Notes on Theory of Distributed Systems

James Aspnes

2024-10-17 22:13

Contents

List of Figures

List of Tables

List of Algorithms

Preface

These are notes for the Yale course CPSC 465/565 Theory of Distributed Systems. This document also incorporates the lecture schedule and assignments, as well as some sample assignments from previous semesters. Because this is a work in progress, it will be updated frequently over the course of the semester.

The most recent version of these notes will be available at https://www.cs.yale.edu/homes/aspnes/classes/465/notes.pdf. More stable archival versions may be found at https://arxiv.org/abs/2001.04235.

Not all topics in the notes will be covered during a particular semester. Some chapters have not been updated and are marked as possibly out of date.

Much of the structure of the course follows Attiya and Welch's *Distributed Computing* [AW04], with some topics based on Lynch's *Distributed Algorithms* [Lyn96] and additional readings from the research literature. In most cases you'll find these materials contain much more detail than what is presented here, so it may be better to consider this document a supplement to them than to treat it as your primary source of information.

Acknowledgments

Many parts of these notes were improved by feedback from students taking various versions of this course, as well as others who have kindly pointed out errors in the notes after reading them online. Many of these suggestions, sadly, went unrecorded, so I must apologize to the many students who should be thanked here but whose names I didn't keep track of in the past. However, I can thank Mike Marmar and Hao Pan in particular for suggesting improvements to some of the posted solutions, Guy Laden for suggesting corrections to Figure 12.1, and Ali Mamdouh for pointing out an error in the original presentation of Algorithm 5.2.

Syllabus

Description

Models of asynchronous distributed computing systems. Fundamental concepts of concurrency and synchronization, communication, reliability, topological and geometric constraints, time and space complexity, and distributed algorithms.

Meeting times

Lectures are Mondays and Wednesdays, from 14:30 to 15:45 in [[[**To be announced.**]]].

On-line course information

The lecture schedule, course notes, and all assignments can be found in a single gigantic PDF file at https://www.cs.yale.edu/homes/aspnes/classes/465/notes.pdf. You should probably bookmark this file, as it will be updated frequently.

Staff

The instructor for the course is James Aspnes. Office: AKW 401. Email: james.aspnes@gmail.com. URL: https://www.cs.yale.edu/homes/aspnes/.

Teaching fellows [[[**To be announced.**]]].

Office hours can be found in the calendar on James Aspnes's web page.

Textbook

The primary course textbook is the lecture notes.

You may also find it helpful to look at the textbook on which the notes were originally based:

Hagit Attiya and Jennifer Welch, *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*, second edition. Wiley, 2004. QA76.9.D5 A75X 2004 (LC). ISBN 0471453242.

On-line version: https://dx.doi.org/10.1002/0471478210. (This may not work outside Yale.)

Errata: http://www.cs.technion.ac.il/~hagit/DC/2nd-errata.html.

Course requirements

If you are taking the class as CPSC 465: Five graded assignments (100% of the semester grade).

If you are taking the class as CPSC 565: Five graded assignments (85% of the semester grade), plus a brief presentation (15%).

Each presentation will be a short description of the main results in a relevant paper chosen in consultation with the instructor, and (circumstances permitting) will be done live during one of the last few lecture slots. If numbers and time permit, it may be possible to negotiate doing a presentation even if you are taking the class as CPSC 465.

Use of outside help

Students are free to discuss homework problems and course material with each other, and to consult with the instructor or a TF. Solutions handed in, however, should be the student's own work. If a student benefits substantially from hints or solutions received from fellow students or from outside sources, then the student should hand in their solution but acknowledge the outside sources, and we will apportion credit accordingly. Using outside resources in solving a problem is acceptable but plagiarism is not.

Questions and comments

Please feel free to send questions or comments on the class or anything connected to it to the instructor at james.aspnes@gmail.com.

For questions about assignments, you may be able to get a faster response using the course Discord server, invite link https://discord.gg/tqnKMAugSG. Note that questions you ask there may be visible to other students if sent to a public channel, so be careful not to broadcast your draft solutions.

Late assignments

Late assignments will not be accepted without a Dean's Excuse.

Academic integrity statement

The graduate school asks that the following statement be included in all graduate course syllabi:

> Academic integrity is a core institutional value at Yale. It means, among other things, truth in presentation, diligence and precision in citing works and ideas we have used, and acknowledging our collaborations with others. In view of our commitment to maintaining the highest standards of academic integrity, the Graduate School Code of Conduct specifically prohibits the following forms of behavior: cheating on examinations, problem sets and all other forms of assessment; falsification and/or fabrication of data; plagiarism, that is, the failure in a dissertation, essay or other written exercise to acknowledge ideas, research, or language taken from others; and multiple submission of the same work without obtaining explicit written permission from both instructors before the material is submitted. Students found guilty of violations of academic integrity are subject to one or more of the following penalties: written reprimand, probation, suspension (noted on a student's transcript) or dismissal (noted on a student's transcript).

Lecture schedule

As always, the future is uncertain, so you should take parts of the schedule that haven't happened yet with a grain of salt. Unless otherwise specified, readings refer to chapters or sections in the course notes.

2025-01-13 What distributed computing is and why we have a theory of it. Basic models: message passing, shared memory, local interactions. Configurations, events, executions, and schedules. The adversary. Basic message-passing model. A simple flooding protocol. Safety properties and invariants. Fairness, liveness properties, and progress measures. Readings: Chapters 1 and 2; §3.1.1.

2025-01-15 [[[**To be announced.**]]]

2025-01-22 [[[**To be announced.**]]]

2025-01-24 [[[**To be announced.**]]]

2025-01-27 [[[**To be announced.**]]]

2025-01-29 [[[**To be announced.**]]]

2025-02-03 [[[**To be announced.**]]]

2025-02-05 [[[**To be announced.**]]]

2025-02-10 [[[**To be announced.**]]]

2025-02-12 [[[**To be announced.**]]]

2025-02-17 [[[**To be announced.**]]]

2025-02-19 [[[**To be announced.**]]]

2025-02-24 [[[**To be announced.**]]]

2025-02-26 [[[**To be announced.**]]]

2025-03-03 [[[**To be announced.**]]]

2025-03-05 [[[**To be announced.**]]]

2025-03-24 [[[**To be announced.**]]]

2025-03-26 [[[**To be announced.**]]]

2025-03-31 [[[**To be announced.**]]]

2025-04-02 [[[**To be announced.**]]]

2025-04-07 [[[**To be announced.**]]]

2025-04-09 [[[**To be announced.**]]]

2025-04-14 [[[**To be announced.**]]]

2025-04-16 [[[**To be announced.**]]]

2025-04-21 CPSC 565 student presentations. (Normal class time and location.)

2025-04-22 CPSC 565 student presentations. (Zoom overflow session, time [[[**To be announced.**]]].)

2025-04-23 CPSC 565 student presentations. (Normal class time and location.)

Chapter 1

Introduction

Distributed systems are characterized by their structure: a typical distributed system will consist of some large number of interacting devices that each run their own programs but that are affected by receiving messages, or observing shared-memory updates or the states of other devices. Examples of distributed systems range from simple systems in which a single client talks to a single server to huge amorphous networks like the Internet as a whole.

As distributed systems get larger, it becomes harder and harder to predict or understand their behavior. Part of the reason for this is that we as programmers have not yet developed a standardized set of tools for managing complexity (like subroutines or objects with narrow interfaces, or even simple structured programming mechanisms like loops or if/then statements) as are found in sequential programming. Part of the reason is that large distributed systems bring with them large amounts of inherent **nondeterminism**—unpredictable events like delays in message arrivals, the sudden failure of components, or in extreme cases the nefarious actions of faulty or malicious machines opposed to the goals of the system as a whole. Because of the unpredictability and scale of large distributed systems, it can often be difficult to test or simulate them adequately. Thus there is a need for theoretical tools that allow us to prove properties of these systems that will let us use them with confidence.

The first task of any theory of distributed systems is modeling: defining a mathematical structure that abstracts out all relevant properties of a large distributed system. There are many foundational models in the literature for distributed systems, but for this class we will follow [AW04] and use simple automaton-based models.

What this means is that we model each process in the system as an automaton that has some sort of local **state**, and model local computation as a transition rule that tells us how to update this state in response to various **events**. Depending on what kinds of system we are modeling, these events might correspond to local computation, to delivery of a message by a network, carrying out some operation on a shared memory, or even something like a chemical reaction between two molecules. The transition rule for a system specifies how the states of all processes involved in the event are updated, based on their previous states. We can think of the transition rule as an arbitrary mathematical function (or relation if the processes are nondeterministic); this corresponds in programming terms to implementing local computation by processes as a gigantic table lookup.

Obviously this is not how we program systems in practice. But what this approach does is allow us to abstract away completely from how individual processes work, and emphasize how all of the processes interact with each other. This can lead to odd results: for example, it's perfectly consistent with this model for some process to be able to solve the halting problem, or carry out arbitrarily complex calculations between receiving a message and sending its response. A partial justification for this assumption is that in practice, the multi-millisecond latencies in even reasonably fast networks are eons in terms of local computation. And as with any assumption, we can always modify it if it gets us into trouble.

1.1 Models

The global state consisting of all process states is called a **configuration**, and we think of the system as a whole as passing from one global state or **configuration** to another in response to each event. When this occurs the processes participating in the event update their states, and the other processes do nothing. This does not model concurrency directly; instead, we interleave potentially concurrent events in some arbitrary way. The advantage of this interleaving approach is that it gives us essentially the same behavior as we would get if we modeled simultaneous events explicitly, but still allows us to consider only one event at a time and use induction to prove various properties of the sequence of configurations we might reach.

We will often use lowercase Greek letters for individual events or sequences of events. Configurations are typically written as capital Latin letters (often C). An **execution** of a schedule is an alternating sequence of configurations and events $C_0\sigma_1C_1\sigma_2C_2\ldots$, where C_{i+1} is the configuration that results from

applying event σ_i to configuration C. A **schedule** is a sequence of events $\sigma_1\sigma_2\ldots$ from some execution. We say that an event σ is **enabled** in C if this event can be carried out in C; an example would be that the event that we deliver a particular message in a message-passing system is enabled only if that message has been sent and not yet delivered. When σ is enabled in C, it is sometime convenient to write $C\sigma$ for the configuration that results from applying σ to C.

What events are available, and what effects they have, will depend on what kind of model we are considering. We may also have additional constraints on what kinds of schedules are **admissible**, which restricts the schedules under consideration to those that have certain desirable properties (say, every message that is sent is eventually delivered). There are many models in the distributed computing literature, which can be divided into a handful of broad categories:

- **Message passing** models (which we will cover in Part I) correspond to systems where processes communicate by sending messages through a network. In **synchronous message-passing**, every process sends out messages at time t that are delivered at time $t + 1$, at which point more messages are sent out that are delivered at time $t + 2$, and so on: the whole system runs in lockstep, marching forward in perfect synchrony.[1] Such systems are difficult to build when the components become too numerous or too widely dispersed, but they are often easier to analyze than **asynchronous** systems, where messages are only delivered eventually after some unknown delay. Variants on these models include **semi-synchronous** systems, where message delays are unpredictable but bounded, and various sorts of timed systems. Further variations come from restricting which processes can communicate with which others, by allowing various sorts of failures: **crash failures** that stop a process dead, **Byzantine failures** that turn a process evil, or **omission failures** that drop messages in transit. Or—on the helpful side—we may supply additional tools like **failure detectors** (Chapter 13) or **randomization** (Chapter 24).

- **Shared-memory** models (Part II) correspond to systems where processes communicate by executing operations on shared objects

[1] In an interleaving model, these apparently simultaneous events are still recorded one at a time. What makes the system synchronous is that we demand that, in any admissible schedule, all n events for time t occur as a sequential block, followed by all n events for time $t + 1$, and so on.

In the simplest case, the objects are simple memory cells supporting read and write operations. These are called **atomic registers**. But in general, the objects could be more complex hardware primitives like **compare-and-swap** (§19.2.3), **load-linked/store-conditional** (§19.2.3), **atomic queues**, or even more exotic objects from the seldom-visited theoretical depths.

Practical shared-memory systems may be implemented as **distributed shared-memory** (Chapter 17) on top of a message-passing system. This gives an alternative approach to designing message-passing systems if it turns out that shared memory is easier to use for a particular problem.

Like message-passing systems, shared-memory systems must also deal with issues of asynchrony and failures, both in the processes and in the shared objects.

Realistic shared-memory systems have additional complications, in that modern CPUs allow out-of-order execution in the absence of special (and expensive) operations called **fences** or **memory barriers**.[AG95] We will effectively be assuming that our shared-memory code is liberally sprinkled with these operations so that nothing surprising happens, but this is not always true of real production code, and indeed there is work in the theory of distributed computing literature on algorithms that don't require unlimited use of memory barriers.

- A third family of models has no communication mechanism independent of the processes. Instead, the processes may directly observe the states of other processes. These models are used in analyzing **self-stabilization**, for some **biologically inspired systems**, and for computation by **population protocols** or **chemical reaction networks**. We will discuss some of this work in Part III.

- Other specialized models emphasize particular details of distributed systems, such as the labeled-graph models used for analyzing routing or the topological models used to give a very high-level picture of various distributed decision problems (see Chapter 29).

We'll see many of these at some point in this course, and examine which of them can simulate each other under various conditions.

1.2 Properties

Properties we might want to prove about a system include:

- **Safety** properties, of the form "nothing bad ever happens" or, more precisely, "there are no bad reachable configurations." These include things like "at most one of the traffic lights at the intersection of Busy Road and Main Street is ever green" or "every value read from a counter equals the number of preceding increment operations." Such properties are typically proved using an , a property of configurations that is true initially and that is preserved by all transitions (this is essentially a disguised induction proof).

- **Liveness** properties, of the form "something good eventually happens." An example might be "my email is eventually either delivered or returned to me." These are not properties of particular states (I might unhappily await the eventual delivery of my email for decades without violating the liveness property just described), but of executions, where the property must hold starting at some finite time. Liveness properties are generally proved either from other liveness properties (e.g., "all messages in this message-passing system are eventually delivered") or from a combination of such properties and some sort of timer argument where some progress metric improves with every transition and guarantees the desirable state when it reaches some bound (also a disguised induction proof).

- **Fairness** properties are a strong kind of liveness property of the form "something good eventually happens to everybody." Such properties exclude **starvation**, a situation where most of the kids are happily chowing down at the orphanage ("some kid eventually eats something" is a liveness property) but poor Oliver Twist is dying in the corner for lack of gruel.

- **Simulations** show how to build one kind of system from another, such as a reliable message-passing system built on top of an unreliable system (TCP [Pos81]), a shared-memory system built on top of a message-passing system (distributed shared memory—see Chapter 17), or a synchronous system build on top of an asynchronous system (synchronizers—see Chapter 7).

- **Impossibility results** describe things we can't do. For example, the classic **Two Generals** impossibility result (Chapter 8) says that it's

> impossible to guarantee agreement between two processes across an unreliable message-passing channel if even a single message can be lost. Other results characterize what problems can be solved if various fractions of the processes are unreliable, or if asynchrony makes timing assumptions impossible. These results, and similar lower bounds that describe things we can't do quickly, include some of the most technically sophisticated results in distributed computing. They stand in contrast to the situation with sequential computing, where the reliability and predictability of the underlying hardware makes proving lower bounds extremely difficult.

There are some basic proof techniques that we will see over and over again in distributed computing.

For **lower bound** and **impossibility** proofs, the main tool is the **indistinguishability** argument. Here we construct two (or more) executions in which some process has the same input and thus behaves the same way, regardless of what algorithm it is running. This exploitation of process's ignorance is what makes impossibility results possible in distributed computing despite being notoriously difficult in most areas of computer science.[2]

For **safety properties**, statements that some bad outcome never occurs, the main proof technique is to construct an **invariant**. An invariant is essentially an induction hypothesis on reachable configurations of the system; an invariant proof shows that the invariant holds in all initial configurations, and that if it holds in some configuration, it holds in any configuration that is reachable in one step.

Induction is also useful for proving **termination** and **liveness** properties, statements that some good outcome occurs after a bounded amount of time. Here we typically structure the induction hypothesis as a **progress measure**, where we argue that each time unit causes the progress measure to advance by some predictable amount, and that when the progress measure reaches a particular value, our desired outcome is achieved.

[2] An exception might be lower bounds for data structures, which also rely on a process's ignorance.

Part I

Message passing

Chapter 2

Model

Message passing models simulate networks. Because any interaction between physically separated processors requires transmitting information from one place to another, all distributed systems are, at a low enough level, message-passing systems. We start by defining a formal model of these systems.

2.1 Basic message-passing model

We have a collection of n **processes** $p_1 \ldots p_2$, each of which has a **state** consisting of a state from from state set Q_i. We think of these processes as nodes in a directed **communication graph** or **network**. The edges in this graph are a collection of point-to-point **channels** or **buffers** b_{ij}, one for each pair of adjacent processes i and j, representing messages that have been sent but that have not yet been delivered. Implicit in this definition is that messages are point-to-point, with a single sender and recipient: if you want broadcast, you have to build it yourself.

A **configuration** of the system consists of a vector of states, one for each process and channel. The configuration of the system is updated by an **event**, in which (1) zero or more messages in channels b_{ij} are delivered to process p_j, removing them from b_{ij}; (2) p_j updates its state in response; and (3) zero or more messages are added by p_j to outgoing channels b_{ji}. We generally think of these events as **delivery events** when at least one message is delivered, and as **computation events** when none are. An **execution segment** is a sequence of alternating configurations and events $C_0, \phi_1, C_1, \phi_2, \ldots$, in which each triple $C_i \phi_{i+1} C_{i+1}$ is consistent with the transition rules for the event ϕ_{i+1}, and the last element of the sequence (if any) is a configuration. If the first configuration C_0 is an **initial configuration** of the system, we have an

execution. A **schedule** is an execution with the configurations removed.

2.1.1 Formal details

Let P be the set of processes, Q the set of process states, and M the set of possible messages.

Each process p_i has a state $\mathsf{state}_i \in Q$. Each channel b_{ij} has a state $\mathsf{buffer}_{ij} \in \mathcal{P}(M)$. We assume each process has a **transition function** $\delta : Q \times \mathcal{P}(M) \to Q \times \mathcal{P}(P \times M)$ that maps tuples consisting of a state and a set of incoming messages a new state and a set of recipients and messages to be sent. An important feature of the transition function is that the process's behavior can't depend on which of its previous messages have been delivered or not. A delivery event $\mathsf{del}(i, A)$, where $A = \{(j_k, m_k)\}$) removes each message m_k from b_{ji}, updates state_i according to $\delta(\mathsf{state}_i, A)$, and adds the outgoing messages specified to $\delta(\mathsf{state}_i, A)$ to the appropriate channels. A computation event $\mathsf{comp}(i)$ does the same thing, except that it applies $\delta(\mathsf{state}_j, \emptyset)$.

Some implicit features in this definition:

- A process can't tell when its outgoing messages are delivered, because the channel states aren't available as input to δ.
- Processes are **deterministic**: The next action of each process depends only on its current state, and not on extrinsic variables like the phase of the moon, coin-flips, etc. We may wish to relax this condition later by allowing coin-flips; to do so, we will need to extend the model to incorporate probabilities.
- It is possible to determine the accessible state of a process by looking only at events that involve that process. Specifically, given a schedule S, define the **restriction** $S|i$ to be the subsequence consisting of all $\mathsf{comp}(i)$ and $\mathsf{del}(i, A)$ events (ranging over all possible A). Since these are the only events that affect the state of i, and only the state of i is needed to apply the transition function, we can compute the state of i looking only at $S|i$. In particular, this means that i will have the same accessible state after any two schedules S and S' where $S|i = S'|i$, and thus will take the same actions in both schedules. This is the basis for **indistinguishability proofs** (§8.2), a central technique in obtaining lower bounds and impossibility results.

Attiya and Welch [AW04] use a different model in which communication channels are not modeled separately from processes, but instead are baked

into processes as outbuf and inbuf variables. This leads to some oddities like having to distinguish the accessible state of a process (which excludes the outbufs) from the full state (which doesn't). Our approach is close to that of Lynch [Lyn96], in that we have separate automata representing processes and communication channels. But since the resulting model produces essentially the same executions, the exact details don't really matter.[1]

2.1.2 Network structure

It may be the case that not all processes can communicate directly; if so, we impose a network structure in the form of a directed graph, where i can send a message to j if and only if there is an edge from i to j in the graph. Typically we assume that each process knows the identity of all its neighbors.

For some problems (e.g., in peer-to-peer systems or other **overlay networks**) it may be natural to assume that there is a fully-connected underlying network but that we have a dynamic network on top of it, where processes can only send to other processes that they have obtained the addresses of in some way.

2.2 Asynchronous systems

In an **asynchronous** model, only minimal restrictions are placed on when messages are delivered and when local computation occurs. A schedule is said to be **admissible** if (a) there are infinitely many computation steps for each process, and (b) every message is eventually delivered. (These are **fairness** conditions.) The first condition (a) assumes that processes do not explicitly terminate, which is the assumption used in [AW04]; an alternative, which we will use when convenient, is to assume that every process either has infinitely many computation steps or reaches an explicit halting state.

[1] The late 1970s and early 1980s saw a lot of research on finding the "right" definition of a distributed system, and some of the disputes from that era were hard fought. But in the end, all the various proposed models turned out to be more or less equivalent, which is not surprising since the authors were ultimately trying to represent the same intuitive understanding of these systems. So most distributed computing papers now just use some phrasing like "we consider the standard model of an asynchronous message-passing system" and leave it to the reader to assume that this standard model is their favorite one.

An example of this trick in action is that you will never see $\mathsf{del}(i, A)$ or $\mathsf{comp}(i)$ again after you finish reading this footnote.

2.2.1 Example: client-server computing

Almost every distributed system in practical use is based on **client-server** interactions. Here one process, the **client**, sends a **request** to a second process, the **server**, which in turn sends back a **response**. We can model this interaction using our asynchronous message-passing model by describing what the transition functions for the client and the server look like: see Algorithms 2.1 and 2.2.

```
1 initially do
2     send request to server
3     upon receiving response do
4         update state
```

Algorithm 2.1: Client-server computation: client code

```
1 upon receiving request do
2     send response to client
```

Algorithm 2.2: Client-server computation: server code

The interpretation of Algorithm 2.1 is that the client sends request (by adding it to its outbuf) in its very first computation event (after which it does nothing). The interpretation of Algorithm 2.2 is that in any computation event where the server observes request in its inbuf, it sends response.

We want to claim that the client eventually receives response in any admissible execution. To prove this, observe that:

1. After finitely many steps, the client carries out a computation event. This computation event puts request in the message buffer between the client and server.

2. After finitely many more steps, a delivery event occurs that delivers request to the server. This causes the server to send response.

3. After finitely many more steps, a delivery event delivers response to the client, causing it to process response.

Each step of the proof is justified by the constraints on admissible executions. If we could run for infinitely many steps without a particular

process doing a computation event or a particular message being delivered, we'd violate those constraints.

Most of the time we will not attempt to prove the correctness of a protocol at quite this level of tedious detail. But if you are only interested in distributed algorithms that people actually use, you have now seen a proof of correctness for 99.9% of them, and do not need to read any further.

2.3 Synchronous systems

A **synchronous message-passing** system is exactly like an asynchronous system, except we insist that the schedule consists of alternating phases in which (a) every process executes a computation step (that may send messages), and (b) all messages are delivered while none are sent.[2] The combination of a computation phase and a delivery phase is called a **round**. Synchronous systems are effectively those in which all processes execute in lock-step, and there is no timing uncertainty. This makes protocols much easier to design, but makes them less resistant to real-world timing oddities. Sometimes this can be dealt with by applying a **synchronizer** (Chapter 7), which transforms synchronous protocols into asynchronous protocols at a small cost in complexity.

2.4 Drawing message-passing executions

Though formally we can describe an execution in a message-passing system as a long list of events, this doesn't help much with visualizing the underlying communication pattern. So it can sometimes be helpful to use a more visual representation of a message-passing execution that shows how information flows through the system.

A typical example is given in Figure 2.1. In this picture, time flows from left to right, and each process is represented by a horizontal line. This convention reflects the fact that processes have memory, so any information available to a process at some time t is also available at all times $t' \geq t$. Events are represented by marked points on these lines, and messages are represented by diagonal lines between events. The resulting picture looks like a collection of **world lines** as used in physics to illustrate the path taken by various objects through spacetime.

[2] Formally, the delivery phase consists of n separate delivery events, in any order, that between them clean out all the channels.

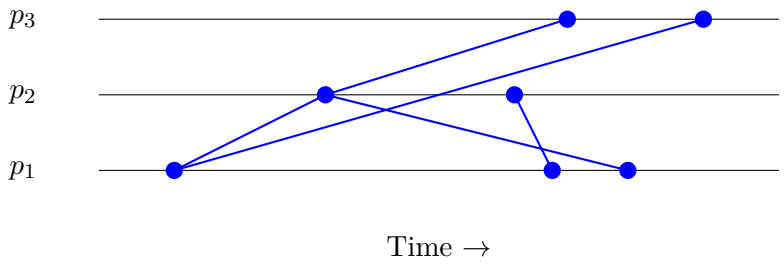


Figure 2.1: Asynchronous message-passing execution. Time flows left-to-right. Horizontal lines represent processes. Nodes represent events. Diagonal edges between events represent messages. In this execution, p_1 executes a computation event that sends messages to p_2 and p_3. When p_2 receives this message, it sends messages to p_1 and p_3. Later, p_2 executes a computation event that sends a second message to p_1. Because the system is asynchronous, there is no guarantee that messages arrive in the same order they are sent.

Pictures like Figure 2.1 can be helpful for illustrating the various constraints we might put on message delivery. In Figure 2.1, the system is completely asynchronous: messages can be delivered in any order, even if sent between the same processes. If we run the same protocol under stronger assumptions, we will get different communication patterns.

For example, Figure 2.2 shows an execution that is still asynchronous but that assumes FIFO (first-in first-out) channels. A **FIFO channel** from some process p to another process q guarantees that q receives messages in the same order that p sends them (this can be simulated by a non-FIFO channel by adding a **sequence number** to each message, and queuing messages at the receiver until all previous messages have been processed).

If we go as far as to assume synchrony, we get the execution in Figure 2.3. Now all messages take exactly one time unit to arrive, and computation events follow each other in lockstep.

2.5 Complexity measures

There is no explicit notion of time in the asynchronous model, but we can define a time measure by adopting the rule that every message is delivered and processed at most 1 time unit after it is sent. Formally, we assign time 0 to the first event, and assign the largest time we can to each subsequent event, subject to the constraints that (a) no event is assigned a larger time than any later event; (b) if a message m from i to j is created by an event at

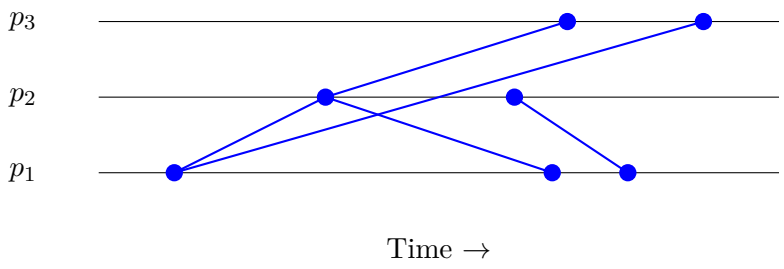


Figure 2.2: Asynchronous message-passing execution with FIFO channels. Multiple messages from one process to another are now guaranteed to be delivered in the order they are sent.

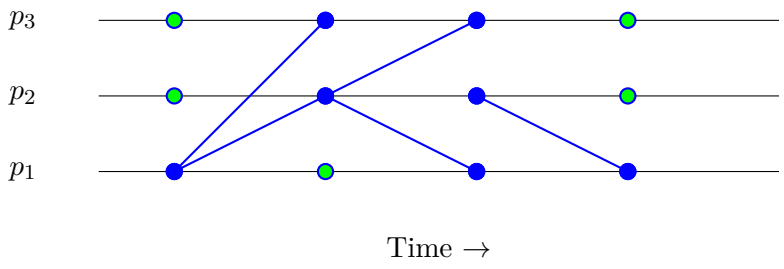


Figure 2.3: Synchronous message-passing execution. All messages are now delivered in exactly one time unit, and computation events immediately follow the delivery events.

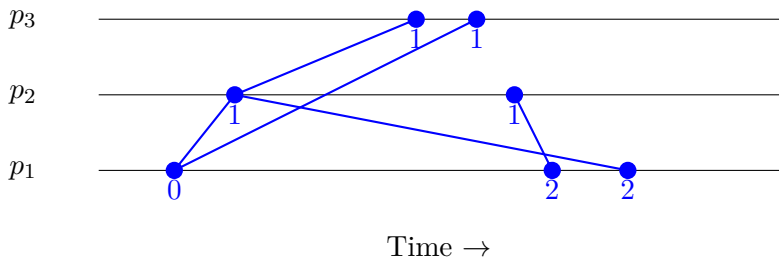


Figure 2.4: Asynchronous message-passing execution with times.

time t, then the time for the delivery of m from i to j is no greater than $t+1$, and (c) any computation step is assigned a time no later than the previous event at the same process (or 0 if the process has no previous events). This is consistent with an assumption that message propagation takes at most 1 time unit and that local computation takes 0 time units.

Another way to look at this is that it is a definition of a time unit in terms of maximum message delay together with an assumption that message delays dominate the cost of the computation. This last assumption is pretty much always true for real-world networks with any non-trivial physical separation between components, thanks to speed of light limitations.

An example of an execution annotated with times in this way is given in Figure 2.4.

The **time complexity** of a protocol (that terminates) is the time of the last event at any process.

Note that looking at **step complexity**, the number of computation events involving either a particular process (**individual step complexity**) or all processes (**total step complexity**) is not useful in the asynchronous model, because a process may be scheduled to carry out arbitrarily many computation steps without any of its incoming or outgoing messages being delivered, which probably means that it won't be making any progress. These complexity measures will be more useful when we look at shared-memory models (Part II).

For a protocol that terminates, the **message complexity** is the total number of messages sent. We can also look at message length in bits, total bits sent, and so on, if these are useful for distinguishing our new improved protocol from last year's model.

For synchronous systems, time complexity becomes just the number of rounds until a protocol finishes. Message complexity is still only loosely connected to time complexity; for example, there are synchronous **leader**

election (Chapter 5) algorithms that, by virtue of grossly abusing the synchrony assumption, have unbounded time complexity but very low message complexity.

Chapter 3

Broadcast and convergecast

Here we'll describe protocols for propagating information throughout a network from some central initiator and gathering information back to that same initiator. We do this both because the algorithms are actually useful and because they illustrate some of the issues that come up with keeping time complexity down in an asynchronous message-passing system.

3.1 Flooding

Flooding is about the simplest of all distributed algorithms. It's dumb and expensive, but easy to implement, and gives you both a broadcast mechanism and a way to build rooted spanning trees.

We'll give a fairly simple presentation of flooding roughly following Chapter 2 of [AW04]. For more recent work on flooding see [**?**].

3.1.1 Basic algorithm

The basic flooding algorithm is shown in Algorithm 3.1. The idea is that when a process receives a message M, it forwards it to all of its neighbors unless it has seen it before, which it tracks using a single bit seen-message.

Theorem 3.1.1. *Every process receives M after at most D time and at most $|E|$ messages, where D is the diameter of the network and E is the set of (directed) edges in the network.*

Proof. Message complexity: Each process only sends M to its neighbors once, so each edge carries at most one copy of M.

Time complexity: By induction on $d(\mathsf{root}, v)$, we'll show that each v receives M for the first time no later than time $d(\mathsf{root}, v) \leq D$. The base

```
1  initially do
2      if pid = root then
3          seen-message ← true
4          send M to all neighbors
5      else
6          seen-message ← false
7  upon receiving M do
8      if seen-message = false then
9          seen-message ← true
10         send M to all neighbors
```

Algorithm 3.1: Basic flooding algorithm

case is when $v = \mathsf{root}$, $d(\mathsf{root}, v) = 0$; here root receives message at time 0. For the induction step, Let $d(\mathsf{root}, v) = k > 0$. Then v has a neighbor u such that $d(\mathsf{root}, u) = k - 1$. By the induction hypothesis, u receives M for the first time no later than time $k - 1$. From the code, u then sends M to all of its neighbors, including v; M arrives at v no later than time $(k - 1) + 1 = k$. □

Note that the time complexity proof also demonstrates correctness: every process receives M at least once.

As written, this is a one-shot algorithm: you can't broadcast a second message even if you wanted to. The obvious fix is for each process to remember which messages it has seen and only forward the new ones (which costs memory) and/or to add a **time-to-live** (TTL) field on each message that drops by one each time it is forwarded (which may cost extra messages and possibly prevents complete broadcast if the initial TTL is too small). The latter method is what was used for searching in http://en.wikipedia.org/wiki/Gnutella, an early peer-to-peer system. An interesting property of Gnutella was that since the application of flooding was to search for huge (multiple MiB) files using tiny (100 byte) query messages, the actual bit complexity of the flooding algorithm was not especially large relative to the bit complexity of sending any file that was found.

We can optimize the algorithm slightly by not sending M back to the node it came from; this will slightly reduce the message complexity in many cases but makes the proof a sentence or two longer. (It's all a question of what you want to optimize.)

3.1.2 Adding parent pointers

To build a spanning tree, modify Algorithm 3.1 by having each process remember who it first received M from. The revised code is given as Algorithm 3.2

```
1  initially do
2      if pid = root then
3          parent ← root
4          send M to all neighbors
5      else
6          parent ← ⊥
7  upon receiving M from p do
8      if parent = ⊥ then
9          parent ← p
10
11         send M to all neighbors
```

Algorithm 3.2: Flooding with parent pointers

We can easily prove that Algorithm 3.2 has the same termination properties as Algorithm 3.1 by observing that if we map parent to seen-message by the rule $\perp \to$ **false**, anything else $\to$ **true**, then we have the same algorithm. We would like one additional property, which is that when the algorithm **quiesces** (has no outstanding messages), the set of parent pointers form a rooted spanning tree. For this we use induction on time:

Lemma 3.1.2. *At any time during the execution of Algorithm 3.2, the following invariant holds:*

1. *If* u.parent $\neq \perp$, *then* u.parent.parent $\neq \perp$ *and following parent pointers gives a path from* u *to* root.

2. *If there is a message* M *in transit from* u *to* v, *then* u.parent $\neq \perp$.

Proof. We have to show that the invariant is true initially, and that any event preserves the invariant. We'll assume that all events are delivery events for a single message, since we can have the algorithm treat a multi-message delivery event as a sequence of single-message delivery events.

We'll treat the initial configuration as the result of the root setting its parent to itself and sending messages to all its neighbors. It's not hard to verify that the invariant holds in the resulting configuration.

For a delivery event, let v receive M from u. There are two cases: if $v.\mathsf{parent}$ is already non-null, the only state change is that M is no longer in transit, so we don't care about $u.\mathsf{parent}$ any more. If $v.\mathsf{parent}$ is null, then

1. $v.\mathsf{parent}$ is set to u. This triggers the first case of the invariant. From the induction hypothesis we have that $u.\mathsf{parent} \neq \perp$ and that there exists a path from u to the root. Then $v.\mathsf{parent}.\mathsf{parent} = u.\mathsf{parent} \neq \perp$ and the path from $v \to u \to \mathsf{root}$ gives the path from v.

2. Message M is sent to all of v's neighbors. Because M is now in transit from v, we need $v.\mathsf{parent} \neq \perp$; but we just set it to u, so we are happy.

□

At the end of the algorithm, the invariant shows that every process has a path to the root, i.e., that the graph represented by the parent pointers is connected. Since this graph has exactly $|V| - 1$ edges (if we don't count the self-loop at the root), it's a tree.

Though we get a spanning tree at the end, we may not get a very good spanning tree. For example, suppose our friend the adversary picks some Hamiltonian path through the network and delivers messages along this path very quickly while delaying all other messages for the full allowed 1 time unit. Then the resulting spanning tree will have depth $|V| - 1$, which might be much worse than D. If we want the shallowest possible spanning tree, we need to do something more sophisticated: see the discussion of **distributed breadth-first search** in Chapter 4. However, we may be happy with the tree we get from simple flooding: if the message delay on each link is consistent, then it's not hard to prove that we in fact get a shortest-path tree. As a special case, flooding always produces a BFS tree in the synchronous model.

Note also that while the algorithm works in a directed graph, the parent pointers may not be very useful if links aren't two-way.

3.1.3 Identifying children

By adding acknowledgment messages, it is possible for each node to learn exactly which of its neighbors become its children. Because the system is asynchronous, this requires each neighbor to inform the node both whether it is a child (using an ack message) and when it is not (using a nack message); only upon receiving one or the other of these messages will the node know that it's not going to receive the other.

The modified code is given in Algorithm 3.3

```
1  initially do
2      nonChildren ← ∅
3      if pid = root then
4          parent ← root
5          children ← {root}
6          send M to all neighbors
7      else
8          parent ← ⊥
9          children ← ∅
10 upon receiving M from p do
11     if parent = ⊥ then
12         parent ← p
13
14         send ack to p
15         send M to all neighbors
16     else
17         send nack to p
18 upon receiving ack from p do
19     children ← children ∪ {p}
20 upon receiving nack from p do
21     nonChildren ← nonChildren ∪ {p}
```

Algorithm 3.3: Flooding tracking children

If we take an execution of Algorithm 3.3 and remove all the ack and nack messages, we get an execution of Algorithm 3.2. So all of the properties that we proved for Algorithm 3.2 continue to hold.

For the improved algorithm, we'd like to show that when the algorithm quiesces, every node p_i has a list of all the nodes p_j for which $p_j.\mathsf{parent} = p_i$ in $p_i.\mathsf{children}$ and a list of all the neighbors p_j for which $p_j.\mathsf{parent} \neq p_i$ in $p_i.\mathsf{nonChildren}$.

We can do this by showing a mix of safety and liveness properties:

1. (Safety) If $p_j \in p_i.\mathsf{children}$, then $p_j.\mathsf{parent} = p_i$. Proof sketch: Verify the strengthening of this property that adds $\mathsf{ack} \in b_{ji}$ implies $p_j.\mathsf{parent} = p_i$ is an invariant.

2. (Safety) If $p_j \in p_i.\mathsf{nonChildren}$, then $p_j.\mathsf{parent} \notin \{p_i, \perp\}$. Proof sketch: Verify the strengthening of this property that adds $\mathsf{nack} \in b_{ji}$ implies $p_j.\mathsf{parent} \notin \{p_i, \perp\}$ is an invariant.

3. (Liveness) Eventually, every neighbor of p_i appears in $p_i.\mathsf{children} \cup p_i.\mathsf{nonChildren}$. Proof: We have previously shown that every node p_i eventually sets $p_i.\mathsf{parent} \neq \emptyset$. From the code we have that whenever a node does this, it sends M to all neighbors. For each neighbor p_j, observe that upon receiving M it responds with exactly one of ack or nack. When this message is eventually delivered, p_j is added to $p_i.\mathsf{children} \cup p_i.\mathsf{nonChildren}$.

Since we assume that each p_i knows which nodes are its neighbors, we can use the property that $p_i.\mathsf{children} \cup p_i.\mathsf{nonChildren}$ includes all neighbors as a kind of local termination test. This can be handy if we want to use flooding as the first step in some larger protocol.

3.2 Convergecast

A **convergecast** is the inverse of broadcast: instead of a message propagating down from a single root to all nodes, data is collected from outlying nodes to the root. Typically some function is applied to the incoming data at each node to summarize it, with the goal being that eventually the root obtains this function of all the data in the entire system. (Examples would be counting all the nodes or taking an average of input values at all the nodes.)

A basic convergecast algorithm is given in Algorithm 3.4; it propagates information up through a previously-computed spanning tree.

```
1  initially do
2      if I am a leaf then
3          send input to parent
4  upon receiving M from c do
5      append (c, M) to buffer
6      if buffer contains messages from all my children then
7          v ← f(buffer, input)
8          if pid = root then
9              return v
10         else
11             send v to parent
```

Algorithm 3.4: Convergecast

The details of what is being computed depend on the choice of f:

- If input $= 1$ for all nodes and f is sum, then we count the number of nodes in the system.
- If input is arbitrary and f is sum, then we get a total of all the input values.
- Combining the above lets us compute averages, by dividing the total of all the inputs by the node count.
- If f just concatenates its arguments, the root ends up with a vector of all the input values.

Running time is bounded by the depth of the tree: we can prove by induction that any node at height h (height is length of the longest path from this node to some leaf) sends a message by time h at the latest. Message complexity is exactly $n - 1$, where n is the number of nodes; this is easily shown by observing that each node except the root sends exactly one message.

Proving that convergecast returns the correct value is similarly done by induction on depth: if each child of some node computes a correct value, then that node will compute f applied to these values and its own input. What the result of this computation is will, of course, depend on f; it generally makes the most sense when f represents some associative operation (as in the examples above).

3.3 Flooding and convergecast together

A natural way to build the spanning tree used by convergecast is to run flooding first. This also provides a mechanism for letting the leaves know that they are leaves and initiating the protocol. The combined algorithm is shown as Algorithm 3.5.

```
1  initially do
2      children ← ∅
3      nonChildren ← ∅
4      if pid = root then
5          parent ← root
6          send init to all neighbors
7      else
8          parent ← ⊥
9  upon receiving init from p do
10     if parent = ⊥ then
11         parent ← p
12         send init to all neighbors
13     else
14         send nack to p
15 upon receiving nack from p do
16     nonChildren ← nonChildren ∪ {p}
17 as soon as children ∪ nonChildren includes all my neighbors do
18     v ← f(buffer, input)
19     if pid = root then
20         return v
21     else
22         send ack(v) to parent
23 upon receiving ack(v) from k do
24     add (k, v) to buffer
25     add k to children
```

Algorithm 3.5: Flooding and convergecast combined

However, this may lead to very bad time complexity for the convergecast stage. Consider a wheel-shaped network consisting of one central node p_0 connected to nodes $p_1, p_2, \ldots, p_{n-1}$, where each p_i is also connected to p_{i+1}.

By carefully arranging for the $p_i p_{i+1}$ links to run much faster than the $p_0 p_i$ links, the adversary can make flooding build a tree that consists of a single path $p_0 p_1 p_2 \dots p_{n-1}$, even though the diameter of the network is only 2. While it only takes 2 time units to build this tree (because every node is only one hop away from the initiator), when we run convergecast we suddenly find that the previously-speedy links are now running only at the guaranteed ≤ 1 time unit per hop rate, meaning that convergecast takes $n - 1$ time.

This may be less of an issue in real networks, where the latency of links may be more uniform over time, meaning that a deep tree of fast links is still likely to be fast when we reach the convergecast step. But in the worst case we will need to be more clever about building the tree. We show how to do this in Chapter 4.

Chapter 4

Distributed breadth-first search

Here we describe some algorithms for building a **breadth-first search** (**BFS**) tree in a network. All assume that there is a designated **initiator** node that starts the algorithm. At the end of the execution, each node except the initiator has a parent pointer and every node has a list of children. These are consistent and define a BFS tree: nodes at distance k from the initiator appear at level k of the tree.

In a synchronous network, **flooding** (§3.1) solves BFS; see [AW04, Lemma 2.8, page 21] or [Lyn96, §4.2]. So the interesting case is when the network is asynchronous.

In an asynchronous network, the complication is that we can no longer rely on synchronous communication to reach all nodes at distance d at the same time. So instead we need to keep track of distances explicitly, or possibly enforce some approximation to synchrony in the algorithm. (A general version of this last approach is to apply a synchronizer to one of the synchronous algorithms using a **synchronizer**; see Chapter 7.)

To keep things simple, we'll drop the requirement that a parent learn the IDs of its children, since this can be tacked on as a separate notification protocol, in which each child just sends one message to its parent once it figures out who its parent is.

4.1 Using explicit distances

This is a translation of the AsynchBFS automaton from [Lyn96, §15.4]. It's a very simple algorithm, closely related to Dijkstra's algorithm for shortest

paths, but there is otherwise no particular reason to use it. Not only does it not detect termination, but it is also dominated by the $O(D)$ time and $O(DE)$ message complexity synchronizer-based algorithm described in §4.3. (Here D is the **diameter** of the network, the maximum distance between any two nodes.)

The idea is to run flooding with distances attached. Each node sets its distance to 1 plus the smallest distance sent by its neighbors and its parent to the neighbor supplying that smallest distance. A node notifies all its neighbors of its new distance whenever its distance changes.

Pseudocode is given in Algorithm 4.1

```
1  initially do
2      if pid = initiator then
3          distance ← 0
4          send distance to all neighbors
5      else
6          distance ← ∞
7  upon receiving d from p do
8      if d + 1 < distance then
9          distance ← d + 1
10         parent ← p
11         send distance to all neighbors
```

Algorithm 4.1: AsynchBFS algorithm (from [Lyn96])

(See [Lyn96] for a precondition-effect description, which also includes code for buffering outgoing messages.)

The claim is that after at most $O(VE)$ messages and $O(D)$ time, all distance values are equal to the length of the shortest path from the initiator to the appropriate node. The proof is by showing the following:

Lemma 4.1.1. *The variable* $\mathsf{distance}_p$ *is always the length of some path from initiator to* p*, and any message sent by* p *is also the length of some path from* initiator *to* p*.*

Proof. The second part follows from the first; any message sent equals p's current value of distance. For the first part, suppose p updates its distance; then it sets it to one more than the length of some path from initiator to p', which is the length of that same path extended by adding the pp' edge. □

We also need a liveness argument that says that $\mathsf{distance}_p = d(\mathsf{initiator}, p)$ no later than time $d(\mathsf{initiator}, p)$. Note that we can't detect when distance stabilizes to the correct value without a lot of additional work.

In [Lyn96], there's an extra $|V|$ term in the time complexity that comes from message pile-ups, since the model used there only allows one incoming message to be processed per time units (the model in [AW04] doesn't have this restriction). The trick to arranging this to happen often is to build a graph where node 1 is connected to nodes 2 and 3, node 2 to 3 and 4, node 3 to 4 and 5, etc. This allows us to quickly generate many paths of distinct lengths from node 1 to node k, which produces k outgoing messages from node k. It may be that a more clever analysis can avoid this blowup, by showing that it only happens in a few places.

4.2 Using layering

This approach is used in the *LayeredBFS* algorithm in [Lyn96], which is due to Gallager [Gal82].

Here we run a sequence of up to $|V|$ instances of the simple algorithm with a distance bound on each: instead of sending out just 0, the initiator sends out $(0, \mathsf{bound})$, where bound is initially 1 and increases at each phase. A process only sends out its improved distance if it is less than bound.

Each phase of the algorithm constructs a partial BFS tree that contains only those nodes within distance bound of the root. This tree is used to report back to the root when the phase is complete. For the following phase, notification of the increase in bound increase is distributed only through the partial BFS tree constructed so far. With some effort, it is possible to prove that in a bidirectional network that this approach guarantees that each edge is only probed once with a new distance (since distance-1 nodes are recruited before distance-2 nodes and so on), and the bound-update and acknowledgment messages contribute at most $|V|$ messages per phase. So we get $O(E + VD)$ total messages. But the time complexity is bad: $O(D^2)$ in the worst case.

4.3 Using local synchronization

The reason the layering algorithm takes so long is that at each phase we have to phone all the way back up the tree to the initiator to get permission to go on to the next phase. We need to do this to make sure that a node is only recruited into the tree once: otherwise we can get pile-ups on the

channels as in the simple algorithm. But we don't necessarily need to do this globally. Instead, we'll require each node at distance d to delay sending out a recruiting message until it has confirmed that none of its neighbors will be sending it a smaller distance. We do this by having two classes of messages:[1]

- exactly(d): "I know that my distance is d."
- more-than(d): "I know that my distance is $> d$."

The rules for sending these messages for a non-initiator are:

1. I can send exactly(d) as soon as I have received exactly($d-1$) from at least one neighbor and more-than($d-2$) from all neighbors.
2. I can send more-than(d) if $d = 0$ or as soon as I have received more-than($d-1$) from all neighbors.

The initiator sends exactly(0) to all neighbors at the start of the protocol (these are the only messages the initiator sends).

My distance will be the unique distance that I am allowed to send in an exactly(d) messages. Note that this algorithm terminates in the sense that every node learns its distance at some finite time.

If you read the discussion of synchronizers in Chapter 7, this algorithm essentially corresponds to building the **alpha synchronizer** into the synchronous BFS algorithm, just as the layered model builds in the **beta synchronizer**. See [AW04, §11.3.2] for a discussion of BFS using synchronizers. The original approach of applying synchronizers to get BFS is due to Awerbuch [Awe85].

We now show correctness. Under the assumption that local computation takes zero time and message delivery takes at most 1 time unit, we'll show that if $d(\text{initiator}, p) = d$, (a) p sends more-than(d') for any $d' < d$ by time d', (b) p sends exactly(d) by time d, (c) p never sends more-than(d') for any $d' \geq d$, and (d) p never sends exactly(d') for any $d' \neq d$. For parts (c) and (d) we use induction on d'; for (a) and (b), induction on time. This is not terribly surprising: (c) and (d) are safety properties, so we don't need to talk about time. But (a) and (b) are liveness properties so time comes in.

Let's start with (c) and (d). The base case is that the initiator never sends any more-than messages at all, and so never sends more-than(0), and

[1] In an earlier version of these notes, these messages where called distance(d) and not-distance(d); the more self-explanatory exactly and more-than terminology is taken from [BDLP08].

any non-initiator never sends exactly(0). For larger d', observe that if a non-initiator p sends more-than(d') for $d' \geq d$, it must first have received more-than($d'-1$) from all neighbors, including some neighbor p' at distance $d-1$. But the induction hypothesis tells us that p' can't send more-than($d'-1$) for $d'-1 \geq d-1$. Similarly, to send exactly(d') for $d' < d$, p must first have received exactly($d'-1$) from some neighbor p', but again p' must be at distance at least $d-1$ from the initiator and so can't send this message either. In the other direction, to send exactly(d') for $d' > d$, p must first receive more-than($d'-2$) from this closer neighbor p', but then $d'-2 > d-2 \geq d-1$ so more-than($d'-2$) is not sent by p'.

Now for (a) and (b). The base case is that the initiator sends exactly(0) to all nodes at time 0, giving (a), and there is no more-than(d') with $d' < 0$ for it to send, giving (b) vacuously; and any non-initiator sends more-than(0) immediately. At time $t+1$, we have that (a) more-than(t) was sent by any node at distance $t+1$ or greater by time t and (b) exactly(t) was sent by any node at distance t by time t; so for any node at distance $t+2$ we send more-than($t+1$) no later than time $t+1$ (because we already received more-than(t) from all our neighbors) and for any node at distance $t+1$ we send exactly($t+1$) no later than time $t+1$ (because we received all the preconditions for doing so by this time).

Message complexity: A node at distance d sends more-than(d') for all $0 < d' < d$ and exactly(d) and no other messages. So we have message complexity bounded by $|E| \cdot D$ in the worst case. Note that this is gives a bound of $O(DE)$, which is slightly worse than the $O(E + DV)$ bound for the layered algorithm.

Time complexity: It's immediate from (a) and (b) that all messages that are sent are sent by time D, and indeed that any node p learns its distance at time $d(\text{initiator}, p)$. So we have optimal time complexity, at the cost of higher message complexity. I don't know if this trade-off is necessary, or if a more sophisticated algorithm could optimize both.

Our time proof assumes that messages don't pile up on edges, or that such pile-ups don't affect delivery time (this is the default assumption used in [AW04]). A more sophisticated proof could remove this assumption.

One downside of this algorithm is that it has to be started simultaneously at all nodes. Alternatively, we could trigger "time 0" at each node by a broadcast from the initiator, using the usual asynchronous broadcast algorithm; this would give us a BFS tree in $O(|E| \cdot D)$ messages (since the $O(|E|)$ messages of the broadcast disappear into the constant) and $2D$ time. The analysis of time goes through as before, except that the starting time 0 becomes the time at which the last node in the system is woken up by the

broadcast. Further optimizations are possible; see, for example, the paper of Boulinier *et al.* [BDLP08], which shows how to run the same algorithm with constant-size messages.

Chapter 5

Leader election

(See also [AW04, Chapter 3] or [Lyn96, Chapter 3].)

The idea of leader election is that we want a single process to declare itself leader and the others to declare themselves non-leaders. The non-leaders may or may not learn the identity of the leader as part of the protocol; if not, we can usually add an extra phase where the leader broadcasts its identity to the others. The leader should be unique in the sense that there is exactly one process that ever decides it is the leader. This excludes protocols that might accidentally elect two or more leaders even if we eventually remove the extras.

Traditionally, leader election has been used as a way to study the effects of symmetry, and many leader election algorithms are designed for networks in the form of a **ring**. These networks consist of a sequence of processes $p_0, p_1, \ldots, p_{n-1}$, with each process p_i able to send messages only to its immediate neighbors p_{i-1} and p_{i+1} (mod n). Some algorithms work in the weaker model of a **unidirectional ring** where p_i can only send messages to p_{i+1}.

A classic result of Angluin [Ang80] shows that leader election in a ring is impossible if the processes do not start with distinct identities. The proof is that if the processes run synchronously, they all receive and send the same messages in each round, update their state identically, and in the end all put on the crown at the same time. We discuss this result in §5.1.

With ordered identities, a simple algorithm due to Le Lann [LL77] and Chang and Roberts [CR79] solves the problem in $O(n)$ time with $O(n^2)$ messages: I send out my own ID clockwise and forward any ID bigger than mine. If I get my ID back, I win. This works with a unidirectional ring, doesn't require synchrony, and never produces multiple leaders. See §5.2.1.

On a bidirectional ring we can get $O(n \log n)$ messages and $O(n)$ time with power-of-2 probing, using an algorithm of Hirschberg and Sinclair [HS80]. See §5.2.2.

An evil trick: if we have synchronized starting, known n, and known ID space, we can have process with ID i wait until round $i \cdot n$ to start sending its ID around, and have everybody else drop out when they receive it; this way only one process (the one with smallest ID) ever starts a message and only n messages are sent [FL87]. But the running time can be pretty bad.

For general networks, we can apply the same basic strategy as in Le Lann-Chang-Roberts by having each process initiate a broadcast/convergecast algorithm that succeeds only if the initiator has the smallest ID. See §5.3.

Some additional algorithms for the asynchronous ring are given in §§5.2.3 and 5.2.4. Lower bounds are shown in §5.4.

5.1 Symmetry

A system exhibits **symmetry** if we can permute the nodes without changing the behavior of the system. More formally, we can define a symmetry as an **equivalence relation** on processes, where we have the additional properties that all processes in the same equivalence class run the same code; and whenever p is equivalent to p', each neighbor q of p is equivalent to a corresponding neighbor q' of p'.

An example of a network with a lot of symmetries would be an **anonymous ring**, which is a network in the form of a cycle (the ring part) in which every process runs the same code (the anonymous part). In this case all nodes are equivalent. If we have a line, then we might or might not have any non-trivial symmetries: if each node has a **sense of direction** that tells it which neighbor is to the left and which is to the right, then we can identify each node uniquely by its distance from the left edge. But if the nodes don't have a sense of direction, we can flip the line over and pair up nodes that map to each other.[1]

Symmetries are convenient for proving impossibility results, as observed by Angluin [Ang80]. The underlying theme is that without some mechanism for **symmetry breaking**, a message-passing system escape from a symmetric initial configuration. The following lemma holds for **deterministic** systems, basically those in which processes can't flip coins:

[1] Typically, this does not mean that the nodes can't tell their neighbors apart. But it does mean that if we swap the labels for all the neighbors (corresponding to flipping the entire line from left to right), we get the same executions.

Lemma 5.1.1. *A symmetric deterministic message-passing system that starts in an initial configuration in which equivalent processes have the same state has a synchronous execution in which equivalent processes continue to have the same state.*

Proof. Easy induction on rounds: if in some round p and p' are equivalent and have the same state, and all their neighbors are equivalent and have the same state, then p and p' receive the same messages from their neighbors and can proceed to the same state (including outgoing messages) in the next round. □

An immediate corollary is that you can't do leader election in an anonymous system with a symmetry that puts each node in a non-trivial equivalence class, because as soon as I stick my hand up to declare I'm the leader, so do all my equivalence-class buddies.

With **randomization**, Lemma 5.1.1 doesn't directly apply, since we can break symmetry by having my coin-flips come up differently from yours. It does show that we can't guarantee convergence to a single leader in any fixed amount of time (because otherwise we could just fix all the coin flips to get a deterministic algorithm). Depending on what the processes know about the size of the system, it may still be possible to show that a randomized algorithm necessarily fails in some cases.[2]

A more direct way to break symmetry is to assume that all processes have **identities**; now processes can break symmetry by just declaring that the one with the smaller or larger identity wins. This approach is taken in the algorithms in the following sections.

5.2 Leader election in rings

Here we'll describe some basic leader election algorithms for rings. Historically, rings were the first networks in which leader election was studied, because they are the simplest networks whose symmetry makes the problem difficult, and because of the connection to token-ring networks, a method for congestion control in local-area networks that is no longer used much.

[2]Specifically, if the processes don't know the size of the ring, we can imagine a ring of size $2n$ in which the first n processes happen to get exactly the same coin-flips as the second n processes for long enough that two matching processes, one in each region, both think they have won the fight in a ring of size n and declare themself to be the leader.

5.2.1 The Le Lann-Chang-Roberts algorithm

This is about the simplest leader election algorithm there is. It works in a **unidirectional ring**, where messages can only travel clockwise.[3] The algorithm does not require synchrony.

Formally, we'll let the state space for each process i consist of two variables: leader, initially 0, which is set to 1 if i decides it's a leader; and maxId, the largest ID seen so far. We assume that i denotes i's position rather than its ID, which we'll write as id_i. We will also treat all positions as values mod n, to simplify the arithmetic.

The initial version of this algorithm was proposed by Le Lann [LL77]; it involved sending every ID all the way around the ring, and having a node decide it was a leader if it had the largest ID. Chang and Roberts [CR79] improved on this by having nodes refuse to forward any ID smaller than the maximum ID seen so far. This means that only the largest ID makes it all the way around the ring, so a node can declare itself leader the moment it sees its own ID. Depending on the writer, the resulting algorithm is known as either Chang-Roberts or Le Lann-Chang-Roberts (LCR). We'll go with the latter because it is always polite to be generous with credit.

Code for the LCR algorithm is given in Algorithm 5.1.

```
1  initially do
2      leader ← 0
3      maxId ← id_i
4      send id_i to clockwise neighbor
5  upon receiving j do
6      if j = id_i then
7          leader ← 1
8      if j > maxId then
9          maxId ← j
10         send j to clockwise neighbor
```

Algorithm 5.1: LCR leader election

Intuitively, this protocol works because whichever process p_{max} holds the maximum ID id_{max} will (a) refuse to forward any smaller ID, and (b)

[3] We'll see later in §5.2.3 that the distinction between unidirectional rings and bidirectional rings is not a big deal, but for now let's imagine that having a unidirectional ring is a serious hardship.

eventually have its value forwarded through all of the other processes, causing it to eventually set its leader bit to 1.

Looking closely at this intuition we see that (a) is a safety property and (b) a liveness property. So we obtain a proof of correctness by converting (a) into an invariant that for each $p_i \neq p_{max}$, id_i is never sent by any process in the range $p_{max} \ldots p_{i-1}$; and converting (b) into an induction argument that each process $p_{\max+j}$ sends id_{max} to $p_{\max+j+1}$ no later than time j. Because the code only has a process p_i set leader to 1 if it receives id_i from p_{i-1}, the invariant tells us that no $p_i \neq p_{max}$ becomes the leader, while the induction argument tells use that eventually p_{max} does.

5.2.1.1 Performance

It's immediate from the correctness proof that the protocol elects a leader within at most n time in the asynchronous model or exactly n rounds in a synchronous model.

To bound message traffic, observe that each process sends at most one copy of each of the n process IDs, for a total of $O(n^2)$ messages. This is a tight bound since if the IDs are in decreasing order $n, n-1, n-2, \ldots 1$, then no messages get eaten until they hit n.

There is a subtlety with the termination guarantee: at the moment the unique leader p_{max} sets its leader bit, the other processes all have $\mathsf{maxId} = \mathsf{id}_{max}$, but they don't actually *know* that they have the correct leader ID, since there is no information available locally at a non-leader process that allows it to detect that there can't be some larger ID out there that just hasn't reached it yet. As with all leader election algorithms, we can have the leader confirm its election with an additional broadcast protocol, which in this case raises the time complexity from n to $2n$ (still $O(n)$) and adds an extra n messages (still $O(n^2)$ in total).

5.2.2 The Hirschberg-Sinclair algorithm

This algorithm improves on Le Lann-Chang-Roberts by reducing the message complexity. The idea is that instead of having each process send a message all the way around a ring, each process will first probe locally to see if it has the largest ID within a short distance. If it wins among its immediate neighbors, it doubles the size of the neighborhood it checks, and continues as long as it has a winning ID. This means that most nodes drop out quickly, giving a total message complexity of $O(n \log n)$. The running time is a constant factor worse than LCR, but still $O(n)$. The algorithm assumes a bidirectional ring,

since the reverse edges are needed to send back responses to probes.

To specify the protocol, it may help to think of messages as mobile agents and the state of each process as being of the form (local-state, {agents I'm carrying}). Then the sending rule for a process becomes *ship any agents in whatever direction they want to go* and the transition rule is *accept any incoming agents and update their state in terms of their own internal transition rules.* An agent state for LCR will be something like (original-sender, direction, hop-count, max-seen) where direction is R or L depending on which way the agent is going, hop-count in phase k is initially 2^k when the agent is sent and drops by 1 each time the agent moves, and max-seen is the biggest ID of any node the agent has visited. An agent turns around (switches direction) when hop-count reaches 0.

To prove this works, we can mostly ignore the early phases (though we have to show that the max-id node doesn't drop out early, which is not too hard). The last phase involves any surviving node probing all the way around the ring, so it will declare itself leader only when it receives its own agent from the left. That exactly one node does so is immediate from the same argument for LCR.

Complexity analysis is mildly painful but basically comes down to the fact that any node that sends a message 2^k hops had to be a winner in phase $2k-1$, which means that it is the largest of some group of 2^{k-1} IDs. Thus the 2^k-hop senders are spaced at least 2^{k-1} away from each other and there are at most $n/2^{k-1}$ of them. Summing up over all $\lceil \lg n \rceil$ phases, we get $\sum_{k=0}^{\lceil \lg n \rceil} 2^k n/2^{k-1} = O(n \log n)$ messages and $\sum_{k=0}^{\lceil \lg n \rceil} 2^k = O(n)$ time.

5.2.3 Peterson's algorithm for the unidirectional ring

This algorithm is due to Peterson [Pet82] and assumes an asynchronous, unidirectional ring. It gets $O(n \log n)$ message complexity in all executions.

Let's start by describing a version with two-way communication. Start with n candidate leaders. In each of at most $\lg n$ asynchronous phases, each candidate probes its nearest surviving neighbors to the left and right; if its ID is larger than the IDs of both neighbors, it survives to the next phase. Non-candidates act as relays passing messages between candidates. As in Hirschberg and Sinclair (§5.2.2), the probing operations in each phase take $O(n)$ messages, and at least half of the candidates drop out in each phase. The last surviving candidate wins when it finds that it's its own surviving neighbor.

To make this work in a 1-way ring, we have to simulate 2-way communication by moving the candidates clockwise around the ring to catch up with

their unsendable counterclockwise messages. Peterson's algorithm does this with a two-hop approach that is inspired by the 2-way case above; in each phase k, a candidate effectively moves two positions to the right, allowing it to look at the IDs of three phase-k candidates before deciding to continue in phase $k+1$ or not. Here is a very high-level description; it assumes that we can buffer and ignore incoming messages from the later phases until we get to the right phase, and that we can execute sends immediately upon receiving messages. Doing this formally in terms of the model of §2.1 means that we have to build explicit internal buffers into our processes, which we can easily do but won't do here (see [Lyn96, pp. 483–484] for the right way to do this).

We can use a similar trick to transform any bidirectional-ring algorithm into a unidirectional-ring algorithm: alternate between phases where we send a message right, then send a virtual process right to pick up any left-going messages deposited for us. The problem with this trick is that it requires two messages per process per phase, which gives us a total message complexity of $O(n^2)$ if we start with an $O(n)$-time algorithm. Peterson's algorithm avoids this by propagating only the surviving candidates.

Pseudocode for Peterson's algorithm is given in Algorithm 5.2.

Note: The phase arguments in the probe messages are useless if one has FIFO channels, which is why [Lyn96] doesn't use them.

Proof of correctness is essentially the same as for the 2-way algorithm. For any pair of adjacent candidates, at most one of their current IDs survives to the next phase. So we get a sole survivor after $\lceil \lg n \rceil$ phases. Each process sends or relays at most 2 messages per phase, so we get at most $2n\lceil \lg n \rceil$ total messages.

Curiously, the time complexity of Peterson's algorithm may be worse than $O(n)$. It's not hard to construct an identity assignment in which all nodes in half the ring drop out, leaving $n/4$ candidates on the other side of the ring. Each subsequent phase may then require as much as $n/2$ time to transmit a message across the missing half. If it takes $\Theta(\log n)$ phases to reduce these $n/4$ candidates to one, this gives $\Theta(n \log n)$ total time.

5.2.4 A simple randomized $O(n \log n)$-message algorithm

An alternative to running a more sophisticated algorithm is to reduce the average cost of LCR using randomization. The presentation here follows the average-case analysis done by Chang and Roberts [CR79].

Run LCR where each ID is constructed by prepending a long random bit-string to the real ID. This gives uniqueness (since the real IDs act as

```
1  procedure candidate()
2      phase ← 0
3      current ← pid
4      while true do
5          send probe(phase, current)
6          wait for probe(phase, x)
7          id_2 ← x
8          send probe(phase + 1/2, id_2)
9          wait for probe(phase + 1/2, x)
10         id_3 ← x
11         if id_2 = current then
12             I am the leader!
13             return
14         else if id_2 > current and id_2 > id_3 do
15             current ← id_2
16             phase ← phase + 1
17         else
18             switch to relay()
19 procedure relay()
20     upon receiving probe(p, i) do
21         send probe(p, i)
```

Algorithm 5.2: Peterson's leader-election algorithm

tie-breakers) and something very close to a random permutation on the constructed IDs. When we have unique random IDs, a simple argument shows that the i-th largest ID only propagates an expected n/i hops, giving a total of $O(nH_n) = O(n \log n)$ hops.[4] Unique random IDs occur with high probability provided the range of the random sequence is $\gg n^2$.

The downside of this algorithm compared to Peterson's is that knowledge of n is required to pick random IDs from a large enough range. It also has higher bit complexity, since Peterson's algorithm is sending only IDs (in the FIFO-channel version) without any random padding. An possible upside is that if the range of random IDs is large enough, we can run it without any initial IDs at all, as long as we are willing to accept a small probability of accidentally electing two leaders.

5.3 Leader election in general networks

For general networks, a simple approach is to have each node initiate a breadth-first-search and convergecast, with nodes refusing to participate in the protocol for any initiator with a lower ID. It follows that only the node with the maximum ID can finish its protocol; this node becomes the leader. If messages from parallel broadcasts are combined, it's possible to keep the message complexity of this algorithm down to $O(DE)$.

More sophisticated algorithms reduce the message complexity by coalescing local neighborhoods similar to what happens in the Hirschberg-Sinclair and Peterson algorithms. A noteworthy example is an $O(n \log n)$ message-complexity algorithm of Afek and Gafni [AG91], who also show an $\Omega(n \log n)$ lower bound on message complexity for any synchronous algorithm in a complete network.

5.4 Lower bounds

Here we present two classic $\Omega(\log n)$ lower bounds on message complexity for leader election in the ring. The first, due to Burns [Bur80], assumes that the system is asynchronous and that the algorithm is **uniform**: it does not depend on the size of the ring. The second, due to Frederickson and Lynch [FL87], allows a synchronous system and relaxes the uniformity

[4] Alternatively, we could consider the **average-case complexity** of the algorithm when we assume all $n!$ orderings of the IDs are equally likely; this also gives $O(n \log n)$ expected message complexity [CR79].

assumption, but requires that the algorithm can't do anything to IDs but copy and compare them.

5.4.1 Lower bound on asynchronous message complexity

Here we describe a lower bound for uniform asynchronous leader election in the ring. The description here is based on [AW04, §3.3.3]; a slightly different presentation can also be found in [Lyn96, §15.1.4]. The original result is due to Burns [Bur80]. We assume the system is deterministic.

The proof constructs a bad execution in which n processes send lots of messages recursively, by first constructing two bad $(n/2)$-process executions and pasting them together in a way that generates many extra messages. If the pasting step produces $\Theta(n)$ additional messages, we get a recurrence $T(n) \geq 2T(n/2) + \Theta(n)$ for the total message traffic, which has solution $T(n) = \Omega(n \log n)$.

We'll assume that all processes are trying to learn the identity of the process with the smallest ID. This is a slightly stronger problem that mere leader election, but it can be solved with at most an additional $2n$ messages once we actually elect a leader. So if we get a lower bound of $f(n)$ messages on this problem, we immediately get a lower bound of $f(n) - 2n$ on leader election.

To construct the bad execution, we consider "open executions" on rings of size n where no message is delivered across some edge (these will be partial executions, because otherwise the guarantee of eventual delivery kicks in). Because no message is delivered across this edge, the processes can't tell if there is really a single edge there or some enormous unexplored fragment of a much larger ring. Our induction hypothesis will show that a line of $n/2$ processes can be made to send at least $T(n/2)$ messages in an open execution (before seeing any messages across the open edge); we'll then show that a linear number of additional messages can be generated by pasting two such executions together end-to-end, while still getting an open execution with n processes.

In the base case, we let $n = 1$. Somebody has to send a message eventually, giving $T(2) \geq 1$.

For larger n, suppose that we have two open executions on $n/2$ processes that each send at least $T(n/2)$ messages. Break the open edges in both executions and replace them with new edges to create a ring of of size n; similarly paste the schedules σ_1 and σ_2 of the two executions together to get a combined schedule $\sigma_1\sigma_2$ with at least $2T(n/2)$ messages. Note that in the combined schedule no messages are passed between the two sides, so the

processes continue to behave as they did in their separate executions.

Let e and e' be the edges we used to past together the two rings. Extend $\sigma_1\sigma_2$ by the longest possible suffix σ_3 in which no messages are delivered across e and e'. Since σ_3 is as long as possible, after $\sigma_1\sigma_2\sigma_3$, there are no messages waiting to be delivered across any edge except e and e' and all processes are **quiescent**—they will send no additional messages until they receive one.

We now consider some suffix σ_4 that causes the protocol to finish when appended to $\sigma_1\sigma_2\sigma_3$. While executing σ_4, construct two sets of processes S and S' by the following rules:

1. If a process is not yet in S or S' and receives a message delivered across e, put it in S; similarly if it receives a message delivered across e', put it in S'.

2. If a process is not yet in S or S' and receives a message that was sent by a process in S, put it in S; similarly for S'.

Observe that this process must eventually make S and S' adjacent, because if there is some node in the half to the ring with the larger minimum id that receives no messages in σ_4 (and thus is never added to S or S'?), that node doesn't learn the global minimum.

So now imagine stopping the process after the shortest prefix σ_4' of σ_4 that makes S and S' adjacent. This gives $|S \cup S'| \geq n/2$, because we include all nodes between e and e' on one side or the other. It follows that at least one of S and S' contains at least $n/4$ nodes after σ_4'.

Assume without loss of generality that it is $|S|$ that is at least $n/4$. Except for the two processes incident to e, every process that is added to S is added in response to a message sent in σ_4'. So there are at least $n/4 - 2$ such messages. We can also argue that all of these messages are sent in the subschedule τ of σ_4' that contains only messages that do not depend on messages delivered across e'. It follows that $\sigma_1\sigma_2\sigma_3\tau$ is an open execution on n processes with at least $2T(n/2) + n/4 - 2$ sent messages. This gives $T(n) \geq 2T(n/2) + n/4 - 2 = 2T(n/2) + \Omega(n)$ as claimed.

5.4.2 Lower bound for comparison-based protocols

Here we give an $\Omega(n \log n)$ lower bound on messages for synchronous-start comparison-based protocols in bidirectional synchronous rings. For full details see [Lyn96, §3.6], [AW04, §3.4.2], or the original JACM paper by Frederickson and Lynch [FL87].

The argument proceeds as follows:

- Two fragments $i \ldots i+k$ and $j \ldots j+k$ of a ring are **order-equivalent** provided $\mathsf{id}_{i+a} > \mathsf{id}_{i+b}$ if and only if $\mathsf{id}_{j+a} > \mathsf{id}_{j+b}$ for $b = 0 \ldots k$.

- A protocol is **comparison-based** if it can't do anything to IDs but copy them and test for $<$. The state of such an protocol is modeled by some non-ID state together with a big bag of IDs, messages have a pile of IDs attached to them, etc. Two states/messages are equivalent under some mapping of IDs if you can translate the first to the second by running all IDs through the mapping.

  An equivalent version uses an explicit equivalence relation between processes. Let executions of p_1 and p_2 be **similar** if both processes send messages in the same direction(s) in the same rounds and both processes declare themselves leader (or not) at the same round. Then an protocol is comparison-based based if order-equivalent rings yield similar executions for corresponding processes. This can be turned into the explicit-copying-ids model by replacing the original protocol with a **full-information protocol** in which each message is replaced by the ID and a complete history of the sending process (including all messages it has every received).

- Define an **active round** as a round in which at least one message is sent. Claim: Actions of i after k active rounds depends, up to an order-equivalent mapping of IDs, only on the order-equivalence class of IDs in $i-k \ldots i+k$, the k**-neighborhood** of i. Proof: by induction on k. Suppose i and j have order-equivalent $(k-1)$-neighborhoods; then after $k-1$ active rounds they have equivalent states by the induction hypothesis. In inactive rounds, i and j both receive no messages and update their states in the same way. In active rounds, i and j receive order-equivalent messages and update their states in an order-equivalent way.

- If we have an order of IDs with a lot of order-equivalent k-neighborhoods, then after k active rounds if one process sends a message, so do a lot of other ones.

Now we just need to build a ring with a lot of order-equivalent neighborhoods. For n a power of 2 we can use the bit-reversal ring, e.g., ID sequence $000, 100, 010, 110, 001, 101, 011, 111$ (in binary) when $n = 8$. Figure 5.1 gives a picture of what this looks like for $n = 32$.

For n not a power of 2 we look up Frederickson and Lynch [FL87] or Attiya *et al.* [ASW88]. In either case we get $\Omega(n/k)$ order-equivalent members

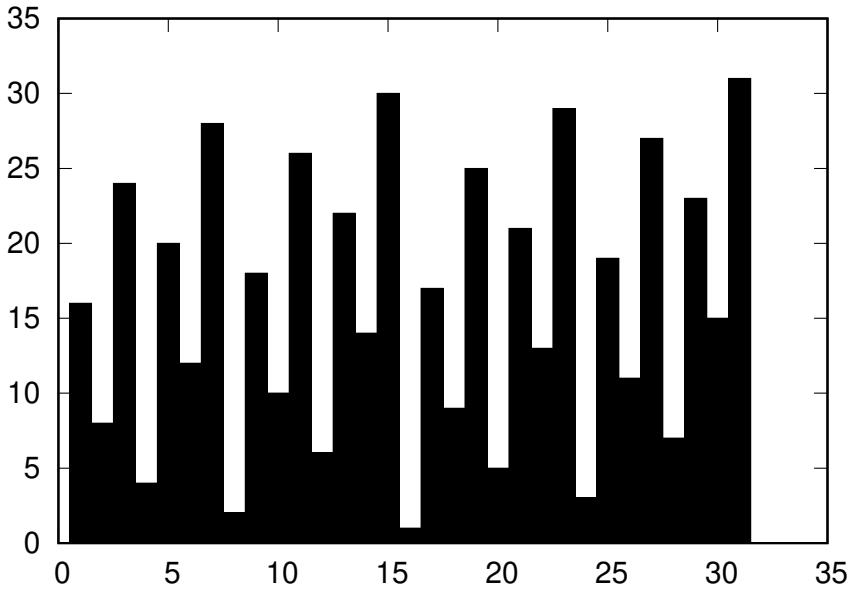


Figure 5.1: Labels in the bit-reversal ring with $n = 32$

of each equivalence class after k active rounds, giving $\Omega(n/k)$ messages per active round, which sums to $\Omega(n \log n)$.

For non-comparison-based protocols we can still prove $\Omega(n \log n)$ messages for time-bounded protocols, but it requires techniques from **Ramsey theory**, the branch of combinatorics that studies when large enough structures inevitably contain substructures with certain properties.[5] Here "time-bounded" means that the running time can't depend on the size of the ID space. See [AW04, §3.4.2] or [Lyn96, §3.7] for the textbook version, or [FL87, §7] for the original result.

The intuition is that for any fixed protocol, if the ID space is large enough, then there exists a subset of the ID space where the protocol acts like a comparison-based protocol. So the existence of an $O(f(n))$-message time-bounded protocol implies the existence of an $O(f(n))$-message comparison-based protocol, and from the previous lower bound we know $f(n)$ is $\Omega(n \log n)$. Note that time-boundedness is necessary: we can't prove

[5] The classic example is **Ramsey's Theorem**, which says that if you color the edges of a complete graph red or blue, while trying to avoid having any subsets of k vertices with all edges between them the same color, you will no longer be able to once the graph is large enough (for any fixed k). See [GRS90] for much more on the subject of Ramsey theory.

the lower bound for non-time-bounded algorithms because of the $i \cdot n$ trick.

Chapter 6

Causal ordering and logical clocks

Logical clocks assign a timestamp to all events in an asynchronous message-passing system that simulates real time, thereby allowing timing-based algorithms to run despite asynchrony. In general, they don't have anything to do with clock synchronization or wall-clock time; instead, they provide numerical values that increase over time and are consistent with the observable behavior of the system. This means that local events on a single process have increasing times, and messages are never delivered before they are sent, when time is measured using the logical clock.

Because the processes in a system don't necessarily know the relative order of distant events, a totally-ordered logical clock may impose an ordering on events that is not observable by the processes. We can capture the observable (partial) ordering using a **causal ordering**, defined in §6.1. A totally-ordered logical clock is correct if it gives an ordering that is a refinement of the causal ordering; some examples are given in §6.2. Alternatively, by using partially-ordered set for the values of our logical clock, it may be possible to capture the causal ordering precisely (§6.2.3).

One application of logical clocks is to implement a **snapshot**, as described in §6.3. The simplest version of this is to have each process record its state at some particular logical clock time. This is not quite an description of the global configuration of the system at some real-time instant in the execution, because asynchronous processes can't guarantee that they all take a snapshot at the same real time. Instead, it's a description of a global configuration that is consistent with the observations of the processes, in the sense that there exists an execution indistinguishable from the real one that contains

this configuration. Causal ordering is the tool that lets us argue that this hypothetical execution exists.

6.1 Causal ordering

Here we define the **causal ordering**, a partial order on events that describes when one event e can shown to occur before some other event e' based only on the sequences of events observed by each process.

For the purpose of defining the causal ordering and logical clocks, we will assume that a schedule consists of **send events** and **receive events**, which correspond to some process sending a single message or receiving a single message, respectively. This is not quite the same as our usual model that allows many messages to be received and sent as part of the same delivery event, but for asynchronous systems we can treat the definitions as equivalent by splitting a multi-message delivery event into a sequence of events, one for each message.

Given two schedules S and S', call S and S' **similar** if $S|p = S'|p$ for all processes p; in other words, S and S' are similar if they are indistinguishable by all participants. We can define a causal ordering on the events of some schedule S implicitly by considering all schedules S' similar to S, and declare that $e < e'$ if e precedes e' in all such S. But it is usually more useful to make this ordering explicit.

Following [AW04, §6.1.1] (and ultimately [Lam78]), define the **happens-before** relation $\underset{S}{\Rightarrow}$ on a schedule S to consist of:

1. All pairs (e, e') where e precedes e' in S and e and e' are events of the same process.

2. All pairs (e, e') where e is a send event and e' is the receive event for the same message.

3. All pairs (e, e') where there exists a third event e'' such that $e \underset{S}{\Rightarrow} e''$ and $e'' \underset{S}{\Rightarrow} e'$. (In other words, we take the **transitive closure** of the relation defined by the previous two cases.)

It is not terribly hard to show that this gives a partial order; the main observation is that if $e \underset{S}{\Rightarrow} e'$, then e precedes e' in S. So $\underset{S}{\Rightarrow}$ is a subset of the total order $<_S$ given by the order of events in S.

A **causal shuffle** S' of a schedule S is a permutation of S that is consistent with the happens-before relation on S; that is, if e happens-before

e' in S, then e precedes e' in S'. The importance of the happens-before relation follows from the following lemma, which says that the causal shuffles of S are precisely the schedules S' that are similar to S.

Lemma 6.1.1. *Let S' be a permutation of the events in S. Then the following two statements are equivalent:*

1. *S' is a causal shuffle of S.*
2. *S' is the schedule of an execution fragment of a message-passing system with $S|p = S'|p$ for all S'.*

Proof. ($1 \Rightarrow 2$). We need to show both similarity and that S' corresponds to some execution fragment. We'll show similarity first. Pick some p; then every event at p in S also occurs in S', and they must occur in the same order by the first case of the definition of the happens-before relation. This gets us halfway to showing S' is the schedule of some execution fragment, since it says that any events initiated by p are consistent with p's programming. To get the rest of the way, observe that any other events are receive events. For each receive event e' in S, there must be some matching send event e also in S; thus e and e' are both in S' and occur in the right order by the second case of the definition of happens-before.

($2 \Rightarrow 1$). First observe that since every event e in S' occurs at some process p, if $S'|p = S|p$ for all p, then there is a one-to-one correspondence between events in S' and S, and thus S' is a permutation of S. Now we need to show that S' is consistent with $\underset{S}{\Rightarrow}$. Let $e \underset{S}{\Rightarrow} e'$. There are three cases.

1. e and e' are events of the same process p and $e <_S e'$. But then $e <_{S'} e'$ because $S|p = S'|p$.
2. e is a send event and e' is the corresponding receive event. Then $e <_{S'} e'$ because S' is the schedule of an execution fragment.
3. $e \underset{S}{\Rightarrow} e'$ by transitivity. Then each step in the chain connecting e to e' uses one of the previous cases, and $e <_{S'} e'$ by transitivity of $<_{S'}$.

□

There are two main applications for causal shuffles:

1. We can prove upper bounds by using a causal shuffle to turn some arbitrary S into a nice S', and argue that the niceness of S' means that even if S might not be nice, it looks nice to the processes. An example of this can be found in Lemma 7.1.1.

2. We can prove lower bounds by using a causal shuffle to turn some specific S into a nasty S', and argue that the existence of S' tells us that there exist nasty schedules for some particular problem. An example of this can be found in §7.4.2. This works particularly well because $\underset{S}{\Rightarrow}$ includes enough information to determine the latest possible time of any event in either S or S', so rearranging schedules like this doesn't change the worst-case time.

In both cases, we are using the fact that if I tell you $\underset{S}{\Rightarrow}$, then you know everything there is to know about the order of events in S that you can deduce from reports from each process together with the fact that messages don't travel back in time.

In the case that we want to use this information *inside* an algorithm, we run into the issue that $\underset{S}{\Rightarrow}$ is a pretty big relation ($\Theta(|S|^2)$ bits with a naive encoding), and seems to require global knowledge of $<_S$ to compute. So we can ask if there is some simpler, easily computable description that works almost as well. This is where logical clocks come in.

6.2 Logical clocks

The idea of a logical clock is to compute a **timestamp** for each event, so that comparing timestamps gives information about $\underset{S}{\Rightarrow}$. Note that these timestamps need not be totally ordered. In general, we will have a relation $<_L$ between timestamps such that $e \underset{S}{\Rightarrow} e'$ implies $e <_L e'$, but it may be that there are some pairs of events that are ordered by the logical clock despite being incomparable in the happens-before relation.

Examples of logical clocks that use small timestamps but add extra ordering are Lamport clocks [Lam78], discussed in §6.2.1; and Neiger-Toueg-Welch clocks [NT87, Wel87], discussed in §6.2.2. These both assign integer timestamps to events and may order events that are not causally related. The main difference between them is that Lamport clocks do not alter the underlying execution, but may allow arbitrarily large jumps in the logical clock values; while Neiger-Toueg-Welch clocks guarantee small increments at the cost of possibly delaying parts of the system.[1]

More informative are **vector clocks** [Fid91, Mat93], discussed in §6.2.3. These use n-dimensional vectors of integers to capture $\underset{S}{\Rightarrow}$ exactly, at the cost of much higher overhead.

[1] This makes them similar to **synchronizers**, which we will discuss in Chapter 7.

6.2.1 Lamport clock

Lamport's **logical clock** [Lam78] runs on top of any other message-passing protocol, adding additional state at each process and additional content to the messages (which is invisible to the underlying protocol). Every process maintains a local variable clock. When a process sends a message or executes an internal step, it sets $\mathsf{clock} \leftarrow \mathsf{clock} + 1$ and assigns the resulting value as the clock value of the event. If it sends a message, it piggybacks the resulting clock value on the message. When a process receives a message with timestamp t, it sets $\mathsf{clock} \leftarrow \max(\mathsf{clock}, t) + 1$; the resulting clock value is taken as the time of receipt of the message. (To make life easier, we assume messages are received one at a time.)

Theorem 6.2.1. *If we order all events by clock value, we get an execution of the underlying protocol that is locally indistinguishable from the original execution.*

Proof. Let $e <_L e'$ if e has a lower clock value than e'. If e and e' are two events of the same process, then $e <_L e'$. If e and e' are send and receive events of the same message, then again $e <_L e'$. So for *any* events e, e', if $e \underset{S}{\Rightarrow} e'$, then $e <_L e'$. Now apply Lemma 6.1.1. □

6.2.2 Neiger-Toueg-Welch clock

Lamport's clock has the advantage of requiring no changes in the behavior of the underlying protocol, but has the disadvantage that clocks are entirely under the control of the logical-clock protocol and may as a result make huge jumps when a message is received. If this is unacceptable—perhaps the protocol needs to do some unskippable maintenance task every 1000 clock ticks—then an alternative approach due to Neiger and Toueg [NT87] and Welch [Wel87] can be used.

Method: Each process maintains its own variable clock, which it increments whenever it feels like it. To break ties, the process extends the clock value to $\langle \mathsf{clock}, \mathsf{id}, \mathsf{eventCount} \rangle$ where eventCount is a count of send and receive events (and possibly local computation steps). As in Lamport's clock, each message in the underlying protocol is timestamped with the current extended clock value. Because the protocol can't change the clock values on its own, when a message is received with a timestamp later than the current extended clock value, its delivery is delayed until clock exceeds the message timestamp, at which point the receive event is assigned the extended clock value of the time of delivery.

Theorem 6.2.2. *If we order all events by clock value, we get an execution of the underlying protocol that is locally indistinguishable from the original execution.*

Proof. Again, we have that (a) all events at the same process occur in increasing order (since the event count rises even if the clock value doesn't, and we assume that the clock value doesn't drop) and (b) all receive events occur later than the corresponding send event (since we force them to). So Lemma 6.1.1 applies. □

The advantage of the Neiger-Toueg-Welch clock is that it doesn't impose any assumptions on the clock values, so it is possible to make clock be a real-time clock at each process and nonetheless have a causally-consistent ordering of timestamps even if the local clocks are not perfectly synchronized. If some process's clock is too far off, it will have trouble getting its messages delivered quickly (if its clock is ahead) or receiving messages (if its clock is behind)—the net effect is to add a round-trip delay to that process equal to the difference between its clock and the clock of its correspondent. But the protocol works well when the processes' clocks are closely synchronized, which is a reasonable assumption in many systems thanks to the Network Time Protocol, cheap GPS receivers, and clock synchronization mechanisms built into most cellular phone networks.[2]

6.2.3 Vector clocks

Logical clocks give a *superset* of the happens-before relation: if $e \underset{S}{\Rightarrow} e'$, then $e <_L e'$ (or conversely, if $e \not<_L e'$, then it is not the case that $e \underset{S}{\Rightarrow} e'$). This is good enough for most applications, but what if we want to compute $\underset{S}{\Rightarrow}$ exactly?

Here we can use a **vector clock**, invented independently by Fidge [Fid91] and Mattern [Mat93]. Instead of a single clock value, each event is stamped with a vector of values, one for each process.

A process p starts with a vector $t^p = 0$ (all components 0). When a process executes a local event or a send event, it increments only its own component t^p_p of the vector, and includes the updated vector clock value with its message. When it receives a message, it increments t^p_p and sets t^p_q for each

[2] As I write this, my computer reports that its clock is an estimated 289 microseconds off from the timeserver it is synchronized to, which is less than a tenth of the round-trip delay to machines on the same local-area network and a tiny fraction of the round-trip delay to machines elsewhere, including the timeserver machine.

q to the max max of its previous value and the value of t_q piggybacked on the message. We define $\mathsf{VC}(e)$ were e is an event p to be the value of t^p at the end of event e. We define $\mathsf{VC}(e) \leq \mathsf{VC}(e')$, where $\mathsf{VC}(e)$ is the value of the vector clock for e, if $\mathsf{VC}(e)_i \leq \mathsf{VC}(e')_i$ for all i.

Theorem 6.2.3. *Fix a schedule S; then for any e, e', $VC(e) < VC(e')$ if and only if $e \underset{S}{\Rightarrow} e'$.*

Proof. We'll start by showing that for any event e at a process p, the value of $\mathsf{VC}(e)_q$ for any $q \neq p$ is equal to the max $\mathsf{VC}(e')_q$ for any event e' of q such that $e' \underset{S}{\Rightarrow} e$, or 0 if there is no such e'.

The proof is by induction on the schedule so far.

If e is a local event or a send event, then there is either no preceding event at the same process (and thus no event e' of q with $e' \underset{S}{\Rightarrow} e$) and $\mathsf{VC}(e)_q = 0$ as required; or there is some preceding event e'' of p. Since e'' is the only immediate predecessor of e' in $\underset{S}{\Rightarrow}$, if there is an event e' of q maximizing $\mathsf{VC}_(e')_q$ such that $e' \underset{S}{\Rightarrow} e$, $e' \underset{S}{\Rightarrow} e''$ and so $\mathsf{VC}(e)_q = \mathsf{VC}(e'')_q = \mathsf{VC}(e')_q$ as required.

Alternatively, if e is a receive event, then there is at most one immediately preceding event e_1 of the same process and a send event e_2 of the same message such that $\mathsf{VC}(e)_q = \max(\mathsf{VC}(e_1)_q, \mathsf{VC}(e_2), q)$. Since any event e' of q with $e' \underset{S}{\Rightarrow} e$ has either $e' \underset{S}{\Rightarrow} e_1$ or $e \underset{S}{\Rightarrow} e_2$, we can apply the induction hypothesis to both e_1 and e_2 and then observe that $\mathsf{VC}(e)_q = \max(\mathsf{VC}(e_1)_q, \mathsf{VC}(e_2)_q)$ satisfies the requirements of the induction hypothesis.

Given this characterization of $\mathsf{VC}(e)_q$, the if part follows immediately from the update rules for the vector clock. For events $e \underset{S}{\Rightarrow} e'$ of the same process, observe that both update rules strictly increase that process's clock, so $\mathsf{VC}(e) < \mathsf{VC}(e')$. Similarly the update rule for receiving a message implies that $\mathsf{VC}(e) < \mathsf{VC}(e')$ when e and e' are matching send and receive events, with the minor issue that we do need to use the observation above to verify that $e_p < e'_p$ for the receiver p.

For the only if part, suppose e does not happen-before e'. Then e and e' are events of distinct processes p and p'. For $\mathsf{VC}(e) < \mathsf{VC}(e')$ to hold, we must have $\mathsf{VC}(e)_p \leq \mathsf{VC}(e')_p$; but as shown above, this can occur only if $e \underset{S}{\Rightarrow} e'$. □

6.3 Consistent snapshots

A **consistent snapshot** of a message-passing computation is a description of the states of the processes (and possibly messages in transit, but we can reduce this down to just states by keeping logs of messages sent and received) that gives the global configuration at some instant of a schedule that is a consistent reordering of the real schedule (a **consistent cut** in the terminology of [AW04, §6.1.2]. Without shutting down the protocol before taking a snapshot this is the about the best we can hope for in a message-passing system.

Logical clocks can be used to obtain consistent snapshots: pick some logical clock time and have each process record its state at this time (i.e., immediately after its last step before the time or immediately before its first step after the time). We have already argued that the logical clock gives a consistent reordering of the original schedule, so the set of values recorded is just the configuration at the end of an appropriate prefix of this reordering. In other words, it's a consistent snapshot.

If we aren't building logical clocks anyway, there is a simpler consistent snapshot algorithm due to Chandy and Lamport [CL85]. Here some central initiator broadcasts a snap message, and each process records its state and immediately forwards the snap message to all neighbors when it first receives a snap message. To show that the resulting configuration is a configuration of some consistent reordering, observe that (with FIFO channels) no process receives a message before receiving snap that was sent after the sender sent snap: thus causality is not violated by lining up all the pre-snap operations before all the post-snap ones.[3]

The full Chandy-Lamport algorithm adds a second marker message that is used to sweep messages in transit out of the communications channels, which avoids the need to keep logs if we want to reconstruct what messages are in transit (this can also be done with the logical clock version). The idea is that when a process records its state after receiving the snap message, it issues a marker message on each outgoing channel. For incoming channels, each process records all messages received between the snapshot and receiving a marker message on that channel (or nothing if it receives marker before receiving snap). A process only reports its value when it has received a marker on each channel. The marker and snap messages can also be combined if the broadcast algorithm for snap resends it on all channels anyway, and a

[3] If FIFO channels are not available, they can be simulated in the absence of failures by adding a sequence number to each outgoing message on a given channel, and processing messages at the recipient only when all previous messages have been processed.

further optimization is often to piggyback both on messages of the underlying protocol if the underlying protocol is chatty enough.

Note that Chandy-Lamport is equivalent to the logical-time snapshot using Lamport clocks, if the snap message is treated as a message with a very large timestamp. For Neiger-Toueg-Welch clocks, we get an algorithm where processes spontaneously decide to take snapshots (since Neiger-Toueg-Welch clocks aren't under the control of the snapshot algorithm) and delay post-snapshot messages until the local snapshot has been taken. This can be implemented as in Chandy-Lamport by separating pre-snapshot messages from post-snapshot messages with a marker message, and essentially turns into Chandy-Lamport if we insist that a process advance its clock to the snapshot time when it receives a marker.

6.3.1 Property testing

Consistent snapshots are in principle useful for debugging (since one can gather a consistent state of the system without being able to talk to every process simultaneously), and in practice are mostly used for detecting **stable properties** of the system. Here a stable property is some predicate on global configurations that remains true in any successor to a configuration in which it is true, or (bending the notion of properties a bit) functions on configurations whose values don't change as the protocol runs. Typical examples are quiescence and its evil twin, deadlock. More exotic examples include total money supply in a banking system that cannot create or destroy money, or the fact that every process has cast an irrevocable vote in favor of some proposal or advanced its Neiger-Toueg-Welch-style clock past some threshold.

The reason we can test such properties using consistent snapshot is that when the snapshot terminates with value C in some configuration C', even though C may never have occurred during the actual execution of the protocol, there *is* an execution which leads from C to C'. So if P holds in C, stability means that it holds in C'.

Naturally, if P doesn't hold in C, we can't say much. So in this case we re-run the snapshot protocol and hope we win next time. If P eventually holds, we will eventually start the snapshot protocol after it holds and obtain a configuration (which again may not correspond to any global configuration that actually occurs) in which P holds.

Chapter 7

Synchronizers

Synchronizers simulate an execution of a failure-free synchronous system in a failure-free asynchronous system. See [AW04, Chapter 11] or [Lyn96, Chapter 16] for a detailed (and rigorous) presentation.

7.1 Definitions

Formally, a synchronizer sits between the underlying network and the processes and does one of two things:

- A **global synchronizer** guarantees that no process receives a message from round r until *all processes* have sent their messages for round r.
- A **local synchronizer** guarantees that no process receives a message from round r until *all of that process's neighbors* have sent their messages for round r.

In both cases, the synchronizer packages all the incoming round r messages m for a single process together and delivers them as a single action $\texttt{recv}(p, m, r)$. Similarly, a process is required to hand over all of its outgoing round-r messages to the synchronizer as a single action $\texttt{send}(p, m, r)$—this prevents a process from changing its mind and sending an extra round-r message or two. It is easy to see that the global synchronizer produces executions that are effectively indistinguishable from synchronous executions, assuming that a synchronous execution is allowed to have some variability in exactly when within a given round each process does its thing. The local synchronizer only guarantees an execution that is locally indistinguishable from an execution of the global synchronizer: an individual process can't

tell the difference, but comparing actions at different (especially widely separated) processes may reveal some process finishing round $r + 1$ while others are still stuck in round r or earlier. Whether this is good enough depends on what you want: it's bad for coordinating simultaneous missile launches, but may be just fine for adapting a synchronous message-passing algorithm (as with distributed breadth-first search as described in §4.3) to an asynchronous system, if we only care about the final states of the processes and not when precisely those states are reached.

Formally, the relation between global and local synchronization is described by the following lemma:

Lemma 7.1.1. *For any schedule S of a locally synchronous execution, there is a schedule S' of a globally synchronous execution such that $S|p = S'|p$ for all processes p.*

Proof. Essentially, we use the same **happens-before** relation as in Chapter 6, and the fact that if a schedule S' is a causal shuffle of another schedule S (i.e., a permutation of T that preserves causality), then $S'|p = S|p$ for all p (Lemma 6.1.1).

Given a schedule S, consider a schedule S' in which the events are ordered first by increasing round and then by putting all sends before receives. This ordering is consistent with $\underset{S}{\Rightarrow}$, so it's a causal shuffle of S and $S'|p = S|p$. But it is globally synchronized, because no round r operation ever happens before a round $(r - 1)$ operation. □

7.2 Implementations

Here we describe several implementations of synchronizers. All of them give at least local synchrony. One of them, the beta synchronizer (§7.2.2), also gives global synchrony.

The names were chosen by their inventor, Baruch Awerbuch [Awe85]. The main difference between them is the mechanism used to determine when round-r messages have been delivered.

In the **alpha synchronizer**, every node sends a message to every neighbor in every round (possibly a dummy message if the underlying protocol doesn't send a message); this allows the receiver to detect when it's gotten all its round-r messages (because it expects to get a message from every neighbor) but may produce huge blow-ups in message complexity in a dense graph.

In the **beta synchronizer**, messages are acknowledged by their receivers (doubling the message complexity), so the senders can detect when all of their messages are delivered. But now we need a centralized mechanism to collect this information from the senders and distribute it to the receivers, since any particular receiver doesn't know which potential senders to wait for. This blows up time complexity, as we essentially end up building a global synchronizer with a central leader.

The **gamma synchronizer** combines the two approaches at different levels to obtain a trade-off between messages and time that depends on the structure of the graph and how the protocol is organized.

Details of each synchronizer are given below.

7.2.1 The alpha synchronizer

The alpha synchronizer uses local information to construct a local synchronizer. In round r, the synchronizer at p sends p's message (tagged with the round number) to each neighbor p' or $\mathsf{noMsg}(r)$ if it has no messages. When it collects a message or noMsg from each neighbor for round r, it delivers all the messages. It's easy to see that this satisfies the local synchronization specification.

This produces no change in time but may drastically increase message complexity because of all the extra noMsg messages flying around. For a synchronous protocol that runs in T rounds with M messages, the same protocol running with the alpha synchronizer will still run in T time units, but the message complexity will go up to $T \cdot |E|$ messages, or worse if the original algorithm doesn't detect termination.

7.2.2 The beta synchronizer

The beta synchronizer centralizes detection of message delivery using a rooted directed spanning tree (previously constructed). When p' receives a round-r message from p, it responds with $\mathsf{ack}(r)$. When p collects an ack for all the messages it sent plus an OK from all of its children, it sends OK to its parent. When the root has all the ack and OK messages it is expecting, it broadcasts go. Receiving go makes p deliver the queued round-r messages.

This works because in order for the root to issue go, every round-r message has to have gotten an acknowledgment, which means that all round-r messages are waiting in the receivers' buffers to be delivered. For the beta synchronizer, message complexity for one round increases slightly from M to

$2M + 2(n-1)$, but time complexity goes up by a factor proportional to the depth of the tree.

7.2.3 The gamma synchronizer

The gamma synchronizer combines the alpha and beta synchronizers to try to get low blowups on both time complexity and message complexity. The essential idea is to cover the graph with a spanning forest and run beta within each tree and alpha between trees. Specifically:

- Every message in the underlying protocol gets acked (including messages that pass between trees).
- When a process has collected all of its outstanding round-r acks, it sends OK up its tree.
- When the root of a tree gets all acks and OK, it sends ready to the roots of all adjacent trees (and itself). Two trees are adjacent if any of their members are adjacent.
- When the root collects ready from itself and all adjacent roots, it broadcasts go through its own tree.

As in the alpha synchronizer, we can show that no root issues go unless it and all its neighbors issue ready, which happens only after both all nodes in the root's tree and all their neighbors (some of whom might be in adjacent trees) have received acks for all messages. This means that when a node receives go it can safely deliver its bucket of messages.

Message complexity is comparable to the beta synchronizer assuming there aren't too many adjacent trees: $2M$ messages for sends and acks, plus $O(n)$ messages for in-tree communication, plus $O(E_{\text{roots}})$ messages for root-to-root communication. Time complexity per synchronous round is proportional to the depth of the trees: this includes both the time for in-tree communication, and the time for root-to-root communication, which might need to be routed through leaves.

In a particularly nice graph, the gamma synchronizer can give costs comparable to the costs of the original synchronous algorithm. An example in [Lyn96] is a ring of k-cliques, where we build a tree in each clique and get $O(1)$ time blowup and $O(n)$ added messages. This is compared to $O(n/k)$ time blowup for the beta synchronizer and $O(k)$ message blowup (or worse) for the alpha synchronizer. Other graphs may favor tuning the size of the

trees in the forest toward the alpha or beta ends of the spectrum, e.g., if the whole graph is a clique (and we didn't worry about contention issues), we might as well just use beta and get $O(1)$ time blowup and $O(n)$ added messages.

7.3 Applications

See [AW04, §11.3.2] or [Lyn96, §16.5]. The one we have seen is distributed breadth-first search, where the two asynchronous algorithms we described in Chapter 4 were essentially the synchronous algorithms with the beta and alpha synchronizers embedded in them. But what synchronizers give us in general is the ability to forget about problems resulting from asynchrony provided we can assume no failures (which may be a very strong assumption) and are willing to accept a bit of overhead.

7.4 Limitations of synchronizers

Here we show some lower bounds on synchronizers, justifying our previous claim that failures are trouble and showing that global synchronizers are necessarily slow in a high-diameter network.

7.4.1 Impossibility with crash failures

These synchronizers all fail badly if some process crashes. In the α synchronizer, the system slowly shuts down as a wave of waiting propagates out from the dead process. In the β synchronizer, the root never gives the green light for the next round. The γ synchronizer, true to its hybrid nature, fails in a way that is a hybrid of these two disasters.

This is unavoidable in the basic asynchronous model, although we don't have all the results we need to prove this yet. The idea is that if we are in a synchronous system with crash failures, it's possible to solve **agreement**, the problem of getting all the processes to agree on a bit (see Chapter 9). But it's not possible to solve this problem in an asynchronous system with even one crash failure (see Chapter 11). Since a synchronous-with-crash-failure agreement protocol on top of a fault-tolerant synchronizer would give a solution to an unsolvable problem, the element of this stack that we don't know an algorithm for must be the one we can't do. Hence there are no fault-tolerant synchronizers.

We'll see more examples of this trick of showing that a particular simulation is impossible because it would allow us to violate impossibility results later, especially when we start looking at the strength of shared-memory objects in Chapter 19.

7.4.2 Unavoidable slowdown with global synchronization

The **session problem** [AFL83] gives a lower bound on the speed of a global synchronizer, or more generally on any protocol that tries to approximate synchrony in a certain sense. Recall that in a global synchronizer, our goal is to produce a simulation that looks synchronous *from the outside*; that is, that looks synchronous to an observer that can see the entire schedule. In contrast, a local synchronizer produces a simulation that looks synchronous *from the inside*—the resulting execution is indistinguishable from a synchronous execution to any of the processes, but an outside observer can see that different processes execute different rounds at different times. The global synchronizer we've seen takes more time than a local synchronizer; the session problem shows that this is necessary.

In our description, we will mostly follow [AW04, §6.2.2].

A solution to the session problem is an asynchronous protocol in which each process repeatedly executes some **special action**. Our goal is to guarantee that these special actions group into s **sessions**, where a session is an interval of time in which every process executes at least one special action. We also want the protocol to terminate: this means that in every execution, every process executes a finite number of special actions.

A synchronous system can solve this problem trivially in s rounds: each process executes one special action per round. For an asynchronous system, a lower bound of Attiya and Mavronicolas [AM94] (based on an earlier bound of Arjomandi, Fischer, and Lynch [AFL83], who defined the problem in a slightly different communication model), shows that if the diameter of the network is D, any solution to the s-session problem takes $(s-1)D$ time or more in the worst case. The argument is based on reordering events in any synchronous execution that takes less time to produce fewer than s sessions, using the happens-before relation described in Chapter 6.

We now give an outline of the proof that this is expensive. (See [AW04, §6.2.2] for the real proof.)

Fix some algorithm A for solving the s-session problem, and suppose that its worst-case time complexity is $(s-1)D$ or less. Consider some synchronous execution of A (that is, one where the adversary scheduler happens to arrange the schedule to be synchronous) that takes $(s-1)D$ rounds or less. Divide

this execution into two segments: an initial segment γ that includes all rounds with special actions, and a suffix δ that includes any extra rounds where the algorithm is still floundering around. We will mostly ignore δ, but we have to leave it in to allow for the possibility that whatever is happening there is important for the algorithm to work (say, to detect termination).

We now want to perform a causal shuffle on γ that leaves it with only $s-1$ sessions. Because causal shuffles don't affect time complexity, this will give us a new bad execution $\gamma'\delta$ that has only $s-1$ sessions despite taking $(s-1)D$ time.

The first step is to chop γ into $s-1$ segments $\gamma_1, \gamma_2, \ldots \gamma_{s-1}$ of at most D rounds each. Because a message sent in round i is not delivered until round $i+1$, if we have a chain of k messages, each of which triggers the next, then if the first message is sent in round i, the last message is not delivered until round $i+k$. If the chain has length D, its events (including the initial send and the final delivery) span $D+1$ rounds $i, i+1, \ldots, i+D$. In this case the initial send and final delivery are necessarily in different segments γ_i and γ_{i+1}.

Now pick processes p and q at distance D from each other. Then any chain of messages starting at p within some segment reaches q after the end of the segment. It follows that for any events e_p of p and e_q of q in the *same* segment γ_i, $e_p \not\underset{\gamma\delta}{\Rightarrow} e_q$. So there exists a causal shuffle of γ_i that puts all events of p after all events of q.[1] By a symmetrical argument, we can similarly put all events of q in a segment after all events of p in the same segment. In both cases the resulting schedule is indistinguishable by all processes from the original.

So now we apply these shuffles to each of the segments γ_i in alternating order: p goes first in the odd-numbered segments and q goes first in the odd-numbered segments. Let's write the shuffled version of γ_i as $\alpha_i\beta_i$ for odd i and $\beta_i\alpha_i$ for even i; in each case, α_i contains only events of p and other processes that aren't q and β_i contains only events of q and other processes that aren't p.

When we put these alternating shuffles together, we get an execution that looks like this example with $s-1=4$:

$$\alpha_1\beta_1\beta_2\alpha_2\alpha_3\beta_3\beta_4\alpha_4\delta$$

Now let's count sessions. Since a session includes special actions by both

[1] Proof: Because $e_p \not\underset{\gamma\delta}{\Rightarrow} e_q$, we can add $e_q < e_p$ for all events e_q and e_p in γ_i and still have a partial order consistent with $\underset{\gamma\delta}{\Rightarrow}$. Now apply topological sort to get the shuffle.

p and q, it can't lie entirely within α intervals or β intervals. contains only steps of p and other processes that aren't q or an interval that contains only steps of q and other processes that aren't p. So any session has to span one of the points in the schedule marked by slashes below:

$$\alpha_1/\beta_1\beta_2/\alpha_2\alpha_3/\beta_3\beta_4/\alpha_4\delta$$

There is one such point for each of our original $s-1$ intervals, so we get at most $s-1$ sessions.

This means that any algorithm that runs in time $(s-1)D$ in the worst case (here, the original synchronous execution) can't guarantee to give s sessions in all cases (it fails in the shuffled asynchronous execution). Note that this is not quite the same as saying that any execution with at least s sessions must take $(s-1)D$ time. Instead, we've shown that algorithm that guarantees we get at least s sessions sometimes takes more than $(s-1)D$ time, even though it might sometimes use less time if it gets lucky.

Chapter 8

Coordinated attack

(See also [Lyn96, §5.1].)

The **Two Generals** problem was the first widely-known distributed consensus problem, described in 1978 by Jim Gray [Gra78, §5.8.3.3.1], although the same problem previously appeared under a different name [AEH75].

The setup of the problem is that we have two generals on opposite sides of an enemy army, who must choose whether to attack the army or retreat. If only one general attacks, his troops will be slaughtered. So the generals need to reach agreement on their strategy.

To complicate matters, the generals can only communicate by sending messages by (unreliable) carrier pigeon. We also suppose that at some point each general must make an irrevocable decision to attack or retreat. The interesting property of the problem is that if carrier pigeons can become lost, there is no protocol that guarantees agreement in all cases unless the outcome is predetermined (e.g., the generals always attack no matter what happens). The essential idea of the proof is that any protocol that does guarantee agreement can be shortened by deleting the last message; iterating this process eventually leaves a protocol with no messages.

Adding more generals turns this into the **coordinated attack** problem, a variant of **consensus**, but it doesn't make things any easier.

8.1 Formal description

To formalize this intuition, suppose that we have $n \geq 2$ generals in a synchronous system with unreliable channels—the set of messages received in round $i + 1$ is always a subset of the set sent in round i, but it may be a proper subset (even the empty set). Each general starts with an input 0

(retreat) or 1 (attack) and must output 0 or 1 after some bounded number of rounds. The requirements for the protocol are that, in all executions:

Agreement All processes output the same decision (0 or 1).

Validity If all processes have the same input x, and no messages are lost, all processes produce output x. (If processes start with different inputs or one or more messages are lost, processes can output 0 or 1 as long as they all agree.)

Termination All processes terminate in a bounded number of rounds.[1]

Sadly, there is not protocol that satisfies all three conditions. We show this in the next section.

8.2 Impossibility proof

To show coordinated attack is impossible,[2] we use an **indistinguishability proof**.

The key steps of an indistinguishability proof usually look like this:

- Show that execution A is **indistinguishable** from execution B for some process p, meaning that p sees the same things (messages or operation results) in both executions.
- Observe that if A is indistinguishable from B for p, then because p can't tell which of these two possible worlds it is in, it returns the same output in both.

So far, pretty dull. But now let's consider a chain of hypothetical executions $A = A_0 A_1 \dots A_k = B$, where each A_i is indistinguishable from A_{i+1} for some process p_i. Suppose also that we are trying to solve an agreement task, where every process must output the same value. Then since p_i outputs the same value in A_i and A_{i+1}, every process outputs the same

[1] **Bounded** means that there is a fixed upper bound on the length of any execution. We could also demand merely that all processes terminate in a *finite* number of rounds. In general, finite is a weaker requirement than bounded, but if the number of possible outcomes at each step is finite (as they are in this case), they're equivalent. The reason is that if we build a tree of all configurations, each configuration has only finitely many successors, and the length of each path is finite, then **König's lemma** (see http://en.wikipedia.org/wiki/Konig's_lemma) says that there are only finitely many paths. So we can take the length of the longest of these paths as our fixed bound. [BG97, Lemma 3.1]

[2] Without making additional assumptions, always a caveat when discussing impossibility.

value in A_i and A_{i+1}. By induction on k, every process outputs the same value in A and B, even though A and B may be very different executions.

This gives us a tool for proving impossibility results for agreement: show that there is a path of indistinguishable executions between two executions that are supposed to produce different output. Another way to picture this: consider a graph whose nodes are all possible executions with an edge between any two indistinguishable executions; then the set of output-0 executions can't be adjacent to the set of output-1 executions. If we prove the graph is connected, we prove the output is the same for all executions.

For coordinated attack, we will show that no protocol satisfies all of agreement, validity, and termination using an indistinguishability argument. The key idea is to construct a path between the all-0-input and all-1-input executions with no message loss via intermediate executions that are indistinguishable to at least one process.

Let's start with $A = A_0$ being an execution in which all inputs are 1 and all messages are delivered. We'll build executions A_1, A_2, etc., by pruning messages. Consider A_i and let m be some message that is delivered in the last round in which any message is delivered. Construct A_{i+1} by not delivering m. Observe that while A_i is distinguishable from A_{i+1} by the recipient of m, on the assumption that $n \geq 2$ there is some other process that can't tell whether m was delivered or not (the recipient can't let that other process know, because no subsequent message it sends are delivered in either execution). Continue until we reach an execution A_k in which all inputs are 1 and no messages are sent. Next, let A_{k+1} through A_{k+n} be obtained by changing one input at a time from 1 to 0; each such execution is indistinguishable from its predecessor by any process whose input didn't change. Finally, construct A_{k+n} through $A_{k+n+k'}$ by adding back messages in the reverse process used for A_0 through A_k; note that this might not result in exactly k new messages, because the number of messages might depend on the inputs. This gets us to an execution $A_{k+n+k'}$ in which all processes have input 0 and no messages are lost. If agreement holds, then the indistinguishability of adjacent executions to some process means that the common output in A_0 is the same as in $A_{k+n+k'}$. But validity requires that A_0 outputs 1 and $A_{k+n+k'}$ outputs 0: so either agreement or validity is violated in some execution.

8.3 Randomized coordinated attack

So we now know that we can't solve the coordinated attack problem. But maybe we want to solve it anyway. The solution is to change the problem.

Randomized coordinated attack is like standard coordinated attack, but with less coordination. Specifically, we'll allow the processes to flip coins to decide what to do, and assume that the communication pattern (which messages get delivered in each round) is fixed and independent of the coin-flips. This corresponds to assuming an **oblivious adversary** that can't see what is going on at all or perhaps a **content-oblivious adversary** that can only see where messages are being sent but not the contents of the messages. We'll also relax the agreement property to only hold with some high probability:

Randomized agreement For any adversary A, the probability that some process decides 0 and some other process decides 1 given A is at most ϵ.

Validity and termination are as before.

8.3.1 An algorithm

Here's an algorithm that gives $\epsilon = 1/r$. (See [Lyn96, §5.2.2] for details or [VL92] for the original version.) A simplifying assumption is that network is complete, although a strongly-connected network with r greater than or equal to the diameter also works.

- First part: tracking information levels
  - Each process tracks its "information level," initially 0. The state of a process consists of a vector of (input, information-level) pairs for all processes in the system. Initially this is (my-input, 0) for itself and $(\perp, -1)$ for everybody else.
  - Every process sends its entire state to every other process in every round.
  - Upon receiving a message m, process i stores any inputs carried in m and, for each process j, sets $\mathsf{level}_i[j]$ to $\max(\mathsf{level}_i[j], \mathsf{level}_m[j])$. It then sets its own information level to $\min_j(\mathsf{level}_i[j]) + 1$.
- Second part: deciding the output
  - Process 1 chooses a random key value uniformly in the range $[1, r]$.

- This key is distributed along with $\mathsf{level}_i[1]$, so that every process with $\mathsf{level}_i[1] \geq 0$ knows the key.
- A process decides 1 at round r if and only if it knows the key, its information level is greater than or equal to the key, and all inputs are 1.

8.3.2 Why it works

Termination Immediate from the algorithm.

Validity

- If all inputs are 0, no process sees all 1 inputs (technically requires an invariant that processes' non-null views are consistent with the inputs, but that's not hard to prove.)
- If all inputs are 1 and no messages are lost, then the information level of each process after k rounds is k (prove by induction) and all processes learn the key and all inputs (immediate from first round). So all processes decide 1.

Randomized Agreement

- First prove a lemma: Define $\mathsf{level}_i^t[k]$ to be the value of $\mathsf{level}_i[k]$ after t rounds. Then for all i, j, k, t, (1) $\mathsf{level}_i[j]^t \leq \mathsf{level}_j[j]^{t-1}$ and (2) $\left|\mathsf{level}_i[k]^t - \mathsf{level}_j[k]^t\right| \leq 1$. As always, the proof is by induction on rounds. Part (1) is easy and boring so we'll skip it. For part (2), we have:
  - After 0 rounds, $\mathsf{level}_i^0[k] = \mathsf{level}_j^0[k] = -1$ if neither i nor j equals k; if one of them is k, we have $\mathsf{level}_k^0[k] = 0$, which is still close enough.
  - After t rounds, consider $\mathsf{level}_i^t[k] - \mathsf{level}_i^{t-1}[k]$ and similarly $\mathsf{level}_j^t[k] - \mathsf{level}_j^{t-1}[k]$. It's not hard to show that each can jump by at most 1. If both deltas are $+1$ or both are 0, there's no change in the difference in views and we win from the induction hypothesis. So the interesting case is when $\mathsf{level}_i[k]$ stays the same and $\mathsf{level}_j[k]$ increases or vice versa.
  - There are two ways for $\mathsf{level}_j[k]$ to increase:
    - If $j \neq k$, then j received a message from some j' with $\mathsf{level}_{j'}^{t-1}[k] > \mathsf{level}_j^{t-1}[k]$. From the induction hypothesis, $\mathsf{level}_{j'}^{t-1}[k] \leq \mathsf{level}_i^{t-1}[k] + 1 = \mathsf{level}_i^t[k]$. So we are happy.
    - If $j = k$, then j has $\mathsf{level}_j^t[j] = 1 + \min_{k \neq j} \mathsf{level}_j^t[k] \leq 1 + \mathsf{level}_j^t[i] \leq 1 + \mathsf{level}_i^t[i]$. Again we are happy.

- Note that in the preceding, the key value didn't figure in; so everybody's level at round r is independent of the key.
- So now we have that $\mathsf{level}_i^r[i]$ is in $\{\ell, \ell+1\}$, where ℓ is some fixed value uncorrelated with the key. The only way to get some process to decide 1 while others decide 0 is if $\ell + 1 \geq \mathsf{key}$ but $\ell < \mathsf{key}$. (If $\ell = 0$, a process at this level doesn't know key, but it can still reason that $0 < \mathsf{key}$ since key is in $[1, r]$.) This can only occur if $\mathsf{key} = \ell + 1$, which occurs with probability at most $1/r$ since key was chosen uniformly.

8.3.3 Almost-matching lower bound

The bound on the probability of disagreement in the previous algorithm is almost tight. Varghese and Lynch [VL92] show that no synchronous algorithm can get a probability of disagreement less than $\frac{1}{r+1}$, using a stronger validity condition that requires that the processes output 0 if any input is 0. This is a natural assumption for database commit, where we don't want to commit if any process wants to abort. We restate their result below:

Theorem 8.3.1. *For any synchronous algorithm for randomized coordinated attack that runs in r rounds that satisfies the additional condition that all non-faulty processes decide 0 if any input is 0,* $\Pr[\textit{disagreement}] \geq 1/(r+1)$.

Proof. Let ϵ be the bound on the probability of disagreement. Define $\mathsf{level}_i^t[k]$ as in the previous algorithm (whatever the real algorithm is doing). We'll show $\Pr[i \text{ decides } 1] \leq \epsilon \cdot (\mathsf{level}_i^r[i] + 1)$, by induction on $\mathsf{level}_i^r[i]$.

- If $\mathsf{level}_i^r[i] = 0$, the real execution is indistinguishable (to i) from an execution in which some other process j starts with 0 and receives no messages at all. In that execution, j must decide 0 or risk violating the strong validity assumption. So i decides 1 with probability at most ϵ (from the disagreement bound).
- If $\mathsf{level}_i^r[i] = k > 0$, the real execution is indistinguishable (to i) from an execution in which some other process j only reaches level $k-1$ and thereafter receives no messages. From the induction hypothesis, $\Pr[j \text{ decides } 1] \leq \epsilon k$ in that pruned execution, and so $\Pr[i \text{ decides } 1] \leq \epsilon(k+1)$ in the pruned execution. But by indistinguishability, we also have $\Pr[i \text{ decides } 1] \leq \epsilon(k+1)$ in the original execution.

Now observe that in the all-1 input execution with no messages lost, $\mathsf{level}_i^r[i] = r$ and $\Pr[i \text{ decides } 1] = 1$ (by validity). So $1 \leq \epsilon(r+1)$, which implies $\epsilon \geq 1/(r+1)$. □

Chapter 9

Synchronous agreement

Here we'll consider synchronous agreement algorithm with stopping failures, where a process stops dead at some point, sending and receiving no further messages. We'll also consider Byzantine failures, where a process deviates from its programming by sending arbitrary messages, but mostly just to see how crash-failure algorithms hold up; for algorithms designed specifically for a Byzantine model, see Chapter 10.

If the model has communication failures instead, we have the coordinated attack problem from Chapter 8.

9.1 Problem definition

We use the usual synchronous model with n processes with binary inputs and binary outputs. Up to f processes may fail at some point; when a process fails, one or one or more of its outgoing messages are lost in the round of failure and all outgoing messages are lost thereafter.

There are two variants on the problem, depending on whether we want a useful algorithm (and so want strong conditions to make our algorithm more useful) or a lower bound (and so want weak conditions to make our lower bound more general). For algorithms, we will ask for these conditions to hold:

Agreement All non-faulty processes decide the same value.

Validity If all processes start with the same input, all non-faulty processes decide it.

Termination All non-faulty processes eventually decide.

For lower bounds, we'll replace validity with **non-triviality** (often called validity in the literature):

Non-triviality There exist failure-free executions A and B that produce different outputs.

Non-triviality follows from validity but doesn't imply validity; for example, a non-trivial algorithm might have the property that if all non-faulty processes start with the same input, they all decide something else.

In §9.2, we'll show that a simple algorithm gives agreement, termination, and validity with f failures using $f + 1$ rounds. We'll then show in §9.3 that non-triviality, agreement, and termination imply that $f + 1$ rounds is the best possible. In Chapter 10, we'll show that the agreement is still possible in $f + 1$ rounds even if faulty processes can send arbitrary messages instead of just crashing, but only if the number of faulty processes is strictly less than $n/3$.

9.2 Solution using flooding

The flooding algorithm, due to Dolev and Strong [DS83] gives a straightforward solution to synchronous agreement for the crash failure case. It runs in $f + 1$ rounds assuming f crash failures. The algorithm given here is a gross simplification of Dolev and Strong's original algorithm, which solves the harder problem of authenticated Byzantine agreement. (This algorithm is also described in more detail in [AW04, §5.1.3] or [Lyn96, §6.2.1].)

Each process keeps a set of (process, input) pairs, initially just $\{(\mathsf{myId}, \mathsf{myInput})\}$. At round r, I broadcast my set to everybody and take the union of my set and all sets I receive. At round $f + 1$, I decide on $f(S)$, where f is some fixed function from sets of process-input pairs to outputs that picks some input in S: for example, f might take the input with the smallest process-id attached to it, take the max of all known input values, or take the majority of all known input values.

Lemma 9.2.1. *After $f + 1$ rounds, all non-faulty processes have the same set.*

Proof. Let S_i^r be the set stored by process i after r rounds. What we'll really show is that if there are no failures in round k, then $S_i^r = S_j^r = S_i^{k+1}$ for all i, j, and $r > k$. To show this, observe that no faults in round k means that all processes that are still alive at the start of round k send their message to all other processes. Let L be the set of live processes in round k. At the

end of round k, for i in L we have $S_i^{k+1} = \bigcup_{j \in L} S_j^k = S$. Now we'll consider some round $r = k + 1 + m$ and show by induction on m that $S_i^{k+m} = S$; we already did $m = 0$, so for larger m notice that all messages are equal to S and so S_i^{k+1+m} is the union of a whole bunch of S's. So in particular we have $S_i^{f+1} = S$ (since some failure-free round occurred in the preceding $f+1$ rounds) and everybody decides the same value $f(S)$. □

9.2.1 Authenticated version

Flooding depends on being able to trust second-hand descriptions of values; it may be that process 1 fails in round 0 so that only process 2 learns its input. If process 2 can suddenly tell 3 (but nobody else) about the input in round $f + 1$—or worse, tell a different value to 3 and 4—then we may get disagreement.

Usually we assume that we don't have access to cryptography, but if we include an authentication mechanism that allows processes to attach unforgeable signatures to messages, then the full version of the Dolev-Strong algorithm solves agreement in $f + 1$ even with f **Byzantine** faults, where a process can send any messages it likes regardless of the protocol. The idea is that instead of sending around unauthenticated input values, I send around input values that are authenticated by a sequence of signatures, one for each process that forwarded it. So a value v_1 that started as the input to process p_1 and reached me via processes p_2 and p_3 might arrive in a message as $\langle v_1, 123, S_3(S_2(S_1(v_1)))\rangle$, giving the value, the path it reached me by, and a nested sequence of signatures allowing me to verify that it did in fact travel this path.

To avoid mischief, a process will accept in round r only a message that appears to have traveled a path involving $f+1$ processes, and will only resend values it accepts. We can limit message complexity by having each process resend only the first copy of each value it accepts, and only to processes that are not already listed in the history.

We now have the property that any value a non-faulty process accepts in round $f + 1$ passed through $f + 1$ processes, including at least one non-faulty process. That non-faulty process will have forwarded it to all non-faulty processes. If a process accepts a value earlier than round $f + 1$, then it forwards it itself. In either case, if you and I are both non-faulty, then I know that my eventual set S is a subset of yours. Since this holds in reverse as well, my S equals your S' and so we decide the same value $f(S) = f(S')$.

The intuition here is that if a Byzantine process can be forced to show its work, Byzantine failures essentially reduce to omission failures, since a

non-faulty process can discard any incoming messages that are obviously bogus. For the most part we will not assume that we have the tools to do this, and that catching Byzantine processes will require more careful protocols.

9.3 Lower bound on rounds

Here we show that synchronous agreement requires at least $f + 1$ rounds if f processes can fail. This proof is modeled on the one in [Lyn96, §6.7] and works backwards from the final state; for a proof of the same result that works in the opposite direction, see [AW04, §5.1.4]. The original result (stated for Byzantine failures) is due to Dolev and Strong [DS83], based on a more complicated proof due to Fischer and Lynch [FL82]; see the chapter notes for Chapter 5 of [AW04] for more discussion of the history.

Note that unlike the algorithms in the preceding and following sections, which provide validity, the lower bound applies even if we only demand non-triviality.

Like the similar proof for coordinated attack (§8.2), the proof uses an indistinguishability argument. But we have to construct a more complicated chain of intermediate executions.

A **crash failure** at process i means that (a) in some round r, some or all of the messages sent by i are not delivered, and (b) in subsequent rounds, no messages sent by i are delivered. The intuition is that i keels over dead in the middle of generating its outgoing messages for a round. Otherwise i behaves perfectly correctly. A process that crashes at some point during an execution is called **faulty**

We will show that if up to f processes can crash, and there are at least $f + 2$ processes,[1] then at least $f + 1$ rounds are needed (in some execution) for any algorithm that satisfies agreement, termination, and non-triviality. In particular, we will show that if all executions run in f or fewer rounds, then the indistinguishability graph is connected; this implies non-triviality doesn't hold, because (as in §8.2), two adjacent states must decide the same value because of the agreement property.[2]

[1] With only $f + 1$ processes, we can solve agreement in f rounds using flooding. The idea is that either (a) at most $f - 1$ processes crash, in which case the flooding algorithm guarantees agreement; or (b) exactly f processes crash, in which case the one remaining non-faulty process agrees with itself. So $f + 2$ processes are needed for the lower bound to work, and we should be suspicious of any lower bound proof that does not use this assumption.

[2] The same argument works with even a weaker version of non-triviality that omits the requirement that A and B are failure-free, but we'll keep things simple.

Now for the proof. To simplify the argument, let's assume that all executions terminate in exactly f rounds (we can always have processes send pointless chitchat to pad out short executions) and that every processes sends a message to every other process in every round where it has not crashed (more pointless chitchat). Formally, this means we have a sequence of rounds $0, 1, 2, \ldots, f-1$ where each process sends a message to every other process (assuming no crashes), and a final round f where all processes decide on a value (without sending any additional messages).

We now want to take any two executions A and B and show that both produce the same output. To do this, we'll transform A's inputs into B's inputs one process at a time, crashing processes to hide the changes. The problem is that just crashing the process whose input changed might change the decision value—so we have to crash later witnesses carefully to maintain indistinguishability all the way across the chain.

Let's say that a process p **crashes fully** in round r if it crashes in round r and no round-r messages from p are delivered. The **communication pattern** of an execution describes which messages are delivered between processes without considering their contents—in particular, it tells us which processes crash and what other processes they manage to talk to in the round in which they crash.

With these definitions, we can state and prove a rather complicated induction hypothesis:

Lemma 9.3.1. *For any f-round protocol with $n \geq f+2$ processes permitting up to f crash failures; any process p; and any execution A in which at most one process crashes per round in rounds $0 \ldots r-1$, p crashes fully in round $r+1$, and no other processes crash; there is a sequence of executions $A = A_0 A_1 \ldots A_k$ such that each A_i is indistinguishable from A_{i+1} by some process, each A_i has at most one crash per round, and the communication pattern in A_k is identical to A except that p crashes fully in round r.*

Proof. By induction on $f - r$. If $r = f$, we just crash p in round r and nobody else notices. For $r < f$, first crash p in round r instead of $r+1$, but deliver all of its round-r messages anyway (this is needed to make space for some other process to crash in round $r+1$). Then choose some message m sent by p in round r, and let p' be the recipient of m. We will show that we can produce a chain of indistinguishable executions between any execution in which m is delivered and the corresponding execution in which it is not.

If $r = f - 1$, this is easy; only p' knows whether m has been delivered, and since $n \geq f+2$, there exists another non-faulty p'' that can't distinguish between these two executions, since p' sends no messages in round f or later.

If $r < f - 1$, we have to make sure p' doesn't tell anybody about the missing message.

By the induction hypothesis, there is a sequence of executions starting with A and ending with p' crashing fully in round $r + 1$, such that each execution is indistinguishable from its predecessor. Now construct the sequence

$$\begin{aligned} A &\to (A \text{ with } p' \text{ crashing fully in } r+1) \\ &\to (A \text{ with } p' \text{ crashing fully in } r+1 \text{ and } m \text{ lost}) \\ &\to (A \text{ with } m \text{ lost and } p' \text{ not crashing}). \end{aligned}$$

The first and last step apply the induction hypothesis; the middle one yields indistinguishable executions since only p' can tell the difference between m arriving or not and its lips are sealed.

We've shown that we can remove one message through a sequence of executions where each pair of adjacent executions is indistinguishable to some process. Now paste together $n - 1$ such sequences (one per message) to prove the lemma. □

The rest of the proof: Crash some process fully in round 0 and then change its input. Repeat until all inputs are changed.

9.4 Variants

So far we have described **binary consensus**, since all inputs are 0 or 1. We can also allow larger input sets. With crash failures, this allows a stronger validity condition: the output must be equal to some non-faulty process's input. It's not hard to see that Dolev-Strong (§9.2) gives this stronger condition.

Chapter 10

Byzantine agreement

Like synchronous agreement (as in Chapter 9) except that we replace crash failures with **Byzantine failures**, where a faulty process can ignore its programming and send any messages it likes. Since we are operating under a universal quantifier, this includes the case where the Byzantine processes appear to be colluding with each other under the control of a centralized adversary.

10.1 Lower bounds

We'll start by looking at lower bounds.

10.1.1 Minimum number of rounds

We've already seen an $f+1$ lower bound on rounds for crash failures (see §9.3). This lower bound applies *a fortiori* to Byzantine failures, since Byzantine failures can simulate crash failures.

10.1.2 Minimum number of processes

We can also show that we need $n > 3f$ processes. For $n = 3$ and $f = 1$ the intuition is that Byzantine B can play non-faulty A and C off against each other, telling A that C is Byzantine and C that A is Byzantine. Since A is telling C the same thing about B that B is saying about A, C can't tell the difference and doesn't know who to believe. Unfortunately, this tragic soap opera is not a real proof, since we haven't actually shown that B can say exactly the right thing to keep A and C from guessing that B is evil.

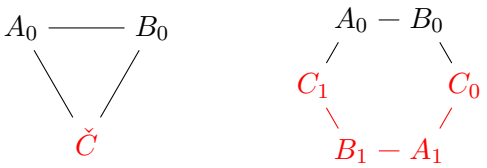


Figure 10.1: Three-process vs. six-process execution in Byzantine agreement lower bound. Processes A_0 and B_0 in right-hand execution receive same messages as in left-hand three-process execution with Byzantine $\check{C}$ simulation C_0 through C_1. So validity forces them to decide 0. A similar argument using Byzantine $\check{A}$ shows the same for C_0.

Here is a real proof, which works by explicitly showing how to construct a bad execution for any given algorithm.[1] Consider an artificial execution where (non-Byzantine) A, B, and C are duplicated and then placed in a ring $A_0B_0C_0A_1B_1C_1$, where the digits indicate inputs. We'll still keep the same code for $n = 3$ on each process, but when A_0 tries to send a message to what it thinks of as just C we'll send it to C_1 while messages from B_0 will instead go to C_0. For any adjacent pair of processes (e.g. A_0 and B_0), the behavior of the rest of the ring could be simulated by a single Byzantine process ($\check{C}$), so each process in the 6-process ring behaves just as it does in some 3-process execution with 1 Byzantine process. It follows that all of the processes terminate and decide in the unholy 6-process Frankenexecution[2] the same value that they would in the corresponding 3-process Byzantine execution. So what do they decide?

Given two processes with the same input, say, A_0 and B_0, the giant execution is indistinguishable from an $A_0B_0\check{C}$ execution where $\check{C}$ is Byzantine (see Figure 10.1. Validity says A_0 and B_0 must both decide 0. Since this works for any pair of processes with the same input, we have each process deciding its input. But now consider the execution of $C_0A_1\check{B}$, where $\check{B}$ is Byzantine. In the big execution, we just proved that C_0 decides 0 and A_1 decides 1, but since the $C_0A_1\check{B}$ execution is indistinguishable from the big execution to C_0 and A_1, they do the same thing here and violate agreement.

This shows that with $n = 3$ and $f = 1$, we can't win. We can generalize this to $n = 3f$. Suppose that there were an algorithm that solved Byzantine

[1] The presentation here is based on [AW04, §5.2.3]. The original impossibility result is due to Pease, Shostak, and Lamport [PSL80]. This particular proof is due to Fischer, Lynch, and Merritt [FLM86].

[2] Not a real word.

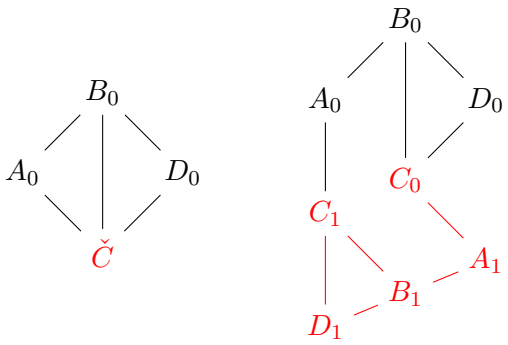


Figure 10.2: Four-process vs. eight-process execution in Byzantine agreement connectivity lower bound. Because Byzantine $\check{C}$ can simulate C_0, D_1, B_1, A_1, and C_1, good processes A_0, B_0 and D_0 must all decide 0 or risk violating validity.

agreement with $n = 3f$ processes. Group the processes into groups of size f, and let each of the $n = 3$ processes simulate one group, with everybody in the group getting the same input, which can only make things easier. Then we get a protocol for $n = 3$ and $f = 1$, an impossibility.

10.1.3 Minimum connectivity

So far, we've been assuming a complete communication graph. If the graph is not complete, we may not be able to tolerate as many failures. In particular, we need the connectivity of the graph (minimum number of nodes that must be removed to split it into two components) to be at least $2f + 1$. See [Lyn96, §6.5] for the full proof. The essential idea is that if we have an arbitrary graph with a vertex cut of size $k < 2f + 1$, we can simulate it on a 4-process graph where A is connected to B and C (but not D), B and C are connected to each other, and D is connected only to B and C. Here B and C each simulate half the processes in the size-k cut, A simulates all the processes on one side of the cut and D all the processes on the other side. We then construct an 8-process artificial execution with two non-faulty copies of each of A, B, C, and D and argue that if one of B or C can be Byzantine then the 8-process execution is indistinguishable to the remaining processes from a normal 4-process execution. (See Figure 10.1.)

An argument similar to the $n > 3f$ proof then shows we violate one of validity or agreement: if we replacing C_0, C_1, and all the nodes on one side of

the $C_0 + C_1$ cut with a single Byzantine $\check{C}$, we force the remaining non-faulty nodes to decide their inputs or violate validity. But then doing the same thing with B_0 and B_1 yields an execution that violates agreement.

Conversely, if we have connectivity $2f+1$, then the processes can simulate a general graph by sending each other messages along $2f+1$ predetermined vertex-disjoint paths and taking the majority value as the correct message. Since the f Byzantine processes can only corrupt one path each (assuming the non-faulty processes are careful about who they forward messages from), we get at least $f+1$ good copies overwhelming the f bad copies. This reduces the problem on a general graph with sufficiently high connectivity to the problem on a complete graph, allowing Byzantine agreement to be solved if the other lower bounds are met.

10.1.4 Weak Byzantine agreement

(Here we are following [Lyn96, §6.6]. The original result is due to Lamport [Lam83].)

Weak Byzantine agreement is like regular Byzantine agreement, but validity is only required to hold if there are no faulty processes at all. If there is a single faulty process, the non-faulty processes can output any value regardless of their inputs (as long as they agree on it). Sadly, this weakening doesn't improve things much: even weak Byzantine agreement can be solved only if $n \geq 3f + 1$.

Proof: As in the strong Byzantine agreement case, we'll construct a many-process Frankenexecution to figure out a strategy for a single Byzantine process in a 3-process execution. The difference is that now the number of processes in our synthetic execution is much larger, since we want to build an execution where at least some of our test subjects think they are in a non-Byzantine environment. The trick is to build a very big, highly-symmetric ring so that at least some of the processes are so far away from the few points of asymmetry that might clue them in to their odd condition that the protocol terminates before they notice.

Fix some protocol that allegedly solves weak Byzantine agreement, and let r be the number of rounds for the protocol. Construct a ring of $6r$ processes $A_{01}B_{01}C_{01}A_{02}B_{02}C_{02}\ldots A_{0r}B_{0r}C_{0r}A_{10}B_{10}C_{10}\ldots A_{1r}B_{1r}C_{1r}$, where each X_{ij} runs the code for process X in the 3-process protocol with input i. For each adjacent pair of processes, there is a 3-process Byzantine execution which is indistinguishable from the $6r$-process execution for that pair: since agreement holds in all Byzantine executions, each adjacent pair decides the same value in the big execution and so either everybody decides

0 or everybody decides 1 in the big execution.

Now we'll show that means that validity is violated in some no-failures 3-process execution. We'll extract this execution by looking at the execution of processes $A_{0,r/2}B_{0,r/2}C_{0,r/2}$. The argument is that up to round r, any input-0 process that is at least r steps in the ring away from the nearest 1-input process acts like the corresponding process in the all-0 no-failures 3-process execution. Since $A_{0,r/2}$ is $3r/2 > r$ hops away from A_{1r} and similarly for $C_{0,r/2}$, our 3 stooges all decide 0 by validity. But now repeat the same argument for $A_{1,r/2}B_{1,r/2}C_{1,r/2}$ and get 3 new stooges that all decide 1. This means that somewhere in between we have two adjacent processes where one decides 0 and one decides 1, violating agreement in the corresponding 3-process execution where the rest of the ring is replaced by a single Byzantine process. This concludes the proof.

This result is a little surprising: we might expect that weak Byzantine agreement could be solved by allowing a process to return a default value if it notices anything that might hint at a fault somewhere. But this would allow a Byzantine process to create disagreement revealing its bad behavior to just one other process in the very last round of an execution otherwise headed for agreement on the non-default value. The chosen victim decides the default value, but since it's the last round, nobody else finds out. Even if the algorithm is doing something more sophisticated, examining the $6r$-process execution will tell the Byzantine process exactly when and how to start acting badly.

10.2 Upper bounds

Here we describe two upper bounds for Byzantine agreement, one of which gets an optimal number of rounds at the cost of many large messages, and the other of which gets smaller messages at the cost of more rounds. (We are following §§5.2.4–5.2.5 of [AW04] in choosing these algorithms.) Neither of these algorithms is state-of-the-art, but they demonstrate some of the issues in solving Byzantine agreement without the sometimes-complicated optimizations needed to get all the parameters of the algorithm down simultaneously.

10.2.1 Exponential information gathering gets $n = 3f + 1$

The idea of **exponential information gathering** is that each process will do a lot of gossiping, but now its state is no longer just a flat set of inputs, but a tree describing who it heard what from. We build this tree out of pairs

of the form $\langle \mathsf{path}, \mathsf{input} \rangle$ where path is a sequence of intermediaries with no repetitions and input is some input. A process i's state at each round is just a set of such pairs, represented by the variables $\mathsf{val}\mathsf{path}, i = \mathsf{input}$. At the end of $f + 1$ rounds of communication (necessary because of the lower bound for crash failures), each non-faulty process i attempts to untangle the complex web of hearsay and second-hand lies to compute the same decision value as the other processes, by computing reconstructed values $\mathsf{val}^*(\mathsf{path}, i)$ that, we hope, will eventually converge to the same values for all processes.

This technique was used by Pease, Shostak, and Lamport [PSL80] to show that their impossibility result is tight: there exists an algorithm for Byzantine agreement that runs in $f + 1$ synchronous rounds and guarantees agreement and validity as long as $n \geq 3f + 1$.

```
   // Set my value to my input
1  val(⟨⟩, i) ← input
2  for round ← 0 . . . f do
       // send step for this round
3      for each non-repeating w, |w| = round, i ∉ w do
4          Send ⟨wi, val(w, i)⟩ to all processes
       // receive step for this round
5      for each non-repeating w, |w| = round do
6          if j sent ⟨wj, v⟩ then
               // Record reported value
7              val(wj, i) ← v
8          else
               // Record default value
9              val(wj, i) ← 0
   // Compute decision value
10 for each path w of length f + 1 with no repeats do
11     val*(w, i) ← val(w, i)
12 for ℓ ← f down to 0 do
13     for each non-repeating w, |w| = ℓ do
14         val*(w, i) ← majority_{j∉w} val*(wj, i)
15 Decide val*(⟨⟩, i)
```

Algorithm 10.1: Exponential information gathering. Code for process i.

The algorithm is given in Algorithm 10.1. The communication phase is just gossiping, where each process starts with its only its input and forwards any values it hears about along with their provenance to all of the other processes. At the end of this phase, each process i has set $\mathsf{val}(\mathsf{path}, i)$ to some value value, where path spans all sequences of 0 to $f+1$ distinct IDs and value is the input value forwarded along that path.

Because we can't trust these $\mathsf{val}(w, i)$ values to be an accurate description of any process's input if there is a Byzantine process in w, each process computes for itself reconstructed values $\mathsf{val}^*(w, i)$ that use majority voting to try to get a more trustworthy picture of the original inputs.

Formally, we think of the set of paths as a tree where w is the parent of wj for each path w and each ID j not in w. To apply EIG in the Byzantine model, ill-formed or missing messages from j are replaced by default values, but otherwise the data-collecting part of EIG proceeds as in the crash failure model. However, we compute the decision value from the last-round values recursively as follows. First, set $\mathsf{val}^*(w, i)$ for any path w with $|w| = f+1$ to $\mathsf{val}(w, i)$. Then for each path w with $|w| < f+1$, define $\mathsf{val}^*(w, i)$ to be the majority value among $\mathsf{val}^*(wj, i)$ for all j. Finally, have process i decide $\mathsf{val}^*(\langle\rangle, i)$. Note that this entire reconstruction process can be computed locally by each process, although we haven't yet shown that i's decision value $\mathsf{val}^*(\langle\rangle, i)$ will necessarily be the same as j's decision value $\mathsf{val}^*(\langle\rangle, j)$.

The majority rule for $w = \langle\rangle$ makes the decision value $\mathsf{val}^*(\langle\rangle, i)$ a majority of reconstructed inputs $\mathsf{val}^*(j, i)$. One way to think about this is that I never trust j to give me the correct value for wj—even when $w = \langle\rangle$ and j is claiming to report its own input—so instead I take a majority of values of wj that j allegedly reported to other people. But since I don't trust those other people either, I use the same process recursively to construct those reports, and hope that all the lies are eventually overcome by the truth.

10.2.1.1 Proof of correctness

This is just a sketch of the proof from [Lyn96, §6.3.2]; essentially the same argument appears in [AW04, §5.2.4].

We start with a basic observation that good processes send and record values correctly. Throughout the proof, we use $\mathsf{val}(w, i)$ for the final value of $\mathsf{val}(w, i)$ recorded by i.

Lemma 10.2.1. *If i and j are both non-faulty, then for all w, $\mathsf{val}(wj, i) = \mathsf{val}(w, j)$.*

Proof. Trivial: j sends $\langle wj, \mathsf{val}(w, i)\rangle$ to i, and i records it in $\mathsf{val}(wj, i)$. □

More involved is this lemma, which says that when we reconstruct a value for a trustworthy process at some level, we get the same value that it sent us. In particular this will be used to show that the reconstructed inputs $\mathsf{val}^*(j,i)$ are all equal to the real inputs for good processes.

Lemma 10.2.2. *If i and j are non-faulty, then for all w, $\mathsf{val}^*(wj,i) = \mathsf{val}(w,j)$.*

Proof. By induction on $f+1-|wj|$. If $|wj| = f+1$, then $\mathsf{val}^*(wj,i) = \mathsf{val}(wj,i) = \mathsf{val}(w,j)$. If $|wj| < f+1$, then then $\mathsf{val}^*(wj,i) = \text{majority}_{k \notin wj}\, \mathsf{val}^*(wjk,i)$. The induction hypothesis says $\mathsf{val}^*(wjk,i) = \mathsf{val}(wj,k)$, which equals $\mathsf{val}(w,j)$ by Lemma 10.2.1. Now observe that there are at least $3f+1-|wj| \geq 2f+1$ possible k, of which at most f are faulty, leaving a non-faulty majority all of which have $\mathsf{val}^*(wjk,i) = \mathsf{val}(w,j)$. □

We call a node w **common** if $\mathsf{val}^*(w,i) = \mathsf{val}^*(w,j)$ for all non-faulty i,j. Lemma 10.2.2 implies that wk is common if k is non-faulty. We can also show that any node whose children are all common is also common, whether or not the last process in its label is faulty.

Lemma 10.2.3. *Let wk be common for all k. Then w is common.*

Proof. Recall that, for $|w| < f+1$, $\mathsf{val}^*(w,i)$ is the majority value among all $\mathsf{val}^*(wk,i)$. If all wk are common, then $\mathsf{val}^*(wk,i) = \mathsf{val}^*(wk,j)$ for all non-faulty i and j. so i and j compute the same majority values and get $\mathsf{val}^*(w,i) = \mathsf{val}^*(w,j)$. □

We can now prove the full result.

Theorem 10.2.4. *Exponential information gathering using $f+1$ rounds in a synchronous Byzantine system with at most f faulty processes satisfies validity and agreement, provided $n \geq 3f+1$.*

Proof. Termination: Protocol finishes after $f+1$ rounds.

Validity: Immediate application of Lemmas 10.2.1 and 10.2.2 when $w = \langle\rangle$. We have $\mathsf{val}^*(j,i) = \mathsf{val}(j,i) = \mathsf{val}(\langle\rangle, j)$ for all non-faulty j and i, which means that a majority of the $\mathsf{val}^*(j,i)$ values equal the common input and thus so does $\mathsf{val}^*(\langle\rangle, i)$.

Agreement: Observe that every path has a common node on it, since a path travels through $f+1$ nodes and one of them is good. If we then suppose that the root is not common: by Lemma 10.2.3, it must have a not-common child, that node must have a not-common child, etc. But this constructs a path from the root to a leaf with no not-common nodes, which we just proved can't happen. □

10.2.2 Phase king gets constant-size messages

The following algorithm, based on work of Berman, Garay, and Perry [BGP89], achieves Byzantine agreement in $2(f+1)$ rounds using constant-size messages, provided $n \geq 4f + 1$. The description here is drawn from [AW04, §5.2.5]. The original Berman-Garay-Perry paper gives somewhat better bounds, but the algorithm and its analysis are more complicated.

10.2.2.1 The algorithm

The main idea of the algorithm is that we avoid the recursive majority voting of EIG by running a vote in each of $f + 1$ *phases* through a **phase king**, some process chosen in advance to run the phase. Since the number of phases exceeds the number of faults, we eventually get a non-faulty phase king. The algorithm is structured so that one non-faulty phase king is enough to generate agreement and subsequent faulty phase kings can't undo the agreement.

Pseudocode appears in Algorithm 10.2. Each processes i maintains an array $\mathsf{pref}_i[j]$, where j ranges over all process IDs. There are also utility values majority, kingMajority and multiplicity for each process that are used to keep track of what it hears from the other processes. Initially, $\mathsf{pref}_i[i]$ is just i's input and $\mathsf{pref}_i[j] = 0$ for $j \neq i$.

The idea of the algorithm is that in each phase, everybody announces their current preference (initially the inputs). If the majority of these preferences is large enough (e.g., all inputs are the same), everybody adopts the majority preference. Otherwise everybody adopts the preference of the phase king. The majority rule means that once the processes agree, they continue to agree despite bad phase kings. The phase king rule allows a good phase king to end disagreement. By choosing a different king in each phase, after $f + 1$ phases, some king must be good. This intuitive description is justified below.

10.2.2.2 Proof of correctness

Termination is immediate from the algorithm.

For validity, suppose all inputs are v. We'll show that all non-faulty i have $\mathsf{pref}_i[i] = v$ after every phase. In the first round of each phase, process i receives at least $n - f$ messages containing v; since $n \geq 4f + 1$, we have $n - f \geq 3f + 1$ and $n/2 + f \leq (4f + 1)/2 + f = 3f + 1/2$, and thus these $n - f$ messages exceed the $n/2 + f$ threshold for adopting them as the new preference. So all non-faulty processes ignore the phase king and stick with v, eventually deciding v after round $2(f + 1)$.

```
1  pref_i[i] = input
2  for j ≠ i do pref_i[j] = 0
3  for k ← 1 to f + 1 do
       // First round of phase k
4      send pref_i[i] to all processes (including myself)
5      pref_i[j] ← v_j, where v_j is the value received from process j
6      majority ← majority value in pref_i
7      multiplicity ← number of times majority appears in pref_i
       // Second round of phase k
8      if i = k then
           // I am the phase king
9          send majority to all processes
10     if received m from phase king then
11         kingMajority ← m
12     else
13         kingMajority ← 0
14     if multiplicity > n/2 + f then
15         pref_i[i] = majority
16     else
17         pref_i[i] = kingMajority
18 return pref_i[i]
```

Algorithm 10.2: Byzantine agreement: phase king

For agreement, we'll ignore all phases up to the first phase with a non-faulty phase king. Let k be the first such phase, and assume that the pref values are set arbitrarily at the start of this phase. We want to argue that at the end of the phase, all non-faulty processes have the same preference. There are two ways that a process can set its new preference in the second round of the phase:

1. The process i observes a majority of more than $n/2 + f$ identical values v and ignores the phase king. Of these values, more than $n/2$ of them were sent by non-faulty processes. So the phase king also receives these values (even if the faulty processes change their stories) and chooses v as its majority value. Similarly, if any other process j observes a majority of $n/2 + f$ identical values, the two $> n/2$ non-faulty parts of the majorities overlap, and so j also chooses v.

2. The process i takes its value from the phase king. We've already shown that i then agrees with any j that sees a big majority; but since the phase king is non-faulty, process i will agree with any process j that also takes its new preference from the phase king.

This shows that after any phase with a non-faulty king, all processes agree. The proof that the non-faulty processes continue to agree is the same as for validity.

10.2.2.3 Performance of phase king

It's not hard to see that this algorithm sends exactly $(f+1)(n^2+n)$ messages of 1 bit each (assuming 1-bit inputs). The cost is doubling the minimum number of rounds and reducing the tolerance for Byzantine processes. As mentioned earlier, a variant of phase-king with 3-round phases gets optimal fault-tolerance with $3(f + 1)$ rounds (but 2-bit messages). Still better is a rather complicated descendant of the EIG algorithm due to Garay and Moses [GM98], which gets $f + 1$ rounds with $n \geq 3f + 1$ while still having polynomial message traffic.

Chapter 11

Impossibility of asynchronous agreement

There's an easy argument that says that you can't do most things in an asynchronous message-passing system with $n/2$ crash failures: partition the processes into two subsets S and T of size $n/2$ each, and allow no messages between the two sides of the partition for some long period of time. Since the processes in each side can't distinguish between the other side being slow and being dead, eventually each has to take action on their own. For many problems, we can show that this leads to a bad configuration. For example, for agreement, we can supply each side of the partition with a different common input value, forcing disagreement because of validity. We can then satisfy the fairness condition that says all messages are eventually delivered by delivering the delayed messages across the partition, but it's too late for the protocol.

The Fischer-Lynch-Paterson (FLP) result [FLP85] says something much stronger: you can't do agreement in an asynchronous message-passing system if even *one* crash failure is allowed.[1] After its initial publication, it was quickly generalized to other models including asynchronous shared memory [LAA87], and indeed the presentation of the result in [Lyn96, §12.2] is given for shared-memory first, with the original result appearing in [Lyn96, §17.2.3] as a corollary of the ability of message passing to simulate shared memory. In these notes, I'll present the original result; the dependence on the model is surprisingly limited, and so most of the proof is the same for both shared memory (even strong versions of shared memory that support operations

[1] Unless you augment the basic model in some way, say by adding randomization (Chapter 24) or failure detectors (Chapter 13).

like atomic snapshots[2]) and message passing.

Section 5.3 of [AW04] gives a very different version of the proof, where it is shown first for two processes in shared memory, then generalized to n processes in shared memory by adapting the classic Borowsky-Gafni simulation [BG93] to show that two processes with one failure can simulate n processes with one failure. This is worth looking at (it's an excellent example of the power of simulation arguments, and BG simulation is useful in many other contexts) but we will stick with the original argument, which is simpler. We will look at this again when we consider BG simulation in Chapter 28.

11.1 Agreement

Usual rules: **agreement** (all non-faulty processes decide the same value), **termination** (all non-faulty processes eventually decide some value), **validity** (for each possible decision value, there an execution in which that value is chosen). Validity can be tinkered with without affecting the proof much.

To keep things simple, we assume the only two decision values are 0 and 1.

11.2 Failures

A failure is an internal action after which all send operations are disabled. The adversary is allowed one failure per execution. Effectively, this means that any group of $n - 1$ processes must eventually decide without waiting for the n-th, because it might have failed.

11.3 Steps

The FLP paper uses a notion of *steps* that is slightly different from the send and receive actions of the asynchronous message-passing model we've been using. Essentially a step consists of receiving zero or more messages followed by doing a finite number of sends. To fit it into the model we've been using, we'll define a step as either a pair (p, m), where p receives message m and performs zero or more sends in response, or $(p, \perp)$, where p receives nothing and performs zero or more sends. We assume that the processes are deterministic, so the messages sent (if any) are determined by p's previous state and the message received. Note that these steps do not correspond

[2] Chapter 20.

precisely to delivery and send events or even pairs of delivery and send events, because what message gets sent in response to a particular delivery may change as the result of delivering some other message; but this won't affect the proof.

The fairness condition essentially says that if (p, m) or $(p, \perp)$ is continuously enabled it eventually happens. Since messages are not lost, once (p, m) is enabled in some configuration C, it is enabled in all successor configurations until it occurs; similarly $(p, \perp)$ is always enabled. So to ensure fairness, we have to ensure that any non-faulty process eventually performs any enabled step.

Comment on notation: I like writing the new configuration reached by applying a step e to C like this: Ce. The FLP paper uses $e(C)$.

11.4 Bivalence and univalence

The core of the FLP argument is a strategy allowing the adversary (who controls scheduling) to steer the execution away from any configuration in which the processes reach agreement. The guidepost for this strategy is the notion of **bivalence**, where a configuration C is **bivalent** if there exist traces T_0 and T_1 starting from C that lead to configurations CT_0 and CT_1 where all processes decide 0 and 1 respectively. A configuration that is not bivalent is **univalent**, or more specifically **0-valent** or **1-valent** depending on whether all executions starting in the configuration produce 0 or 1 as the decision value. (Note that bivalence or univalence are the only possibilities because of termination.) The important fact we will use about univalent configurations is that any successor to an x-valent configuration is also x-valent.

It's clear that any configuration where some process has decided is not bivalent, so if the adversary can keep the protocol in a bivalent configuration forever, it can prevent the processes from ever deciding. The adversary's strategy is to start in an initial bivalent configuration C_0 (which we must prove exists) and then choose only bivalent successor configurations (which we must prove is possible). A complication is that if the adversary is only allowed one failure, it must eventually allow any message in transit to a non-faulty process to be received and any non-faulty process to send its outgoing messages, so we have to show that the policy of avoiding univalent configurations doesn't cause problems here.

11.5 Existence of an initial bivalent configuration

We can specify an initial configuration by specifying the inputs to all processes. If one of these initial configurations is bivalent, we are done. Otherwise, let C and C' be two initial configurations that differ only in the input of one process p; by assumption, both C and C' are univalent. Consider two executions starting with C and C' in which process p is faulty; we can arrange for these executions to be indistinguishable to all the other processes, so both decide the same value x. It follows that both C and C' are x-valent. But since any two initial configurations can be connected by some chain of such indistinguishable configurations, we have that all initial configurations are x-valent, which violations validity.

11.6 Staying in a bivalent configuration

Now start in a failure-free bivalent configuration C with some step $e = (p, m)$ or $e = (p, \perp)$ enabled in C. Let S be the set of configurations reachable from C without doing e or failing any processes, and let $e(S)$ be the set of configurations of the form $C'e$ where C' is in S. (Note that e is always enabled in S, since once enabled the only way to get rid of it is to deliver the message.) We want to show that $e(S)$ contains a failure-free bivalent configuration.

The proof is by contradiction: suppose that $C'e$ is univalent for all C' in S. We will show first that there are C_0 and C_1 in S such that each C_ie is i-valent. To do so, consider any pair of i-valent A_i reachable from C; if A_i is in S, let $C_i = A_i$. If A_i is not in S, let C_i be the last configuration before executing e on the path from C to A_i (C_ie is univalent in this case by assumption).

So now we have C_0e and C_1e with C_ie i-valent in each case. We'll now go hunting for some configuration D in S and step e' such that De is 0-valent but $De'e$ is 1-valent (or vice versa); such a pair exists because S is connected and so some step e' crosses the boundary between the $C'e = 0$-valent and the $C'e = 1$-valent regions.

By a case analysis on e and e' we derive a contradiction:

1. Suppose e and e' are steps of different processes p and p'. Let both steps go through in either order. Then $Dee' = De'e$, since in an asynchronous system we can't tell which process received its message first. But De is 0-valent, which implies Dee' is also 0-valent, which contradicts $De'e$ being 1-valent.

2. Now suppose e and e' are steps of the same process p. Again we let both go through in either order. It is not the case now that $Dee' = De'e$, since p knows which step happened first (and may have sent messages telling the other processes). But now we consider some finite sequence of steps $e_1 e_2 \dots e_k$ in which no message sent by p is delivered and some process decides in $Dee_1 \dots e_k$ (this occurs since the other processes can't distinguish Dee' from the configuration in which p died in D, and so have to decide without waiting for messages from p). This execution fragment is indistinguishable to all processes except p from $De'ee_1 \dots e_k$, so the deciding process decides the same value i in both executions. But Dee' is 0-valent and $De'e$ is 1-valent, giving a contradiction.

It follows that our assumption was false, and there is some reachable bivalent configuration $C'e$.

Now to construct a fair execution that never decides, we start with a bivalent configuration, choose the oldest enabled action and use the above to make it happen while staying in a bivalent configuration, and repeat.

11.7 Generalization to other models

The FLP results extends to any asynchronous model where it is impossible to tell which of two events happened first. The main idea is to replace the definition of a step to whatever is available in the new model, and adapt the resulting case analysis of 0-valent $De'e$ vs 1-valent Dee' as appropriate. For example, in asynchronous shared memory, if e and e' are operations on different memory locations, they commute (just like steps of different processes), and if they are operations on the same location, either they commute (two reads) or only one process can tell whether both happened (with a write and a read, only the reader knows, and with two writes, only the first writer knows). Killing the witness yields two indistinguishable configurations with different valencies, a contradiction.

Loui and Abu-Amara [LAA87] first proved this generalization to shared memory using standard read-write registers. Herlihy [Her91b] later provided similar arguments for a wide variety of shared-memory primitives that may provide additional operations beyond reads and writes. We will see many of these latter arguments in Chapter 19.

Chapter 12

Paxos

The **Paxos** algorithm for consensus in a message-passing system was first described by Lamport in 1990 in a tech report that was widely considered to be a joke (see http://research.microsoft.com/users/lamport/pubs/pubs.html#lamport-paxos for Lamport's description of the history). The algorithm was finally published in 1998 [Lam98], and after the algorithm continued to be ignored, Lamport finally gave up and translated the results into readable English [Lam01]. It is now understood to be one of the most efficient practical algorithms for achieving consensus in a message-passing system with failure detectors, mechanisms that allow processes to give up on other stalled processes after some amount of time (which can't be done in a normal asynchronous system because giving up can be made to happen immediately by the adversary).

We will describe the basic Paxos algorithm in §12.1. This is a one-shot version of Paxos that solves a single agreement problem. The version that is more typically used, called **multi-Paxos**, uses repeated executions of the basic Paxos algorithm to implement a replicated state machine; we'll describe this in §12.7.

There are many more variants of Paxos in use. The WikiPedia article on Paxos (http://en.wikipedia.org/wiki/Paxos_(computer_science)) gives a reasonably good survey of subsequent developments and applications.

12.1 The Paxos algorithm

The algorithm runs in a message-passing model with asynchrony and fewer than $n/2$ crash failures (but not Byzantine failures, at least in the original algorithm). As always, we want to get agreement, validity, and termination.

The Paxos algorithm itself is mostly concerned with guaranteeing agreement and validity, while allowing for the possibility of termination if there is a long enough interval in which no process restarts the protocol. A noteworthy feature of Paxos is that it is robust even to omission failures, in the sense that lost messages can prevent termination, but if new messages start being delivered again, the protocol can recover.

Processes are classified as **proposers**, **accepters**, and **learners** (a single process may have all three roles). The idea is that a proposer attempts to ratify a proposed decision value (from an arbitrary input set) by collecting acceptances from a majority of the accepters, and this ratification is observed by the learners. Agreement is enforced by guaranteeing that only one proposal can get the votes of a majority of accepters, and validity follows from only allowing input values to be proposed. The tricky part is ensuring that we don't get deadlock when there are more than two proposals or when some of the processes fail. The intuition behind how this works is that any proposer can effectively restart the protocol by issuing a new proposal (thus dealing with lockups), and there is a procedure to release accepters from their old votes if we can prove that the old votes were for a value that won't be getting a majority any time soon.

To organize this vote-release process, we attach a distinct proposal number to each proposal. The safety properties of the algorithm don't depend on anything but the proposal numbers being distinct, but since higher numbers override lower numbers, to make progress we'll need them to increase over time. The simplest way to do this in practice is to make the proposal number be a timestamp with the proposer's ID appended to break ties. We could also have the proposer poll the other processes for the most recent proposal number they've seen and add 1 to it.

The revoting mechanism now works like this: before taking a vote, a proposer tests the waters by sending a $\mathsf{prepare}(r)$ message to all accepters, where r is the proposal number. An accepter responds to this with a promise never to accept any proposal with a number less than r (so that old proposals don't suddenly get ratified) together with the highest-numbered proposal that the accepter has accepted (so that the proposer can substitute this value for its own, in case the previous value was in fact ratified). If the proposer receives a response from a majority of the accepters, the proposer then does a second phase of voting where it sends $\mathsf{accept!}(r, v)$ to all accepters and wins if receives a majority of votes. (The exclamation point on $\mathsf{accept!}$ is not in the original paper, but has become a common convention to emphasize that it's a command, not a response.)

So for each proposal, the algorithm proceeds as follows:

1. The proposer sends a message $\mathsf{prepare}(r)$ to all accepters. (Sending to only a majority of the accepters is enough, assuming they will all respond.)

2. Each accepter compares r to the highest-numbered proposal for which it has responded to a $\mathsf{prepare}$ message and the highest-numbered proposal it has accepted. If r is greater than both, it responds with $\mathsf{ack}(r, v, r_v)$, where v is the highest-numbered proposal it has accepted and r_v is the number of that proposal (or $\perp$ and $-\infty$ if there is no such proposal).

   An optimization at this point is to allow the accepter to send back $\mathsf{nack}(r, r')$ where r' is some higher number to let the proposer know that it's doomed and should back off and try again with a higher proposal number. (This keeps a confused proposer who thinks it's the future from locking up the protocol until 2087.)

3. The proposer waits (possibly forever) to receive ack from a majority of accepters. If any ack contained a value, it sets v to the most recent (in proposal number ordering) value that it received. It then sends $\mathsf{accept!}(r, v)$ to all accepters (or just a majority). You should think of $\mathsf{accept!}$ as a demand ("Accept!") rather than acquiescence ("I accept")—the accepters still need to choose whether to accept or not.

4. Upon receiving $\mathsf{accept!}(r, v)$, an accepter accepts v unless it has already received $\mathsf{prepare}(r')$ for some $r' > r$. If a majority of accepters accept the value of a given proposal, that value becomes the decision value of the protocol.

Implementing these rules require only that each accepter track r_{ack}, the highest number of any proposal for which it sent an ack, and $\langle v, r_v \rangle$, the last proposal that it accepted. Pseudocode showing the behavior of proposer and accepters in the core Paxos protocol is given in Algorithm 12.1.

Note that acceptance is a purely local phenomenon; additional messages are needed to detect which if any proposals have been accepted by a majority of accepters. Typically this involves a fourth round, where accepters send $\mathsf{accepted}(r, v)$ to all learners.

There is no requirement that only a single proposal is sent out (indeed, if proposers can fail we will need to send out more to jump-start the protocol). The protocol guarantees agreement and validity no matter how many proposers there are and no matter how often they start.

```
1  procedure Propose(r, v)
       // Issue proposal number r with value v
       // Assumes r is unique
2      send prepare(r, v) to all accepters
3      wait to receive ack(r, v', r_v') from a majority of accepters
4      if some v' is not ⊥ then
5          v ← v' with maximum r_v'
6      send accept!(r, v) to all accepters
7  procedure accepter()
8      initially do
9          r_ack ← −∞
10         v ← ⊥
11         r_v ← −∞
12     upon receiving prepare(r) from p do
13         if r > max(r_ack, r_v) then
               // Respond to proposal
14             send ack(r, v, r_v) to p
15             r_ack ← r
16     upon receiving accept!(r, v') do
17         if r ≥ max(r_ack, r_v) then
               // Accept proposal
18             send accepted(r, v') to all learners
19             if r > r_v then
                   // Update highest accepted proposal
20                 ⟨r_v, v⟩ ← ⟨r, v'⟩
```

Algorithm 12.1: Paxos

12.2 Informal analysis: how information flows between rounds

Call a **round** the collection of all messages labeled with some particular proposal r. The structure of the algorithm simulates a sequential execution in which higher-numbered rounds follow lower-numbered ones, even though there is no guarantee that this is actually the case in a real execution.

When an accepter sends $\mathsf{ack}(r, v, r_v)$, it is telling the round-r proposer the last value preceding round r that it accepted. The rule that an accepter only acknowledges a proposal higher than any proposal it has previously accepted prevents it from sending information "back in time"—the round r_v in an acknowledgment is always less than r. The rule that an accepter doesn't accept any proposal earlier than a round it has acknowledged means that the value v in an $\mathsf{ack}(r, v, r_v)$ message never goes out of date—there is no possibility that an accepter might retroactively accept some later value in round r' with $r_v < r' < r$. So the ack message values tell a consistent story about the history of the protocol, even if the rounds execute out of order.

The second trick is to use overlapping majorities to make sure that any value that is accepted is not lost. If the only way to decide on a value in round r is to get a majority of accepters to accept it, and the only way to make progress in round r' is to get acknowledgments from a majority of accepters, these two majorities overlap. So in particular the overlapping process reports the round-r proposal value to the proposer in round r', and we can show by induction on r' that this round-r proposal value becomes the proposal value in all subsequent rounds that proceed past the acknowledgment stage. So even though it may not be possible to detect that a decision has been reached in round r (say, because some of the accepters in the accepting majority die without telling anybody what they did), no later round will be able to choose a different value. This ultimately guarantees agreement.

12.3 Example execution

For Paxos to work well, proposal numbers should increase over time. But there is no requirement that proposal numbers are increasing or even that proposals with different proposal numbers don't overlap. When thinking about Paxos, it is easy to make the mistake of ignore cases where proposals are processed concurrently or out of order. In Figure 12.1, we give an example of an execution with three proposals running concurrently.

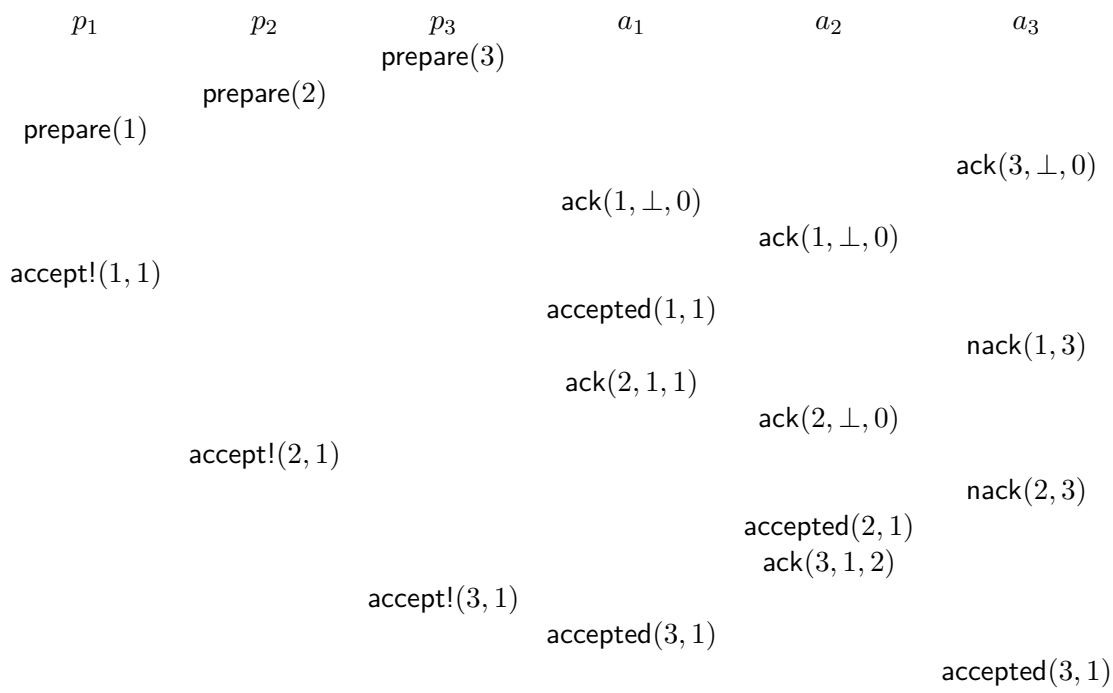


Figure 12.1: Example execution of Paxos. Time increases downward. Each column records messages sent by one of three proposers p_1, p_2, and p_3 and three accepters a_1, a_2, and a_3. Proposer p_1's proposed value 1 is not accepted by a majority of processes in round 1, but it is picked up by proposer p_2 in round 2, and is eventually adopted and accepted in round 3.

12.4 Safety properties

We now present a more formal analysis of the Paxos protocol. We consider only the safety properties of the protocol, corresponding to validity and agreement. Without additional assumptions, Paxos does *not* guarantee termination.

Call a value *chosen* if it is accepted by a majority of accepters. The safety properties of Paxos are:

- No value is chosen unless it is first proposed. (This gives validity.)
- No two distinct values are both chosen. (This gives agreement.)

The first property is immediate from examination of the algorithm: every value propagated through the algorithm is ultimately a copy of some proposer's original input. We can formalize this observation by checking that, for any set of values S, the property that all values contained in messages or processes' internal state are in S is an invariant.

For the second property, we'll show by induction on proposal number that a value v chosen with proposal number r is the value chosen by any proposer $p_{r'}$ with proposal number r'. There are two things that make this true:

1. Any $\mathsf{ack}(r', v', r_{v'})$ message received by $p_{r'}$ has $r_{v'} < r'$. Proof: Immediate from the code.
2. If a majority of accepters accept a proposal with number r at some point during the execution, and $p_{r'}$ receives $\mathsf{ack}(r', -, -)$ messages from a majority of accepters, then $p_{r'}$ receives at least one $\mathsf{ack}(r', v', r_{v'})$ message with $r' \geq r$. Proof: Let S be the set of processes that issue $\mathsf{accepted}(r, v)$ and let T be the set of processes that send $\mathsf{ack}(r', -, -)$ to p'. Because S and T are both majorities, there is at least one accepter a in $S \cap T$. Suppose $p_{r'}$ receives $\mathsf{ack}(r, v'', r'')$ from a. If $r'' < r$, then at the time a sends its $\mathsf{ack}(r, v'', r'')$ message, it has not yet accepted a proposal with number r. But then when it does receive $\mathsf{accept!}(r, v)$, it rejects it. This contradicts $a \in S$.

These two properties together imply that $p_{r'}$ receives at least one $\mathsf{ack}(r, v'', r'')$ with $r \leq r'' < r'$ and no such messages with $r'' < r$. So the maximum proposal number it sees is r'' where $r \leq r'' < r$. By the induction hypothesis, the corresponding value is v. It follows that $p_{r'}$ also chooses v.

12.5 Learning the results

Somebody has to find out that a majority accepted a proposal in order to get a decision value out. The usual way to do this is to have a fourth round of messages where the accepters send accepted(v, r) to some designated learners. These are often the processes that need to implement whatever decision was made by the agreement protocol, but in principle could be any processes that care about the outcome.

12.6 Liveness properties

We'd like the protocol to terminate eventually. Suppose there is a single proposer, and that it survives long enough to collect a majority of acks and to send out accept!s to a majority of the accepters. If everybody else cooperates, we get termination in 4 message delays, including the time for the learners to detect acceptance.

If there are multiple proposers, then they can step on each other. For example, it's enough to have two carefully-synchronized proposers alternate sending out prepare messages to prevent any accepter from every accepting (since an accepter promises not to accept accept!(r, v) once it has responded to prepare$(r + 1)$). The solution is to ensure that there is eventually some interval during which there is exactly one proposer who doesn't fail. One way to do this is to use exponential random backoff (as popularized by Ethernet): when a proposer decides it's not going to win a round (e.g., by receiving a nack or by waiting long enough to realize it won't be getting any more acks soon), it picks some increasingly large random delay before starting a new round. Unless something strange is going on, new rounds will eventually start far enough apart in time that one will get done without interference.

A more abstract solution is to assume some sort of weak leader election mechanism, which tells each accepter who the "legitimate" proposer is at each time. The accepters then discard messages from illegitimate proposers, which prevents conflict at the cost of possibly preventing progress. Progress is however obtained if the mechanism eventually reaches a state where a majority of the accepters bow to the same non-faulty proposer long enough for the proposal to go through.

Such a weak leader election method is an example of a more general class of mechanisms known as **failure detectors**, in which each process gets hints about what other processes are faulty that eventually converge to reality. The weak-leader-election failure detector needed for Paxos is called the Ω

failure detector. There are other still weaker failure detectors that can also be used to solve consensus. We will discuss failure detectors in detail in Chapter 13.

Since implementing this kind of leader election allows us to solve consensus, the FLP result (Chapter 11) implies that we can't build it using only the tools available in the asynchronous message-passing model. In practice, detecting failures and electing a non-faulty leader involves using lots of timeouts. An example of a Paxos-like protocol that does this is the Raft protocol of Ongaro and Osterhout [OO14], which may be the most commonly implemented protocol in this family.

12.7 Replicated state machines and multi-Paxos

The most common practical use of Paxos is to implement a **replicated state machine** [Lam78]. The idea is to maintain many copies of some data structure, each on a separate machine, and guarantee that each copy (or **replica**) stays in sync with all the others as new operations are applied to them. This requires some mechanism to ensure that all the different replicas apply the same sequence of operations, or in other words that the machines that hold the replicas solve a sequence of agreement problems to agree on these operations. The payoff is that the state of the data structure survives the failure of some of the machines, without having to copy the entire structure every time it changes.

Making all copies consistent requires solving a new version of agreement every time we want to add another operation. Paxos works well for this because we can have the proposer simply issue a new proposal without taking into account any lower-numbered values, assuming that it has verified that lower-numbered values have in fact been accepted. The round-number mechanism means that all of the accepters will switch to working on the new proposal without any modifications to their code.

Typically for this application, we'll have a single active proposer that is responsible for serializing any incoming operations to the replicated state machine. If the proposer doesn't change very often, a further optimization allows skipping the prepare and ack messages in between agreement protocols for consecutive operations. This reduces the time to certify each operation to a single round-trip for the accept! and accepted messages, which is about the best we can reasonably hope for.

One detail is that to make this work, we need to distinguish between consecutive proposals by the same proposer, and "new" proposals that change

the proposer in addition to reaching agreement on some value. This is done by splitting the proposal number into a major and minor number, with proposals ordered lexicographically. A proposer that wins $\langle x, 0\rangle$ is allowed to make further proposals numbered $\langle x, 1\rangle$, $\langle x, 2\rangle$, etc. But a different proposer will need to increment x.

Lamport calls this optimization Paxos in [Lam01]; other authors have called it **multi-Paxos** to distinguish it from the basic Paxos algorithm.

Chapter 13

Failure detectors

Failure detectors were proposed by Chandra and Toueg [CT96] as a mechanism for solving consensus in an asynchronous message-passing system with crash failures by distinguishing between slow processes and dead processes. This involves extending the model by giving each process a **failure detector** module that continuously outputs an estimate of which processes in the system have failed. The output does not need to be correct; indeed, the main contribution of Chandra and Toueg's paper (and a companion paper by Chandra, Hadzilacos, and Toueg [CHT96]) is characterizing just how bogus the output of a failure detector can be and still be useful.

We will mostly follow Chandra and Toueg in these notes; see the paper for the full technical details.

To emphasize that the output of a failure detector is merely a hint at the actual state of the world, a failure detector (or the process it's attached to) is said to **suspect** a process at time t if it outputs failed at that time. Failure detectors can then be classified based on when their suspicions are correct.

We use the usual asynchronous message-passing model, and in particular assume that non-faulty processes execute infinitely often, get all their messages delivered, etc. From time to time we will need to talk about time, and unless we are clearly talking about real time this just means any steadily increasing count (e.g., of total events), and will be used only to describe the ordering of events.

13.1 How to build a failure detector

Failure detectors are only interesting if you can actually build them. In a fully asynchronous system, you can't (this follows from the FLP result and the existence of failure-detector-based consensus protocols). But with timeouts, it's not hard: have each process ping each other process from time to time, and suspect the other process if it doesn't respond to the ping within twice the maximum round-trip time for any previous ping. Assuming that ping packets are never lost and there is an (unknown) upper bound on message delay, this gives what is known as an **eventually perfect failure detector**: once the max round-trip times rise enough and enough time has elapsed for the live processes to give up on the dead ones, all and only dead processes are suspected.

13.2 Classification of failure detectors

Chandra and Toueg define eight classes of failure detectors, based on when they suspect faulty processes and non-faulty processes. Suspicion of faulty processes comes under the heading of **completeness**; of non-faulty processes, **accuracy**.

13.2.1 Degrees of completeness

Strong completeness Every faulty process is eventually permanently suspected by every non-faulty process.

Weak completeness Every faulty process is eventually permanently suspected by some non-faulty process.

There are two temporal logic operators embedded in these statements: "eventually permanently" means that there is some time t_0 such that for all times $t \geq t_0$, the process is suspected. Note that completeness says nothing about suspecting non-faulty processes: a paranoid failure detector that permanently suspects everybody has strong completeness.

13.2.2 Degrees of accuracy

These describe what happens with non-faulty processes, and with faulty processes that haven't crashed yet.

Strong accuracy No process is suspected (by anybody) before it crashes.

Weak accuracy Some non-faulty process is never suspected.

Eventual strong accuracy After some initial period of confusion, no process is suspected before it crashes. This can be simplified to say that no non-faulty process is suspected after some time, since we can take end of the initial period of chaos as the time at which the last crash occurs.

Eventual weak accuracy After some initial period of confusion, some non-faulty process is never suspected.

Note that "strong" and "weak" mean different things for accuracy vs completeness: for accuracy, we are quantifying over suspects, and for completeness, we are quantifying over suspectors. Even a weakly-accurate failure detector guarantees that all processes trust the one visibly good process.

13.2.3 Boosting completeness

It turns out that any weakly-complete failure detector can be boosted to give strong completeness. Recall that the difference between weak completeness and strong completeness is that with weak completeness, somebody suspects a dead process, while with strong completeness, everybody suspects it. So to boost completeness we need to spread the suspicion around a bit. On the other hand, we don't want to break accuracy in the process, so there needs to be some way to undo a premature rumor of somebody's death. The simplest way to do this is to let the alleged corpse speak for itself: I will suspect you from the moment somebody else reports you dead until the moment you tell me otherwise.

Pseudocode is given in Algorithm 13.1.

```
1 initially do
2     suspects ← ∅
3 while true do
4     Let S be the set of all processes my weak detector suspects.
5     Send S to all processes.
6 upon receiving S from q do
7     suspects ← (suspects ∪ S) \ {q}
```

Algorithm 13.1: Boosting completeness

It's not hard to see that this boosts completeness: if p crashes, somebody's weak detector eventually suspects it, this process tells everybody else, and p never contradicts it. So eventually everybody suspects p.

What is slightly trickier is showing that it preserves accuracy. The essential idea is this: if there is some good-guy process p that everybody trusts forever (as in weak accuracy), then nobody ever reports p as suspect—this also covers strong accuracy since the only difference is that now every non-faulty process falls into this category. For eventual weak accuracy, wait for everybody to stop suspecting p, wait for every message ratting out p to be delivered, and then wait for p to send a message to everybody. Now everybody trusts p, and nobody every suspects p again. Eventual strong accuracy is again similar.

This will justify ignoring the weakly-complete classes.

13.2.4 Failure detector classes

Two degrees of completeness times four degrees of accuracy gives eight classes of failure detectors, each of which gets its own name. But since we can boost weak completeness to strong completeness, we can use this as an excuse to consider only the strongly-complete classes.

P **(perfect)** Strongly complete and strongly accurate: non-faulty processes are never suspected; faulty processes are eventually suspected by everybody. Easily achieved in synchronous systems.

S **(strong)** Strongly complete and weakly accurate. The name is misleading if we've already forgotten about weak completeness, but the corresponding W (weak) class is only weakly complete and weakly accurate, so it's the strong completeness that the S is referring to.

$\Diamond P$ **(eventually perfect)** Strongly complete and eventually strongly accurate.

$\Diamond S$ **(eventually strong)** Strongly complete and eventually weakly accurate.

Jumping to the punch line: P can simulate any of the others, S and $\Diamond P$ can both simulate $\Diamond S$ but can't simulate P or each other, and $\Diamond S$ can't simulate any of the others (See Figure 13.1—we'll prove all of this later.) Thus $\Diamond S$ is the weakest class of failure detectors in this list. However, $\Diamond S$ *is* strong enough to solve consensus, and in fact any failure detector (whatever

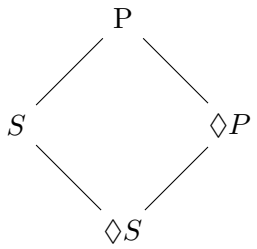


Figure 13.1: Partial order of failure detector classes. Higher classes can simulate lower classes but not vice versa.

its properties) that can solve consensus is strong enough to simulate $\Diamond S$ (this is the result in the Chandra-Hadzilacos-Toueg paper [CHT96])—this makes $\Diamond S$ the "weakest failure detector for solving consensus" as advertised. Continuing our tour through Chandra and Toueg [CT96], we'll show the simulation results and that $\Diamond S$ can solve consensus, but we'll skip the rather involved proof of $\Diamond S$'s special role from Chandra-Hadzilacos-Toueg.

13.3 Consensus with S

With the strong failure detector S, we can solve consensus for any number of failures.

In this model, the failure detectors as applied to most processes are completely useless. However, there is some non-faulty process c that nobody every suspects, and this is enough to solve consensus with as many as $n-1$ failures.

The protocol is carried out in three phases. In the first phase, the processes gossip about input values for $n-1$ asynchronous rounds. In the second, they exchange all the values they've seen and prune out any that are not universally known. In the third, each process decides on the lowest-id input that hasn't been pruned (minimum input also works since at this point everybody has the same view of the inputs).

Pseudocode is given in Algorithm 13.2

In Phase 1, each process p maintains two partial functions V_p and δ_p, where V_p lists all the input values $\langle q, v_q \rangle$ that p has ever seen and δ_p lists only those input values seen in the most recent of $n-1$ asynchronous rounds. Both V_p and δ_p are initialized to $\{\langle p, v_p \rangle\}$. In round i, p sends (i, δ_p) to all processes. It then collects $\langle i, \delta_q \rangle$ from each q that it doesn't suspect and sets δ_p to $\bigcup_q \delta_q \setminus V_p$ (where q ranges over the processes from which p received a

```
1  V_p ← {⟨p, v_p⟩} // All values known to p
2  δ_p ← {⟨p, v_p⟩} // New values p learned last round
   // Phase 1: add values
3  for i ← 1 to n − 1 do
4      Send ⟨i, δ_p⟩ to all processes.
5      Wait to receive ⟨i, δ_q⟩ from all q I do not suspect.
6      δ_p ← (⋃_q δ_q) \ V_p
7      V_p ← (⋃_q δ_q) ∪ V_p
   // Phase 2: subtract values
8  Send ⟨n, V_p⟩ to all processes.
9  Wait to receive ⟨n, V_q⟩ from all q I do not suspect.
10 V_p ← (⋂_q V_q) ∩ V_p
   // Phase 3: decide on something
11 return some input from V_p chosen via a consistent rule.
```

Algorithm 13.2: Consensus with a strong failure detector

message in round i) and sets V_p to $V_p \cup \delta_p$. In the next round, it repeats the process. Note that each pair $\langle q, v_q \rangle$ is only sent by a particular process p the first round after p learns it: so any value that is still kicking around in round $n-1$ had to go through $n-1$ processes.

In Phase 2, each process p sends $\langle n, V_p \rangle$, waits to receive $\langle n, V_q \rangle$ from every process it does not suspect, and sets V_p to the intersection of V_p and all received V_q. At the end of this phase all V_p values will in fact be equal, as we will show.

In Phase 3, everybody picks some input from their V_p vector according to a consistent rule.

13.3.1 Proof of correctness

Let c be a non-faulty process that nobody every suspects.

The first observation is that the protocol satisfies validity, since every V_p contains v_c after round 1 and each V_p can only contain input values by examination of the protocol. Whatever it may do to the other values, taking intersections in Phase 2 still leaves v_c, so all processes pick some input value from a nonempty list in Phase 3.

To get termination we have to prove that nobody ever waits forever for a message it wants; this basically comes down to showing that the first

non-faulty process that gets stuck eventually is informed by the S-detector that the process it is waiting for is dead.

For agreement, we must show that in Phase 3, every V_p is equal; in particular, we'll show that every $V_p = V_c$. First it is necessary to show that at the end of Phase 1, $V_c \subseteq V_p$ for all p. This is done by considering two cases:

1. If $\langle q, v_q \rangle \in V_c$ and c learns $\langle q, v_q \rangle$ before round $n-1$, then c sends $\langle q, v_q \rangle$ to p no later than round $n-1$, p waits for it (since nobody ever suspects c), and adds it to V_p.

2. If $\langle q, v_q \rangle \in V_c$ and c learns $\langle q, v_q \rangle$ only in round $n-1$, then $\langle q, v_q \rangle$ was previously sent through $n-1$ other processes, i.e., all of them. Each process $p \neq c$ thus added $\langle q, v_q \rangle$ to V_p before sending it and again $\langle q, v_q \rangle$ is in V_p.

(The missing case where $\langle q, v_q \rangle$ isn't in V_c we don't care about.)

But now Phase 2 knocks out any extra elements in V_p, since V_p gets set to $V_p \cap V_c \cap$ (some other V_q's that are supersets of V_c). It follows that, at the end of Phase 2, $V_p = V_c$ for all p. Finally, in Phase 3, everybody applies the same selection rule to these identical sets and we get agreement.

13.4 Consensus with $\Diamond S$ and $f < n/2$

The consensus protocol for S depends on some process c never being suspected; if c is suspected during the entire (finite) execution of the protocol—as can happen with $\Diamond S$—then it is possible that no process will wait to hear from c (or anybody else) and the processes will all decide their own inputs. So to solve consensus with $\Diamond S$ we will need to assume fewer than $n/2$ failures, allowing any process to wait to hear from a majority no matter what lies its failure detector is telling it.

The resulting protocol, known as the **Chandra-Toueg consensus protocol**, is structurally similar to the consensus protocol in Paxos.[1] The difference is that instead of proposers blindly showing up, the protocol is divided into rounds with a rotating **coordinator** p_i in each round r with $r = i \pmod{n}$. The termination proof is based on showing that in any round where the coordinator is not faulty and nobody suspects it, the protocol finishes.

[1] See Chapter 12.

The consensus protocol uses as a subroutine a protocol for **reliable broadcast**, which guarantees that any message that is sent is either received by no non-faulty processes or exactly once by all non-faulty processes. Pseudocode for reliable broadcast is given as Algorithm 13.3. It's easy to see that if a process p is non-faulty and receives m, then the fact that p is non-faulty means that is successfully sends m to everybody else, and that the other non-faulty processes also receive the message at least once and deliver it.

```
1 procedure broadcast(m)
2     send m to all processes.
3 upon receiving m do
4     if I haven't seen m before then
5         send m to all processes
6         deliver m to myself
```

Algorithm 13.3: Reliable broadcast

Here's a sketch of the actual consensus protocol:

- Each process keeps track of a preference (initially its own input) and a timestamp, the round number in which it last updated its preference.
- The processes go through a sequence of asynchronous rounds, each divided into four phases:
    1. All processes send (round, preference, timestamp) to the coordinator for the round.
    2. The coordinator waits to hear from a majority of the processes (possibly including itself). The coordinator sets its own preference to some preference with the largest timestamp of those it receives and sends (round, preference) to all processes.
    3. Each process waits for the new proposal from the coordinator *or* for the failure detector to suspect the coordinator. If it receives a new preference, it adopts it as its own, sets timestamp to the current round, and sends (round, ack) to the coordinator. Otherwise, it sends (round, nack) to the coordinator.
    4. The coordinator waits to receive ack or nack from a majority of processes. If it receives ack from a majority, it announces the current preference as the protocol decision value using reliable broadcast.

- Any process that receives a value in a reliable broadcast decides on it immediately.

Pseudocode is in Algorithm 13.4.

```
1  preference ← input
2  timestamp ← 0
3  for round ← 1 . . . ∞ do
4      Send ⟨round, preference, timestamp⟩ to coordinator
5      if I am the coordinator then
6          Wait to receive ⟨round, preference, timestamp⟩ from majority of
           processes.
7          Set preference to value with largest timestamp.
8          Send ⟨round, preference⟩ to all processes.
9      Wait to receive ⟨round, preference′⟩ from coordinator or to suspect
       coordinator.
10     if I received ⟨round, preference′⟩ then
11         preference ← preference′
12         timestamp ← round
13         Send ack(round) to coordinator.
14     else
15         Send nack(round) to coordinator.
16     if I am the coordinator then
17         Wait to receive ack(round) or nack(round) from a majority of
           processes.
18         if I received no nack(round) messages then
19             Broadcast preference using reliable broadcast.
```

Algorithm 13.4: Consensus with an eventually-strong failure detector

13.4.1 Proof of correctness

For validity, observe that the decision value is an estimate and all estimates start out as inputs.

For termination, observe that no process gets stuck in Phase 1, 2, or 4, because either it isn't waiting or it is waiting for a majority of non-faulty processes who all sent messages unless they have already decided (this is why we need the nacks in Phase 3). The loophole here is that processes that

decide stop participating in the protocol; but because any non-faulty process retransmits the decision value in the reliable broadcast, if a process is waiting for a response from a non-faulty process that already terminated, eventually it will get the reliable broadcast instead and terminate itself. In Phase 3, a process might get stuck waiting for a dead coordinator, but the strong completeness of $\Diamond S$ means that it suspects the dead coordinator eventually and escapes. So at worst we do finitely many rounds.

Now suppose that after some time t there is a process c that is never suspected by any process. Then in the next round in which c is the coordinator, in Phase 3 all surviving processes wait for c and respond with ack, c decides on the current estimate, and triggers the reliable broadcast protocol to ensure everybody else decides on the same value. Since reliable broadcast guarantees that everybody receives the message, everybody decides this value *or some value previously broadcast*—but in either case everybody decides.

Agreement is the tricky part. It's possible that two coordinators both initiate a reliable broadcast and some processes choose the value from the first and some the value from the second. But in this case the first coordinator collected acks from a majority of processes in some round r, and all subsequent coordinators collected estimates from an overlapping majority of processes in some round $r' > r$. By applying the same induction argument as for Paxos, we get that all subsequent coordinators choose the same estimate as the first coordinator, and so we get agreement.

13.5 $f < n/2$ is still required even with $\Diamond P$

We can show that with a majority of failures, we're in trouble with just $\Diamond P$ (and thus with $\Diamond S$, which is trivially simulated by $\Diamond P$). The reason is that $\Diamond P$ can lie to us for some long initial interval of the protocol, and consensus is required to terminate eventually despite these lies. So the usual partition argument works: start half of the processes with input 0, half with 1, and run both halves independently with $\Diamond P$ suspecting the other half until the processes in both halves decide on their common inputs. We can now make $\Diamond P$ happy by letting it stop suspecting the processes, but it's too late.

13.6 Relationships among the classes

It's easy to see that P simulates S and $\Diamond P$ simulates $\Diamond S$ without modification. It's also immediate that P simulates $\Diamond P$ and S simulates $\Diamond S$ (make "eventually" be "now"), which gives a diamond-shaped lattice structure between

the classes. What is trickier is to show that this structure doesn't collapse: $\Diamond P$ can't simulate S, S can't simulate $\Diamond P$, and $\Diamond S$ can't simulate any of the other classes.

First let's observe that $\Diamond P$ can't simulate S: if it could, we would get a consensus protocol for $f \geq n/2$ failures, which we can't do. It follows that $\Diamond P$ also can't simulate P (because P can simulate S).

To show that S can't simulate $\Diamond P$, choose some non-faulty victim process v and consider an execution in which S periodically suspects v (which it is allowed to do as long as there is some other non-faulty process it never suspects). If the $\Diamond P$-simulator ever responds to this by refusing to suspect v, there is an execution in which v really is dead, and the simulator violates strong completeness. But if not, we violate eventual strong accuracy. Note that this also implies S can't simulate P, since P can simulate $\Diamond P$. It also shows that $\Diamond S$ can't simulate either of $\Diamond P$ or P.

We are left with showing $\Diamond S$ can't simulate S. Consider a system where p's $\Diamond S$ detector suspects q but not p from the start of the execution. Run p until p's S-simulator gives up and suspects q, which it must do eventually by strong completeness, since this run is indistinguishable from one in which q is faulty. Then wake up q and crash p. Since q is the only non-faulty process, and the alleged S-simulator suspected it, we've violated weak accuracy.

13.7 Terminating reliable broadcast with P

If we look carefully at the arguments so far, we haven't actually shown anything that P is good for: we only know that S and $\Diamond P$ can't simulate P because neither can simulate the other. This raises the obvious question of whether there is something we might actually want to do that requires P.

Chandra and Toueg [CT96] give as an example of a natural problem that can be solved only with P the problem of **terminating reliable broadcast**. In this problem, a leader process ℓ sends a message m, and all processes eventually decide on m or a no-message value $\perp$. Validity in this case says that if ℓ is non-faulty, every non-faulty process decides m. Agreement says that all non-faulty processes must decide the same value (which will be one of m or $\perp$) whether ℓ is faulty or not. Terminating is the usual condition that all processes eventually decide on some value.

This problem is equivalent to having the processes reach consensus on a value that defaults to $\perp$ if no message is received from ℓ. Since P implements S, we can do this using our already-known algorithm for solving consensus for any number of failures using S. The resulting algorithm runs in two

phases:

1. In the first phase, ℓ transmits m to all processes, and each process waits to either receive m (and use m as the input to the next phase) or suspect ℓ (and use $\perp$ as the input to the next phase).

2. In the second phase, use Algorithm 13.2 to reach agreement on m or $\perp$. (We can do this because P is also an instance of S.)

If ℓ is non-faulty, all non-faulty processes start the consensus phase with m and end with m. Whether ℓ is faulty or not, all non-faulty processes end the consensus phase with the same value. So validity and agreement are satisfied.

It's not hard to see that we can't solve terminating reliable broadcast with either S or $\Diamond P$. If we try to solve it using S, the weak accuracy of S means that some non-faulty p is never suspected, but p doesn't have to be ℓ. So if all the processes start off suspecting ℓ, either they wait forever for a faulty ℓ to wake up (violating termination), or they finish the protocol and decide on the wrong value before a non-faulty ℓ wakes up (violating validity). The same argument works for $\Diamond P$: during the initial period of confusion, a non-faulty ℓ might be suspected by all processes, and if we wait to decide until ℓ starts sending messages or becomes non-suspect, we violate termination in the case where ℓ really is faulty.

Chapter 14

Quorum systems

Last updated 2014. Some material may be out of date.

14.1 Basics

In the past few chapters, we've seen many protocols that depend on the fact that if I talk to more than $n/2$ processes and you talk to more than $n/2$ processes, the two groups overlap. This is a special case of a **quorum system**, a family of subsets of the set of processes with the property that any two subsets in the family overlap. By choosing an appropriate family, we may be able to achieve lower load on each system member, higher availability, defense against Byzantine faults, etc.

The exciting thing from a theoretical perspective is that these turn a systems problem into a combinatorial problem: this means we can ask combinatorialists how to solve it.

14.2 Simple quorum systems

- Majority and weighted majorities
- Specialized read/write systems where write quorum is a column and read quorum a row of some grid.
- Dynamic quorum systems: get more than half of the most recent copy.
- Crumbling walls [PW97b, PW97a]: optimal small-quorum system for good choice of wall sizes.

14.3 Goals

- Minimize **quorum size**.
- Minimize **load**, defined as the minimum over all access strategies (probability distributions on quorums) of the maximum over all servers of probability it gets hit.
- Maximize **capacity**, defined as the maximum number of quorum accesses per time unit in the limit if each quorum access ties up a quorum member for 1 time unit (but we are allowed to stagger a quorum access over multiple time units).
- Maximize **fault-tolerance**: minimum number of server failures that blocks all quorums. Note that for standard quorum systems this is directly opposed to minimizing quorum size, since killing the smallest quorum stops us dead.
- Minimize **failure probability** = probability that every quorum contains at least one bad server, assuming each server fails with independent probability.

Naor and Wool [NW98] describe trade-offs between these goals (some of these were previously known, see the paper for citations):

- capacity = 1/load; this is obtained by selecting the quorums independently at random according to the load-minimizing distribution. In particular this means we can forget about capacity and just concentrate on minimizing load.
- load $\geq \max(c/n, 1/c)$ where c is the minimum quorum size. The first case is obvious: if every access hits c nodes, spreading them out as evenly as possible still hits each node c/n of the time. The second is trickier: Naor and Wool prove it using LP duality, but the argument essentially says that if we have some quorum Q of size c, then since every other quorum Q' intersects Q in at least one place, we can show that every Q' adds at least 1 unit of load in total to the c members of Q. So if we pick a random quorum Q', the average load added to all of Q is at least 1, so the average load added to some particular element of Q is at least $1/|Q| = 1/c$. Combining the two cases, we can't hope to get load better than $1/\sqrt{n}$, and to get this load we need quorums of size at least $\sqrt{n}$.

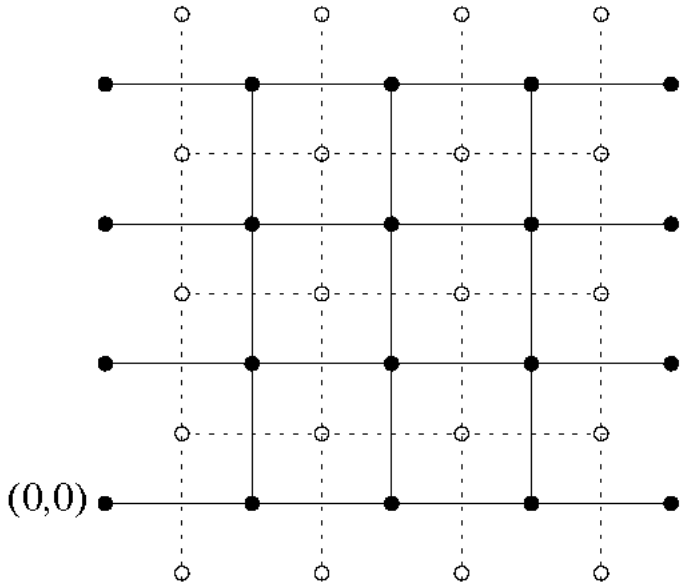


Figure 14.1: Figure 2 from [NW98]. Solid lines are $G(3)$; dashed lines are $G^*(3)$.

- failure probability is at least p when $p > 1/2$ (and optimal system is to just pick a single leader in this case), failure probability can be made exponentially small in size of smallest quorum when $p < 1/2$ (with many quorums). These results are due to Peleg and Wool [PW95].

14.4 Paths system

This is an optimal-load system from Naor and Wool [NW98] with exponentially low failure probability, based on percolation theory.

For this system, we build a $d \times d$ mesh-like graph where a quorum consists of the union of a top-to-bottom path (TB path) and a left-to-right path (LR path); this gives quorum size $O(\sqrt{n})$ and load $O(1/\sqrt{n})$. Note that the TB and LR paths are not necessarily direct: they may wander around for a while in order to get where they are going, especially if there are a lot of failures to avoid. But the smallest quorums will have size $2d + 1 = O(\sqrt{n})$.

The actual mesh is a little more complicated. Figure 14.1 reproduces the picture of the $d = 3$ case from the Naor and Wool paper.

Each server corresponds to a *pair* of intersecting edges, one from the

$G(d)$ grid and one from the $G^*(d)$ grid (the star indicates that $G^*(d)$ is the **dual graph**[1] of $G(d)$. A quorum consists of a set of servers that produce an LR path in $G(d)$ and a TB path in $G^*(d)$. Quorums intersect, because any LR path in $G(d)$ must cross some TB path in $G^*(d)$ at some server (in fact, each pair of quorums intersects in at least two places). The total number of elements n is $(d+1)^2$ and the minimum size of a quorum is $2d+1 = \Theta(\sqrt{n})$.

The symmetry of the mesh gives that there exists a LR path in the mesh if and only if there does not exist a TB path in its **complement**, the graph that has an edge only if the mesh doesn't. For a mesh with failure probability $p < 1/2$, the complement is a mesh with failure probability $q = 1-p > 1/2$. Using results in percolation theory, it can be shown that for failure probability $q > 1/2$, the probability that there exists a left-to-right path is exponentially small in d (formally, for each p there is a constant $\phi(p)$ such that $\Pr[\exists \text{LR path}] \leq \exp(-\phi(p)d)$). We then have

$$\begin{aligned}
\Pr[\exists(\text{live quorum})] &= \Pr[\exists(\text{TB path}) \wedge \exists(\text{LR path})] \\
&= \Pr[\neg\exists(\text{LR path in complement}) \vee \neg\exists(\text{TB path in complement})] \\
&\leq \Pr[\neg\exists(\text{LR path in complement})] + \Pr[\neg\exists(\text{TB path in complement})] \\
&\leq 2\exp(-\phi(1-p)d) \\
&= 2\exp(-\Theta(\sqrt{n})).
\end{aligned}$$

So the failure probability of this system is exponentially small for any fixed $p < 1/2$.

See the paper [NW98] for more details.

14.5 Byzantine quorum systems

Standard quorum systems are great when you only have crash failures, but with Byzantine failures you have to worry about finding a quorum that includes a Byzantine serve who lies about the data. For this purpose you need something stronger. Following Malkhi and Reiter [MR98] and Malkhi *et al.* [MRWW01], one can define:

- A **b-disseminating quorum system** guarantees $|Q_1 \cap Q_2| \geq b+1$ for all quorums Q_1 and Q_2. This guarantees that if I update a quorum Q_1 and you update a quorum Q_2, and there are at most

[1] The dual of a graph G embedded in the plane has a vertex for each region of G, and an edge connecting each pair of vertices corresponding to adjacent regions, where a region is a subset of the plane that is bounded by edges of G.

b Byzantine processes, then there is some non-Byzantine process in both our quorums. Mostly useful if data is "self-verifying," that is, signed with digital signatures that the Byzantine processes can't forge. Otherwise, I can't tell which of the allegedly most recent data values is the right one since the Byzantine processes lie.

- A **b-masking quorum system** guarantees $|Q_1 \cap Q_2| \geq 2b + 1$ for all quorums Q_1 and Q_2. (In other words, it's the same as a $2b$-disseminating quorum system.) This allows me to defeat the Byzantine processes through voting: given $2b + 1$ overlapping servers, if I want the most recent value of the data I take the one with the most recent timestamp that appears on at least $b + 1$ servers, which the Byzantine guys can't fake.

An additional requirement in both cases is that for any set of servers B with $|B| \leq b$, there is some quorum Q such that $Q \cap B = \emptyset$. This prevents the Byzantine processes from stopping the system by simply refusing to participate.

Note: these definitions are based on the assumption that there is some fixed bound on the number of Byzantine processes. Malkhi and Reiter [MR98] give more complicated definitions for the case where one has an arbitrary family $\{\mathcal{B}\}$ of potential Byzantine sets. The definitions above are actually simplified versions from [MRWW01].

The simplest way to build a b-disseminating quorum system is to use supermajorities of size at least $(n + b + 1)/2$; the overlap between any two such supermajorities is at least $(n + b + 1) - n = b + 1$. This gives a load of substantially more than $\frac{1}{2}$. There are better constructions that knock the load down to $\Theta(\sqrt{b/n})$; see [MRWW01].

For more on this topic in general, see the survey by by Merideth and Reiter [MR10].

14.6 Probabilistic quorum systems

The problem with all standard (or **strict**) quorum systems is that we need big quorums to get high fault tolerance, since the adversary can always stop us by knocking out our smallest quorum. A **probabilistic quorum system** or more specifically an **ϵ-intersecting quorum system** [MRWW01] improves the fault-tolerance by relaxing the requirements. For such a system we have not only a set system Q, but also a probability distribution w supplied by

the quorum system designer, with the property that $\Pr[Q_1 \cap Q_2 = \emptyset] \leq \epsilon$ when Q_1 and Q_2 are chosen independently according to their weights.

14.6.1 Example

Let a quorum be any set of size $k\sqrt{n}$ for some k and let all quorums be chosen uniformly at random. Pick some quorum Q_1; what is the probability that a random Q_2 does not intersect Q_1? Imagine we choose the elements of Q_2 one at a time. The chance that the first element x_1 of Q_2 misses Q_1 is exactly $(n - k\sqrt{n})/n = 1 - k/\sqrt{n}$, and conditioning on x_1 through x_{i-1} missing Q_1 the probability that x_i also misses it is $(n - k\sqrt{n} - i + 1)/(n - i + 1) \leq (n - k\sqrt{n})/n = 1 - k/\sqrt{n}$. So taking the product over all i gives $\Pr[\text{all miss } Q_1] \leq (1 - k/\sqrt{n})^{k\sqrt{n}} \leq \exp(-k\sqrt{n})^{k/\sqrt{n})} = \exp(-k^2)$. So by setting $k = \Theta(\ln 1/\epsilon)$, we can get our desired ϵ-intersecting system.

14.6.2 Performance

Failure probabilities, if naively defined, can be made arbitrarily small: add low-probability singleton quorums that are hardly ever picked unless massive failures occur. But the resulting system is still ϵ-intersecting.

One way to look at this is that it points out a flaw in the ϵ-intersecting definition: ϵ-intersecting quorums may cease to be ϵ-intersecting conditioned on a particular failure pattern (e.g., when all the non-singleton quorums are knocked out by massive failures). But Malkhi *et al.* [MRWW01] address the problem in a different way, by considering only survival of **high quality** quorums, where a particular quorum Q is δ**-high-quality** if $\Pr[Q_1 \cap Q_2 = \emptyset | Q_1 = Q] \leq \delta$ and high quality if it's $\sqrt{\epsilon}$-high-quality. It's not hard to show that a random quorum is δ-high-quality with probability at least ϵ/δ, so a high quality quorum is one that fails to intersect a random quorum with probability at most $\sqrt{\epsilon}$ and a high quality quorum is picked with probability at least $1 - \sqrt{\epsilon}$.

We can also consider load; Malkhi *et al.* [MRWW01] show that essentially the same bounds on load for strict quorum systems also hold for ϵ-intersecting quorum systems: $\text{load}(S) \geq \max((\mathrm{E}(|Q|)/n, (1-\sqrt{\epsilon})^2/\mathrm{E}(|Q|))$, where $\mathrm{E}(|Q|)$ is the expected size of a quorum. The left-hand branch of the max is just the average load applied to a uniformly-chosen server. For the right-hand side, pick some high quality quorum Q' with size less than or equal to $(1 - \sqrt{\epsilon})\,\mathrm{E}(|Q|)$ and consider the load applied to its most loaded member by its nonempty intersection (which occurs with probability at least $1 - \sqrt{\epsilon}$) with a random quorum.

14.7 Signed quorum systems

A further generalization of probabilistic quorum systems gives **signed quorum systems** [Yu06]. In these systems, a quorum consists of some set of positive members (servers you reached) and negative members (servers you tried to reach but couldn't). These allow $O(1)$-sized quorums while tolerating $n - O(1)$ failures, under certain natural probabilistic assumptions. Because the quorums are small, the load on some servers may be very high: so these are most useful for fault-tolerance rather than load-balancing. See the paper for more details.

Chapter 15

Blockchains

All of the results we have considered so far for message-passing systems have made a critical assumption: the number of processes n is known and fixed, so we can sensibly talk about things like majorities of processes, fewer than $n/3$ Byzantine faults, and so on. This assumption is not unreasonable for systems operated by a single organization, but it may not make sense for large distributed systems that can in principle be joined by anybody. In this case, to solve a problem like agreement, we need some mechanism other than simply counting machines to produce overlapping quorums or to outvote Byzantine coalitions.

This is particularly tricky because it is possible for a single machine on the Internet to masquerade as many, by using routing trickery to simulate many distinct IP addresses. This is not something we can practically remove from the IP protocol stack, since it's used for positive ends like allowing a single machine to simulate multiple low-use servers or, in the other direction, allowing a warehouse full of machines to simulate a single high-use server. But this possibility allows for a **Sybil attack** [Dou02], where an algorithm naively implemented on the assumption that faulty processes form a small minority is suddenly overwhelmed by a single faulty machine backed by an army of virtual clones. This requires re-examining how (or if) we can achieve consensus in systems that allow arbitrary participants.

The current dominant strategy for doing so is to use cryptographic mechanisms to substitute majorities expressed in terms of unforgeable resources like computing power, storage, or simulated currencies for majorities expressed in mere counts of IP addresses. This is often coupled with a certified replicated-state-machine approach that replaces agreement with weaker various **eventual consistency** guarantees, all of which is encompassed by the

notion of a **blockchain**, which has no universally-accepted formal definition, but which we can think of roughly as a global-scale replicated-state-machine algorithm that allows arbitrary participants and enforces consistency in the long run using a combination of cryptographic tools and social engineering.

The blockchain world is a bit of a moving target, and constructing a practical blockchain that will see wide adoption involves a number of political and social issues that go beyond simply putting together the right technology. But from a distributed-systems perspective, we can look at the systems that people have actually built and try to learn something from them. In this chapter, we'll start by looking at the problem of defending against arbitrary participants in a distributed system, and then look at how the Bitcoin system [Nak08] appears to do so successfully even though it arguably shouldn't.

15.1 Sybil attacks

The idea of the **Sybil attack** is that one bad machine can masquerade as many different machines using routing tricks. This defeats any distributed algorithm based on assuming a fixed fraction of the processes are bad. This is particularly difficult to defend against in the current Internet as most machines are now buried behind Network Address Translation (NAT) mechanisms to allow a single IP to be shared between multiple machines, making it trivial for an army of bogus clones to masquerade as separate machines behind a NAT.

Whatever the source of the bogus clones, they are a problem for any system with open admissions, where any machine on the network can join it. Examples of such systems are the SMTP-based mail system, the HTTP-based World-Wide Web, and many multiplayer games. The openness of these systems makes them inherently vulnerable to malicious actors (spammers for SMTP, various kinds of undesirable users for HTTP, cheaters in games), especially if new identities can be manufactured for free.

The name "Sybil attack" was popularized by a paper by John Douceur [Dou02], in a paper that analyzes several methods for attempting to defeat them. The term itself is credited to Brian Zill in Douceur's paper, and is based on the book (and later movie) *Sybil* [Sch73] about a psychiatric patient diagnosed with multiple-personality disorder.

Note that Sybil attacks do not include attacks using botnets, where the problem is that we really do have 10,000,000 bad nodes overwhelming our system; instead, we are worrying about the case where a bad router can

claim to have 10,000,000 bad nodes behind it but these nodes are simulated by only a small number of machines.

15.1.1 Resource-based defenses

Douceur considers an abstract model involving interactions between **entities**, which may or may not correspond to actual machines. (For consistency with the rest of these notes, we will just call them "processes.") The communication model is a generic broadcast channel (called a "cloud" in Douceur's terminology) that, unlike our usual model, does not record the source of messages. It is assumed that processes are computationally bounded, allowing the use of public-key cryptography. In particular, an process can establish an **identity** by creating a public/secret key pair and signing all of its messages using the secret key.

Non-faulty processes will do this once, establishing a single **legitimate** identity. Faulty processes will attempt to construct as many extra **counterfeit** identities as they can get away with.

Assuming that there is no external agent (like a centralized certificate authority) that prevents them from doing this, so need a mechanism to constrain how many identities a faulty process can construct. One solution is to assume that all processes have limited access to some resource needed to construct identities. Typically this is computational power, allowing for a **proof-of-work** strategy where any new identity is validated by demonstrating that the process using it has burned some minimal amount of computational time.

This approach was first proposed by Dwork and Naor [DN93] as a tool for combating email spam, and is frequently reinvented. The usual approach is to pick a cryptographically-secure hash function h that produces n bits of output, a puzzle input x that should be unique to this instance of the problem, and demand that the process find some value y such that $h(xy) = 0$; if we assume that h acts like a random function, it should require 2^n computations of h on average to find such a y.

Proof-of-work allows for direct validation of identities: if you present me with an identity that incorporates xy with $h(xy) = 0$, I can be reasonably confident that you spent computed approximately 2^n hashes since you learned x. The cost of checking a valid identity is relatively cheap, since I only have to compute one hash (although the cost of checking a bogus identity might be more expensive than generating the bogus identity). Assuming that the each faulty process spends at most ρ times as much processing power than any non-faulty process, Douceur observes that the expected number of counterfeit

identities for each faulty process will average around ρ.

A key assumption here is that the proof-of-work tasks are carried out over a bounded interval. If the faulty processes can prepare identities well in advance, Douceur observes that a faulty process can spend as much time as it likes to construct as many identities as it likes.

(There is a third main result in this paper, which shows that using an initial assignment of identities to validate later identities using some sort of vouching processes just leads the initial army of counterfeits to recruit more counterfeits. This is mostly interesting because it was still used at the time as a way to try to validate identities in PGP [ASZ96], a popular open-source public-key encryption system for email messages.)

15.1.2 Limitations of resource-based defenses

Douceur's paper was interpreted by many researchers as a sign that proof-of-work is fundamentally useless for defending against Sybil attacks, at least in the context of problems like consensus where a constant fraction of faulty agents can disrupt the protocol. The usual argument goes like this:[1]

1. For any instance of a problem to be solved using proof-of-work, non-faulty processes need to burn resources that are a constant fraction of the resources burned by faulty processes.

2. This resource burning needs to exceed the value of whatever target is being defended, or the faulty processes can obtain a net profit by burning enough resources to overwhelm the non-faulty processes.

3. The resource burning by the non-faulty processes needs to be repeated every time the target is defended, because the non-faulty processes only need to get lucky once. In contrast, the faulty processes can wait and burn their resources for only one instance of the protocol.

It follows that the non-faulty processes quickly expend more resources defending the target than the target is worth: proof-of-work can't work.[2]

[1] As far as I can tell, this argument initially arose as a folk theorem. But see [BGM+18] for references to more serious game-theoretic analyses that are similarly pessimistic and some reasons to be less pessimistic.

[2] This did not stop some of us from trying anyway. One of the earliest written examples of attempting to use proof-of-work to solve Byzantine agreement despite Sybil attacks is a Yale CS tech report derived from Collin Jackson's CPSC 490 senior project [AJK05]. Sadly the two co-authors who advised him on this project didn't think it was worth trying to publish this obviously silly idea anywhere more visible.

The description above is a little vague about what it means to protect a target. As a concrete example, suppose I am purchasing some real-world good from you using a transaction that is recorded in a **distributed ledger**, a replicated state machine that records payments. If I can subvert the consensus protocol used to update the distributed ledger, I might be able to show you a ledger that includes a payment from me to you (causing you to turn over the valuable good), but then convince all other participants to adopt a different update in which this transaction never happened, and explain that you are simply a Byzantine process trying to steal my money.[3]

15.1.3 Alternative defenses

CAPTCHAs [vABHL03] have been used in the context of web sites interacting with human users, by requiring the users to complete tasks that are hard to automate. This raises the cost of a fake identity by a bit, since a human being needs to be involved in the process somewhere, but it's still fundamentally a resource-burning technique, just now the resource is human time instead of computer time. As with proof-of work, the problem is that non-faulty users are required to spend the same effort as faulty users, and this adds up fast via the folk theorem. This becomes particularly annoying when attackers can arbitrage low wage rates in some countries or even apply man-in-the-middle attacks that get would-be visitors to one site to solve CAPTCHAs for another site.

Some other approaches that have been proposed use physical locations or social networks to attempt to detect counterfeit identities generated by the same process. Bazzi and Konjevod [BK07] proposed that a process that wants to authenticate itself could a **geometric certificate** consisting of verified ping times to a collection of standardized beacon nodes. Multiple virtual machines located at the same physical location will end up with essentially the same certificate, and can be treated as one (possibly corrupted) node. Unfortunatley, so will multiple users at large institutions with a single outgoing pipe to the rest of the network. The idea does avoid resource-burning, but it never really caught on on practice, and if tried now could probably be easily defeated by geographically-distributed botnets.

SybilGuard [YKGF06] was proposed by Yu *et al.* as a defense against

[3] A reasonable objection is that if you demand that I sign my transactions using a private key, I won't be able to repudiate my payment even if you only have a private copy. In this case what I do is show you a version of the ledger where I have plenty of funds to pay you, and then show everybody else a version where my payment to you sadly does not go through because I already gave all my money to my suspiciously identical twin.

Sybil attacks based on the structure of social networks. The idea is that a social network graph with many Sybil nodes is likely to decompose into a subnetwork consisting mostly of legitimate nodes and a subnetwork consisting mostly of counterfeit nodes, with the majority of links between nodes within each subnetwork and few links between legitimate nodes and counterfeit nodes. This approach is pretty clever, and subsequent work explored in depth efficient algorithms for separating these two subnetworks, but it causes trouble for users that wish to disconnect their activities from their social-network identity, and more practically is trivially defeated if the faulty processes can amass enough bogus social network accounts that they are not longer an obvious disconnected minority.

15.2 Bitcoin

Since proof-of-work is too expensive, and other approaches are easily defeated, what do we do if we really want to solve consensus in an open system? It turns out we bite the bullet and accept the huge cost of proof-of-work. This was the approach taken by the pseudonymous person or persons Satoshi Nakamoto in Bitcoin [Nak08]. This system evades some of the issues in the folk theorem by (a) convincing lots of non-faulty processes to join by including a lottery awarding tokens to participants and (b) relying on the would-be faulty processes not to be coordinated enough or have enough available processing power relative to the huge horde of non-faulty lottery-ticket buyers to target a specific round of the protocol.

Bitcoin is an implementation of a **cryptocurrency**, a mechanism for exchanging cryptographic tokens between users that can be used analogously to standard currencies. To make all transfers visible thus prevent **double-spending**, it implements what is now usually called a **distributed ledger** consisting of a **chain** (sequence) of **blocks**, each of which contains a set of transactions that record transfers of tokens between participants. Participants are identified by cryptographic keys, and a transaction must be signed by the sender of the tokens to be valid.

A cryptographic hash of the entire ledger is updated with the addition of each block, to prevent tampering and to construct the key for the proof-of-work puzzle used to select the next block to be added. This technique, which gave rise to the name **blockchain** for systems of this kind, was originally developed by Haber and Stornetta [HS91], without the proof-of-work consensus algorithm, as a tool for making it difficult to backdate digital documents by storing their hashes in a centrally-maintained sequence of

signed blocks of this type whose full hash is published from time to time in a difficult-to-corrupt location. (Haber and Stornetta's company Surety uses a weekly classified advertisement in the New York Times.)

Bitcoin takes this idea and adds a proof-of-work based consensus protocol on top, while including side payments to reward participation in the protocol. The rule for the consensus protocol is that every interested process tries to extend the current chain as best it can, but only a process that provably solves a cryptographic puzzle can do so. So the first process to solve the puzzle wins, and if the majority of the computation power belongs to non-faulty processes, this process is likely to be non-faulty. In the case of a tie (possibly created by faulty processes that refuse to admit defeat), longer chain wins. In this way the computationally-strong majority eventually overcomes the computationally-weak minority, since even if the minority gets lucky a few times they are unlikely to win the race against the more powerful faction.

To analyze this, let's assume a synchronous message-passing system where messages are distributed through an anonymous broadcast channel. Synchrony is obtained by assuming roughly-synchronized clocks and setting a very long timeout of 10 minutes for each round. Because the identities of processes are not relevant to the protocol, there is no need to identify the sender of a message, although the proof-of-work mechanism used to select blocks also has the useful side effect of limiting propagation of spam updates.

In distributed computing terms, Bitcoin implements a replicated state machine, using a probabilistic version of consensus to choose between possible extensions. Using randomization evades the Dolev-Strong [DS83] and FLP [FLP85] lower bounds, because the bad executions constructed in these bounds are either (a) highly improbable or (b) require the adversary to predict the future (we'll come back to this idea in Chapter 24). The Nakamoto paper does not reference the distributed computing literature, and its definition of consensus deviates substantially from the traditional termination-validity-agreement framework of Pease *et al.* [PSL80]. Instead of guaranteeing termination and validity, the protocol attempts to provide an **eventual consistency** where over time, the copies of the state machine continuously converge to agreeing on an initial prefix of the operation history that includes all but a few recently-added blocks.

15.2.1 Obtaining eventual consistency

In this section, we'll describe the operation of the Bitcoin consensus protocol, often called **Nakamoto consensus**. There is a somewhat heuristic analysis of this protocol in the original Bitcoin whitepaper. We'll give a slightly less

heuristic analysis that is still pretty sloppy. For a more serious analysis, see [GKL15], which influenced some of the less suspicious parts of the discussion below.

Our model is already strong enough to trivially guarantee agreement in each round: since every non-faulty process sees the same chains in the broadcast channel, it's enough to discard any invalid chains (which we will define soon), and apply some consistent tie-breaking rule to choose among the remaining valid chains. So the goal of the consensus step will be to guarantee **eventual consistency** between rounds, which we will take to mean that any block buried deep enough in the chain C_r for round r also appears in any chain $C_{r'}$ for $r' > r$.

The mechanism for doing this is to generate each C_{r+1} as an extension of C_r. To construct an extension, a process i that wishes to add block x_i must first solve a hash puzzle by finding some y such that $h(C_r, x_i, y) \leq D$, where h is a hash function that is sufficiently cryptographically secure that we can pretend it's a random function, and D is a difficulty parameter that can be tuned to adjust the likelihood of finding a solution within the time bounds associated with the round. If successful, the process can propose an extension $C_r \langle x_i, y \rangle$ that is valid if it satisfies both application-specific requirements like x_i doesn't include transactions that spend money the spender doesn't have after C_r, and protocol-specific requirements like C_r is valid and $h(C_r, x_i, y) \leq D$. These conditions are easily checked by any process.

For the tie-breaking rule, we will favor longer chains over shorter ones, and otherwise break ties consistently. As noted previously, consistent tie-breaking means all non-faulty processes adopt the same value C_r for each r. To replace a buried block, the faulty processes will need to supply an alternative chain that wins the tie-breaking rule by being the same length or longer as the chain built by the non-faulty processes.

The resulting protocol is given in Algorithm 15.1.

The main issue with this protocol is that if the faulty processes get lucky, they can construct a chain that is longer than the chain of the non-faulty processes, and use this to hijack the protocol. We'd like to show that when this happens, the bad chain shares all but a small suffix with the good chain it displaces. If we are willing to cut a few corners in the argument, this comes down to demonstrating that the faulty processes can't win the race to extend their evil chain past the non-faulty processes' preferred chain over long sequences of rounds. We will consider the specific case where the non-faulty and faulty processes both start off with some common $C_r = \check{C}_r$, and over the next m rounds the non-faulty processes extend C_r as best they can using

```
1  C ← some initial chain.
   // Do infinitely many synchronous rounds
2  for r ← 0 . . . ∞ do
3      Let x be the block I want to add to C
       // Attempt to extend C
4      for i ← 1 . . . q do
5          Choose a random value y
6          if h(C, x, y) ≤ D then
7              C ← C ⟨x, y⟩
8              Broadcast C
9              break
       // Take best valid C′
10     for each C′ received this round do
11         if C′ is valid and tie-breaker favors C′ over C then
12             C ← C′
```

Algorithm 15.1: Nakamoto consensus

Algorithm 15.1 while the faulty processes extend $\check{C}_r$ in secret. The faulty processes win if the resulting $\check{C}_{r+m}$ is longer than the non-faulty processes' C_{r+m}. (There is a lot of unjustified simplification sneaking in here. For a much more sophisticated argument that doesn't cheat, see [GKL15].)

For each process i, let p_i be the expected number of puzzle solutions it finds in a single round. If i is non-faulty, this is just the probability that it finds a solution, since non-faulty processes stop after finding one solution. If i is faulty, i can generate more than one solution, which might make p_i a bit larger than it would be for a non-faulty process with the same computational power. If p_i is very small in either case the difference will be slight.

To simplify things, we'll assume that the set of processes and their p_i values are fixed over time. Let α be the sum of p_i over all the non-faulty processes, and β the sum of p_i over all the faulty processes. These give the expected number of solutions obtained in one round by all non-faulty or faulty processes respectively.

Inclusion-exclusion says that the probability that the non-faulty processes solve at least one puzzle in a given round is at least $\sum_i p_i - \sum_{i \neq j} p_i p_j \geq \alpha - \alpha^2$. Letting X_i be the indicator for the event that the non-faulty processes add a new block in round $r+i$, they add at least an expected $\sum \mathrm{E}[X_i] \geq m(\alpha - \alpha^2)$ blocks in m rounds. We can similarly argue that the faulty processes add

at most an expected $m\beta = \sum \mathrm{E}[Y_i]$ blocks in m rounds, where Y_i is the indicator variable for success of the i-th puzzle attempt by a non-faulty process. In both cases we are looking at a sum of 0–1 random variable with known mean, so Chernoff bounds apply and we get, for any δ,

$$\Pr\left[\sum X_i \leq (1-\delta)m(\alpha-\alpha^2)\right] \leq e^{-\delta^2 m(\alpha-\alpha^2)/2}$$
$$\Pr\left[\sum Y_i \geq (1+\delta)m\beta\right] \leq e^{-\delta^2 m\beta/2}$$

Let's suppose β is less than half of $\alpha - \alpha^2$, corresponding to the faulty processes having a bit less than a third of the total computational power. Writing $k = m(\alpha-\alpha^2) = \mathrm{E}[\sum X_i]$ we get $\mathrm{E}[\sum Y_i] = m\beta \leq k/2$. Set $\delta = 1/3$ to get

$$\Pr\left[\sum X_i \leq (1-\delta)k = (2/3)k\right] \leq e^{-k/18}$$
$$\Pr\left[\sum Y_i \geq (1+\delta)(k/2) = (2/3)k\right] \leq e^{-k/36}.$$

This gives a total probability of at most $e^{-k/18} + e^{-k/36} = e^{-\Theta(k)}$ that either the bad chain gets extended by $(2/3)k$ or more blocks the good chain gets extended by $(2/3)k$ or fewer blocks. If neither of these events happen, the good chain wins. This means that as we consider longer and longer suffixes, it becomes exponentially more improbable that the suffix in the chain the non-faulty processes currently agree on will suddenly be replaced by an alternative chain prepared in secret.

This is not as good a guarantee as we get from iterating traditional Byzantine agreement, where the output of the protocol at each step will never be retracted, but it seems to be good enough in practice that users are willing to tolerate it.

15.2.2 Does Bitcoin disprove the folk theorem?

The short answer is no, and a proof can be found in a paper by Leshno *et al.* [LPS23] (which also gives an alternative open distributed ledger construction that is less vulnerable). And yet Bitcoin is still in use as of 2023.

I'm not really qualified to answer why Bitcoin seems to work anyway, but I suspect that some of its survival is a result of it being uniquely huge. This has consequences that don't apply to a smaller system:

1. The amount of work needed for a sustained attack on Bitcoin is enormous. Given that most of the proof-of-work puzzles in the Bitcoin

as currently implemented are solved using custom parallel hardware running off of low-cost power sources, the likelihood of any attacker (other than a few large state actors) amassing comparable hardware in secret is low.

2. While the volume of transactions on the Bitcoin blockchain increases the potential rewards of a successful attack, their diversity makes the chances of collecting that reward low. It's much easier to imagine convincing a single participant of a low-volume blockchain to trade their valuable cartoon ape for a handful of virtual fairy gold that turns into virtual dirt by dawn. It is probably much harder to do this to every participant in a high-volume chain over a long enough interval to make a sufficient profit.

3. Though Bitcoin was designed to be decentralized, in practice economies of scale mean that most of the protocol is run by a small number of organizations. A profitable attack on Bitcoin might lead these organization to simply roll back and fork the chain, erasing the attacker's gains. (A similar rollback happened after a 2016 attack on Etherium.) So the political and social factors surrounding successful blockchains aren't taken into account in the abstract model underlying the folk theorem.

At the same time, Bitcoin is still absurdly costly, and the guarantees it provides are not as strong as can be obtained by running iterated Byzantine agreement on a small number of semi-trusted parties. This may be why more recent systems have been moving away from proof-of-work, and suggests that Bitcoin's unusual status as the first widely-used blockchain may, in the long run, not save it from being outcompeted by better systems.

Perhaps the way to think about the enormous cost of proof-of-work based systems is that they are paying a **price of anarchy** [KP09] for avoiding any kind of centralized management in the form of a privileged set of servers. Unfortunately, much of this cost appears to be unavoidable without such management [PS18].

Part II

Shared memory

Chapter 16

Model

Shared memory models describe the behavior of processes in a multiprocessing system. These processes might correspond to actual physical processors or processor cores, or might use time-sharing on a single physical processor. In either case the assumption is that communication is through some sort of shared data structures.

Here we describe the basic shared-memory model. See also [AW04, §4.1].

Where shared memory differs from message passing is that processes can't communicate with each other directly, but instead communicate through a pool of shared **objects**. These are typically **registers** supporting read and write operations, but fancier objects corresponding to more sophisticated data structures or synchronization primitives may also be included in the model.

It is usually assumed that the shared objects do not experience faults. This means that the shared memory can be used as a tool to prevent partitions and other problems that can arise in message passing if the number of faults get too high. As a result, for large numbers of processor failures, shared memory is a more powerful model than message passing, although we will see in Chapter 17 that both models can simulate each other provided a majority of processes are non-faulty.

16.1 Atomic registers

An **atomic register** supports read and write operations. We think of these as happening instantaneously, and think of operations of different processes as interleaved in some sequence. Each read operation on a particular register returns the value written by the last previous write operation. Write

operations return nothing.

A process is defined by giving, for each state, the operation that it would like to do next, together with a transition function that specifies how the state will be updated in response to the return value of that operation. A configuration of the system consists of a vector of states for the processes and a vector of value for the registers. A sequential execution consists of a sequence of alternating configurations and operations $C_0, \pi_1, C_1, \pi_2, C_2 \ldots$, where in each triple C_i, π_{i+1}, C_{i+1}, the configuration C_{i+1} is the result of applying π_{i+1} to configuration C_i. For read operations, this means that the state of the reading process is updated according to its transition function. For write operations, the state of the writing process is updated, and the state of the written register is also updated.

Pseudocode for shared-memory protocols is usually written using standard pseudocode conventions, with the register operations appearing either as explicit subroutine calls or implicitly as references to shared variables. Sometimes this can lead to ambiguity; for example, in the code fragment

$$\mathsf{done} \leftarrow \mathsf{leftDone} \wedge \mathsf{rightDone},$$

it is clear that the operation `write`(done, −) happens after `read`(leftDone) and `read`(rightDone), but it is not clear which of `read`(leftDone and `read`(rightDone) happens first. When the order is important, we'll write the sequence out explicitly:

```
1 leftIsDone ← read(leftDone)
2 rightIsDone ← read(rightDone)
3 write(done, leftIsDone ∧ rightIsDone)
```

Here leftIsDone and rightIsDone are internal variables of the process, so using them does not require read or write operations to the shared memory.

16.2 Single-writer versus multi-writer registers

One variation that does come up even with atomic registers is what processes are allowed to read or write a particular register. A typical assumption is that registers are **single-writer multi-reader**—there is only one process that can write to the register (which simplifies implementation since we don't have to arbitrate which of two near-simultaneous writes gets in last and thus leaves the long-term value), although it's also common to assume **multi-writer**

multi-reader registers, which if not otherwise available can be built from single-writer multi-reader registers using atomic snapshot (see Chapter 20). Less common are **single-writer single-reader** registers, which act much like message-passing channels except that the receiver has to make an explicit effort to pick up its mail.

16.3 Fairness and crashes

From the perspective of a schedule, the fairness condition says that every processes gets to perform an operation infinitely often, unless it enters either a crashed or halting state where it invokes no further operations. (Note that unlike in asynchronous message-passing, there is no way to wake up a process once it stops doing operations, since the only way to detect that any activity is happening is to read a register and notice it changed.) Because the registers (at least in in multi-reader models) provide a permanent fault-free record of past history, shared-memory systems are much less vulnerable to crash failures than message-passing systems (though a version FLP[1] still applies [LAA87]); so in extreme cases, we may assume as many as $n - 1$ crash failures, which makes the fairness condition very weak. The $n - 1$ crash failures case is called the **wait-free** case—since no process can wait for any other process to do anything—and has been extensively studied in the literature.

For historical reasons, work on shared-memory systems has tended to assume crash failures rather than Byzantine failures—possibly because Byzantine failures are easier to prevent when you have several processes sitting in the same machine than when they are spread across the network, or possibly because in multi-writer situations a Byzantine process can do much more damage. But the model by itself doesn't put any constraints on the kinds of process failures that might occur.

16.4 Concurrent executions

Often, the operations on our shared objects will be implemented using lower-level operations. When this happens, it no longer makes sense to assume that the high-level operations occur one at a time—although an implementation may try to give that impression to its users. To model to possibility of concurrency between operations, we split an operation into an **invocation**

[1] See Chapter 11.

and **response**, corresponding roughly to a procedure call and its return. The user is responsible for invoking the object; the object's implementation (or the shared memory system, if the object is taken as a primitive) is responsible for responding. Typically we will imagine that an operation is invoked at the moment it becomes pending, but there may be executions in which that does not occur. The time between the invocation and the response for an operation is the **interval** of the operation.

A **concurrent execution** is a sequence of invocations and responses, where after any prefix of the execution, every response corresponds to some preceding invocation, and there is at most one invocation for each process—always the last—that does not have a corresponding response. A concurrent execution is **complete** if every invocation has a matching response, and it is **sequential** if the operations don't overlap, meaning that there is at most one invocation without a corresponding response in any prefix of the execution.

Sequential executions correspond to executions of a sequential object, which doesn't allow (or at least doesn't experience) concurrent operations. How a given concurrent execution may or may not relate to a sequential execution depends on the consistency properties of the implementation, as described below.

16.5 Consistency properties

Different shared-memory systems may provide various **consistency properties**, which describe how views of an object by different processes mesh with each other. The strongest consistency property generally used is **linearizability** [HW90], which says roughly that an implementation of an object is **linearizable** if, for any complete concurrent execution of the object, there is a sequential execution of the object with the same operations and return values, where the (total) order of operations in the sequential execution is a linearization of the (partial) order of operations in the concurrent execution. The order in each case is defined as $a <_H b$ if the response event for operation a in execution H precedes the invoke event for operation b in the same execution.

The actual definition is a little bit more technical, since it has to deal with the issue of concurrent executions that may include incomplete operations for which there is an invoke event but no response. We'd like to give the implementation the flexibility of deciding whether these operations have taken effect or not, so given an incomplete concurrent execution H, a **linearization**

of H involves three steps:

1. Extend H by adding zero or more response events, obtaining a new execution H'.

2. Remove any invoke events in H' that don't have a matching response event, obtaining a new execution H''.

3. Construct a sequential S such that S meets the sequential specification of the object, $H''|p = S|p$ for all p, and $<_H \subseteq <_S$.

An execution is now linearizable if it has a linearization as defined above.

Most of the complexity of the above definition is needed only to be able to decide if incomplete executions are linearizable. If we consider only complete executions, we can skip the H' and H'' steps, since neither changes H. Even better, if we are asking if an implementation of an object is linearizable—meaning that all executions of the object are linearizable—then we can usually prove this by proving it only for complete executions, since if the implementation has the property that any operation in progress can eventually finish, we can extend any incomplete H to a complete $H' = H''$ by simply running any pending operations to completion. (If our implementation does not have this property, we will need to use the more general definition, but this may be the least of our problems.)

Linearization is usually proved for complete H by constructing the total order $<_S$ explicitly, which gives S as the unique sequential execution equivalent to H that assigns this order to operations. An alternative method is to assign each operation a **linearization point** somewhere between when its invocation and response, and obtain S by assuming that all operations occur atomically at their linearization points is consistent with the specification of the object; this is equivalent to constructing $<_S \supseteq <_H$ since given $<_S$ we can always find consistent linearization points. I personally find constructing a linearization ordering easier for most implementations, but linearization points are useful because they emphasize that to the user, it really does look like a linearizable implementation executes all operations atomically. Using either definition, we are given a fair bit of flexibility in how to order overlapping operations, which can sometimes be exploited by clever implementations (or lower bounds).

A weaker condition is **sequential consistency** [Lam79]. This says that for any concurrent execution of the object, there exists some sequential execution that is indistinguishable to all processes; however, this sequential execution might include operations that occur out of order from a global

perspective. (Essentially we are dropping the requirement $<_H \subseteq <_S$ from the linearizability definition.) For example, we could have an execution of an atomic register where you write to it, then I read from it, but I get the initial value that precedes your write. This is sequentially consistent but not linearizable.

Linearizability has the useful property of being composable, in the sense that if $H|A$ is linearizable for any particular object A, then H is linearizable. Sequential consistency does not generally have this property. For this reason, we will usually ask any implementations we consider to be linearizable. However, both linearizability and sequential consistency are much stronger than the consistency conditions provided by real multiprocessors. For some examples of weaker memory consistency rules, a good place to start might be the dissertation of Jalal Y. Kawash [Kaw00].

16.6 Complexity measures

There are several complexity measures for shared-memory systems.

Time Assume that no process takes more than 1 time unit between operations (but some fast processes may take less). Assign the first operation in the schedule time 1 and each subsequent operation the largest time consistent with the bound. The time of the last operation is the **time complexity**. This is also known as the **big-step** or **round** measure because the time increases by 1 precisely when every non-faulty process has taken at least one step, and a minimum interval during which this occurs counts as a big step or a round.

Total work The **total work** or **total step complexity** is just the length of the schedule, i.e., the number of operations. This doesn't consider how the work is divided among the processes, e.g., an $O(n^2)$ total work protocol might dump all $O(n^2)$ operations on a single process and leave the rest with almost nothing to do. There is usually not much of a direct correspondence between total work and time. For example, any algorithm that involves **busy-waiting**—where a process repeatedly reads a register until it changes—may have unbounded total work (because the busy-waiter might spin very fast) even though it runs in bounded time (because the register gets written to as soon as some slower process gets around to it). However, it is trivially the case that the time complexity is never greater than the total work.

Per-process work The **per-process work**, **individual work**, **per-process step complexity**, or **individual step complexity** measures the maximum number of operations performed by any single process. Optimizing for per-process work produces more equitably distributed workloads (or reveals inequitably distributed workloads). Like total work, per-process work gives an upper bound on time, since each time unit includes at least one operation from the longest-running process, but time complexity might be much less than per-process work (e.g., in the busy-waiting case above).

Remote memory references As we've seen, step complexity doesn't make much sense for processes that busy-wait. An alternative measure is **remote memory reference** complexity or **RMR** complexity. This measure charges one unit for write operations and the first read operation by each process following a write, but charges nothing for subsequent read operations if there are no intervening writes (see §18.6 for details). In this measure, a busy-waiting operation is only charged one unit. RMR complexity can be justified to a certain extent by the cost structure of multi-processor caching [MCS91, And90].

Contention In multi-writer or multi-reader situations, it may be bad to have too many processes pounding on the same register at once. The **contention** measures the maximum number of pending operations on any single register during the schedule (this is the simplest of several definitions out there). A single-reader single-writer algorithm always has contention at most 2, but achieving such low contention may be harder for multi-reader multi-writer algorithms. Of course, the contention is never worse that n, since we assume each process has at most one pending operation at a time.

Space Just how big are those registers anyway? Much of the work in this area assumes they are *very* big.[2] But we can ask for the maximum number of bits in any one register (**width**) or the total size (**bit complexity**) or number (**space complexity**) of all registers, and will try to minimize these quantities when possible. We can also look at the size of the internal states of the processes for another measure of space complexity.

[2] A typical justification for this assumption is that an arbitrarily-large register can be simulated by a smaller register that holds pointers to single-use collections of registers holding the actual values. But even using this technique there are problems for which individual registers of unbounded size are necessary [DFF+23].

16.7 Fancier registers

In addition to stock read-write registers, one can also imagine more tricked-out registers that provide additional operations. These usually go by the name of **read-modify-write** (**RMW**) registers, since the additional operations consist of reading the state, applying some function to it, and writing the state back, all as a single atomic action. Examples of RMW registers that have appeared in real machines at various times in the past include:

Test-and-set bits A **test-and-set** operation sets the bit to 1 and returns the old value.

Fetch-and-add registers A **fetch-and-add** operation adds some increment (typically -1 or 1) to the register and returns the old value.

Compare-and-swap registers A **compare-and-swap** operation writes a new value only if the previous value is equal to a supplied test value.

These are all designed to solve various forms of **mutual exclusion** or locking, where we want at most one process at a time to work on some shared data structure.

Some more exotic read-modify-write registers that have appeared in the literature are

Fetch-and-cons Here the contents of the register is a linked list; a **fetch-and-cons** adds a new head and returns the old list.

Sticky bits (or sticky registers) With a **sticky bit** or **sticky register** [Plo89], once the initial empty value is overwritten, all further writes fail. The writer is not notified that the write fails, but may be able to detect this fact by reading the register in a subsequent operation.

Bank accounts Replace the write operation with `deposit`, which adds a non-negative amount to the state, and `withdraw`, which subtracts a non-negative amount from the state provided the result would not go below 0; otherwise, it has no effect.

These solve problems that are hard for ordinary read/write registers under bad conditions. Note that they all have to return something in response to an invocation.

There are also blocking objects like locks or semaphores, but these don't fit into the RMW framework.

We can also consider generic read-modify-write registers that can compute arbitrary functions (passed as an argument to the read-modify-write operation) in the modify step. Here we typically assume that the read-modify-write operation returns the old value of the register. Generic read-modify-write registers are not commonly found in hardware but can be easily simulated (in the absence of failures) using mutual exclusion.[3]

[3] See Chapter 18.

Chapter 17

Distributed shared memory

In **distributed shared memory**, our goal is to simulate a collection of memory locations or **registers**, each of which supports a **read** operation that returns the current state of the register and a **write** operation that updates the state. Our implementation should be **linearizable** [HW90], meaning that read and write operations appear to occur instantaneously (**atomically**) at some point in between when the operation starts and the operation finishes; equivalently, there should be some way to order all the operations on the registers to obtain a **sequential execution** consistent with the behavior of a real register (each read returns the value of the most recent write) while preserving the observable partial order on operations (where π_1 precedes π_2 if π_1 finishes before π_2 starts). Implicit in this definition is the assumption that implemented operations take place over some interval, between an **invocation** that starts the operation and a **response** that ends the operation and returns its value.[1]

In the absence of process failures, we can just assign each register to some process, and implement both read and write operations by remote procedure calls to the process (in fact, this works for arbitrary shared-memory objects). With process failures, we need to make enough copies of the register that failures can't destroy all of them. This creates an asymmetry between simulations of message-passing from shared-memory and vice versa; in the former case (discussed briefly in §17.1 below), a process that fails in the underlying shared-memory system only means that the same process fails in the simulated message-passing system. But in the other direction, not only does the failure of a process in the underlying message-passing system mean that the same process fails in the simulated shared-memory system, but the

[1] More details on the shared-memory model are given in Chapter 16.

simulation collapses completely if a majority of processes fail.

17.1 Message passing from shared memory

We'll start with the easy direction. We can build a reliable FIFO channel from single-writer single-reader registers using polling. The naive approach is that for each edge uv in the message-passing system, we create a (very big) register r_{uv}, and u writes the entire sequence of every message it has ever sent to v to r_{uv} every time it wants to do a new send. To receive messages, v polls all of its incoming registers periodically and delivers any messages in the histories that it hasn't processed yet.[2]

The ludicrous register width can be reduced by adding in an acknowledgment mechanism in a separate register ack_{vu}; the idea is that u will only write one message at a time to r_{uv}, and will queue subsequent messages until v writes in ack_{vu} that the message in r_{uv} has been received. With some tinkering, it is possible to knock r_{uv} down to only three possible states (sending 0, sending 1, and reset) and ack_{vu} down to a single bit (value-received, reset-received), but that's probably overkill for most applications.

Process failures don't affect any of these protocols, except that a dead process stops sending and receiving.

17.2 Shared memory from message passing: the Attiya-Bar-Noy-Dolev algorithm

Here we show how to implement shared memory from message passing. We'll assume that our system is asynchronous, that the network is complete, and that we are only dealing with $f < n/2$ crash failures. We'll also assume we only want to build single-writer registers, just to keep things simple; we can extend to multi-writer registers later.

Here's the algorithm, which is due to Attiya, Bar-Noy, and Dolev [ABND95]; see also [Lyn96, §17.1.3]. (Section 9.3 of [AW04] gives an equivalent algorithm, but the details are buried in an implementation of totally-ordered broadcast). We'll make n copies of the register, one on each process. Each process's copy will hold a pair $(\mathsf{value}, \mathsf{timestamp})$ where timestamps are (unbounded) integer values. Initially, everybody starts with $(\perp, 0)$. A process updates its copy

[2] If we are really cheap about using registers, and are willing to accept even more absurdity in the register size, we can just have u write every message it ever sends to r_u, and have each v poll all the r_u and filter out any messages intended for other processes.

with new values (v, t) upon receiving $\texttt{write}(v, t)$ from any other process p, provided t is greater than the process's current timestamp. It then responds to p with $\mathsf{ack}(v, t)$, whether or not it updated its local copy. A process will also respond to a message $\texttt{read}(u)$ with a response $\mathsf{ack}(\mathsf{value}, \mathsf{timestamp}, u)$; here u is a **nonce**[3] used to distinguish between different read operations so that a process can't be confused by out-of-date acknowledgments.

To write a value, the writer increments its timestamp, updates its value and sends $\texttt{write}(\mathsf{value}, \mathsf{timestamp})$ to all other processes. The write operation terminates when the writer has received acknowledgments containing the new timestamp value from a majority of processes.

To read a value, a reader does two steps:

1. It sends $\texttt{read}(u)$ to all processes (where u is any value it hasn't used before) and waits to receive acknowledgments from a majority of the processes. It takes the value v associated with the maximum timestamp t as its return value (no matter how many processes sent it).

2. It then sends $\texttt{write}(v, t)$ to all processes, and waits for response $\mathsf{ack}(v, t)$ from a majority of the processes. Only then does it return.

(Any extra messages, messages with the wrong nonce, etc., are discarded.)

Both reads and writes cost $\Theta(n)$ messages ($\Theta(1)$ per process).

Intuition: Nobody can return from a write or a read until they are sure that subsequent reads will return the same (or a later) value. A process can only be sure of this if it knows that the values collected by a read will include at least one copy of the value written or read. But since majorities overlap, if a majority of the processes have a current copy of v, then the majority read quorum will include it. Sending $\texttt{write}(v, t)$ to all processes and waiting for acknowledgments from a majority is just a way of ensuring that a majority do in fact have timestamps that are at least t.

If we omit the `write` stage of a `read` operation, we may violate linearizability. An example would be a situation where two values (1 and 2, say), have been written to exactly one process each, with the rest still holding the initial value $\perp$. A reader that observes 1 and $(n-1)/2$ copies of $\perp$ will return 1, while a reader that observes 2 and $(n-1)/2$ copies of $\perp$ will return 2. In the absence of the `write` stage, we could have an arbitrarily long sequence of readers return 1, 2, 1, 2, ..., all with no concurrency. This would not be

[3] A **nonce** is any value that is guaranteed to be used at most once (the term originally comes from cryptography, which in turn got it from linguistics). In practice, a reader will most likely generate a nonce by combining its process ID with a local timestamp.

consistent with any sequential execution in which 1 and 2 are only written once.

17.3 Proof of linearizability

Our intuition may be strong, but we still need a proof the algorithm works. In particular, we want to show that for any trace T of the ABD protocol, there is an trace of an atomic register object that gives the same sequence of invoke and response events. The usual way to do this is to find a **linearization** of the read and write operations: a total order that extends the observed order in T where $\pi_1 < \pi_2$ in T if and only if π_1 ends before π_2 starts. Sometimes it's hard to construct such an order, but in this case it's easy: we can just use the timestamps associated with the values written or read in each operation. Specifically, we define the timestamp of a write or read operation as the timestamp used in the $\texttt{write}(v, t)$ messages sent out during the implementation of that operation, and we put π_1 before π_2 if:

1. π_1 has a lower timestamp than π_2, or
2. π_1 has the same timestamp as π_2, π_1 is a write, and π_2 is a read, or
3. π_1 has the same timestamp as π_2 and $\pi_1 <_T \pi_2$, or
4. none of the other cases applies, and we feel like putting π_1 first.

The intent is that we pick some total ordering that is consistent with both $<_T$ and the timestamp ordering (with writes before reads when timestamps are equal). To make this work we have to show (a) that these two orderings are in fact consistent, and (b) that the resulting ordering produces values consistent with an atomic register: in particular, that each read returns the value of the last preceding write.

Part (b) is easy: since timestamps only increase in response to writes, each write is followed by precisely those reads with the same timestamp, which are precisely those that returned the value written.

For part (a), suppose that $\pi_1 <_T \pi_2$. The first case is when π_2 is a read. Then before the end of π_1, a set S of more than $n/2$ processes send the π_1 process an $\mathsf{ack}(v1, t_1)$ message. Since local timestamps only increase, from this point on any $\mathsf{ack}(v_2, t_2, u)$ message sent by a process in S has $t_2 \geq t_1$. Let S' be the set of processes sending $\mathsf{ack}(v_2, t_2, u)$ messages processed by π_2. Since $|S| > n/2$ and $|S'| > n/2$, we have $S \cap S'$ is nonempty and so S' includes a process that sent $\mathsf{ack}(v_2, t_2)$ with $t_2 \geq t_1$. So π_2 is serialized after

π_1. The second case is when π_2 is a write; but then π_1 returns a timestamp that precedes the writer's increment in π_2, and so again is serialized first.

17.4 Proof that $f < n/2$ is necessary

This is pretty much the standard partition argument that $f < n/2$ is necessary to do anything useful in a message-passing system. Split the processes into two sets S and S' of size $n/2$ each. Suppose the writer is in S. Consider an execution where the writer does a write operation, but all messages between S and S' are delayed. Since the writer can't tell if the S' processes are slow or dead, it eventually returns. Now let some reader in S' attempt to read the simulated register, again delaying all messages between S and S'; now the reader is forced to return some value without knowing whether the S processes are slow or dead. If the reader doesn't return the value written, we lose. If by some miracle it does, then we lose in the execution where the write didn't happen and all the processes in S really were dead.

17.5 Multiple writers

So far we have assumed a single writer. The main advantage of this approach is that we don't have to do much to manage timestamps: the single writer can just keep track of its own. With multiple writers we can use essentially the same algorithm, but each write needs to perform an initial round of gathering timestamps so that it can pick a new timestamp bigger than those that have come before. We also extend the timestamps to be of the form $\langle \mathsf{count}, \mathsf{id} \rangle$, lexicographically ordered, so that two timestamps with the same count field are ordered by process ID. The modified write algorithm is:

1. Send $\texttt{read}(u)$ to all processes and wait to receive acknowledgments from a majority of the processes.

2. Set my timestamp to $t = (\max_q \mathsf{count}_q + 1, \mathsf{id})$ where the max is taken over all processes q that sent me an acknowledgment. Note that this is a two-field timestamp that is compared lexicographically, with the id field used only to prevent duplicate timestamps.

3. Send $\texttt{write}(v, t)$ to all processes, and wait for a response $\mathsf{ack}(v, t)$ from a majority of processes.

This increases the cost of a write by a constant factor, but in the end we still have only a linear number of messages. The proof of linearizability is

essentially the same as for the single-writer algorithm, except now we must consider the case of two write operations by different processes. Here we have that if $\pi_1 <_T \pi_2$, then π_1 gets acknowledgments of its write with timestamp t_1 from a majority of processes before π_2 starts its initial phase to compute count. Since π_2 waits for acknowledgments from a majority of processes as well, these majorities overlap, so π_2's timestamp t_2 must exceed t_1. So the linearization ordering previously defined still works.

17.6 Other operations

The basic ABD framework can be extended to support other operations.

One such operation is a **collect** [SSW91], where we read n registers in parallel with no guarantee that they are read at the same time. This can trivially be implemented by running n copies of ABD in parallel, and can be implemented with the same time and message complexity as ABD for a single register by combining the messages from the parallel executions into single (possibly very large) messages.

The ABD algorithm can also be used to implement a max register, which is a register that returns the largest value ever written to it instead of the most recent value (see Chapter 22). The idea is that the multi-writer version of ABD already implements a max register for timestamps. So we can discard the value field entirely and just set each timestamp to a writer's input, and have each reader return the largest timestamp it sees.

17.7 Byzantine failures

With effort, it is possible to adapt the ABD algorithm [ABND95] to handle Byzantine failures. Because a Byzantine writer can overwrite a simulated register with garbage, this mostly makes sense for SWMR registers, where we can limit the damage done by a Byzantine process to the contents of its own simulated register.

Mostéfaoui *et al.* [MPRJ17] give an ABD-like algorithm that simulates a SWMR register in an asynchronous message-passing system with $t < n/3$ Byzantine faults, without resorting to cryptography. The main change is to replace the broadcast done by the writer with a Byzantine reliable broadcast due to Bracha [Bra87]. This has the unfortunate side-effect of increasing the message complexity of a write operation to $O(n^2)$. Fortunately, the authors are able to show that read operations can skip the reliable broadcast and

still run in $O(n)$ messages. The details are messy enough that we will not attempt to reproduce them here; see the cited paper if you are interested.

Chapter 18

Mutual exclusion

For full details see [AW04, Chapter 4] or [Lyn96, Chapter 10].

18.1 The problem

The goal is to share some critical resource between processes without more than one using it at a time—this is *the* fundamental problem in time-sharing systems.

The solution is to only allow access while in a specially-marked block of code called a **critical section**, and only allow one process at a time to be in a critical section.

A **mutual exclusion protocol** guarantees this, usually in an asynchronous shared-memory model.

Formally: We want a process to cycle between states **trying** (trying to get into critical section), **critical** (in critical section), **exiting** (cleaning up so that other processes can enter their critical sections), and **remainder** (everything else—essentially just going about its non-critical business). Only in the trying and exiting states does the process run the mutual exclusion protocol to decide when to switch to the next state; in the critical or remainder states it switches to the next state on its own.

The ultimate payoff is that mutual exclusion solves for systems without failures what consensus solves for systems with failures: if the only way to update a data structure is to hold a lock on it, we are guaranteed to get a nice clean sequence of atomic-looking updates. Of course, once we allow failures back in, mutex becomes less useful, as our faulty processes start crashing without releasing their locks, and with the data structure in some

broken, half-updated state.[1]

18.2 Goals

(See also [AW04, §4.2], [Lyn96, §10.2].)

Core mutual exclusion requirements:

Mutual exclusion At most one process is in the critical state at a time.

No deadlock (progress) If there is at least one process in a trying state, then eventually some process enters a critical state; similarly for exiting and remainder states.

Note that the protocol is not required to guarantee that processes leave the critical or remainder state, but we generally have to insist that the processes at least leave the critical state on their own to make progress.

An additional useful property (not satisfied by all mutual exclusion protocols; see [Lyn96, §10.4)]:

No lockout (**lockout-freedom**): If there is a particular process in a trying or exiting state, that process eventually leaves that state. This means that I don't starve because somebody else keeps jumping past me and seizing the critical resource before I can.

Stronger starvation guarantees include explicit time bounds (how many rounds can go by before I get in) or **bounded bypass** (nobody gets in more than k times before I do). Each of these imply lockout-freedom assuming no deadlock.

18.3 Mutual exclusion using strong primitives

See [AW04, §4.3] or [Lyn96, 10.9]. The idea is that we will use some sort of **read-modify-write** register, where the RMW operation computes a new value based on the old value of the register and writes it back as a single atomic operation, usually returning the old value to the caller as well.

[1] In principle, if we can detect that a process has failed, we can work around this problem by allowing some other process to bypass the lock and clean up. This may require that the original process leaves behind notes about what it was trying to do, or perhaps copies the data it is going to modify somewhere else before modifying it. But even this doesn't work if some zombie process can suddenly lurch to life and scribble its ancient out-of-date values all over our shiny modern data structure.

18.3.1 Test and set

A **test-and-set** operation does the following sequence of actions atomically:

```
1 oldValue ← read(bit)
2 write(bit, 1)
3 return oldValue
```

Typically there is also a second **reset** operation for setting the bit back to zero. For some implementations, this reset operation may only be used safely by the last process to get 0 from the test-and-set bit.

Because a test-and-set operation is atomic, if two processes both try to perform test-and-set on the same bit, only one of them will see a return value of 0. This is not true if each process simply executes the above code on a stock atomic register: there is an execution in which both processes read 0, then both write 1, then both return 0 to whatever called the non-atomic test-and-set subroutine.

Test-and-set provides a trivial implementation of mutual exclusion, shown in Algorithm 18.1.

```
1 while true do
      // trying
2     while TAS(lock) = 1 do nothing
      // critical
3     (do critical section stuff)
      // exiting
4     reset(lock)
      // remainder
5     (do remainder stuff)
```

Algorithm 18.1: Mutual exclusion using test-and-set

It is easy to see that this code provides mutual exclusion, as once one process gets a 0 out of lock, no other can escape the inner while loop until that process calls the `reset` operation in its exiting state. It also provides progress (assuming the lock is initially set to 0); the only part of the code that is not straight-line code (which gets executed eventually by the fairness condition) is the inner loop, and if lock is 0, some process escapes it, while if lock is 1, some process is in the region between the `TAS` call and the `reset`

call, and so it eventually gets to `reset` and lets the next process in (or itself, if it is very fast).

The algorithm does *not* provide lockout-freedom: nothing prevents a single fast process from scooping up the lock bit every time it goes through the outer loop, while the other processes ineffectually grab at it just after it is taken away. Lockout-freedom requires a more sophisticated turn-taking strategy.

18.3.2 A lockout-free algorithm using an atomic queue

Basic idea: In the trying phase, each process enqueues itself on the end of a shared queue (assumed to be an atomic operation). When a process comes to the head of the queue, it enters the critical section, and when exiting it dequeues itself. So the code would look something like Algorithm 18.2.

Note that this requires a queue that supports a `peek` operation that returns the head of the queue. Not all implementations of queues have this property.

```
1 while true do
      // trying
2     enq(q, myId)
3     while peek(q) ≠ myId do nothing
      // critical
4     (do critical section stuff)
      // exiting
5     deq(q)
      // remainder
6     (do remainder stuff)
```

Algorithm 18.2: Mutual exclusion using a queue

Here the proof of mutual exclusion is that only the process whose ID is at the head of the queue can enter its critical section. Formally, we maintain an invariant that any process whose program counter is between the inner while loop and the call to `deq`(q) must be at the head of the queue; this invariant is easy to show because a process can't leave the while loop unless the test fails (i.e., it is already at the head of the queue), no `enq` operation changes the head value (if the queue is nonempty), and the `deq` operation (which does change the head value) can only be executed by a process already at the head (from the invariant).

Deadlock-freedom follows from proving a similar invariant that every element of the queue is the ID of some process in the trying, critical, or exiting states, so eventually the process at the head of the queue passes the inner loop, executes its critical section, and dequeues its ID.

Lockout-freedom follows from the fact that once a process is at position k in the queue, every execution of a critical section reduces its position by 1; when it reaches the front of the queue (after some finite number of critical sections), it gets the critical section itself. Alternatively, we can argue lockout-freedom by showing bounded bypass: once I am in the queue, no process can execute two critical sections before I do, because once it leaves its first critical section, it enqueues behind me.

18.3.2.1 Replacing the queue with RMW

Following [AW04, §4.3.2], we can give an implementation of this algorithm using a single read-modify-write (RMW) register instead of a queue; this drastically reduces the (shared) space needed by the algorithm. The reason this works is because we don't really need to keep track of the position of each process in the queue itself; instead, we can hand out numerical tickets to each process and have the process take responsibility for remembering where its place in line is.

The RMW register has two fields, first and last, both initially 0. Incrementing last simulates an enqueue, while incrementing first simulates a dequeue. The trick is that instead of testing if it is at the head of the queue, a process simply remembers the value of the last field when it "enqueued" itself, and waits for the first field to equal it.

Algorithm 18.3 shows the code from Algorithm 18.2 rewritten to use this technique. The way to read the `RMW` operations is that the first argument specifies the variable to update and the second specifies an expression for computing the new value. Each `RMW` operation returns the old state of the object, before the update.

In practice, this algorithm is usually implemented using two objects, one of which implements a **fetch-and-increment** operation that increments a register and returns the value before the increment, and one of which is an ordinary atomic register. As in Algorithm 18.3, a process takes a position in line by calling the fetch-and-increment, and the head of the line is marked by the second register, which can only be incremented by a process in the exiting section. This implementation has the same properties of mutual exclusion and starvation-freedom as the single-RMW version.

```
1 while true do
      // trying
2     position ← RMW(V, ⟨V.first, V.last + 1⟩)
      // enqueue
3     while RMW(V, V).first ≠ position.last do
4         nothing
      // critical
5     (do critical section stuff)
      // exiting
6     RMW(V, ⟨V.first + 1, V.last⟩)
      // dequeue
      // remainder
7     (do remainder stuff)
```

Algorithm 18.3: Mutual exclusion using read-modify-write

18.4 Mutual exclusion and linearizability

Beyond controlling access to shared resources, mutual exclusion can instantly give us a linearizable implementation of any object for which we have a sequential implementation. The reason is that we can use a mutex to guard access to the shared data structure implementing the object.

Formally, we imagine that we have a read-modify-write object of some sort and an implementation from atomic registers that works for sequential executions. The simplest way to model this is to imagine that we have a single register r that contains the entire state of the object. A read-modify-write operation reads an old state q from r, computes a new state $f(q)$ and writes it back to r, and finally returns the old value q. This works as long as we don't have two or more processes executing operations concurrently. But we can enforce this with a mutex, as in Algorithm 18.4.

```
1 procedure RMW(f)
2     Enter critical section.
3     q ← r
4     r ← f(q)
5     Leave critical section.
6     return q
```

Algorithm 18.4: Building a concurrent RMW object using mutex

To show that this implementation is linearizable, observe that for any concurrent history H we can construct a sequential history S by assigning the invoke/respond times for each operation to when that operation enters and leaves the critical section. This gives a total order $<_S$ since no process can enter the critical section until the previous one leaves. Since the processes carry out the same operations on r in both H and S, both produce identical views. Given two operations $a <_H b$, a leaves its critical section before b enters its critical section, so $<_H \subseteq <_S$. We thus have a linearization of any given H.

18.5 Mutual exclusion using only atomic registers

While mutual exclusion is easier using powerful primitives, we can also solve the problem using only registers.

18.5.1 Peterson's algorithm

Algorithm 18.5 shows Peterson's lockout-free mutual exclusion protocol for two processes p_0 and p_1 [Pet81] (see also [AW04, §4.4.2] or [Lyn96, §10.5.1]). It uses only atomic registers.

This uses three bits to communicate: present[0] and present[1] indicate which of p_0 and p_1 are participating, and waiting enforces turn-taking. The protocol requires that waiting be multi-writer, but it's OK for present[0] and present[1] to be single-writer.

In the description of the protocol, we write Lines 8 and 10 as two separate lines because they include two separate read operations, and the order of these reads is important.

18.5.1.1 Correctness of Peterson's protocol

Intuitively, let's consider all the different ways that the entry code of the two processes could interact. There are basically two things that each process does: it sets its own present variable in Line 5 and grabs the waiting variable in Line 6. Here's a typical case where one process gets in first:

1. p_0 sets present[0] $\leftarrow 1$
2. p_0 sets waiting $\leftarrow 0$
3. p_0 reads present[1] $= 0$ and enters critical section

```
   shared data:
 1 waiting, initially arbitrary
 2 present[i] for i ∈ {0, 1}, initially 0
 3 Code for process i:
 4 while true do
       // trying
 5     present[i] ← 1
 6     waiting ← i
 7     while true do
 8         if present[¬i] = 0 then
 9             break
10         if waiting ≠ i then
11             break
       // critical
12     (do critical section stuff)
       // exiting
13     present[i] = 0
       // remainder
14     (do remainder stuff)
```

Algorithm 18.5: Peterson's mutual exclusion algorithm for two processes

4. p_1 sets $\mathsf{present}[1] \leftarrow 1$
5. p_1 sets $\mathsf{waiting} \leftarrow 1$
6. p_1 reads $\mathsf{present}[0] = 1$ and $\mathsf{waiting} = 1$ and loops
7. p_0 sets $\mathsf{present}[0] \leftarrow 0$
8. p_1 reads $\mathsf{present}[0] = 0$ and enters critical section

The idea is that if I see a 0 in your present variable, I know that you aren't playing, and can just go in.

Here's a more interleaved execution where the waiting variable decides the winner:

1. p_0 sets $\mathsf{present}[0] \leftarrow 1$
2. p_0 sets $\mathsf{waiting} \leftarrow 0$
3. p_1 sets $\mathsf{present}[1] \leftarrow 1$
4. p_1 sets $\mathsf{waiting} \leftarrow 1$
5. p_0 reads $\mathsf{present}[1] = 1$
6. p_1 reads $\mathsf{present}[0] = 1$
7. p_0 reads $\mathsf{waiting} = 1$ and enters critical section
8. p_1 reads $\mathsf{present}[0] = 1$ and $\mathsf{waiting} = 1$ and loops
9. p_0 sets $\mathsf{present}[0] \leftarrow 0$
10. p_1 reads $\mathsf{present}[0] = 0$ and enters critical section

Note that it's the process that set the waiting variable last (and thus sees its own value) that stalls. This is necessary because the earlier process might long since have entered the critical section.

Sadly, examples are not proofs, so to show that this works in general, we need to formally verify each of mutual exclusion and lockout-freedom. Mutual exclusion is a safety property, so we expect to prove it using invariants. The proof in [Lyn96] is based on translating the pseudocode directly into automata (including explicit program counter variables); we'll do essentially the same proof but without doing the full translation to automata. Below, we write that p_i is at line k if it the operation in line k is enabled but has not occurred yet.

Lemma 18.5.1. *If* present$[i] = 0$*, then* p_i *is at Line 5 or 14.*

Proof. Immediate from the code. □

Lemma 18.5.2. *If* p_i *is at Line 12, and* $p_{\neg i}$ *is at Line 8, 10, or 12, then* waiting $= \neg i$*.*

Proof. We'll do the case $i = 0$; the other case is symmetric. The proof is by induction on the schedule. We need to check that any event that makes the left-hand side of the invariant true or the right-hand side false also makes the whole invariant true. The relevant events are:

- Transitions by p_0 from Line 8 to Line 12. These occur only if present$[1] = 0$, implying p_1 is at Line 5 or 14 by Lemma 18.5.1. In this case the second part of the left-hand side is false.
- Transitions by p_0 from Line 10 to Line 12. These occur only if waiting $\neq 0$, so the right-hand side is true.
- Transitions by p_1 from Line 6 to Line 8. These set waiting to 1, making the right-hand side true.
- Transitions that set waiting to 0. These are transitions by p_0 from Line 6 to Line 10, making the left-hand side false.

□

We can now read mutual exclusion directly off of Lemma 18.5.2: if both p_0 and p_1 are at Line 12, then we get waiting $= 1$ and waiting $= 0$, a contradiction.

To show progress, observe that the only place where both processes can get stuck forever is in the loop at Lines 8 and 10. But then waiting isn't changing, and so some process i reads waiting $= \neg i$ and leaves. To show lockout-freedom, observe that if p_0 is stuck in the loop while p_1 enters the critical section, then after p_1 leaves it sets present$[1]$ to 0 in Line 13 (which lets p_0 in if p_0 reads present$[1]$ in time), but even if it then sets present$[1]$ back to 1 in Line 5, it still sets waiting to 1 in Line 6, which lets p_0 into the critical section. With some more tinkering this argument shows that p_1 enters the critical section at most twice while p_0 is in the trying state, giving 2-bounded bypass; see [Lyn96, Lemma 10.12]. With even more tinkering we get a constant time bound on the waiting time for process i to enter the critical section, assuming the other process never spends more than $O(1)$ time inside the critical section.

18.5.1.2 Generalization to n processes

(See also [AW04, §4.4.3].)

The easiest way to generalize Peterson's two-process algorithm to n processes is to organize a tournament in the form of log-depth binary tree; this method was invented by Peterson and Fischer [PF77]. At each node of the tree, the roles of the two processes are taken by the winners of the subtrees, i.e., the processes who have entered their critical sections in the two-process algorithms corresponding to the child nodes. The winner of the tournament as a whole enters the real critical section, and afterwards walks back down the tree unlocking all the nodes it won in reverse order. It's easy to see that this satisfies mutual exclusion, and not much harder to show that it satisfies lockout-freedom—in the latter case, the essential idea is that if a winner at some node reaches the root infinitely often, then lockout-freedom at that node means that a winner of each child node reaches the root infinitely often.

The most natural way to implement the nodes is to have present[0] and present[1] at each node be multi-writer variables that can be written to by any process in the appropriate subtree. Because the present variables don't do much, we can also implement them as the OR of many single-writer variables (this is what is done in [Lyn96, §10.5.3]), but there is no immediate payoff to doing this since the waiting variables are still multi-writer.

Nice properties of this algorithm are that it uses only bits and that it's very fast: $O(\log n)$ time in the absence of contention.

18.5.2 Fast mutual exclusion

With a bit of extra work, we can reduce the no-contention cost of mutual exclusion to $O(1)$, while keeping whatever performance we previously had in the high-contention case. The trick (due to Lamport [Lam87]) is to put an object at the entrance to the protocol that diverts a solo process onto a "fast path" that lets it bypass the n-process mutex that everybody else ends up on.

Our presentation mostly follows [AW04][§4.4.5], which uses the **splitter** abstraction of Moir and Anderson [MA95] to separate out the mechanism for diverting a lone process.[2] Code for a splitter is given in Algorithm 18.6.

A splitter assigns to each processes that arrives at it the value right, down, or stop. The useful properties of splitters are that if at least one process

[2] Moir and Anderson call these things **one-time building blocks**, but the name **splitter** has become standard in subsequent work.

```
   shared data:
1  atomic register race, big enough to hold an ID, initially ⊥
2  atomic register door, big enough to hold a bit, initially open
3  procedure splitter(id)
4      race ← id
5      if door = closed then
6          return right
7      door ← closed
8      if race = id then
9          return stop
10     else
11         return down
```

Algorithm 18.6: Implementation of a splitter

arrives at a splitter, then (a) at least one process returns right or stop; and (b) at least one process returns down or stop; (c) at most one process returns stop; and (d) any process that runs by itself returns stop. The first two properties will be useful when we consider the problem of **renaming** in Chapter 25; we will prove them there. The last two properties are what we want for mutual exclusion.

The names of the variables race and door follow the presentation in [AW04, §4.4.5]; Moir and Anderson [MA95], following Lamport [Lam87], call these X and Y. As in [MA95], we separate out the right and down outcomes—even though they are equivalent for mutex—because we will need them later for other applications.

The intuition behind Algorithm 18.6 is that setting door to closed closes the door to new entrants, and the last entrant to write its ID to race wins (it's a slow race), assuming nobody else writes race and messes things up. The added cost of the splitter is always $O(1)$, since there are no loops.

To reset the splitter, write open to door. This allows new processes to enter the splitter and possibly return stop.

Lemma 18.5.3. *After each time that* door *is set to* open*, at most one process running Algorithm 18.6 returns* stop*.*

Proof. To simplify the argument, we assume that each process calls `splitter` at most once.

Let t be some time at which door is set to open ($-\infty$ in the case of the initial value). Let S_t be the set of processes that read open from door after

time t and before the next time at which some process writes closed to door, and that later return stop by reaching Line 9.

Then every process in S_t reads door before any process in S_t writes door. It follows that every process in S_t writes race before any process in S_t reads race. If some process p is not the *last* process in S_t to write race, it will not see its own ID, and will not return stop. But only one process can be the last process in S_t to write race.[3] □

Lemma 18.5.4. *If a process runs Algorithm 18.6 by itself starting from a configuration in which* door = open*, it returns* stop*.*

Proof. Follows from examining a solo execution: the process sets race to id, reads open from door, then reads id from race. This causes it to return stop as claimed. □

To turn this into an n-process mutex algorithm, we use the splitter to separate out at most one process (the one that gets stop) onto a **fast path** that bypasses the **slow path** taken by the rest of the processes. The slow-path process first fight among themselves to get through an n-process mutex; the winner then fights in a 2-process mutex with the process (if any) on the fast path.

Releasing the mutex is the reverse of acquiring it. If I followed the fast path, I release the 2-process mutex first then reset the splitter. If I followed the slow path, I release the 2-process mutex first then the n-process mutex. This gives mutual exclusion with $O(1)$ cost for any process that arrives before there is any contention ($O(1)$ for the splitter plus $O(1)$ for the 2-process mutex).

A complication is that if nobody wins the splitter, there is no fast-path process to reset it. If we don't want to accept that the fast path just breaks forever in this case, we have to include a mechanism for a slow-path process to reset the splitter if it can be assured that there is no fast-path process left in the system. The simplest way to do this is to have each process mark a bit in an array to show it is present, and have each slow-path process, while still holding all the mutexes, check on its way out if the door bit is set and no processes claim to be present. If it sees all zeros (except for itself) after seeing door = closed, it can safely conclude that there is no fast-path process and reset the splitter itself. The argument then is that the last slow-path process to leave will do this, re-enabling the fast path once there is

[3] It's worth noting that this last process still might not return stop, because some later process—not in S_t—might overwrite race. This can happen even if nobody ever resets the splitter.

no contention again. This approach is taken implicitly in Lamport's original algorithm, which combines the splitter and the mutex algorithms into a single miraculous blob.

18.5.3 Lamport's Bakery algorithm

See [AW04, §4.4.1] or [Lyn96, §10.7] for some textbook presentations; the original algorithm is found in [Lam74].

This is a lockout-free mutual exclusion algorithm that uses only single-writer registers (although some of the registers may end up holding arbitrarily large values). Code for the Bakery algorithm is given as Algorithm 18.7.

```
   shared data:
1  choosing[i], an atomic bit for each i, initially 0
2  number[i], an unbounded atomic register, initially 0
3  Code for process i:
4  while true do
       // trying
5      choosing[i] ← 1
6      number[i] ← 1 + max_{j≠i} number[j]
7      choosing[i] ← 0
8      for j ≠ i do
9          loop until choosing[j] = 0
10         loop until number[j] = 0 or ⟨number[i], i⟩ < ⟨number[j], j⟩
       // critical
11     (do critical section stuff)
       // exiting
12     number[i] ← 0
       // remainder
13     (do remainder stuff)
```

Algorithm 18.7: Lamport's Bakery algorithm

Note that several of these lines are actually loops; this is obvious for Lines 9 and 10, but is also true for Line 6, which includes an implicit loop to read all $n - 1$ values of number$[j]$.

Intuition for mutual exclusion is that if you have a lower number than I do, then I block waiting for you; for lockout-freedom, eventually I have the smallest number. (There are some additional complications involving the choosing bits that we are sweeping under the rug here.) For a real proof

see [AW04, §4.4.1] or [Lyn96, §10.7].

Selling point is a strong near-FIFO guarantee and the use of only single-writer registers (which need not even be atomic—it's enough that they return correct values when no write is in progress). Weak point is unbounded registers.

18.6 RMR complexity

It's not hard to see that we can't build a shared-memory mutex without busy-waiting: any process that is waiting can't detect that the critical section is safe to enter without reading a register, but if that register tells it that it should keep waiting, it is back where it started and has to read it again. This makes our standard step-counting complexity measures useless for describe the worst-case complexity of a mutual exclusion algorithm.

However, the same argument that suggests we can ignore local computation in a message-passing model suggests that we can ignore local operations on registers in a shared-memory model. Real multiprocessors have memory hierarchies where memory that is close to the CPU (or one of the CPUs) is generally much faster than memory that is more distant. This suggests charging only for **remote memory references**, or RMRs, where each register is local to one of the processes and only operations on non-local registers are expensive. This has the advantage of more accurately modeling real costs [MCS91, And90], and allowing us to build busy-waiting mutual exclusion algorithms with costs we can actually analyze.

As usual, there is a bit of a divergence here between theory and practice. Practically, we are interested in algorithms with good real-time performance, and RMR complexity becomes a heuristic for choosing how to assign memory locations. This gives rise to very efficient mutual exclusion algorithms for real machines, of which the most widely used is the beautiful MCS algorithm of Mellor-Crummey and Scott [MCS91]. Theoretically, we are interested in the question of how efficiently we can solve mutual exclusion in our formal model, and RMR complexity becomes just another complexity measure, one that happens to allow busy-waiting on local variables.

18.6.1 Cache-coherence vs. distributed shared memory

The basic idea of RMR complexity is that a process doesn't pay for operations on local registers. But what determines which operations are local?

In the **cache-coherent** model (CC for short), once a process reads a register it retains a local copy as long as nobody updates it. So if I do a

sequence of read operations with no intervening operations by other processes, I may pay an RMR for the first one (if my cache is out of date), but the rest are free. The assumption is that each process can cache registers, and there is some cache-coherence protocol that guarantees that all the caches stay up to date. We may or may not pay RMRs for write operations or other read operations, depending on the details of the cache-coherence protocol, but for upper bounds it is safest to assume that we do.

In the **distributed shared memory** model (DSM), each register is assigned permanently to a single process. Other processes can read or write the register, but only the owner gets to do so without paying an RMR. Here memory locations are nailed down to specific processes.

In general, we expect the cache-coherent model to be cheaper than the distributed shared-memory model, if we ignore constant factors. The reason is that if we run a DSM algorithm in a CC model, then the process p to which a register r is assigned incurs an RMR only if some other process q accesses p since p's last access. But then we can amortize p's RMR by charging q double. Since q incurs an RMR in the CC model, this tells us that we pay at most twice as many RMRs in DSM as in CC for any algorithm.

The converse is not true: there are (mildly exotic) problems for which it is known that CC algorithms are asymptotically more efficient than DSM algorithms [Gol11, DH04].

18.6.2 RMR complexity of Peterson's algorithm

As a warm-up, let's look at the RMR complexity of Peterson's two-process mutual exclusion algorithm (Algorithm 18.5). Acquiring the mutex requires going through mostly straight-line code, except for the loop that tests $\mathsf{present}[\neg i]$ and $\mathsf{waiting}$.

In the DSM model, spinning on $\mathsf{present}[\neg i]$ is not a problem (we can make it a local variable of process i). But $\mathsf{waiting}$ is trouble. Whichever process we don't assign it to will pay an RMR every time it looks at it. So Peterson's algorithm behaves badly by the RMR measure in this model.

Things are better in the CC model. Now process i may pay RMRs for its first reads of $\mathsf{present}[\neg i]$ and $\mathsf{waiting}$, but any subsequent reads are free unless process $\neg i$ changes one of them. But any change to either of the variables causes process i to leave the loop. It follows that process i pays at most 3 RMRs to get through the busy-waiting loop, giving an RMR complexity of $O(1)$.

RMR complexities for parts of a protocol that access different registers add just like step complexities, so the Peterson-Fischer tree construction

described in §18.5.1.2 works here too. The result is $O(\log n)$ RMRs per critical section access, but only in the CC model.

18.6.3 Mutual exclusion in the DSM model

Yang and Anderson [YA95] give a mutual exclusion algorithm for the DSM model that requires $\Theta(\log n)$ RMRs to reach the critical section. This is now known to be optimal for deterministic algorithms [AHW08]. The core of the algorithm is a 2-process mutex similar to Peterson's, with some tweaks so that each process spins only on its own registers. Pseudocode is given in Algorithm 18.8; this is adapted from [YA95, Figure 1].

```
1  C[side(i)] ← i
2  T ← i
3  P[i] ← 0
4  rival ← C[¬side(i)]
5  if rival ≠ ⊥ and T = i then
6      if P[rival] = 0 then
7          P[rival] = 1
8      while P[i] = 0 do spin
9      if T = i then
10         while P[i] ≤ 1 do spin
   // critical section goes here
11 C[side(i)] ← ⊥
12 rival ← T
13 if rival ≠ i then
14     P[rival] ← 2
```

Algorithm 18.8: Yang-Anderson mutex for two processes

The algorithm is designed to be used in a tree construction where a process with ID in the range $\{1 \ldots n/2\}$ first fights with all other processes in this range, and similarly for processes in the range $\{n/2+1 \ldots n\}$. The function side(i) is 0 for the first group of processes and 1 for the second. The variables $C[0]$ and $C[1]$ are used to record which process is the winner for each side, and also take the place of the present variables in Peterson's algorithm. Each process has its own variable $P[i]$ that it spins on when blocked; this variable is initially 0 and ranges over $\{0, 1, 2\}$; this is used to signal a process that it is safe to proceed, and tests on P substitute for tests

on the non-local variables in Peterson's algorithm. Finally, the variable T is used (like waiting in Peterson's algorithm) to break ties: when $T = i$, it's i's turn to wait.

Initially, $C[0] = C[1] = \perp$ and $P[i] = 0$ for all i.

When I want to enter my critical section, I first set $C[\texttt{side}(i)]$ so you can find me; this also has the same effect as setting $\mathsf{present}[\texttt{side}(i)]$ in Peterson's algorithm. I then point T to myself and look for you. I'll block if I see $C[\neg\texttt{side}(i)] \neq \perp$ and $T = i$. This can occur in two ways: one is that I really write T after you did, but the other is that you only wrote $C[\neg\texttt{side}(i)]$ but haven't written T yet. In the latter case, you will signal to me that T may have changed by setting $P[i]$ to 1. I have to check T again (because maybe I really did write T later), and if it is still i, then I know that you are ahead of me and will succeed in entering your critical section. In this case I can safely spin on $P[i]$ waiting for it to become 2, which signals that you have left.

There is a proof that this actually works in [YA95], but it's 27 pages of very meticulously-demonstrated invariants (in fairness, this includes the entire algorithm, including the tree parts that we omitted here). For intuition, this is not much more helpful than having a program mechanically check all the transitions, since the algorithm for two processes is effectively finite-state if we ignore the issue with different processes i jumping into the role of $\texttt{side}(i)$.

A slightly less rigorous but more human-accessible proof would be analogous to the proof of Peterson's algorithm. We need to show two things: first, that no two processes ever both enter the critical section, and second, that no process gets stuck.

For the first part, consider two processes i and j, where $\texttt{side}(i) = 0$ and $\texttt{side}(j) = 1$. We can't have both i and j skip the loops, because whichever one writes T last sees itself in T. Suppose that this is process i and that j skips the loops. Then $T = i$ and $P[i] = 0$ as long as j is in the critical section, so i blocks. Alternatively, suppose i writes T last but does so after j first reads T. Now i and j both enter the loops. But again i sees $T = i$ on its second test and blocks on the second loop until j sets $P[i]$ to 2, which doesn't happen until after j finishes its critical section.

Now let us show that i doesn't get stuck. Again we'll assume that i wrote T second.

If j skips the loops, then j sets $P[i] = 2$ on its way out as long as $T = i$; this falsifies both loop tests. If this happens after i first sets $P[i]$ to 0, only i can set $P[i]$ back to 0, so i escapes its first loop, and any j' that enters from the 1 side will see $P[i] = 2$ before attempting to set $P[i]$ to 1, so $P[i]$ remains at 2 until i comes back around again. If j sets $P[i]$ to 2 before i sets

$P[i]$ to 0 (or doesn't set it at all because $T = j$, then $C[\texttt{side}(j)]$ is set to $\perp$ before i reads it, so i skips the loops.

If j doesn't skip the loops, then $P[i]$ and $P[j]$ are both set to 1 after i and j enter the loopy part. Because j waits for $P[j] \neq 0$, when it looks at T the second time it will see $T = i \neq j$ and will skip the second loop. This causes it to eventually set $P[i]$ to 2 or set $C[\texttt{side}(j)]$ to $\perp$ before i reads it as in the previous case, so again i eventually reaches its critical section.

Since the only operations inside a loop are on local variables, the algorithm has $O(1)$ RMR complexity. For the full tree this becomes $O(\log n)$.

18.6.4 Lower bounds

For deterministic algorithms, there is a lower bound due to Attiya, Hendler, and Woelfel [AHW08] that shows that any one-shot mutual exclusion algorithm for n processes incurs $\Omega(n \log n)$ total RMRs in either the CC or DSM models (which implies that some single process incurs $\Omega(\log n)$ RMRs). This is based on an earlier breakthrough lower bound of Fan and Lynch [FL06] that proved the same lower bound for the number of times a register changes state. Both bounds are information-theoretic: a family of $n!$ executions is constructed containing all possible orders in which the processes enter the critical section, and it is shown that each RMR or state change only contributes $O(1)$ bits to choosing between them.

For randomized algorithms, Hendler and Woelfel [HW11] have an algorithm that uses $O(\log n / \log \log n)$ expected RMRs against an adaptive adversary, beating the deterministic lower bound. This is the best possible for an adaptive adversary, due to a matching lower bound of Giakkoupis and Woelfel [GW12b] that holds even for systems that provide compare-and-swap objects.

For an oblivious adversary, an algorithm of Giakkoupis and Woelfel [GW14] achieves $O(1)$ expected RMRs using compare-and-swap in the DSM model. A more recent algorithm of Giakkoupis and Woelfel [GW17] gives the same $O(1)$ expected RMRs in the CC model; this also uses compare-and-swap. Curiously, there also exist linearizable $O(1)$-RMR implementations of CAS from registers in this model [GHHW12]; however, it is not clear that these implementations can be combined with the Giakkoupis-Woelfel algorithm to give $O(1)$ expected RMRs using registers, because variations in scheduling of randomized implementations may produce subtle conditioning that gives different behavior from actual atomic objects in the context of a randomized algorithm [GHW11].

18.7 Space complexity

There is a famous result due to Burns and Lynch [BL93] that any mutual exclusion protocol using only read/write registers requires at least n of them. Details are in [Lyn96, §10.8]. A slightly different version of the argument is given in [AW04, §4.4.4]. The proof is another nice example of an indistinguishability proof, where we use the fact that if a group of processes can't tell the difference between two executions, they behave the same in both.

Assumptions: We have a protocol that guarantees mutual exclusion and progress. Our base objects are all atomic registers.

Key idea: In order for some process p to enter the critical section, it has to do at least one write to let the other processes know it is doing so. If not, they can't tell if p ever showed up at all, so eventually either some p' will enter the critical section and violate mutual exclusion or (in the no-p execution) nobody enters the critical section and we violate progress. Now suppose we can park a process p_i on each register r_i with a pending write to i; in this case we say that p_i **covers** r_i. If every register is so covered, we can let p go ahead and do whatever writes it likes and then deliver all the covering writes at once, wiping out anything p did. Now the other processes again don't know if p exists or not. So we can say something stronger: before some process p can enter a critical section, it has to write to an uncovered register.

The hard part is showing that we can cover all the registers without letting p know that there are other processes waiting—if p can see that other processes are waiting, it can just sit back and wait for them to go through the critical section and make progress that way. So our goal is to produce states in which (a) processes $p_1 \ldots, p_k$ (for some k) between them cover k registers, and (b) the resulting configuration is indistinguishable from an **idle configuration** to $p_{k+1} \ldots p_n$, where an idle configuration is one in which every process is in its remainder section.

Lemma 18.7.1. *Starting from any idle configuration C, there exists an execution in which only processes $p_1 \ldots p_k$ take steps that leads to a configuration C' such that (a) C' is indistinguishable by any of $p_{k+1} \ldots p_n$ from some idle configuration C'' and (b) k distinct registers are covered by $p_1 \ldots p_k$ in C'.*

Proof. The proof is by induction on k. For $k = 0$, let $C'' = C' = C$.

For larger k, the essential idea is that starting from C, we first run to a configuration C_1 where $p_1 \ldots p_{k-1}$ cover $k - 1$ registers and C_1 is indistinguishable from an idle configuration by the remaining processes, and

then run p_k until it covers one more register. If we let $p_1 \dots p_{k-1}$ go, they overwrite anything p_k wrote. Unfortunately, they may not come back to covering the same registers as before if we rerun the induction hypothesis (and in particular might cover the same register that p_k does). So we have to look for a particular configuration C_1 that not only covers $k-1$ registers but also has an extension that covers the same $k-1$ registers.

Here's how we find it: Start in C. Run the induction hypothesis to get C_1; here there is a set W_1 of $k-1$ registers covered in C_1. Now let processes p_1 through p_{k-1} do their pending writes, then each enter the critical section, leave it, and finish, and rerun the induction hypothesis to get to a state C_2, indistinguishable from an idle configuration by p_k and up, in which $k-1$ registers in W_2 are covered. Repeat to get sets W_3, W_4, etc. Since this sequence is unbounded, and there are only $\binom{r}{k-1}$ distinct sets of registers to cover (where r is the number of registers), eventually we have $W_i = W_j$ for some $i \neq j$. The configurations C_i and C_j are now our desired configurations covering the same $k-1$ registers.

Now that we have C_i and C_j, we run until we get to C_i. We now run p_k until it is about to write some register not covered by C_i (it must do so, or otherwise we can wipe out all of its writes while it's in the critical section and then go on to violate mutual exclusion). Then we let the rest of p_1 through p_{k-1} do all their writes (which immediately destroys any evidence that p_k ran at all) and run the execution that gets them to C_j. We now have $k-1$ registers covered by p_1 through p_{k-1} and a k-th register covered by p_k, in a configuration that is indistinguishable from idle: this proves the induction step. □

The final result follows by the fact that when $k = n$ we cover n registers; this implies that there are n registers to cover.

It's worth noting that the execution constructed in this proof might be *very, very long*. It's not clear what happens if we consider executions in which, say, the critical section is only entered a polynomial number of times. If we are willing to accept a small probability of failure over polynomially-many entries, there is a randomized mutual exclusion protocol that uses $O(\log n)$ space [AHTW18], at the cost of $O(n)$ amortized RMR complexity in the cache-coherent model. It is still open whether it is possible to reduce the space complexity below $O(n)$ for polynomial-length executions without allowing for a small probability of failure or without having such high RMR complexity.

Chapter 19

The wait-free hierarchy

In a shared memory model, it may be possible to solve some problems using **wait-free** protocols, in which any process can finish the protocol in a bounded number of steps, no matter what the other processes are doing (see Chapter 27 for more on this and some variants).

The **wait-free hierarchy** h_m^r classifies asynchronous shared-memory object types T by **consensus number**, where a type T has consensus number n if with objects of type T and atomic registers (all initialized to appropriate values[1]) it is possible to solve wait-free consensus (i.e., agreement, validity, wait-free termination) for n processes but not for $n+1$ processes. The consensus number of any type is at least 1, since 1-process consensus requires no interaction, and may range up to ∞ for particularly powerful objects.

The general idea is that a type T with consensus number c can't simulate at type T' with a higher consensus number c', because then we could use the simulation to convert a c'-process consensus protocol using T' into a c'-process consensus protocol using T. The converse claim, that objects with the same or higher consensus numbers can simulate those with lower

[1] The justification for assuming that the objects can be initialized to an arbitrary state is a little tricky. The idea is that if we are trying to implement consensus from objects of type T that are themselves implemented in terms of objects of type S, then it's natural to assume that we initialize our simulated type-T objects to whatever states are convenient. Conversely, if we are using the ability of type-T objects to solve n-process consensus to show that they can't be implemented from type-S objects (which can't solve n-process consensus), then for both the type-T and type-S objects we want these claims to hold no matter how they are initialized.

If we don't like the convenient initialization assumption, we can also use the algorithm of Borowsky *et al.* [BGA94] to enforce initialization to any reachable state. See §19.1.2 for a discussion of how this works.

ones, is not necessarily true: even though n-process consensus can implement any object for n processes (see §19.3), it may be that for more than n processes there is some object that has consensus number n but that cannot be implemented from an arbitrary n-consensus object.[2]

The wait-free hierarchy was suggested by work by Maurice Herlihy [Her91b] that classified many common (and some uncommon) shared-memory objects by consensus number, and showed that an unbounded collection of objects with consensus number n together with atomic registers gives a wait-free implementation of any object in an n-process system.

19.1 Formal version

Various subsequent authors noticed that this did not give a **robust hierarchy** in the sense that combining two types of objects with consensus number n could solve wait-free consensus for larger n, and the hierarchy h_m^r was proposed by Prasad Jayanti [Jay97] as a way of classifying objects that might be robust: an object is at level n of the h_m^r hierarchy if having unboundedly many objects plus unboundedly many registers solves n-process wait-free consensus but not $(n+1)$-process wait-free consensus.[3]

There is some flexibility in what assumptions we make about initialization and what version of consensus we solve. This is discussed below in §§19.1.2 and 19.1.3.

19.1.1 Robustness

Whether or not the resulting hierarchy is in fact robust for arbitrary deterministic objects is still open, but Ruppert [Rup00] subsequently showed that it is robust for RMW registers and objects with a read operation that returns the current state, and there is a paper by Borowsky, Gafni, and Afek [BGA94] that sketches a proof based on a topological characterization of computability[4] that h_m^r is robust for deterministic objects that don't discriminate between processes (unlike, say, single-writer registers). So for well-behaved shared-memory objects (deterministic, symmetrically accessible,

[2] The existence of such objects was eventually demonstrated by Afek, Ellen, and Gafni [AEG16].

[3] The r in h_m^r stands for the registers, the m for having many objects of the given type. Jayanti [Jay97] also defines a hierarchy h_1^r where you only get finitely many objects. The h stands for "hierarchy," or, more specifically, $h(T)$ stands for the level of the hierarchy at which T appears [Jay11].

[4] See Chapter 29.

with read operations, etc.), consensus number appears to give a real classification that allows us to say for example that any collection of read-write registers (consensus number 1), fetch-and-increments (2), test-and-set bits (2), and queues (2) is not enough to build a compare-and-swap (∞).[5]

We won't attempt to do the robustness proofs of Borowsky *et al.* [BGA94] or Ruppert [Rup00]. Instead, we'll concentrate (in §19.2) on Herlihy's original results and show that specific objects have specific consensus numbers when used in isolation. The procedure in each case will be to show an upper bound on the consensus number using a variant of Fischer-Lynch-Paterson (made easier because we are wait-free and don't have to worry about fairness) and then show a matching lower bound (for non-trivial upper bounds) by exhibiting an n-process consensus protocol for some n. Most of what we show below is taken directly from Herlihy's paper [Her91b], so reading that may make more sense than reading these notes.

19.1.2 Initialization

Another useful result from the Borowsky *et al.*paper [BGA94] mentioned above is that the consensus number is not generally dependent on what assumptions we make about the initial state of the objects. Specifically, [BGA94, Lemma 3.2] states that as long as there is some sequence of operations that takes an object from a fixed initial state to a desirable initial state for consensus, then we can safely assume that the object is in the desirable state. The core idea of the proof is that each process can initialize its own copy of the object and then announce that it is ready; each process will then participate in a sequence of consensus protocols using the objects that they observe are ready, with the output of each protocol used as the input to the next. Because the first object S_i to be announced as initialized will be visible to all processes, they will all do consensus using S_i. Any subsequent protocols that may be used by only a subset of the processes will not change the common agreed output from the S_i protocol.[6] This justifies our assumption that objects can be initialized to any desired value.

[5] Ruppert's paper is particularly handy because it gives an algorithm for computing the consensus number of the objects it considers. However, for infinite-state objects, this requires solving the halting problem (as previously shown by Jayanti and Toueg [JT92]).

[6] The result in the paper is stated for a consensus protocol that uses a single copy of the object, but it generalizes in the obvious way to those that use multiple copies of the object.

19.1.3 Output value of the consensus protocol

Depending on what we are interested in, we can imagine several different conventions for the output of a consensus protocol. These correspond to different choices for the validity condition:

1. **Binary consensus** outputs a value 0 or 1 that is equal to the input of some participating process.
2. **Id consensus** outputs the id of some participating process.
3. **Multivalued consensus**consensus!multivalued outputs a value that is equal to the input of some participating process. Unlike binary consensus, the range of inputs and outputs is arbitrary.

It is trivial to show that multivalued consensus can implement both binary consensus and id consensus.

In the other direction, if we have id consensus, we can implement multivalued consensus using a standard trick: have each process i write its input to a register r_i not used by the id-consensus protocol. Then each process that learns a winner j from the id-consensus protocol can read r_j to obtain j's value.

The tricky case is going from binary consensus to id-consensus. Here the idea is to perform a tournament similar to Peterson-Fischer [PF77]. Build a binary tree whose internal nodes are binary-consensus protocols C_b, each indexed by a binary string of length equal to its depth. Each process starts at a leaf determined by the binary expansion of its id and fights its way to the top. Unlike mutual exclusion, a process continues to fight on behalf of its subtree even if it loses. Once the outcome at the root $C_{\langle\rangle}$ is determined, we can work backwards to figure out which leaf is the actual winner. (See Algorithm 19.1.)

A complication here is that this may require processes that didn't participate in a particular subtree on the way up to be able to detect the outcome of the consensus protocol for that subtree on the way down. Fortunately, since we only do this after the winner of the subtree is determined, it's safe for a curious process to just join the subtree's consensus protocol with a default input value, since this default input won't change the outcome. We'll leave the actual proof of correctness as an exercise.

19.1.4 Multiple objects vs multiple operations

When considering multiple objects, the usual assumption is that objects are combined by putting them next to each other. If we can combine two objects

```
  // Returns the id of a participating process
1 procedure idConsensus()
2     Let x_1 ... x_ℓ = binary expansion of my id
3     for i ← ℓ − 1 down to 0 do
          // C_{x_1...x_{i−1}} is a binary consensus object
4         C_{x_1...x_{i−1}}(x_i)
      // Reconstruct winning sequence
5     for i ← 0 to ℓ − 1 do
          // Get previously decided output
6         y_{i+1} ← C_{y_1...y_i}(0)
7     return y_1 ... y_ℓ
```

Algorithm 19.1: Id consensus from binary consensus

by constructing a single object with operations of both—which is essentially what happens when we apply different machine language instructions to the same memory location—then the object with both operations may have a higher consensus number than the object with either operation individually. This was observed by Ellen *et al.* [EGSZ20]. A simple example would be a register than supports increment ($+1$) and doubling ($\times 2$) operations. A register with only one of these operations is equivalent to a counter and has consensus number 1. But a register with both operations has consensus number at least 2, since if it is initialized to 2, we can tell which of the two operations went first by looking at the final value: $3 = 2 + 1, 4 = 2 \times 2, 5 = (2 \times 2) + 1, 6 = (2 + 1) \times 2$.

19.2 Classification by consensus number

Here we show the position of various types in the wait-free hierarchy. The quick description is shown in Table 19.1; more details (mostly adapted from [Her91b]) are given below.

19.2.1 Level 1: atomic registers, counters, other interfering RMW registers that don't return the old value

First observe that any type has consensus number at least 1, since 1-process consensus is trivial.

We'll argue that a large class of particularly weak objects has consensus

Consensus number	Defining characteristic	Examples
1	Read with interfering no-return RMW.	Registers, counters, generalized counters, max registers, atomic snapshots.
2	Interfering RMW; queue-like structures.	Test-and-set, fetch-and-add, queues, process-to-memory swap.
m	First of $\leq m$ write-like operations wins	m-process consensus objects, m-sliding window registers.
$2m-2$		Atomic m-register write.
∞	First write-like operation wins.	Queue with peek, fetch-and-cons, sticky bits, compare-and-swap, memory-to-memory swap, memory-to-memory copy.

Table 19.1: Position of various types in the wait-free hierarchy

number exactly 1, by running FLP with 2 processes. Recall from Chapter 11 that in the Fischer-Lynch-Paterson [FLP85] proof we classify states as bivalent or univalent depending on whether both decision values are still possible, and that with at least one failure we can always start in a bivalent state (this doesn't depend on what objects we are using, since it depends only on having invisible inputs). Since the system is wait-free there is no constraint on adversary scheduling, and so if any bivalent state has a bivalent successor we can just do it. So to solve consensus we have to reach a bivalent configuration C that has only univalent successors, and in particular has a 0-valent and a 1-valent successor produced by applying operations x and y of processes p_x and p_y.

Assuming objects don't interact with each other behind the scenes, x and y must be operations of the same object. Otherwise $Cxy = Cyx$ and we get a contradiction.

Now let's suppose we are looking at atomic registers, and consider cases:

- x and y are both reads, Then x and y commute: $Cxy = Cyx$, and we get a contradiction.
- x is a read and y is a write. Then p_y can't tell the difference between

Cyx and Cxy, so running p_y to completion gives the same decision value from both Cyx and Cxy, another contradiction.

- x and y are both writes. Now p_y can't tell the difference between Cxy and Cy, so we get the same decision value for both, again contradicting that Cx is 0-valent and Cy is 1-valent.

There's a pattern to these cases that generalizes to other objects. Suppose that an object has a read operation that returns its state and one or more read-modify-write operations that don't return anything (perhaps we could call them "modify-write" operations). We'll say that the MW operations are **interfering** if, for any two operations x and y, either:

- x and y **commute**: $Cxy = Cyx$.
- One of x and y **overwrites** the other: $Cxy = Cy$ or $Cyx = Cx$.

Then no pair of read or modify-write operations can get us out of a bivalent state, because (a) reads commute; (b) for a read and MW, the non-reader can't tell which operation happened first; (c) and for any two MW operations, either they commute or the overwriter can't detect that the first operation happened. So any MW object with uninformative, interfering MW operations has consensus number 1.

For example, consider a counter that supports operations read, increment, decrement, and write: a write overwrites any other operation, and increments and decrements commute with each other, so the counter has consensus number 1. The same applies to a generalized counter that supports an atomic $x \leftarrow x + a$ operation; as long as this operation doesn't return the old value, it still commutes with other atomic increments.

Max registers [AACH12], which have read operations that return the largest value previously written, also have commutative updates, so they also have consensus number 1. This gives an example of an object not invented at the time of Herlihy's paper that is still covered by Herlihy's argument.

19.2.2 Level 2: interfering RMW objects that return the old value, queues (without peek)

Suppose now that we have a RMW object that returns the old value, and suppose that it is *non-trivial* in the sense that it has at least one RMW operation where the embedded function f that determines the new value is not the identity (otherwise RMW is just read). Then there is some value v such that $f(v) \neq v$. To solve two-process consensus, have each process p_i first

write its preferred value to a register r_i, then execute the non-trivial RMW operation on the RMW object initialized to v. The first process to execute its operation sees v and decides its own value. The second process sees $f(v)$ and decides the first process's value (which it reads from the register).[7] It follows that a non-trivial RMW object has consensus number *at least* 2.

In many cases, this is all we get. Suppose that the operations of some RMW type T are non-interfering in a way analogous to the previous definition, where now we say that x and y commute if they leave the object in the same state (regardless of what values are returned) and that y overwrites x if the object is always in the same state after both x and xy (again regardless of what is returned). The two processes p_x and p_y that carry out x and y know what happened, but a third process p_z doesn't. So if we run p_z to completion we get the same decision value after both Cx and Cy, which means that Cx and Cy can't be 0-valent and 1-valent. It follows that no collection of RMW registers with interfering operations can solve 3-process consensus, and thus all such objects have consensus number 2. Examples of these objects include **test-and-set** bits, **fetch-and-add** registers, and **swap** registers that support an operation `swap` that writes a new value and returns the previous value.

There are some other objects with consensus number 2 that don't fit this pattern. Define a **wait-free queue** as an object with enqueue and dequeue operations (like normal queues), where dequeue returns $\perp$ if the queue is empty (instead of blocking). To solve 2-process consensus with a wait-free queue, initialize the queue with a single value (it doesn't matter what the value is). We can then treat the queue as a non-trivial RMW register where a process wins if it successfully dequeues the initial value and loses if it gets empty.[8]

However, enqueue operations are non-interfering: if p_x enqueues v_x and p_y enqueues v_y, then any third process can detect which happened first; similarly we can distinguish $\texttt{enq}(x)\texttt{deq}()$ from $\texttt{deq}()\texttt{enq}(x)$. So to show we can't do three process consensus we do something sneakier: given a bivalent state C with allegedly 0- and 1-valent successors $C\texttt{enq}(x)$ and $C\texttt{enq}(y)$,

[7] The extra registers are just implementing the standard construction of multivalued consensus from id-consensus; see §19.1.3.

[8] But wait! What if the queue starts empty?

This turns out to be a surprisingly annoying problem, and was one of the motivating examples for h^r_m as opposed to Herlihy's vaguer initial definition.

With one empty queue and nothing else, Jayanti and Toueg [JT92, Theorem 7] show that there is no solution to consensus for two processes. This is also true for stacks (Theorem 8 from the same paper). But adding a register (Theorem 9) lets you do it. A second empty queue also works.

consider both C`enq`(x)`enq`(y) and C`enq`(y)`enq`(x) and run p_x until it does a `deq`() (which it must, because otherwise it can't tell what to decide) and then stop it. Now run p_y until it also does a `deq`() and then stop it. We've now destroyed the evidence of the split and poor hapless p_z is stuck. In the case of C`deq`()`enq`(x) and C`enq`(x)`deq`() on a non-empty queue we can kill the initial dequeuer immediately and then kill whoever dequeues x or the value it replaced, and if the queue is empty only the dequeuer knows. In either case we reach indistinguishable states after killing only 2 witnesses, and the queue has consensus number at most 2.

Similar arguments work on stacks, deques, and so forth—these all have consensus number exactly 2.

19.2.3 Level ∞: objects where the first write wins

These are objects that can solve consensus for any number of processes. Here are a bunch of level-∞ objects:

Queue with peek Has operations `enq`(x) and `peek`(), which returns the first value enqueued. (Maybe also `deq`(), but we don't need it for consensus). Protocol is to enqueue my input and then peek and return the first value in the queue.

Fetch-and-cons Returns old cdr and adds new car on to the head of a list. Use preceding protocol where `peek`() = `tail`(car :: cdr).

Sticky bit Has a `write` operation that has no effect unless register is in the initial $\perp$ state. Whether the `write` succeeds or fails, it returns nothing. The consensus protocol is to write my input and then return result of a read.

Compare-and-swap Has `CAS`(old, new) operation that writes new only if previous value is old. Use it to build a sticky bit.

Load-linked/store-conditional Like compare-and-swap split into two operations. The **o**peration reads a memory location and marks it. The **o**peration succeeds only if the location has not been changed since the preceding load-linked by the same process. Can be used to build a sticky bit.

Memory-to-memory swap Has `swap`(r_i, r_j) operation that atomically swaps contents of r_i with r_j, as well as the usual read and write operations for all registers. Use to implement fetch-and-cons. Alternatively, use two registers input$[i]$ and victory$[i]$ for each process i, where

victory[i] is initialized to 0, and a single central register prize, initialized to 1. To execute consensus, write your input to input[i], then swap victory[i] with prize. The winning value is obtained by scanning all the victory registers for the one that contains a 1, then returning the corresponding input value.)

Memory-to-memory copy Has a `copy`(r_i, r_j) operation that copies r_i to r_j atomically. Use the same trick as for memory-to-memory swap, where a process copies prize to victory[i]. But now we have a process follow up by writing 0 to prize. As soon as this happens, the victory values are now fixed; take the leftmost 1 as the winner.[9]

Herlihy [Her91b] gives a slightly more complicated version of this procedure, where there is a separate prize[i] register for each i, and after doing its copy a process writes 0 to all of the prize registers. This shows that memory-to-memory copy solves consensus for arbitrarily many processes even if we insist that copy operations can never overlap. The same trick also works for memory-to-memory swap, since we can treat a memory-to-memory swap as a memory-to-memory copy given that we don't care what value it puts in the prize[i] register.

Bank accounts A **bank account** object stores a non-negative integer, and supports a `read` operation that returns the current value and a `withdraw`(k) operation that reduces the value by k, unless this would reduce the value below 0, in which case it has no effect.

To solve (binary) consensus with a bank account, start it with 3, and have each process with input b attempt to withdraw $3 - b$ from the account. After the first withdrawal, the object will hold either 0 or 1, and no further withdrawals will have any effect. So the bank account acts exactly like a sticky bit where 3 represents $\perp$.[10]

For many years, I assumed that this example demonstrated why cryptocurrencies all seem to use embedded consensus protocols of some sort. However, it turns out that there is a critical assumption needed for this proof, which is that more than one process can spend from the same account. Without this assumption, it has been shown by Guerraoui *et al.* [GKM+19] that the consensus number of a single-spender

[9] Or use any other rule that all processes apply consistently.

[10] If you have more money, you can extend this construction to any fixed set of values. For example, to choose among values v in $0 \ldots m-1$, start with $2m$ and have a process with input v subtract $2m - v$.

bank account is 1, and more generally that the consensus number of a k-spender bank account is exactly k.

19.2.4 Level $2m - 2$: simultaneous m-register write

Here we have a (large) collection of atomic registers augmented by an m-register write operation that performs all the writes simultaneously. The intuition for why this is helpful is that if p_1 writes r_1 and r_{shared} while p_2 writes r_2 and r_{shared} then any process can look at the state of r_1, r_2 and r_{shared} and tell which write happened first. Code for this procedure is given in Algorithm 19.2; note that up to 4 reads may be necessary to determine the winner because of timing issues.[11]

The workings of Algorithm 19.2 are straightforward:

- If the process reads $r_1 = r_2 = \perp$, then we don't care which went first, because the reader (or somebody else) already won.
- If the process reads $r_1 = 1$ and then $r_2 = \perp$, then p_1 went first.
- If the process reads $r_2 = 2$ and then $r_1 = \perp$, then p_2 went first. (This requires at least one more read after checking the first case.)
- Otherwise the process saw $r_1 = 1$ and $r_2 = 2$. Now read r_{shared}: if it's 1, p_2 went first; if it's 2, p_1 went first.

Algorithm 19.2 requires 2-register writes, and will give us a protocol for 2 processes (since the reader above has to participate somewhere to make the first case work). For m processes, we can do the same thing with m-register writes. We have a register $r_{pq} = r_{qp}$ for each pair of distinct processes p and q, plus a register r_{pp} for each p; this gives a total of $\binom{m}{2} + m = O(m^2)$ registers. All registers are initialized to $\perp$. Process p then writes its initial preference to some single-writer register pref_p and then simultaneously writes p to r_{pq} for all q (including r_{pp}). It then attempts to figure out the first writer by applying the above test for each q to r_{pq} (standing in for r_{shared}), r_{pp} (r_1) and r_{qq} (r_2). If it won against all the other processes, it decides its own value. If not, it repeats the test recursively for some p' that beat it until

[11] The main issue is that processes can only read the registers one at a time. An alternative to running Algorithm 19.2 is to use a double-collect snapshot (see §20.1) to simulate reading all three registers at once. However, this might require as many as twelve read operations, since a process doing a snapshot has to re-read all three registers if any of them change.

```
1  v_1 ← r_1
2  v_2 ← r_2
3  if v_1 = v_2 = ⊥ then
4      return no winner
5  if v_1 = 1 and v_2 = ⊥ then
       // p_1 went first
6      return 1
   // read r_1 again
7  v'_1 ← r_1
8  if v_2 = 2 and v'_1 = ⊥ then
       // p_2 went first
9      return 2
   // both p_1 and p_2 wrote
10 if r_shared = 1 then
11     return 2
12 else
13     return 1
```

Algorithm 19.2: Determining the winner of a race between 2-register writes. The assumption is that p_1 and p_2 each wrote their own IDs to r_i and r_{shared} simultaneously. This code can be executed by any process (including but not limited to p_1 or p_2) to determine which of these 2-register writes happened first.

it finds a process that beat everybody, and returns its value. So m-register writes solve m-process wait-free consensus.

A further tweak gets $2m-2$: run two copies of an $(m-1)$-process protocol using separate arrays of registers to decide a winner for each group. Then add a second phase where processes contend across the groups. This involves each process p from group 1 writing the winning ID for its group simultaneously into s_p and s_{pq} for each q in the other group. The first process to do this will be the only process that wins against every process in the other group, so we can pick a winning group by looking for some such process. We can then return the input value for whichever process won within the winning group.

One thing to note about the second phase is that, unlike mutex, we can't just have the winners of the two groups fight each other, since this would not give the wait-free property for non-winners. Instead, we have to allow a non-winner p to pick up the slack for a slow winner and fight on behalf of the entire group. This requires an m-process write operation to write s_p and all s_{pq} at once.

19.2.4.1 Matching impossibility result

It might seem that the technique used to boost from m-process consensus to $(2m-2)$-process consensus could be repeated to get up to at least $\Theta(m^2)$, but this turns out not to be the case. The essential idea is to show that in order to escape bivalence, we have to get to a configuration C where *every* process is about to do an m-register write leading to a univalent configuration (since reads don't help for the usual reasons, and normal writes can be simulated by m-register writes with an extra $m-1$ dummy registers), and then argue that these writes can't overlap too much. So suppose we are in such a configuration, and suppose that Cx is 0-valent and Cy is 1-valent, and we also have many other operations $z_1 \dots z_k$ that lead to univalent states. Following Herlihy [Her91b], we argue in two steps:

1. There is some register that is written to by x alone out of all the pending operations. Proof: Suppose not. Then the 0-valent configuration $Cxyz_1 \dots z_k$ is indistinguishable from the 1-valent configuration $Cyz_1 \dots z_k$ by any process except p_x, and we're in trouble.

2. There is some register that is written to by x and y but not by any of the z_i. Proof:: Suppose not. The each register is written by at most one of x and y, making it useless for telling which went first; or it is overwritten by some z_i, hiding the value that tells which went first.

So $Cxyz_1 \dots z_k$ is indistinguishable from $Cyxz_1 \dots z_k$ for any process other than p_x and p_y, and we're still in trouble.

Now suppose we have $2m - 1$ processes. The first part says that each of the pending operations (x, y, all of the z_i) writes to 1 single-writer register and at least k two-writer registers where k is the number of processes leading to a different univalent value. This gives $k + 1$ total registers simultaneously written by this operation. Now observe that with $2m - 1$ process, there is some set of m processes whose operations all lead to a b-valent state; so for any process to get to a $(\neg b)$-valent state, it must write $m + 1$ registers simultaneously. It follows that with only m simultaneous writes we can only do $(2m - 2)$-consensus.

Curiously, we can see the last bivalent configuration in the algorithm given earlier: as long as we have not had any process contend with the processes in the other group, it is still possible for the winner of either group to win the overall protocol. If we run each process until it is about to do its final m-register write, we get exactly the situation where the processes in one group give exactly $m - 1$ pending writes that lead to 0-valent configurations and the processes in the other group give exactly $m - 1$ pending writes that lead to 1-valent configurations, with all of these pending writes overlapping in exactly the way required by the impossibility argument. In principle this happens for any consensus implementation that is subject to this kind of bivalence argument, but it is nice to see the structure of the upper bound and lower bound matching up so directly in this case.

19.2.5 Level m: m-process consensus objects, m-sliding window registers

An m-process **consensus object** has a single `consensus` operation that, the first m times it is called, returns the input value in the first operation, and thereafter returns only $\perp$. Clearly this solves m-process consensus. To show that it doesn't solve $(m + 1)$-process consensus even when augmented with registers, run a bivalent initial configuration to a configuration C where any further operation yields a univalent state. By an argument similar to the m-register write case, we can show that the pending operations in C must all be consensus operations on the same consensus object (anything else commutes or overwrites). Now run $Cxyz_1 \dots z_{m-1}$ and $Cyxz_1 \dots z_{m-1}$, where x and y lead to 0-valent and 1-valent states, and observe that the process that did z_{m-1} can't distinguish the resulting configurations because all it got was $\perp$. (Note: this works even if the consensus object isn't in

its initial state, since we know that before x or y the configuration is still bivalent.)

So the m-process consensus object has consensus number m. This shows that h_m^r is nonempty at each level.

A natural question at this point is whether the inability of m-process consensus objects to solve $(m+1)$-process consensus implies robustness of the hierarchy. One might consider the following argument: given any object at level m, we can simulate it with an m-process consensus object, and since we can't combine m-process consensus objects to boost the consensus number, we can't combine any objects they can simulate either. The problem here is that while m-process consensus objects can simulate any object in a system with m processes (see below), it may be that some objects can do more in a system with $m+1$ objects while still not solving $(m+1)$-process consensus. A simple way to see this would be to imagine a variant of the m-process consensus object that doesn't fail completely after m operations; for example, it might return one of the first two inputs given to it instead of $\perp$. This doesn't help with solving consensus, but it might (or might not) make it too powerful to implement using standard m-process consensus objects.

An m-process consensus object is arguably a very artificial way to populate all levels of the consensus hierarchy. Mostefaoui *et al.* [MPR18] proposed **m-sliding window registers** as a "natural" class of objects that has this property. An m-sliding window register RW_m possesses a write operation and a read operation that returns the last m values written to the register in the order they were written.

It's easy to solve m-process consensus using this object. We assume that the initial state of the register does not contain any process IDs, and have each contending process write its ID to the register. The first writer wins.

The proof that an m-sliding window register can't solve consensus for $m+1$ processes is similar to that for m-process consensus objects. Given a system consisting of read-write registers and RW_m objects, choosing the bivalent successor of any configuration either works forever or eventually reaches a configuration C with only univalent successors. By the usual argument, the $m+1$ pending operations in C must all be operations on the same m-sliding window register.

We can easily show that none of these operations can be read operations. Suppose x is a read operation such that Cx is b-valent, and let y be any operation such that Cy is $\neg b$-valent. Then Cxy and Cy are indistinguishable to the $n-1$ processes that do not execute x, giving a contradiction.

Now let x and y be write operations where Cx is 0-valent and Cy is 1-valent. Let $z_1, \ldots, z_{m-1}$ be the remaining operations enabled in C. Then

$Cxyz_1 \dots z_{m-1}$ and $Cyz_1 \dots z_{m-1}$ apply the same last m writes to the sliding window register, leaving the resulting configurations indistinguishable to all processes if the process carrying out x takes no more steps.

Mostefaoui *et al.*observe that taking this argument to the limit shows that a unbounded distributed ledger has infinite consensus number, which is not entirely surprising given that such an object is equivalent to fetch-and-cons (§19.2.3).

19.3 Universality of consensus

Universality of consensus says that any type that can implement n-process consensus can, together with atomic registers, give a wait-free implementation of any object in a system with n processes. That consensus is universal was shown by Herlihy [Her91b] and Plotkin [Plo89]. Both of these papers spend a lot of effort on making sure that both the cost of each operation and the amount of space used is bounded. But if we ignore these constraints, the same result can be shown using a mechanism similar to the replicated state machines of §12.7.

Here the processes repeatedly use consensus to decide between candidate histories of the simulated object, and a process successfully completes an operation when its operation (tagged to distinguish it from other similar operations) appears in a winning history. A round structure avoids too much confusion.

Details are given in Algorithm 19.3.

There are some subtleties to this algorithm. The first time that a process calls consensus (on $c[r]$), it may supply a dummy input; the idea is that it is only using the consensus object to obtain the agreed-upon history from a round it missed. It's safe to do this, because no process writes r to its round register until $c[r]$ is complete, so the dummy input can't be accidentally chosen as the correct value.

It's not hard to see that whatever h_{r+1} is chosen in $c[r+1]$ is an extension of h_r (it is constructed by appending operations to h_r), and that all processes agree on it (by the agreement property of the consensus object $c[r+1]$. So this gives us an increasing sequence of consistent histories. We also need to show that these histories are linearizable. The obvious linearization is just the most recent version of h_r. Suppose some call to `apply`(π_1) finishes before a call to `apply`(π_2) starts. Then π_1 is contained in some h_r when `apply`(π_1) finishes, and since π_2 can only enter h by being appended at the end, we get π_1 linearized before π_2.

```
1  procedure apply(π)
       // announce my intended operation
2      op[i] ← π
3      while true do
           // find a recent round
4          r ← max_j round[j]
           // obtain the history as of that round
5          if h_r = ⊥ then
6              h_r ← consensus(c[r], ⊥)
7          if π ∈ h_r then
8              return value π returns in h_r
           // else attempt to advance
9          h' ← h_r
10         for each j do
11             if op[j] ∉ h' then
12                 append op[j] to h'
13         h_{r+1} ← consensus(c[r + 1], h')
14         round[i] ← r + 1
```

Algorithm 19.3: A universal construction based on consensus

Finally, we need to show termination. The algorithm is written with a loop, so in principle it could run forever. But we can argue that no process after executes the loop more than twice. The reason is that a process p puts its operation in $\mathsf{op}[p]$ before it calculates r; so any process that writes $r' > r$ to round sees p's operation before the next round. It follows that p's value gets included in the history no later than round $r + 2$. (We'll see this sort of thing again when we do atomic snapshots in Chapter 20.)

A minor complication with this construction is that it assumes consensus over arbitrary inputs, while some objects directly implement only binary consensus. Fortunately there is a straightforward reduction of general consensus to a tree of binary consensus protocols. Assign a register to the root of each subtree (including leaves representing the individual processes). To do consensus, I first write my input to my leaf. I then fight my way up through the tree solving binary consensus at each node, with input equal to the side (left or right) I am coming from. Whichever value wins a node, each process participating in the node will copy the winning value from the appropriate subtree to the register for that node. Eventually a single value prevails at the root.

Building a consistent shared history is easier with some particular objects that solve consensus. For example, a **fetch-and-cons** object that supplies an operation that pushes a new head onto a linked list and returns the old head trivially implements the common history above without the need for helping. One way to implement fetch-and-cons is with memory-to-memory swap; to add a new element to the list, create a cell with its next pointer pointing to itself, then swap the next field with the head pointer for the entire list.

The solutions we've described here have a number of deficiencies that make them impractical in a real system (even more so than many of the algorithms we've described). If we store entire histories in a register, the register will need to be very, very wide. If we store entire histories as a linked list, it will take an unbounded amount of time to read the list. For solutions to these problems, see [AW04, 15.3] or the papers of Herlihy [Her91b] and Plotkin [Plo89].

Chapter 20

Atomic snapshots

We've seen in the previous chapter that there are a lot of things we can't make wait-free with just registers. But there are a lot of things we can. Atomic snapshots are a tool that let us do a lot of these things easily.

An **atomic snapshot object** acts like a collection of n single-writer multi-reader atomic registers with a special **snapshot** operation that returns (what appears to be) the state of all n registers at the same time. This is easy without failures: we simply lock the whole register file, read them all, and unlock them to let all the starving writers in. But it gets harder if we want a protocol that is wait-free, where any process can finish its own snapshot or write even if all the others lock up.

We'll give the usual sketchy description of a couple of snapshot algorithms. More details on early snapshot results can be found in [AW04, §10.3] or [Lyn96, §13.3]. There is also a reasonably recent survey by Fich on upper and lower bounds for the problem [Fic05].

20.1 The basic trick: two identical collects equals a snapshot

Let's tag any value written with a sequence number, so that each value written has a seqno field attached to it that increases over time. We can now detect if a new write has occurred between two reads of the same variable. Suppose now that we repeatedly perform **collects**—reads of all n registers—until two successive collects return exactly the same vector of values and sequence numbers. We can then conclude that precisely these values were present in the registers at some time in between the two collects. This gives us a very simple algorithm for snapshot. Unfortunately, it doesn't

terminate if there are a lot of writers around.[1] So we need some way to slow the writers down, or at least get them to do snapshots for us.

20.2 Snapshots using double collects with helping

This is the approach taken by Afek and his five illustrious co-authors [AAD+93] (see also [AW04, §10.3] or [Lyn96, §13.3.2]): before a process can write to its register, it first has to complete a snapshot and leave the results behind with its write.[2] This means that if some slow process (including a slow writer, since now writers need to do snapshots too) is prevented from doing the two-collect snapshot because too much writing is going on, eventually it can just grab and return some pre-packaged snapshot gathered by one of the many successful writers.

Specifically, if a process executing a single snapshot operation σ sees values written by a single process i with three different sequence numbers s_1, s_2 and s_3, then it can be assured that the snapshot σ_3 gathered with sequence number s_3 started no earlier than s_2 was written (and thus no earlier than σ started, since σ read s_1 after it started) and ended no later than σ ended (because σ saw it). It follows that the snapshot can safely return σ_3, since that represents the value of the registers at some time inside σ_3's interval, which is contained completely within σ's interval.

So a snapshot repeatedly does collects until either (a) it gets two identical collects, in which case it can return the results (a **direct scan**, or (b) it sees three different values from the same process, in which case it can take the snapshot collected by the second write (an **indirect scan**). See pseudocode in Algorithm 20.1.

Amazingly, despite the fact that updates keep coming and everybody is trying to do snapshots all the time, a snapshot operation of a single process is guaranteed to terminate after at most $n + 1$ collects. The reason is that

[1] This isn't always a problem, since there may be external factors that keep the writers from showing up too much. Maurice Herlihy and I got away with using exactly this snapshot algorithm in an ancient, pre-snapshot paper on randomized consensus [AH90a]. The reread-until-no-change idea was used as early as 1977 by Lamport [Lam77].

[2] The algorithm is usually called the AADGMS algorithm by people who can remember all the names—or at least the initials—of the team of superheroes who came up with it (Afek, Attiya, Dolev, Gafni, Merritt, and Shavit). Historically, this was one of three independent solutions to the problem that appeared at about the same time. A similar algorithm for **composite registers** was given by James Anderson [And94] and a somewhat different algorithm for **consistent scan** was given by Aspnes and Herlihy [AH90b]. The Afek *et al.* algorithm had the advantage of using bounded registers (in its full version), and so it and its name for atomic snapshot prevailed over its competitors.

in order to prevent case (a) from holding, the adversary has to supply at least one new value in each collect after the first. But it can only supply one new value for each of the $n-1$ processes that aren't doing collects before case (b) is triggered (it's triggered by the first process that shows up with a second new value). Adding up all the collects gives $1+(n-1)+1 = n+1$ collects before one of the cases holds. Since each collect takes $n-1$ read operations (assuming the process is smart enough not to read its own register), a snapshot operation terminates after at most n^2-1 reads.

```
1  procedure update_i(A, v)
2      s ← scan(A)
3      A[i] ← ⟨A[i].count + 1, v, s⟩
4  procedure scan(A)
5      initial ← collect(A)
6      previous ← initial while true do
7          s ← collect(A)
8          if s = previous then
               // Two identical collects
9              return s
10         else if ∃j : s[j].count ≥ initial[j].count + 2 do
               // Three different counts from j
11             return s[j].snapshot
12         else
               // Nothing useful, try again
13             previous ← s
```

Algorithm 20.1: Snapshot of [AAD+93] using unbounded registers

For a write operation, a process first performs a snapshot, then writes the new value, the new sequence number, and the result of the snapshot to its register (these are very wide registers). The total cost is n^2-1 read operations for the snapshot plus 1 write operation.

20.2.1 Linearizability

We now need to argue that the snapshot vectors returned by the Afek *et al.* algorithm really work, that is, that between each matching `invoke-snapshot` and `respond-snapshot` there was some actual time where the registers in the array contained precisely the values returned in the respond-snapshot action.

We do so by assigning a **linearization point** to each snapshot vector, a time at which it appears in the registers (which for correctness of the protocol had better lie within the interval between the snapshot invocation and response). For snapshots obtained through case (a), take any time between the two collects. For snapshots obtained through case (b), take the linearization point already assigned to the snapshot vector provided by the third write. In the latter case we argue by induction on termination times that the linearization point lies inside the snapshot's interval.

Note that this means that all snapshots were ultimately collected by two successive collects returning identical values, since any case-(b) snapshot sits on top of a finite regression of case-(b) snapshots that must end with a case-(a) snapshot. This means that any snapshot corresponds to an actual global state of the registers at some point in the execution, which is not true of all snapshot algorithms. It also means that we can replace the registers in the snapshot array with other objects that allow us to detect updates (say, counters or max registers) and still get snapshots.

In an actual execution, the fact that we are waiting for double collects with no intervening updates means that if there are many writers, eventually all of them will stall waiting for a case-(a) snapshot to complete. So that snapshot will complete because all the writers are stuck. In a sense, requiring writers to do snapshots first almost gives us a form of locking, but without the vulnerability to failures of a real lock.

20.2.2 Using bounded registers

The simple version of the Afek *et al.* algorithm requires unbounded registers (since sequence numbers may grow forever). One of the reasons why this algorithm required so many smart people was to get rid of this assumption: the paper describes a (rather elaborate) mechanism for recycling sequence numbers that prevents unbounded growth (see also [Lyn96, 13.3.3]). In practice, unbounded registers are probably not really an issue once one has accepted very large registers, but getting rid of them is an interesting theoretical problem.

It turns out that with a little cleverness we can drop the sequence numbers entirely. The idea is that we just need a mechanism to detect when somebody has done a lot of writes while a snapshot is in progress. A naive approach would be to have sequence numbers wrap around mod m for some small constant modulus m; this fails because if enough snapshots happen between two of my collects, I may notice none of them because all the sequence numbers wrapped around all the way. But we can augment mod-m sequence

numbers with a second handshaking mechanism that detects when a large enough number of snapshots have occurred; this acts like the guard bit on an automobile odometer, than signals when the odometer has overflowed to prevent odometer fraud by just running the odometer forward an extra million miles or so.

The result is the full version of Afek *et al.* [AAD+93]. (Our presentation here follows [AW04, 10.3].) The key mechanism for detecting odometer fraud is a **handshake**, a pair of single-writer bits used by two processes to signal each other that they have done something. Call the processes S (for *same*) and D (for *different*), and supposed we have handshake bits h_S and h_D. We then provide operations `tryHandshake` (signal that something is happening) and `checkHandshake` (check if something happened) for each process; these operations are asymmetric. The code is:

`tryHandshake`(S)**:** $h_S \leftarrow h_D$ (make the two bits the same)

`tryHandshake`(D)**:** $h_D \leftarrow \neg h_S$ (make the two bits different)

`checkHandshake`(S)**:** return $h_S \neq h_D$ (return true if D changed its bit)

`checkHandshake`(D)**:** return $h_S = h_D$ (return true if S changed its bit)

The intent is that `checkHandshake` returns true if the other process called `tryHandshake` after I did. The situation is a bit messy, however, since `tryHandshake` involves two register operations (reading the other bit and then writing my own). So in fact we have to look at the ordering of these read and write events. Let's assume that `checkHandshake` is called by S (so it returns true if and only if it sees different values). Then we have two cases:

1. `checkHandshake`(S) returns true. Then S reads a different value in h_D from the value it read during its previous call to `tryHandshake`(S). It follows that D executed a write as part of a `tryHandshake`(D) operation in between S's previous read and its current read.

2. `checkHandshake`(S) returns false. Then S reads the same value in h_D as it read previously. This does not necessarily mean that D didn't write h_D during this interval—it is possible that D is just very out of date, and did a write that didn't change the register value—but it does mean that D didn't perform both a read and a write since S's previous read.

How do we use this in a snapshot algorithm? The idea is that before performing my two collects, I will execute `tryHandshake` on my end of a

pair of handshake bits for every other process. After performing my two collects, I'll execute `checkHandshake`. I will also assume each update (after performing a snapshot) toggles a mod-2 sequence number bit on the value stored in its segment of the snapshot array. The hope is that between the toggle and the handshake, I detect any changes. (See [AW04, Algorithm 30] for the actual code.)

Does this work? Let's look at cases:

1. The toggle bit for some process q is unchanged between the two snapshots taken by p. Since the bit is toggled with each update, this means that an even number of updates to $q's$ segment occurred during the interval between p's writes. If this even number is 0, we are happy: no updates means no call to `tryHandshake` by q, which means we don't see any change in q's segment, which is good, because there wasn't any. If this even number is 2 or more, then we observe that each of these events precedes the following one:

   - p's call to `tryHandshake`.
   - p's first read.
   - q's first write.
   - q's call to `tryHandshake` at the start of its second scan.
   - q's second write.
   - p's second read.
   - p's call to `checkHandshake`.

   It follows that q both reads and writes the handshake bits in between p's calls to `tryHandshake` and `checkHandshake`, so p correctly sees that q has updated its segment.

2. The toggle bit for q has changed. Then q did an odd number of updates (i.e., at least one), and p correctly detects this fact.

What does p do with this information? Each time it sees that q has done a scan, it updates a count for q. If the count reaches 3, then p can determine that q's last scanned value is from a scan that is contained completely within the time interval of p's scan. Either this is a **direct scan**, where q actually performs two collects with no changes between them, or it's an **indirect scan**, where q got its value from some other scan completely contained within q's scan. In the first case p is immediately happy; in the second, we observe that this other scan is also contained within the interval of p's scan, and so

(after chasing down a chain of at most $n-1$ indirect scans) we eventually reach a direct scan contained within it that provided the actual value. In either case p returns the value of pair of adjacent collects with no changes between them that occurred during the execution of its scan operation, which gives us linearizability.

20.3 Faster snapshots using lattice agreement

The Afek *et al.* algorithm and its contemporaries all require $O(n^2)$ operations for each snapshot. It is possible to get this bound down to $O(n)$ using a more clever algorithm, [IMCT94] which is the best we can reasonably hope for in the worst case given that (a) even a collect (which doesn't guarantee anything about linearizability) requires $\Theta(n)$ operations when implemented in the obvious way, and (b) there is a linear lower bound, due to Jayanti, Tan, and Toueg [JTT00], on a large class of wait-free objects that includes snapshot.[3]

The first step, due to Attiya, Herlihy, and Rachman [AHR95], is a reduction to a related problem called **lattice agreement**.

20.3.1 Lattice agreement

A **lattice** is a partial order in which every pair of elements x, y has a least upper bound $x \vee y$ called the **join** of x and y and a greatest lower bound $x \wedge y$ called the **meet** of x and y. For example, we can make a lattice out of sets by letting join be union and meet be intersection; or we can make a lattice out of integers by making join be max and meet be min.

In the lattice agreement problem, each process starts with an input x_i and produces an output y_i, where both are elements of some lattice. The requirements of the problem are:

Comparability For all i, j, $y_i \leq y_j$ or $y_j \leq y_i$.

Downward validity For all i, $x_i \leq y_i$.

Upward validity For all i, $y_i \leq x_1 \vee x_2 \vee x_3 \vee \ldots \vee x_n$.

These requirements are analogous to the requirements for consensus. Comparability acts like agreement: the views returned by the lattice-agreement protocol are totally ordered. Downward validity says that each process will

[3] But see §22.6 for a faster alternative if we allow either randomization or limits on the number of times the array is updated.

include its own input in its output. Upward validity acts like validity: an output can't include anything that didn't show up in some input.

For the snapshot algorithm, we also demand **wait-freedom**: each process terminates after a bounded number of its own steps, even if other processes fail.

Note that if we are really picky, we can observe that we don't actually need meets; a **semi-lattice** that provides only joins is enough. In practice we almost always end up with a full-blown lattice, because (a) we are working with finite sets, and (b) we generally want to include a bottom element $\perp$ that is less than all the other elements, to represent the "empty" state of our data structure. But any finite join-semi-lattice with a bottom element turns out to be a lattice, since we can define $x \wedge y$ as the join of all elements z such that $z \leq x$ and $z \leq y$. We don't *use* the fact that we are in a lattice anywhere, but it does save us two syllables not to have to say "semi-lattice agreement."

20.3.2 Connection to vector clocks

The first step in reducing snapshot to lattice agreement is to have each writer generate a sequence of increasing timestamps $r_1, r_2, \ldots$, and a snapshot corresponds to some vector of timestamps $\langle t_1, t_2 \ldots t_n \rangle$, where t_i indicates the most recent write by p_i that is included in the snapshot (in other words, we are using vector clocks again; see §6.2.3). Now define $v \leq v'$ if $v_i \leq v'_i$ for all i; the resulting partial order is a lattice, and in particular we can compute $x \vee y$ by the rule $(x \vee y)_i = x_i \vee y_i$.

Suppose now that we have a bunch of snapshots that satisfy the comparability requirement. This means they are totally ordered. Then we can construct a sequential execution by ordering the snapshots in increasing order with each update operation placed before the first snapshot that includes it. This sequential execution is not necessarily a linearization of the original execution, and a single lattice agreement object won't support more than one operation for each process, but the idea is that we can nonetheless use lattice agreement objects to enforce comparability between concurrent executions of snapshot, while doing some other tricks (exploiting, among other things, the validity properties of the lattice agreement objects) to get linearizability over the full execution.

20.3.3 The full reduction

The Attiya-Herlihy-Rachman algorithm is given as Algorithm 20.2. It uses an array of registers R_i to hold round numbers (timestamps); an array S_i to hold values to scan; an unboundedly humongous array V_{ir} to hold views obtained by each process in some round; and a collection of lattice-agreement objects LA_r, one for each round.

```
1  procedure scan()
2      for attempt ← 1 to 2 do
3          R_i ← r ← max(R_1 ... R_n; R_i + 1)
4          collect ← read(S_1 ... S_n)
5          view ← LA_r(collect)
           // max computation requires a collect
6          if max(R_1 ... R_n) ≤ R_i then
7              V_ir ← view
8              return V_ir
       // finding nonempty V_jr also requires a collect
9      V_ir ← some nonempty V_jr
10     return V_ir
```

Algorithm 20.2: Lattice agreement snapshot

The algorithm makes two attempts to obtain a snapshot. In both cases, the algorithm advances to the most recent round it sees (or its previous round plus one, if nobody else has reached this round yet), attempts a collect, and then runs lattice-agreement to try to get a consistent view. If after getting its first view it finds that some other process has already advanced to a later round, it makes a second attempt at a new, higher round r' and uses some view that it obtains in this second round, either directly from lattice agreement, or (if it discovers that it has again fallen behind), it uses an indirect view from some speedier process.

The reason why I throw away my view if I find out you have advanced to a later round is not because the view is bad for me but because it's bad for you: I might have included some late values in my view that you didn't see, breaking consistency between rounds. But I don't have to do this more than once; if the same thing happens on my second attempt, I can use an indirect view as in [AAD+93], knowing that it is safe to do so because any collect that went into this indirect view started after I did.

The update operation is the usual update-and-scan procedure; for com-

pleteness this is given as Algorithm 20.3. To make it easier to reason about the algorithm, we assume that an update returns the result of the embedded scan.

```
1 procedure update_i(v)
2     S_i ← (S_i.seqno + 1, v)
3     return scan()
```

Algorithm 20.3: Update for lattice agreement snapshot

20.3.4 Why this works

We need to show three facts:

1. All views returned by the scan operation are comparable; that is, there exists a total order on the set of views (which can be extended to a total order on scan operations by breaking ties using the execution order).

2. The view returned by an update operation includes the update (this implies that future views will also include the update, giving the correct behavior for snapshot).

3. The total order on views respects the execution order: if π_1 and π_2 are scan operations that return v_1 and v_2, then $\pi_1 <_S \pi_2$ implies $v_1 \leq v_2$. (This gives us linearization.)

Let's start with comparability. First observe that any view returned is either a direct view (obtained from LA_r) or an indirect view (obtained from V_{jr} for some other process j). In the latter case, following the chain of indirect views eventually reaches some direct view. So all views returned for a given round are ultimately outputs of LA_r and thus satisfy comparability.

But what happens with views from different rounds? The lattice-agreement objects only operate within each round, so we need to ensure that any view returned in round r is included in any subsequent rounds. This is where checking round numbers after calling LA_r comes in.

Suppose some process i returns a direct view; that is, it sees no higher round number in either its first attempt or its second attempt. Then at the time it starts checking the round number in Line 6, no process has yet written a round number higher than the round number of i's view (otherwise

i would have seen it). So no process with a higher round number has yet executed the corresponding collect operation. When such a process does so, it obtains values that are at least as large as those fed into LA_r, and i's round-r view is less than or equal to the vector of these values by upward validity of LA_r, and thus less than or equal to the vector of values returned by $LA_{r'}$ for $r' > r$, by downward validity of $\mathsf{LA}_{r'}$. So we have comparability of all direct views, which implies comparability of all indirect views as well.

To show that each view returned by a scan includes any preceding update, we observe that either a process returns its first-try scan (which includes the update by downward validity) or it returns the results of a scan in the second-try round (which includes the update by downward validity in the later round, since any collect in the second-try round starts after the update occurs). So no updates are missed.

Now let's consider two scan operations π_1 and π_2 where π_1 precedes π_2 in the execution. We want to show that, for the views v_1 and v_2 that these scans return, $v_1 \leq v_2$. Pick some time between when π_1 finishes and π_2 starts, and let s be the contents of the registers at this time. Then $v_1 \leq s$ by upward validity, since any input fed to a lattice agreement object before π_1 finishes was collected from a register whose value was no greater than it is in s. Similarly, $s \leq v_2$ by downward validity, because v_2 is at least as large as the $\mathsf{collect}$ value read by π_2, and this is at least as large as s. So $v_1 \leq s \leq v_2$.

20.3.5 Implementing lattice agreement

There are several known algorithms for implementing lattice agreement, including the original algorithm of Attiya, Herlihy, and Rachman [AHR95] and an adaptive algorithm of Attiya and Fouren [AF01]. The best of them (assuming multi-writer registers) is Inoue *et al.*'s linear-time lattice agreement protocol [IMCT94].

The intuition behind this protocol is to implement lattice agreement using divide-and-conquer. The processes are organized into a tree, with each leaf in the tree corresponding to some process's input. Internal nodes of the tree hold data structures that will report increasingly large subsets of the inputs under them as they become available. At each internal node, a double-collect snapshot is used to ensure that the value stored at that node is always the union of two values that appear in its children at the same time. This is used to guarantee that, so long as each child stores an increasing sequence of sets of inputs, the parent does so also.

Each process ascends the tree updating nodes as it goes to ensure that its value is included in the final result. A clever data structure is used to

ensure that out-of-date smaller sets don't overwrite larger ones at any node, and the cost of using this data structure and carrying out the double-collect snapshot at a node with m leaves below it is shown to be $O(m)$. So the total cost of a snapshot is $O(n + n/2 + n/4 + \ldots 1) = O(n)$, giving the linear time bound.

Let's now look at the details of this protocol. There are two main components: the `Union` algorithm used to compute a new value for each node of the tree, and the `ReadSet` and `WriteSet` operations used to store the data in the node. These are both rather specialized algorithms and depend on the details of the other, so it is not trivial to describe them in isolation from each other; but with a little effort we can describe exactly what each component demands from the other, and show that it gets it.

The `Union` algorithm does the usual two-collects-without change trick to get the values of the children and then stores the result. In slightly more detail:

1. Perform `ReadSet` on both children. This returns a set of leaf values.
2. Perform `ReadSet` on both children again.
3. If the values obtained are the same in both collects, call `WriteSet` on the current node to store the union of the two sets and proceed to the parent node. Otherwise repeat the preceding step.

The requirement of the `Union` algorithm is that calling `ReadSet` on a given node returns a non-decreasing sequence of sets of values; that is, if `ReadSet` returns some set S at a particular time and later returns S', then $S \subseteq S'$. We also require that the set returned by `ReadSet` is a superset of any set written by a `WriteSet` that precedes it, and that it is equal to some such set. This last property only works if we guarantee that the values stored by `WriteSet` are all comparable (which is shown by induction on the behavior of `Union` at lower levels of the tree).

Suppose that all these conditions hold; we want to show that the values written by successive calls to `Union` are all comparable, that is, for any values S, S' written by union we have $S \subseteq S'$ or $S' \subseteq S$. Observe that $S = L \cup R$ and $S' = L' \cup R'$ where L, R and L', R' are sets read from the children. Suppose that the `Union` operation producing S completes its snapshot before the operation producing S'. Then $L \subseteq L'$ (by the induction hypothesis) and $R \subseteq R'$, giving $S \subseteq S'$.

We now show how to implement the `ReadSet` and `WriteSet` operations. The main thing we want to avoid is the possibility that some large set gets

overwritten by a smaller, older one. The solution is to have m registers $a[1 \ldots m]$, and write a set of size s to every register in $a[1 \ldots s]$ (each register gets a copy of the entire set). Because register $a[s]$ gets only sets of size s or larger, there is no possibility that our set is overwritten by a smaller one. If we are clever about how we organize this, we can guarantee that the total cost of all calls to `ReadSet` by a particular process is $O(m)$, as is the cost of the single call to `WriteSet` in `Union`.

Pseudocode for both is given as Algorithm 20.4. This is a simplified version of the original algorithm from [IMCT94], which does the writes in increasing order and thus forces readers to finish incomplete writes that they observe, as in Attiya-Bar-Noy-Dolev [ABND95] (see also Chapter 17).

```
   shared data: array a[1 ... m] of sets, initially ∅
   local data: index p, initially 0

1  procedure WriteSet(S)
2      for i ← |S| down to 1 do
3          a[i] ← S

4  procedure ReadSet()
       // update p to last nonempty position
5      while true do
6          s ← a[p]
7          if p = m or a[p + 1] = ∅ then
8              break
9          else
10             p ← p + 1
11     return s
```

Algorithm 20.4: Increasing set data structure

Naively, one might think that we could just write directly to $a[|S|]$ and skip the previous ones, but this makes it harder for a reader to detect that $a[|S|]$ is occupied. By writing all the previous registers, we make it easy to tell if there is a set of size $|S|$ or bigger in the sequence, and so a reader can start at the beginning and scan forward until it reaches an empty register, secure in the knowledge that no larger value has been written.[4] Since we

[4] This trick of reading in one direction and writing in another dates back to a paper by Lamport from 1977 [Lam77].

want to guarantee that no reader every spends more that $O(m)$ operations on an array of m registers (even if it does multiple calls to `ReadSet`), we also have it remember the last location read in each call to `ReadSet` and start there again on its next call. For `WriteSet`, because we only call it once, we don't have to be so clever, and can just have it write all $|S| \leq m$ registers.

We need to show linearizability. We'll do so by assigning a specific linearization point to each high-level operation. Linearize each call to `ReadSet` at the last time that it reads $a[p]$. Linearize each call to `WriteSet`(S) at the first time at which $a[|S|] = S$ and $a[i] \neq \emptyset$ for every $i < |S|$ (in other words, at the first time that some reader might be able to find and return S); if there is no such time, linearize the call at the time at which it returns. Since every linearization point is inside its call's interval, this gives a linearization that is consistent with the actual execution. But we have to argue that it is also consistent with a sequential execution, which means that we need to show that every `ReadSet` operation returns the largest set among those whose corresponding `WriteSet` operations are linearized earlier.

Let R be a call to `ReadSet` and W a call to `WriteSet`(S). If R returns S, then at the time that R reads S from $a[|S|]$, we have that (a) every register $a[i]$ with $i < |S|$ is non-empty (otherwise R would have stopped earlier), and (b) $|S| = m$ or $a[|S| + 1] = \emptyset$ (as otherwise R would have kept going after later reading $a[|S| + 1]$. From the rule for when `WriteSet` calls are linearized, we see that the linearization point of W precedes this time and that the linearization point of any call to `WriteSet` with a larger set follows it. So the return value of R is consistent.

The payoff: unless we do more updates than snapshots, don't want to assume multi-writer registers, are worried about unbounded space, have a beef with huge registers, or care about constant factors, it costs no more time to do a snapshot than a collect. So in theory we can get away with assuming snapshots pretty much wherever we need them.

20.4 Practical snapshots using LL/SC

Though atomic registers are enough for snapshots, it is possible to get a much more efficient snapshot algorithm using stronger synchronization primitives. An algorithm of Riany, Shavit, and Touitou [RST01] uses **load-linked/store-conditional** objects to build an atomic snapshot protocol with linear-time snapshots and constant-time updates using small registers. We'll give a sketch of this algorithm here.

The RST algorithm involves two basic ideas: the first is a snapshot

algorithm for a single scanner (i.e., only one process can do snapshots) in which each updater maintains two copies of its segment, a high copy (that may be more recent than the current scan) and a low copy (that is guaranteed to be no more recent than the current scan). The idea is that when a scan is in progress, updaters ensure that the values in memory at the start of the scan are not overwritten before the scan is completed, by copying them to the low registers, while the high registers allow new values to be written without waiting for the scan to complete. Unbounded sequence numbers, generated by the scanner, are used to tell which values are recent or not.

As long as there is only one scanner, nothing needs to be done to ensure that all scans are consistent, and indeed the single-scanner algorithm can be implemented using only atomic registers. But extending the algorithm to multiple scanners is tricky. A simple approach would be to keep a separate low register for each concurrent scan—however, this would require up to n low registers and greatly increase the cost of an update. Instead, the authors devise a mechanism, called a **coordinated collect**, that allows the scanners collectively to implement a sequence of *virtual scans* that do not overlap. Each virtual scan is implemented using the single-scanner algorithm, with its output written to a common *view* array that is protected from inconsistent updates using LL/SC operations (CAS also works). A scanner participates in virtual scans until it obtains a virtual scan that is useful to it (this means that the virtual scan has to take place entirely within the interval of the process's actual scan operation); the simplest way to arrange this is to have each scanner perform two virtual scans and return the value obtained by the second one.

The paper puts a fair bit of work into ensuring that only $O(n)$ view arrays are needed, which requires handling some extra special cases where particularly slow processes don't manage to grab a view before it is reallocated for a later virtual scan. We avoid this complication by simply assuming an unbounded collection of view arrays; see the paper for how to do this right.

A more recent paper by Fatourou and Kallimanis [FK07] gives improved time and space complexity using the same basic technique.

20.4.1 Details of the single-scanner snapshot

The single-scanner snapshot is implemented using a shared currSeq variable (incremented by the scanner but used by all processes) and an array memory of n snapshot segments, each of which is divided into a high and low component consisting of a value and a timestamp. Initially, currSeq is 0, and all memory locations are initialized to $(\perp, 0)$. This part of the algorithm does not require

LL/SC.

A call to `scan` copies the first of memory[j].high or memory[j].low that has a sequence number less than the current sequence number. Pseudocode is given as Algorithm 20.5.

```
1 procedure scan()
2     currSeq ← currSeq + 1
3     for j ← 0 to n − 1 do
4         h ← memory[j].high
5         if h.seq < currSeq then
6             view[j] ← h.value
7         else
8             view[j] ← memory[j].low.value
```

Algorithm 20.5: Single-scanner snapshot: `scan`

The `update` operation for process i cooperates by copying memory[i].high to memory[i].low if it's old.

The `update` operation always writes its value to memory[i].high, but preserves the previous value in memory[i].low if its sequence number indicates that it may have been present at the start of the most recent call to `scan`. This means that `scan` can get the old value if the new value is too recent. Pseudocode is given in Algorithm 20.6.

```
1 procedure update()
2     seq ← currSeq
3     h ← memory[i].high
4     if h.seq ≠ seq then
5         memory[i].low ← h
6     memory[i].high ← (value, seq)
```

Algorithm 20.6: Single-scanner snapshot: `update`

To show this actually works, we need to show that there is a linearization of the scans and updates that has each scan return precisely those values whose corresponding updates are linearized before it. The ordering is based on when each `scan` operation S increments currSeq and when each `update` operation U reads it; specifically:

- If U reads currSeq after S increments it, then $S < U$.

- If U reads currSeq before S increments it and S reads memory$[i]$.high (where i is the process carrying out U) before U writes it, then $S < U$.
- If U reads currSeq before S increments it, but S reads memory$[i]$.high after U writes it, then $U < S$.

Updates are ordered based on intervening scans (i.e., $U_1 < U_2$ if $U_1 < S$ and $S < U_2$ by the above rules), or by the order in which they read currSeq if there is no intervening scan.

To show this is a linearization, we need first to show that it extends the ordering between operations in the original schedule. Each of the above rules has $\pi_1 < \pi_2$ only if some low-level operation of π_1 precedes some low-level operation of π_2, with the exception of the transitive ordering of two update events with an intervening scan. But in this last case we observe that if $U_1 < S$, then U_1 writes memory$[i]$.high before S reads it, so if U_1 precedes U_2 in the actual execution, U_2 must write memory$[i]$.high after S reads it, implying $S < U_2$.

Now we show that the values returned by `scan` are consistent with the linearization ordering; that, is, for each i, `scan` copies to view$[i]$ the value in the last `update` by process i in the linearization. Examining the code for `scan`, we see that a `scan` operation S takes memory$[i]$.high if its sequence number is less than currSeq, i.e., if the `update` operation U that wrote it read currSeq before S incremented it and wrote memory$[i]$.high before S read it; this gives $U < S$. Alternatively, if `scan` takes memory$[i]$.low, then memory$[i]$.low was copied by some update operation U' from the value written to memory$[i]$.high by some update U that read currSeq before S incremented it. Here U' must have written memory$[i]$.high before S read it (otherwise S would have taken the old value left by U) and since U precedes U' (being an operation of the same process) it must therefor also have written memory$[i]$.high before S read it. So again we get the first case of the linearization ordering and $U < S$.

So far we have shown only that S obtains values that were linearized before it, but not that it ignores values that were linearized after it. So now let's consider some U with $S < U$. Then one of two cases holds:

- U reads currSeq after S increments it. Then U writes a sequence number in memory$[i]$.high that is greater than or equal to the currSeq value used by S; so S returns memory$[i]$.low instead, which can't have a sequence number equal to currSeq and thus can't be U's value either.
- U reads currSeq before S increments it but writes memory$[i]$.high after S reads it. Now S won't return U's value from memory$[i]$.high (it didn't

read it), and won't get it from memory[i].low either (because the value that *is* in memory[i].high will have seq $<$ currSeq, and so S will take that instead).

So in either case, if $S < U$, then S doesn't return U's value. This concludes the proof of correctness.

20.4.2 Extension to multiple scanners

See the paper for details.

The essential idea: view now represents a *virtual scan* view$_r$ generated cooperatively by all the scanners working together in some asynchronous round r. To avoid conflicts, we update view$_r$ using LL/SC or compare-and-swap (so that only the first scanner to write wins), and pretend that reads of memory[i] by losers didn't happen. When view$_r$ is full, start a new virtual scan and advance to the next round (and thus the next view$_{r+1}$).

20.5 Applications

Here we describe a few things we can do with snapshots.

20.5.1 Multi-writer registers from single-writer registers

One application of atomic snapshot is building multi-writer registers from single-writer registers. The idea is straightforward: to perform a write, a process does a snapshot to obtain the maximum sequence number, tags its own value with this sequence number plus one, and then writes it. A read consists of a snapshot followed by returning the value associated with the largest sequence number (breaking ties by process ID). (See [Lyn96, §13.5] for a proof that this actually works.) This requires using a snapshot that doesn't use multi-writer registers, and turns out to be overkill in practice; there are simpler algorithms that give $O(n)$ cost for reads and writes based on timestamps (see [AW04, 10.2.3]).

With additional work, it is even possible to eliminate the requirement of multi-reader registers, and get a simulation of multi-writer multi-reader registers that goes all the way down to single-writer single-read registers, or even single-writer single-reader bits. See [AW04, §§10.2.1–10.2.2] or [Lyn96, §13.4] for details.

20.5.2 Counters

Given atomic snapshots, it's easy to build a counter (supporting increment, decrement, and read operations); or, in more generality, a generalized counter (supporting increments by arbitrary amounts); or, in even more generality, an object supporting any collection of commutative and associative update operations (as long as these operations don't return anything). The idea is that each process stores in its segment the total of all operations it has performed so far, and a read operation is implemented using a snapshot followed by summing the results. This is a case where it is reasonable to consider multi-writer registers in building the snapshot implementation, because there is not necessarily any circularity in doing so.

20.5.3 Resilient snapshot objects

The previous examples can be generalized to objects with operations that either read the current state of the object but don't update it or update the state but return nothing, provided the update operations either overwrite each other (so that $Cxy = Cy$ or $Cyx = Cx$) or commute (so that $Cxy = Cyx$).

This was shown by Aspnes and Herlihy [AH90b] and improved on by Anderson and Moir [AM93] by eliminating unbounded space usage. Anderson and Moir also defined the terms **snapshot objects** for those with separate read and update operations and **resilience** for the property that all operations commute or overwrite. The basic idea underneath both of these papers is to use the multi-writer register construction given above, but break ties among operations with the same sequence numbers by first placing overwritten operations before overwriting operations and only then using process IDs.

This *almost* shows that snapshots can implement any object with consensus number 1 where update operations return nothing, because an object that is not resilient violates the commute-or-overwrite condition in some configuration has consensus number at least 2 (see §19.2.2)—in Herlihy's terminology, non-resilient objects have interfering operations. It doesn't quite work (as observed in the Anderson-Moir paper), because the tie-breaking procedure assumes a static ordering on which operations overwrite each other, so that given operations x and y where y overwrites x, y overwrites x in any configuration. But there may be objects with a *dynamic* ordering to how operations interfere, where y overwrites x in some configuration, x overwrites y in another, and perhaps even the two operations commute in yet another. This prevents us from achieving consensus, but also breaks the

tie-breaking technique. So it may be possible that there are objects with consensus number 1 and no-return updates that we still can't implement using only registers.

Chapter 21

Lower bounds on perturbable objects

Being able to do snapshots in linear time means that we can build linearizable counters, generalized counters, max registers, and so on, in linear time, by having each reader take a snapshot and combine the contributions of each updater using the appropriate commutative and associative operation. A natural question is whether we can do better by exploiting the particular features of these objects.

Unfortunately, the Jayanti-Tan-Toueg [JTT00] lower bound for **perturbable** objects says each of these objects requires $n-1$ space and $n-1$ steps for a read operation in the worst case, for any solo-terminating deterministic implementation from historyless objects. Like Burns-Lynch, this is a worst-case bound based on a covering argument, so it may be possible to evade it in some cases using either randomization or a restriction on the length of an execution (see Chapter 22).

Perturbable means that the object has a particular property that makes the proof work, essentially that the outcome of certain special executions can be changed by stuffing lots of extra update operations in the middle (see below for details).

Solo-terminating means that a process finishes its current operation in a finite number of steps if no other process takes steps in between; it is a much weaker condition, for example, than wait-freedom.

Historyless objects are those for which any operation either never changes the state (like a read, but it could be weaker) or always sets the state to a value that depends only on the operation and not the previous value (like a write, but it may also return some information about the old

state). The point of historyless objects is that covering arguments work for them: if there is a process with a pending update operations on some object, the adversary can use it at any time to wipe out the state of the object and hide any previous operations from any process except the updater (who, in a typical covering argument, is quietly killed to keep it from telling anybody what it saw).

Atomic registers are a common example of a historyless object: the read never changes the state, and the write always replaces it. **Swap objects** (with a swap operation that writes a new state while returning the old state) are the canonical example, since they can implement any other historyless object (and even have consensus number 2, showing that even extra consensus power doesn't necessarily help here). Test-and-sets (which are basically one-bit swap objects where you can only swap in 1) are also historyless. In contrast, anything that looks like a counter or similar object where the new state is a combination of the old state and the operation is *not* historyless. This is important because many of these objects turn out to be perturbable, and if they were also historyless, we'd get a contradiction.

Below is a sketch of the proof. See the original paper [JTT00] for more details.

The basic idea is to build a sequence of executions of the form $\Lambda_k\Sigma_k\pi$, where Λ_k is a preamble consisting of various complete update operations and k incomplete update operations by processes p_1 through p_{n-1}, Σ_k delivers k delayed writes from the incomplete operations in Λ_k, and π is a operation by p_n that returns some information about the object that is affected by previous operations. To make our life easier, we'll assume that π performs only read steps.[1]

We'll expand $\Lambda_k\Sigma_k$ to $\Lambda_{k+1}\Sigma_{k+1}$ by inserting new operations in between Λ_k and Σ_k, and argue that because those operations can change the value returned by π, one of them must write an object not covered in Σ_k, which will (after some more work) allow us to cover yet another object.

In order for these covered objects to keep accumulating, the reader has to keep looking at them. To a first approximation, this means that we want the first k reads done by π to be from objects written in Σ_k: since the

[1] The idea is that if π does anything else, then the return values of other steps can be simulated by doing a `read` in place of the first step and using the property of being historyless to compute the return values of subsequent steps. There is still a possible objection that we might have some historyless objects that don't even provide `read` steps. The easiest way to work around this is to assume that our objects do in fact provide a `read` step, because taking the `read` step away isn't going to make implementing the candidate perturbable object any easier.

values seen by the reader for these objects never change, the (deterministic) reader will continue to read them even as we add more operations before Σ_k. Unfortunately, this does not quite match all possible cases, because it may be that π performs useless reads of objects that aren't covered in Σ_k but that aren't written to by anybody anyway. So we have the more technical condition that π has an initial prefix that only includes covered reads and useless reads: formally, there is a prefix π' of π that includes at least one read operation of every object covered by Σ_k, such that any other read operation in π' reads an object whose state cannot be changed by any step that can be performed by any sequence of operations by processes p_1 through p_{n-1} that can be inserted between Λ_k and $\Sigma_k\pi$.

The induction hypothesis is that an execution $\Lambda_k\Sigma_k$ with these properties exists for each $k \leq n-1$.

For the base case, $\Lambda_0\Sigma_0 = \langle\rangle$. This covers 0 reads by π.

For the induction step, we start with $\Lambda_k\Sigma_k$, and look for a partial execution γ that we can insert in between Λ_k and Σ_k that changes what π returns in $\Lambda_k\gamma\Sigma_k\pi$ from what it returned in $\Lambda_k\gamma\Sigma_k$.

This is where perturbability comes in: an object is defined to be **perturbable** if such a partial execution γ always exists.

Some examples of γ:

- For a snapshot object, let γ write to a component that is not written to by any of the operations in Σ_k.
- For a max register, let γ include a bigger write than all the others.
- For a counter, let γ include at least n increments. We need n increments, because with fewer increments, we can make π return the same value by being sneaky about when the partial increments represented in Σ_k are linearized. The same choice works for a mod-m counter if m is at least $2n$, and similarly we can argue that a fetch-and-increment or fetch-and-add is perturbable by a γ that includes at least n fetch-and-increments.

In contrast, historyless objects (including atomic registers) are not perturbable: if Σ_k includes a write that sets the value of the object, no set of operations inserted before it will change this value. This is good, because we know that it only takes one atomic register to implement an atomic register.

Assuming that our object is perturbable, now we want to use the existence of γ to generate our bigger execution $\Lambda_{k+1}\Sigma_{k+1}$. As in the Burns-Lynch mutex bound [BL93], we will be arguing that γ must include a write to an

object that is not covered by the k delayed writes. Also as in Burns-Lynch, it turns out that it is not enough just to delay this particular write, because it might not cover the specific object we want.

Instead, we look for an alternative γ' that changes the value of the earliest object read by π that can be changed. We know that some such γ' exists, because γ writes to some such object, so there must be a first place in the execution of π where the output of an object can change, and there must be some γ' that makes that change. Note however that γ' that hits that earliest object need not be the same as the γ used to demonstrate perturbability, and indeed it may be that γ' is very different from γ—in particular, it may be much longer.

So now we expand $\gamma' = \alpha\beta\delta$, where β is the magic write to the uncovered object, and let $\Lambda_{k+1} = \Lambda_k\alpha\delta'$ and $\Sigma_{k+1} = \beta\Sigma_k$, where δ' consists of running all incomplete operations in α except the one that includes β to completion. We've now covered $k+1$ distinct objects in Σ_k and have no incomplete operations in Λ_{k+1} except the $k+1$ operations that cover these objects. It remains only to show that the technical condition that any uncovered object that π reads before reading all the covered objects can't have its value changed by inserting additional operations.

Suppose that there is a sequence of operations κ such that $\Lambda_{k+1}\kappa$ changes one of these forbidden uncovered objects. But $\Lambda_{k+1}\kappa = \Lambda_k\alpha\kappa$, and so $\gamma'' = \alpha\kappa$ changes an object that either (a) can't be changed because of the technical condition in the induction hypothesis for k, or (b) changes an object that π reads before the object covered by β. In the second case, this γ'' changes an earlier object that γ', contradicting the choice of γ'.

It follows that we do in fact manage to cover $k+1$ objects while satisfying the technical condition, and the induction hypothesis holds for $k+1$.

We can repeat this step until we've covered $n-1$ objects. This implies that there *are* at least $n-1$ objects (the space lower bound), and in the worst case some reader reads all of them (the step complexity lower bound).

Chapter 22

Restricted-use objects

The Jayanti-Tan-Toueg bound puts a hard floor under the worst-case complexity of almost anything interesting we'd like to implement with solo termination in a system that provides only historyless objects as primitives. As with the consensus hierarchy lower bounds, we could interpret this as a reason to demand stronger primivitives. Or we could look for ways to bypass the JTT bound.

One approach is to modify our target objects so that they are no longer perturbable. This can be done by limiting their use: a counter or max register that can only change its value a limited number of times is not perturbable, because once we hit the limit, there is no perturbing sequence of operations that we can insert between Λ_k and Σ_k in the JTT execution that changes the value returned by the eventual reader. This observation motivated a line of work on restricted-use max registers [AACH12] and restricted-use snapshots [AACHE15] that have polylogarithmic worst-case individual step complexity assuming a polynomial limit on updates. While restricted-use objects might not be all that exciting on their own, they in turn have served as building blocks for implementations of snapshots with polylogarithmic polylogarithmic amortized individual step complexity [ABHMT20].

In this chapter, we will concentrate on the original restricted-use max register construction of Aspnes, Attiya, and Censor-Hillel [AACH12], and its extension to give restricted-use snapshots by Aspnes *et al.* [AACHE15].

22.1 Max registers

We will start by implementing a restricted-use **max register** [AACH12], for which read operation returns the largest value previously written, as opposed

to the last value previously written. So after writes of 0, 3, 5, 2, 6, 11, 7, 1, 9, a read operation will return 11.

In general, max registers are perturbable objects in the sense of the Jayanti-Tan-Toueg bound, so in the worst case a max-register read will have to read at least $n-1$ distinct atomic registers, giving an $n-1$ lower bound on both step complexity and space. But we can get around this by considering bounded max registers, which only hold values in some range $0 \dots m-1$. These are not perturbable because once we hit the upper bound we can no longer insert new operations to change the value returned by a read. This allows for a much more efficient implementation (at least in terms of step complexity) when m is not too big.

22.2 Implementing bounded max registers

This implementation is from a paper by Aspnes, Attiya, and Censor-Hillel [AACH12]. The same paper shows that it is in a certain sense the only possible implementation of a wait-free restricted max register (see §22.5).

For $m = 1$, the implementation is trivial: write does nothing and read always returns 0.

For larger m, we'll show how to paste together two max registers left and right with m_0 and m_1 values together to get a max register r with $m_0 + m_1$ values. We'll think of each value stored in the max register as a bit-vector, with bit-vectors ordered lexicographically. In addition to left and right, we will need a 1-bit atomic register switch used to choose between them. The read procedure is straightforward and is shown in Algorithm 22.1; essentially we just look at switch, read the appropriate register, and prepend the value of switch to what we get.

```
1 procedure read(r)
2     if switch = 0 then
3         return 0 : read(left)
4     else
5         return 1 : read(right)
```

Algorithm 22.1: Max register read operation

For write operations, we have two somewhat asymmetrical cases depending on whether the value we are writing starts with a 0 bit or a 1 bit. These are shown in Algorithm 22.2.

```
1 procedure write(r, 0x)
2     if switch = 0 then
3         write(left, x)
4 procedure write(r, 1x)
5     write(right, x)
6     switch ← 1
```

Algorithm 22.2: Max register write operations

The intuition is that the max register is really a big tree of switch variables, and we store a particular bit-vector in the max register by setting to 1 the switches needed to make `read` follow the path corresponding to that bit-vector. The procedure for writing $0x$ tests switch first, because once switch gets set to 1, any $0x$ values are smaller than the largest value, and we don't want them getting written to left where they might confuse particularly slow readers into returning a value we can't linearize. The procedure for writing $1x$ sets switch second, because (a) it doesn't need to test switch, since $1x$ always beats $0x$, and (b) it's not safe to send a reader down into right until some value has actually been written there.

It's easy to see that `read` and `write` operations both require exactly one operation per bit of the value read or written. To show that we get linearizability, we give an explicit linearization ordering (see the paper for a full proof that this works):

1. All operations that read 0 from switch go in the first pile.
   (a) Within this pile, we sort operations using the linearization ordering for left.
2. All operations that read 1 from switch or write 1 to switch go in the second pile, which is ordered after the first pile.
   (a) Within this pile, operations that touch right are ordered using the linearization ordering for right. Operations that don't (which are the "do nothing" writes for $0x$ values) are placed consistently with the actual execution order.

To show that this gives a valid linearization, we have to argue first that any `read` operation returns the largest earlier `write` argument and that we don't put any non-concurrent operations out of order.

For the first part, any `read` in the 0 pile returns 0 : `read`(left), and `read`(left) returns (assuming left is a linearizable max register) the largest value previously written to left, which will be the largest value linearized before the `read`, or the all-0 vector if there is no such value. In either case we are happy. Any `read` in the 1 pile returns 1 : `read`(right). Here we have to guard against the possibility of getting an all-0 vector from `read`(right) if no `write` operations linearize before the `read`. But any `write` operation that writes $1x$ doesn't set switch to 1 until after it writes to right, so no `read` operation ever starts `read`(right) until after at least one `write` to right has completed, implying that that `write` to right linearizes before the `read` from right. So in all the second-pile operations linearize as well.

22.3 Encoding the set of values

If we structure our max register as a balanced tree of depth k, we are essentially encoding the values $0 \ldots 2^k - 1$ in binary, and the cost of performing a read or write operation on an m-valued register is exactly $k = \lceil \lg m \rceil$. But if we are willing to build an unbalanced tree, any **prefix code** will work.

The paper describes a method of building a max register where the cost of each operation that writes or reads a value v is $O(\log v)$. The essential idea is to build a tree consisting of a rightward path with increasingly large left subtrees hanging off of it, where each of these left subtrees is twice as big as the previous. This means that after following a path encoded as 1^k0, we hit a 2^k-valued max register. The value returned after reading some v' from this max register is $v' + (2^k - 1)$, where the $2^k - 1$ term takes into account all the values represented by earlier max registers in the chain. Formally, this is equivalent to encoding values using an **Elias gamma code** [Eli75], tweaked slightly by changing the prefixes from 0^k1 to 1^k0 to get the ordering right.

22.4 Unbounded max registers

While the unbalanced-tree construction could be used to get an unbounded max register, it is possible that read operations might not terminate (if enough writes keep setting 1 bits on the right path before the read gets to them) and for very large values the cost even of terminating reads becomes higher than what we can get out of a snapshot.

Here is the snapshot-based method: if each process writes its own contribution to the max register to a single-writer register, then we can read the max register by taking a snapshot and returning the maximum value. (It is

not hard to show that this is linearizable.) This gives an unbounded max register with read and write cost $O(n)$. So by choosing this in preference to the balanced tree when m is large, the cost of either operation on a max register is $\min(\lceil \lg m \rceil, O(n))$.

We can combine this with the unbalanced tree by terminating the right path with a snapshot-based max register. This gives a cost for reads and writes of values v of $O(\min(\log v, n))$.

22.5 Lower bound

The $\min(\lceil \lg m \rceil, O(n))$ cost of a max register read turns out to be exactly optimal, at least for the $\lceil \lg m \rceil$ part; there is a lower bound [AACH12] of $\min(\lceil \lg m \rceil, n-1)$. Intuitively, we can show by a covering argument that once some process attempts to write to a particular atomic register, then any subsequent writes convey no additional information (because they can be overwritten by the first delayed write). So in effect, no algorithm can use get more than one bit of information out of each atomic register, and any max register read ends up looking like chasing a path through a tree of switches. But as always, turning this intuition into an actual proof requires a bit more work.

We will consider solo-terminating executions in which $n-1$ writers do any number of max-register writes in some initial prefix Λ, followed by a single max-register read π by process p_n. Let $T(m,n)$ be the optimal reader cost for executions with this structure with m values, and let r be the first register read by process p_n, assuming it is running an algorithm optimized for this class of executions (we do not even require it to be correct for other executions).

We are now going split up our set of values based on which will cause a write operation to write to r. Let S_k be the set of all sequences of writes that only write values $\leq k$. Let t be the smallest value such that some execution in S_t writes to r (there must be some such t, or our reader can omit reading r, which contradicts the assumption that it is optimal).

Case 1 Since t is smallest, no execution in S_{t-1} writes to r. If we restrict writes to values $\leq t-1$, we can omit reading r, giving $T(t,n) \leq T(m,n) - 1$, from which $T(m,n) \geq T(t,n) + 1$.

Case 2 Let α be some execution in S_t that writes to r.

- Split α as $\alpha' \delta \beta$ where δ is the first write to r by some process p_i.

- Construct a new execution $\alpha'\eta$ by letting all the max-register writes except the one performing δ finish.
- Now consider any execution $\alpha'\eta\gamma\delta$, where γ is any sequence of max-register writes with values $\geq t$ that excludes p_i and p_n. Then p_n always sees the same value in r following these executions, but otherwise (starting after $\alpha'\eta$) we have an $(n-1)$-process max-register with values t through $m-1$.
- Omit the read of r again to get $T(m,n) \geq T(m-t,n-1)+1$.

We've shown the recurrence $T(m,n) \geq \min_t(\max(T(t,n),T(m-t,n)))+1$, with base cases $T(1,n)=0$ and $T(m,1)=0$. The solution to this recurrence is exactly $\min(\lceil \lg m \rceil, n-1)$, which is the same, except for a constant factor on n, as the upper bound we got by choosing between a balanced tree for small m and a snapshot for $m \geq 2^{n-1}$. For small m, the recursive split we get is also the same as in the tree-based algorithm: call the r register switch and you can extract a tree from whatever algorithm somebody gives you. So this says that the tree-based algorithm is (up to choice of the tree) essentially the unique optimal bounded max register implementation for $m \leq 2^{n-1}$.

It is also possible to show lower bounds on randomized implementations of max registers and other restricted-use objects. See [AACH12, ACAH16, HK14] for examples.

22.6 Max-register snapshots

With some tinkering, it's possible to extend the max-register construction to get an array of max registers that supports snapshots. The description in this section follows [AACHE15], with some updates to fix a bug noted in the original paper in an erratum published by the authors [AACHE18].

Formally, a **max array** is an object a that supports an operation `write`$(a[i],v)$ that sets $a[i] \leftarrow \max(v,a[i])$, and an operation `read`$(a)$ that returns a snapshot of all components of the array. The first step in building this beast is to do it for two components. The resulting **2-component max array** can then be used as a building block for larger max arrays and for fast restricted-used snapshots in general.

A $k \times \ell$ max array a is one that permits values in the range $0 \ldots k-1$ in $a[0]$ and $0 \ldots \ell - 1$ in $a[1]$. We think of $a[0]$ as the head of the max array and $a[1]$ as the tail. We'll show how to construct such an object recursively from smaller objects of the same type, analogous to the construction of an m-valued max register (which we can think of as a $m \times 1$ max array). The

idea is to split head into two pieces left and right as before, while representing tail as a master copy stored in a max register at the top of the tree plus cached copies at every internal node. These cached copies are updated by readers at times carefully chosen to ensure linearizability.

The base of the construction is an ℓ-valued max register r, used directly as a $1 \times \ell$ max array; this is the case where the head component is trivial and we only need to store $a.\mathsf{tail} = r$. Here calling $\texttt{write}(a[0], v)$ does nothing, while $\texttt{write}(a[1], v)$ maps to $\texttt{write}(r, v)$, and $\texttt{read}(a)$ returns $\langle 0, \texttt{read}(r)\rangle$.

For larger values of k, paste a $k_{\mathsf{left}} \times \ell$ max array left and a $k_{\mathsf{right}} \times \ell$ max array right together to get a $(k_{\mathsf{left}} + k_{\mathsf{right}}) \times \ell$ max array. This construction uses a switch variable as in the basic construction, along with an ℓ-valued max register tail that is used to store the value of $a[1]$.

Calls to $\texttt{write}(a[0], v)$ and $\texttt{read}(a)$ follow the structure of the corresponding operations for a simple max register, with some extra work in read to make sure that the value in tail propagates into left and right as needed to ensure the correct value is returned.

A call to $\texttt{write}(a[1], v)$ operation writes tail directly, and then calls $\texttt{read}(a)$ to propagate the new value as well.[1]

Pseudocode is given in Algorithm 22.3.

The individual step complexity of each operation is easily computed. Assuming a balanced tree, $\texttt{write}(a[0], v)$ takes exactly $\lceil \lg k \rceil$ steps, while $\texttt{write}(a[1], v)$ costs exactly $\lceil \lg \ell \rceil$ steps plus the cost of $\texttt{read}(a)$. Read operations are more complicated. In the worst case, we have two reads of $a.\mathsf{tail}$ and a write to $a.\mathsf{right}[1]$ at each level, plus up to two operations on $a.\mathsf{switch}$, for a total cost of at most $(3\lceil \lg k \rceil - 1)(\lceil \lg \ell \rceil + 2) = O(\log k \log \ell)$ steps. This dominates other costs in $\texttt{write}(a[1], v)$, so the asymptotic cost of both write and read operations is $O(\log k \log \ell)$.

In the special case where $k = \ell$, both writes and reads have their step complexities squared compared to a single-component k-valued max register.

22.6.1 Linearizability

In broad outline, the proof of linearizability follows the proof for a simple max register. But as with snapshots, we have to show that the ordering of the head and tail components are consistent.

The key observation is the following lemma.

[1] This call to $\texttt{read}(a)$ was omitted in the original published version of the algorithm [AACHE15], but was added in an erratum by the authors [AACHE18]. Without it, the implementation can violate linearizability in some executions.

```
1  procedure write(a[i], v)
2      if i = 0 then
3          if v < k_left then
4              if a.switch = 0 then
5                  write(a.left[0], v)
6          else
7              write(a.right[0], v − k_left)
8              a.switch ← 1
9      else
10         write(a.tail, v)
11         read(a)

12 procedure read(a)
13     x ← read(a.tail)
14     if a.switch = 0 then
15         write(a.left[1], x)
16         return read(a.left)
17     else
18         x ← read(a.tail)
19         write(a.right[1], x)
20         return ⟨k_left, 0⟩ + read(a.right)
```

Algorithm 22.3: Recursive construction of a 2-component max array

Lemma 22.6.1. *Fix some execution of a max array a implemented as in Algorithm 22.3. Suppose this execution contains a* `read`(a) *operation* π_{left} *that returns* v_{left} *from* $a.\mathsf{left}$ *and a* `read`(a) *operation* π_{right} *that returns* v_{right} *from* $a.\mathsf{right}$*. Then* $v_{\mathsf{left}}[1] \leq v_{\mathsf{right}}[1]$*.*

Proof. Both $v_{\mathsf{left}}[1]$ and $v_{\mathsf{right}}[1]$ are values that were previously written to their respective max arrays by `read`(a) operations (such writes necessarily exist because any process that reads $a.\mathsf{left}$ or $a.\mathsf{right}$ writes $a.\mathsf{left}[1]$ or $a.\mathsf{right}[1]$ first). From examining the code, we have that any value written to $a.\mathsf{left}[1]$ was read from $a.\mathsf{tail}$ before $a.\mathsf{switch}$ was set to 1, while any value written to $a.\mathsf{right}[1]$ was read from $a.\mathsf{tail}$ after $a.\mathsf{switch}$ was set to 1. Since max-register reads are non-decreasing, we have than any value written to $a.\mathsf{left}[1]$ is less than or equal to any value written to $a.\mathsf{right}[1]$, proving the claim. □

The rest of the proof is tedious but straightforward: we linearize the `read`(a) and `write`$(a[0])$ operations as in the max-register proof, then fit the `write`$(a[1])$ operations in based on the tail values of the reads. The full result is:

Theorem 22.6.2. *If $a.\mathsf{left}$ and $a.\mathsf{right}$ are linearizable max arrays, and $a.\mathsf{tail}$ is a linearizable max register, then Algorithm 22.3 implements a linearizable max array.*

It's worth noting that the same unbalanced-tree construction used in §§22.3 and 22.4 can be used here as well. This makes the step complexity for `read`(a) scale as $O(\log v[0] \log v[1])$, where v is the value returned. For writes the step complexity may depend in a complicated way on what values are being written and to which side, but in the worst case, it is $O(\log v[0] \log v[1])$, where v is the value in the register when the write finishes. (This is a consequence of the embedded `read`(a) in `write`$(a, 1, v)$.)

22.7 Restricted-use snapshots

To build an ordinary snapshot object from 2-component max arrays, we construct a balanced binary tree in which each leaves holds a pointer to an individual snapshot element and each internal node holds a pointer to a partial snapshot containing all of the elements in the subtree of which it is the root. The pointers themselves are non-decreasing indices into arrays of values that consist of ordinary (although possibly very wide) atomic registers.

When a process writes a new value to its component of the snapshot object, it increases the pointer value in its leaf and then propagates the new

value up the tree by combining together partial snapshots at each step, using 2-component max arrays to ensure linearizability. The resulting algorithm is similar in many ways to the lattice agreement procedure of Inoue *et al.* [IMCT94] (see §20.3.5), except that it uses a more contention-tolerant snapshot algorithm than double collects and we allow processes to update their values more than once. It is also similar to the ***f*-array** construction of Jayanti [Jay02] for efficient computation of array aggregates (sum, min, max, etc.) using LL/SC, the main difference being that because the index values are non-decreasing, max arrays can substitute for LL/SC.

Each node in the tree except the root is represented by one component of a 2-component max array that we can think of as being owned by its parent, with the other component being the node's sibling in the tree. To propagate a value up the tree, at each level the process takes a snapshot of the two children of the node and writes the sum of the indices to the node's component in its parent's max array (or to an ordinary max register if we are at the root). Before doing this last write, a process will combine the partial snapshots from the two child nodes and write the result into a separate array indexed by the sum. In this way any process that reads the node's component can obtain the corresponding partial snapshot in a single register operation. At the root this means that the cost of obtaining a complete snapshot is dominated by the cost of the max-register read, at $O(\log v)$, where v is the number of updates ever performed.

A picture of this structure, adapted from the proceedings version of [AACHE15], appears in Figure 22.1. The figure depicts an update in progress, with red values being the new values written as part of the update. Only some of the tables associated with the nodes are shown.

The cost of an update is dominated by the $O(\log n)$ max-array operations needed to propagate the new value to the root. This takes $O(\log^2 v \log n)$ steps. Here v can be taken to be the number of update operations, which controls the maximum value on either side of the 2-component max arrays.

The linearizability proof is trivial: linearize each update by the time at which a snapshot containing its value is written to the root (which necessarily occurs within the interval of the update, since we don't let an update finish until it has propagated its value to the top), and linearize reads by when they read the root. This immediately gives us an $O(\log^3 n)$ implementation—as long as we only want to use it polynomially many times—of anything we can build from snapshot, including counters, generalized counters, and (by [AH90b, AM93]) any other object whose operations all commute with or overwrite each other in a static pattern.

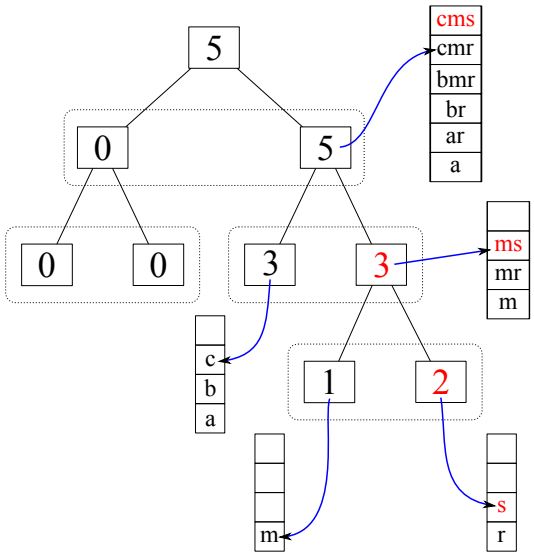


Figure 22.1: Snapshot from max arrays; taken from [AACHE15, Fig. 2]

22.7.1 Randomized and amortized snapshots

Aspnes and Censor-Hillel [ACH13] claimed to give an unrestricted, randomized snapshot with $O(\log^3 n)$ This claimed result is somewhat suspect because (a) it is based on the original, uncorrected version of the max array from [AACHE15], (b) the paper incorrectly computes the running time of the algorithm, and (c) the claim is supported by a rather rococo proof of linearizability that is dubious in various additional ways. So it is not clear that this algorithm actually works.

Fortunately, this result is largely dominated by a much less questionable result by Ahad Baig *et al.* [ABHMT20] that gives a deterministic snapshot implementation with $O(\log^3 n)$ amortized individual step complexity.

As in the restricted-use case, the Ahad Baig *et al.* snapshot assumes arbitrarily-wide registers. An alternative suggested by Bashari and Woelfel [BW21] is to implement an **adaptive partial snapshot** where a scan effectively returns a sensibly-sized index from which individual values can be extracted using a separate `observe` operation. Bashari and Woelfel show that such snapshots can be implemented in $O(\log n)$ steps using fetch-and-add and compare-and-swap primitives. Whether it is possible to improve on the $O(\log^3 n)$ bound of Ahad Baig *et al.*without using stronger primitives is still open.

Neither of these algorithms contradict the JTT lower bound: in the worst case, each will have operations that take $\Omega(n)$ steps. But the hope is that these operations are rare, and in the amortized case, paid for by many cheap operations. Also, even though we may beat JTT most of the time, other lower bounds may still apply; see for example [ACAH16, HK14].

Chapter 23

Common2

Last updated 2019. Some material may be out of date.

The **common2** class, defined by Afek, Weisberger, and Weisman [AWW93] consists of all read-modify-write objects where the modify functions either (a) all commute with each other or (b) all overwrite each other. We can think of it as the union of two simpler classes, the set of read-modify-write objects where all update operations commute, called **commuting objects** [AW99]; and the set of read-modify-write objects where all updates produce a value that doesn't depend on the previous state, called **historyless objects** [FHS98]).

From §19.2.2, we know that both commuting objects and historyless objects have consensus number at most 2, and that these objects have consensus number exactly 2 provided they supply at least one non-trivial update operation. The main result of Afek *et al.* [AWW93] is that commuting and historyless objects can all be implemented from any object with consensus number 2, even in systems with more than 2 processes. This gives a **completeness** result analogous to completeness results in complexity theory: any non-trivial common2 object can be used to implement any other common2 object.

The **common2 conjecture** was that common2 objects could also implement any object with consensus number 2, This is now known to be false [AEG16].

The main result in the paper has two parts, reflecting the two parts of the common2 class: a proof that 2-process consensus plus registers is enough to implement all commuting objects (which essentially comes down to build a generalized fetch-and-add that returns an unordered list of all preceding operations); and a proof that 2-process consensus plus registers is enough to implement all overwriting objects (which is equivalent to showing that we can

implement swap objects). The construction of the generalized fetch-and-add is pretty nasty, so we'll concentrate on the implementation of swap objects. We will also skip the swap implementation in [AWW93], and instead describe, in §§23.3 and 23.4, a simpler (though possibly less efficient) algorithm from a later paper by Afek, Morrison, and Wertheim [AMW11]. Before we do this, we'll start with some easier results from the older paper, including an implementation of n-process test-and-set from 2-process consensus. This will show that anything we can do with test-and-set we can do with any common2 object.

23.1 Test-and-set and swap for two processes

The first step is to get test-and-set.

Algorithm 23.1 shows how to turn 2-process consensus into 2-process test-and-set. The idea is that whoever wins the consensus protocol wins the test-and-set. This is linearizable, because if I run `TAS2` before you do, I win the consensus protocol by validity.

```
1 procedure TAS2()
2     if Consensus2(myId) = myId then
3         return 0
4     else
5         return 1
```

Algorithm 23.1: Building 2-process TAS from 2-process consensus

Once we have test-and-set for two processes, we can easily get one-shot swap for two processes. The trick is that a one-shot swap object always returns $\perp$ to the first process to access it and returns the other process's value to the second process. We can distinguish these two roles using test-and-set and add a register to send the value across. Pseudocode is in Algorithm 23.2.

23.2 Building n-process TAS from 2-process TAS

To turn the `TAS2` into full-blown n-process `TAS`, start by staging a tournament along the lines of [PF77] (§18.5.1.2). Each process walks up a tree of nodes, and at each node it attempts to beat every process from the other subtree using a $\texttt{TAS}_2$ object (we can't just have it fight one process, because we don't know which one process will have won the other subtree, and our $\texttt{TAS}_2$

```
1 procedure swap(v)
2     a[myId] = v
3     if TAS2() = 0 then
4         return ⊥
5     else
6         return a[¬myId]
```

Algorithm 23.2: Two-process one-shot swap from TAS

objects may only work for two specific processes). A process drops out if it ever sees a 1. We can easily show that at most one process leaves each subtree with all zeros, including the whole tree itself.

Unfortunately, this process does not give a *linearizable* test-and-set object. It is possible that p_1 loses early to p_2, but then p_3 starts (elsewhere in the tree) after p_1 finishes, and races to the top, beating out p_2. To avoid this, we can follow [AWW93] and add a gate bit that locks out latecomers.[1]

The resulting construction looks something like Algorithm 23.3. This gives a slightly different interface from straight TAS; instead of returning 0 for winning and 1 for losing, the algorithm returns $\perp$ if you win and the id of some process that beats you if you lose.[2] It's not hard to see that this gives a linearizable test-and-set after translating the values back to 0 and 1 (the trick for linearizability is that any process that wins saw an empty gate, and so started before any other process finished). It also sorts the processes into a rooted tree, with each process linearizing after its parent (this latter claim is a little trickier, but basically comes down to a loser linearizing after the process that defeated it either on gate or on one of the TAS2 objects).

This algorithm is kind of expensive: the losers that drop out early are relatively lucky, but the winning process has to win a TAS2 against everybody, for a total of $\Theta(n)$ TAS operations. We can reduce the cost to $O(\log n)$ if our TAS2 objects allow arbitrary processes to execute them. This is done, for example, in the RatRace test-and-set implementation of Alistarh *et al.* [AAG+10], using a randomized implementation of TAS2 due to Tromp and Vitányi [TV02] (see §25.5.2).

[1] The original version of this trick is from an earlier paper [AGTV92], where the gate bit is implemented as an array of single-writer registers.

[2] Note that this process may also be a loser, just one that made it further up the tree than you did. We can't expect to learn the ID of the ultimate winner, because that would solve n-process consensus.

```
1  procedure compete(i)
       // check the gate
2      if gate ≠ ⊥ then
3          return gate
4      gate ← i
       // Do tournament, returning id of whoever I lose to
5      node ← leaf for i
6      while node ≠ root do
7          for each j whose leaf is below sibling of node do
8              if TAS2(t[i, j]) = 1 then
9                  return j
10         node ← node.parent
       // I win!
11     return ⊥
```

Algorithm 23.3: Tournament algorithm with gate

23.3 Obstruction-free swap from test-and-set

We'll start by describing the "strawman algorithm" from the AMW paper. This is presented by the authors as a stepping-stone to their real algorithm, which we will describe below in §23.4.

The code is given in Algorithm 23.4. This implements a swap object that is linearizable but not wait-free.

This algorithm uses two infinite arrays s and t of test-and-set objects and an infinite array r of atomic registers. The s_i objects are essentially being used to implement a fetch-and-increment, and if we have a fetch-and-increment lying around we can replace the loop at Line 4 with an operation on that object instead. The r_i registers record values to return. The t_i registers implement a block/pass mechanism where a later process can force an earlier process to try again if it didn't record its value in time. This solves the problem of a process going to sleep after acquiring a particular slot i from the fetch-and-increment but before writing down a value that somebody else can use.

The algorithm is obstruction-free, because in any reachable configuration, only finitely many test-and-sets have been accessed, so there is some value i with $s_j = t_j = 0$ for all $j \geq i$. A process running in isolation will eventually hit one of these slots, win both test-and-sets, and return.

```
1  procedure swap(v)
2      i ← 0
3      while true do
           // Look for a starting point
4          while TAS(s_i) = 1 do
5              i ← i + 1
6          v_i ← v
           // Check if we've been blocked
7          if TAS(t_i) = 0 then
               // We win, find our predecessor
8              for j ← i − 1 down to 0 do
9                  if TAS(t_j) = 1 then
                       // Use this value
10                     return v_j
               // Didn't find anybody, we are first
11             return ⊥
12         else
               // Pick a new start and try again
```

Algorithm 23.4: Obstruction-free swap from test-and-set

For linearizability, the value of i when each operation returns gives an obvious linearization ordering. This ordering is consistent with the observed history, because if I finish with value i_1 before you start, then at the time that I finish all s_j for $j \leq i_1$ have $s_j = 1$. So you can't win any of them, and get a slot $i_2 > i_1$. But we still have to show that the return values make sense.

Consider some swap operation π.

Suppose that π starts at position i and wins every t_j down to position k, where it loses. Then no other operation wins any t_j with $k < j < i$, so there is no process that leaves with any slot between k and i. In addition, the operation π'that did win t_k must have taken slot k in Line 7, because any other process would have needed to win t_{k+1} before attempting to win t_k. So π' linearizes immediately before π, which is good, because π returns the value v_k that π' wrote before it won t_k.

Alternatively, suppose that π never loses t_j for any $j \leq i$. Then no other operation takes a slot less than i, and π linearizes first. In this case, it must return $\perp$, which it does.

23.4 Wait-free swap from test-and-set

Now we want to make the strawman algorithm wait-free. The basic idea is similar: we will have an ordered collection of test-and-set objects, and a process will move right until it can capture one that determines its place in the linearization ordering, and then it will move left to block any other processes from taking an earlier place unless they have already written out their values. To avoid starvation, we assign a disjoint collection of test-and-set objects to each operation, so that every operation eventually wins one of its own test-and-sets. Unfortunately this only works if we make the ordering dense, so that between each pair of test-and-sets there are infinitely many other test-and-sets.

AMW do this in terms of a binary tree, but I find it easier to think of the test-and-sets as being indexed by dyadic rationals strictly between 0 and 1.[3] The idea is that the i-th operation to start executing the swap object will use test-and-sets t_q where $q = k/2^i$ for all odd k in the range $1 \ldots 2^i - 1$. In order to avoid having to check the infinitely many possible values smaller than q, we will use two auxiliary objects: a readable fetch-and-increment maxDepth that hands out denominators and tracks the largest denominator used so far,

[3]The two representations are isomorphic: make each value $k/2^q$ be the parent of $k/2^q \pm 1/2^{q+1}$.

and a max register accessed that keeps track of the largest position accessed so far.

AMW implement accessed using a snapshot, which we will do as well to avoid complications from trying to build a max register out of an infinitely deep tree.[4] Note that AMW don't call this data structure a max register, but we will, because we like max registers.

Code for the swap procedure is given in Algorithm 23.5.

To show Algorithm 23.5 works, we need the following technical lemma, which, among other things, implies that node $1 - 2^{\mathsf{depth}}$ is always available to be captured by the process at depth depth. This is essentially just a restatement of Lemma 1 from [AMW11].

Lemma 23.4.1. *For any $x = k/2^q$, where k is odd, no process attempts to capture any $y \in [x, x + 1/2^q)$ before some process writes x to* accessed*.*

Proof. Suppose that the lemma fails, let $y = \ell/2^r$ be the first node captured in violation of the lemma, and let $x = k/2^q$ be such that $y \in [x, x + 1/2^q)$ but x has not been written to accessed when y is captured. Let p be the process that captures y.

Now consider $y' = x - 1/2^r$, the last node to the left of x at the same depth as y. Why didn't p capture y'?

One possibility is that some other process p' blocked y' during its return phase. This p' must have captured a node $z > y'$. If $z > y$, then p' would have blocked y first, preventing p from capturing it. So $y' < z < y$.

The other possibility is that p never tried to capture y', because some other process p' wrote some value $z > y'$ to accessed first. This value z must also be less than y (or else p would not have tried to capture y).

In both cases, there is a process p' that captures a value z with $y' < z < y$, before p captures y and thus before anybody writes x to accessed.

Since $y' < x$ and $y' < z$, either $y' < z < x$ or $y' < x < z$. In the first case, $z \in [y', y' + 1/2^r)$ is captured before y' is written to accessed. In the second case $z \in [x, x + 1/2^q)$ is captured before x is written to accessed. Either way, y is not the first capture to violate the lemma, contradicting our initial assumption. □

Using Lemma 23.4.1, it is straightforward to show that Algorithm 23.5 is wait-free. If I get q for my value of depth, then no process will attempt to

[4] The issue is not so much that we can't store arbitrary dyadics, since we can encode them using an order-preserving prefix-free code, but that, without some sort of helping mechanism, a read running concurrently with endlessly increasing writes (e.g. $1/2, 3/4, 7/8, \ldots$) might not be wait-free. Plus as soon as the denominator exceeds 2^n, which happens after only n calls to `swap`, $O(n)$-step snapshots are cheaper anyway.

```
1  procedure swap(v)
       // Pick a new row just for me
2      depth ← fetchAndIncrement(maxDepth)
       // Capture phase
3      repeat
           // Pick leftmost node in my row greater than accessed
4          cap ← min{x | x = k/2^depth for odd k, x > accessed}
           // Post my value
5          reg[cap] ← v
           // Try to capture the test-and-set
6          win ← TAS(tst[cap]) = 0
7          writeMax(accessed, cap)
8      until win
       // Return phase
       // Max depth reached by anybody left of cap
9      maxPreviousDepth ← read(maxDepth)
10     ret ← cap
       // Block previous nodes until we find one we can take
11     repeat
12         ret ← max{x = k/2^q | q ≤ maxPreviousDepth, k odd, x < ret}
13         if ret < 0 then
14             return ⊥
15     until TAS(tst[ret]) = 1
16     return reg[ret]
```

Algorithm 23.5: Wait-free swap from test-and-set [AMW11]

capture any y in $[1-2^q, 1)$ before I write $1-2^q$ to accessed. But this means that nobody can block me from capturing $1-2^q$, because processes can only block values smaller than the one they already captured. I also can't get stuck in the return phase, because there are only finitely many values with denominator less than $2^{\mathsf{maxPreviousDepth}}$.

It remains to show that the implementation is linearizable. The obvious linearization ordering is given by sorting each operation i by its captured node cap. Linearizability requires then that if we imagine a directed graph containing an edge ij for each pair of operations i and j such that i captures cap_i and returns $\mathsf{reg}[\mathsf{cap}_j]$, then this graph forms a path that corresponds to this linearization ordering.

Since each process only returns one value, it trivially holds that each node in the graph has out-degree at most 1. For the in-degree, suppose that we have operations i, j, and k with $\mathsf{cap}_i < \mathsf{cap}_j < \mathsf{cap}_k$ such that j and k both return $\mathsf{reg}[\mathsf{cap}_i]$. Before k reaches $\mathsf{tst}[\mathsf{cap}_i]$, it must first capture all the test-and-sets between cap_i and cap_k that have depth less than or equal to $\mathsf{maxPreviousDepth}_k$. This will include $\mathsf{tst}[\mathsf{cap}_j]$, because j must write to maxDepth before doing anything, and this must occur before k starts the return phase if j sees a value of accessed that is less that cap_k.

A similar argument show that there is at most one process that returns $\perp$; this implies that there is at most one process with out-degree 0.

So now we have a directed graph where every process has in-degree and out-degree at most one, which implies that each weakly-connected component will be a path. But each component will also have exactly one terminal node with out-degree 0. Since there is only one such node, there is only one component, and the entire graph is a single path. This concludes the proof of linearizability.

23.5 Implementations using stronger base objects

The terrible step complexity of known wait-free implementations of Common2 objects like `swap` or `fetchAndIncrement` from 2-process consensus objects and registers has led to work on finding better implementations assuming stronger base objects. Using load-linked/store-conditional, Ellen and Woelfel [EW13] provide implementations of several Common2 objects, including `fetchAndIncrement`, `fetchAndAdd`, and `swap` that all have $O(\log n)$ individual step complexity.[5] This is know to be optimal due to a lower bound

[5] What they actually implement is the ability to do fetch-and-f, where f is any binary associative function, using an object they call an **aggregator**. Each of these objects is

of Jayanti [Jay98].

The lower bound applies *a fortiori* to the case where we don't have LL/SC or CAS and have to rely on 2-process consensus objects. But it's not out of the question that there is a matching upper bound in this case.

obtained by choosing an appropriate f.

Chapter 24

Randomized consensus and test-and-set

We've seen that we can't solve **consensus** in an asynchronous system message-passing or shared-memory system with one crash failure [FLP85, LAA87], but that the problem becomes solvable using failure detectors [CT96]. An alternative that also allows us to solve consensus is to allow the processes to use randomization, by providing each process with a **local coin** that can generate random values that are immediately visible only to that process. The resulting **randomized consensus** problem replaces the **termination** requirement with **probabilistic termination**: all processes terminate with probability 1. The agreement and validity requirements remain the same.

In this chapter, we will describe how randomization interacts with the adversary, give a bit of history of randomized consensus, and then concentrate on recent algorithms for randomized consensus and the closely-related problem of randomized test-and-set. Much of the material in this chapter is adapted from notes for a previous course on randomized algorithms [Asp11] and a few of my own papers [Asp12b, AE11, Asp12a].

24.1 Role of the adversary in randomized algorithms

Because randomized processes are unpredictable, we need to become a little more sophisticated in our handling of the adversary. As in previous asynchronous protocols, we assume that the adversary has control over timing, which we model by allowing the adversary to choose at each step which process performs the next operation. But now the adversary may do

so based on knowledge of the state of the protocol and its past evolution. How much knowledge we give the adversary affects its power. Several classes of adversaries have been considered in the literature; ranging from strongest to weakest, we have:

1. An **adaptive adversary**. This adversary is a function from the state of the system to the set of processes; it can see everything that has happened so far (including coin-flips internal to processes that have not yet been revealed to anybody else), but can't predict the future. It's known that an adaptive adversary can force any randomized consensus protocol to take $\Theta(n^2)$ total steps [AC08]. The adaptive adversary is also called a **strong adversary** following a foundational paper of Abrahamson [Abr88].

2. An **intermediate adversary** or **weak adversary** [Abr88] is one that limits the adversary's ability to observe or control the system in some way, without completely eliminating it. For example, a **content-oblivious adversary** [Cha96] or **value-oblivious adversary** [Aum97] is restricted from seeing the values contained in registers or pending write operations and from observing the internal states of processes directly. A **location-oblivious adversary** [Asp12b] can distinguish between values and the types of pending operations, but can't discriminate between pending operations based one which register they are operating on. These classes of adversaries are modeled by imposing an equivalence relation on partial executions and insisting that the adversary make the same choice of processes to go next in equivalent situations. Typically they arise because somebody invented a consensus protocol for the oblivious adversary (described below) and then looked for the next most powerful adversary that still let the protocol work.

   Weak adversaries often allow much faster consensus protocols than adaptive adversaries. Each of the above adversaries permits consensus to be achieved in $O(\log n)$ expected individual work using an appropriate algorithm. But from a mathematical standpoint, weak adversaries are a bit messy, and once you start combining algorithms designed for different weak adversaries, it's natural to move all the way down to the weakest reasonable adversary, the oblivious adversary.

3. A **oblivious adversary** has no ability to observe the system at all; instead, it fixes a sequence of process IDs in advance, and at each step the next process in the sequence runs.

> We will describe below a protocol that guarantees $O(\log \log n)$ expected individual work for an oblivious adversary. It is not known whether this is optimal; in fact, is is consistent with the best known lower bound (due to Attiya and Censor [AC08]) that consensus can be solved in $O(1)$ expected individual steps against an oblivious adversary.

Each of these adversaries is defined based on choosing steps of particular objects, with particular constraints on knowledge based on the states of those objects. This interacts badly with abstractions like linearizability: an adversary might be able to play games with the internals of an implementation of an object that allows it more power than it would have with an actual sequential version of the object. So even though linearizable implementations are indistinguishable from sequential objects for deterministic protocols, for randomized protocols they can give very different results for both adaptive and oblivious adversaries [GHW11]; and in the specific case of consensus, it can be shown that there are randomized consensus protocols that terminate with probability 1 against an adaptive adversary when implemented with atomic registers, but fail to terminate with nonzero probability when implemented using an arbitrary linearizable implementation [HHT20].

These results don't necessarily imply the failure of any specific consensus protocol implemented using a specific atomic register simulation, but they do justify suspicion. The easiest way to deal with this suspicion is to assume that our atomic registers are, in fact, atomic, so that's what we will do here.

24.2 History

The use of randomization to solve consensus in an asynchronous system with crash failures was proposed by Ben-Or [BO83] for a message-passing model. Chor, Israeli, and Li [CIL94] gave the first wait-free consensus protocol for a shared-memory system, which assumed a particular kind of weak adversary. Abrahamson [Abr88] defined strong and weak adversaries and gave the first wait-free consensus protocol for a strong adversary; its expected step complexity was $\Theta\left(2^{n^2}\right)$. After failing to show that exponential time was necessary, Aspnes and Herlihy [AH90a] showed how to do consensus in $O(n^4)$ total step complexity, a value that was soon reduced to $O(n^2 \log n)$ by Bracha and Rachman [BR91]. This remained the best known bound for the strong-adversary model until Attiya and Censor [AC08] showed matching $\Theta(n^2)$ upper and lower bounds on total step complexity. A later paper by Aspnes and Censor [AC09] showed that it was also possible to get an $O(n)$ bound on individual step complexity.

For weak adversaries, the best known upper bound on individual step complexity was $O(\log n)$ for a long time [Cha96, Aum97, Asp12b], with an $O(n)$ bound on total step complexity for some models [Asp12b]. More recent work has lowered the individual step complexity bound to $O(\log \log n)$, under the assumption of an oblivious adversary [Asp12a]. No non-trivial lower bound on expected individual step complexity is known, although there is a known lower bound on the distribution of the individual step complexity [ACH10].

In the following sections, we will concentrate on the more recent weak-adversary algorithms. These have the advantage of being fast enough that one might reasonably consider using them in practice, assuming that the weak-adversary assumption does not create trouble, and they are also require less probabilistic machinery to analyze than the strong-adversary algorithms.

24.3 Reduction to simpler primitives

To show how to solve consensus using randomization, it helps to split the problem in two: we will first see how to detect *when* we've achieved agreement, and then look at *how* to achieve agreement.

24.3.1 Adopt-commit objects

Most known randomized consensus protocols have a round-based structure that alternates between generating and detecting agreement. Gafni [Gaf98] proposed **adopt-commit protocols** as a tool for detecting agreement, and these protocols were later abstracted as **adopt-commit objects** [MRRT08, AGGT09]. The version described here is largely taken from [AE11], which shows bounds on the complexity of adopt-commit objects.

An adopt-commit object supports a single operation, `AdoptCommit`(u), where u is an input from a set of m **values**. The result of this operation is an output of the form (commit, v) or (adopt, v), where the second component is a value from this set and the first component is a **decision bit** that indicates whether the process should decide value v immediately or adopt it as its preferred value in later rounds of the protocol.

The requirements for an adopt-commit object are the usual requirements of validity and termination, plus:

1. **Coherence.** If the output of some operation is (commit, v), then every output is either (adopt, v) or (commit, v).

2. **Convergence.** If all inputs are v, all outputs are (commit, v).

These last two requirement replace the agreement property of consensus. They are also strictly weaker than consensus, which means that a consensus object (with all its output labeled commit) is also an adopt-commit object.

The reason we like adopt-commit objects is that they allow the simple consensus protocol shown in Algorithm 24.1.

```
1 preference ← input
2 for r ← 1 . . . ∞ do
3     (b, preference) ← AdoptCommit(AC[r], preference)
4     if b = commit then
5         return preference
6     else
7         do something to generate a new preference
```

Algorithm 24.1: Consensus using adopt-commit

The idea is that the adopt-commit takes care of ensuring that once somebody returns a value (after receiving commit), everybody else who doesn't return adopts the same value (follows from coherence). Conversely, if everybody already has the same value, everybody returns it (follows from convergence). The only missing piece is the part where we try to shake all the processes into agreement. For this we need a separate object called a *conciliator*.

24.3.2 Conciliators

Conciliators are a weakened version of randomized consensus that replace agreement with **probabilistic agreement**: the processes can disagree sometimes, but must agree with constant probability despite interference by the adversary. An algorithm that satisfies termination, validity, and probabilistic agreement is called a **conciliator**.[1]

The important feature of conciliators is that if we plug a conciliator that guarantees agreement with probability at least δ into Algorithm 24.1, then on average we only have to execute the loop $1/\delta$ times before every process agrees. This gives an expected cost equal to $1/\delta$ times the total cost of `AdoptCommit` and the conciliator. Typically we will aim for constant δ.

[1] Warning: This name has not really caught on in the general theory-of-distributed-computing community, and so far only appears in papers that have a particular researcher as a co-author [Asp12a, AE11, Asp12b, AACV17]. Unfortunately, there doesn't seem to be a better name for the same object that has caught on. So we are stuck with it for now.

24.4 Implementing an adopt-commit object

What's nice about adopt-commit objects is that they can be implemented deterministically. Here we'll give a simple adopt-commit object for two values, 0 and 1. Optimal (under certain assumptions) constructions of m-valued adopt-commits can be found in [AE11].

Pseudocode is given in Algorithm 24.2.

```
   shared data: a[0], a[1], initially 0; proposal, initially ⊥
 1 procedure AdoptCommit(v)
 2     a[v] ← 1
 3     if proposal = ⊥ then
 4         proposal ← v
 5     else
 6         v ← proposal
 7     if a[¬v] = 0 then
 8         return (commit, v)
 9     else
10         return (adopt, v)
```

Algorithm 24.2: A 2-valued adopt-commit object

Structurally, this is pretty similar to a splitter (see §18.5.2), except that we use values instead of process IDs.

We now show correctness. Termination and validity are trivial. For coherence, observe that if I return (commit, v) I must have read $a[\neg v] = 0$ before any process with $\neg v$ writes $a[\neg v]$; it follows that all such processes will see $\mathsf{proposal} \neq \bot$ and return (adopt, v). For convergence, observe that if all processes have the same input v, they all write it to $\mathsf{proposal}$ and all observe $a[\neg v] = 0$, causing them all to return (commit, v).

24.5 Conciliators and shared coins

For an adaptive adversary, the usual way to implement a conciliator is from a **weak shared coin** [AH90a], which is basically a non-cryptographic version of the **common coin** [Rab83] found in many cryptographic Byzantine agreement protocols. Formally, a weak shared coin is an object that has no inputs and returns either 0 or 1 to all processes with some minimum probability δ. By itself this does not give validity, so converting a weak

shared coin into a conciliator requires extra machinery to bypass the coin if the processes that have accessed the conciliator so far are all in agreement; see Algorithm 24.3. The intuition is that having some processes (who all agree with each other) skip the shared coin is not a problem, because with probability δ the remaining processes will agree with them as well.

```
   shared data:
     binary registers r_0 and r_1, initially 0;
     weak shared coin sharedCoin
1  procedure coinConciliator()
2      r_v ← 1
3      if r_¬v = 1 then
4          return sharedCoin()
5      else
6          return v
```

Algorithm 24.3: Shared coin conciliator from [Asp12b]

This still leaves the problem of how to build a shared coin. In the message-passing literature, the usual approach is to use cryptography,[2] but because we are assuming an arbitrarily powerful adversary, we can't use cryptography.

If we don't care how small δ gets, we could just have each process flip its own local coin and hope that they all come up the same. (This is more or less what was done by Abrahamson [Abr88].) But that might take a while. If we aren't willing to wait exponentially long, a better approach is to combine many individual local coins using some sort of voting.

A version of this approach, based on a random walk, was used by Aspnes and Herlihy [AH90a] to get consensus in (bad) polynomial expected time against an adaptive adversary. A better version was developed by Bracha and Rachman [BR91]. In their version, each process repeatedly generates a random ± 1 vote and adds it to a common pool (which just means writing the sum and count of all its votes so far out to a single-writer register). Every $\Theta(n/\log n)$ votes, the process does a collect (giving an overhead of $\Theta(\log n)$ operations per vote) and checks to see if the total number of votes is greater than a $\Theta(n^2)$ threshold. If it is, the process returns the sign of the total vote.

Bracha and Rachman showed that despite processes seeing different combinations of votes (due to the collects running at possibly very different

[2] For example, Canetti and Rabin [CR93] solved Byzantine agreement in $O(1)$ time by building a shared coin on top of secret sharing.

speeds), the difference between what each process sees and the actual sum of all votes ever generated is at most $O(n)$ with high probability. This means that if the total vote is more than cn from 0 for some c, which occurs with constant probability, then every processes is likely to return the same value. This gives a weak shared coin with constant bias, and thus also a consensus protocol, that runs in $O(n^2 \log n)$ expected total steps.

This remained the best known protocol for many years, leaving an annoying gap between the upper bound and the best known lower bound of $\Omega(n^2/\log^2 n)$ [Asp98]. Eventually, Attiya and Censor [AC08] produced an entirely new argument to bring the lower bound up to $\Omega(n^2)$ and at the same time gave a simple tweak to the Bracha-Rachman protocol to bring the upper bound down to $O(n^2)$, completely settling (up to constant factors) the asymptotic expected total step complexity of strong-adversary consensus. But the question of how quickly one could solve weak-adversary adversary consensus remained (and still remains) open.

24.6 A one-register conciliator for an oblivious adversary

```
   shared data: register r, initially ⊥
1  k ← 0
2  while r = ⊥ do
3      with probability 2^k/(2n) do
4          write v to r
5      else
6          do a dummy operation
7      k ← k + 1
8  return r
```

Algorithm 24.4: Impatient first-mover conciliator from [Asp12b]

Algorithm 24.4 implements a conciliator for an oblivious adversary[3] using a single register. This particular construction is taken from [Asp12b], and is based on an earlier algorithm of Chor, Israeli, and Li [CIL94]. The cost of this algorithm is expected $O(n)$ total work and $O(\log n)$ individual work. Later (§24.7.2), we will see a different algorithm [Asp12a] that reduces the

[3] Or any adversary weak enough not to be able to block the write based on how the coin-flip turned out.

individual work to $O(\log \log n)$, although the total work for that algorithm may be $O(n \log \log n)$.

The basic idea is that processes alternate between reading a register r and (maybe) writing to the register; if a process reads a non-null value from the register, it returns it. Any other process that reads the same non-null value will agree with the first process; the only way that this can't happen is if some process writes a different value to the register before it notices the first write.

The random choice of whether to write the register or not avoids this problem. The idea is that even though the adversary can schedule a write at a particular time, because it's oblivious, it won't be able to tell if the process wrote (or was about to write) or did a no-op instead.

The basic version of this algorithm, due to Chor, Israeli, and Li [CIL94], uses a fixed $\frac{1}{2n}$ probability of writing to the register. So once some process writes to the register, the chance that any of the remaining $n-1$ processes write to it before noticing that it's non-null is at most $\frac{n-1}{2n} < 1/2$. It's also not hard to see that this algorithm uses $O(n)$ total operations, although it may be that one single process running by itself has to go through the loop $2n$ times before it finally writes the register and escapes.

Using increasing probabilities avoids this problem, because any process that executes the main loop $\lceil \lg n \rceil + 1$ times will write the register. This establishes the $O(\log n)$ per-process bound on operations. At the same time, an $O(n)$ bound on total operations still holds, since each write has at least a $\frac{1}{2n}$ chance of succeeding. The price we pay for the improvement is that we increase the chance that an initial value written to the register gets overwritten by some high-probability write. But the intuition is that the probabilities can't grow too much, because the probability that I write on my next write is close to the sum of the probabilities that I wrote on my previous writes—suggesting that if I have a high probability of writing next time, I should have done a write already.

Formalizing this intuition requires a little bit of work. Fix the schedule, and let p_i be the probability that the i-th write operation in this schedule succeeds. Let t be the least value for which $\sum_{i=1}^{t} p_i \geq 1/4$. We're going to argue that with constant probability one of the first t writes succeeds, and that the next $n-1$ writes by different processes all fail.

The probability that none of the first t writes succeed is

$$\begin{aligned}\prod_{i=1}^{t}(1-p_i) &\leq \prod_{i=1}^{t} e^{-p_i} \\ &= \exp\left(\sum_{i=1}^{t} p_i\right) \\ &\leq e^{-1/4}.\end{aligned}$$

Now observe that if some process p writes at or before the t-th write, then any process q with a pending write either did no writes previously, or its last write was among the first $t-1$ writes, whose probabilities sum to less than $1/4$. In either case, q has a $\sum_{i\in S_q} p_i + \frac{1}{2n}$ chance of writing on its pending attempt, where S_q is the set of indices in $1 \ldots t-1$ where q previously attempted to write.

Summing up these probabilities over all processes gives a total of $\frac{n-1}{2n} + \sum_q \sum_{i\in S_q} p_i \leq 1/2 + 1/4 = 3/4$. So with probability at least $e^{-1/4}(1-3/4) = e^{-1/4}/4$, we get agreement.

24.7 Sifters

A faster conciliator can be obtained using a **sifter**, which is a mechanism for rapidly discarding processes using randomization [AA11] while keeping at least one process around. The simplest sifter has each process either write a register (with low probability) or read it (with high probability); all writers and all readers that see $\perp$ continue to the next stage of the protocol, while all readers who see a non-null value drop out. If the probability of writing is tuned carefully, this will reduce n processes to at most $2\sqrt{n}$ processes on average; by iterating this mechanism, the expected number of remaining processes can be reduced to $1+\epsilon$ after $O(\log\log n + \log(1/\epsilon))$ phases.

As with previous implementations of test-and-set (see Algorithm 23.3), it's often helpful to have a sifter return not only that a process lost but which process it lost to. This gives the implementation shown in Algorithm 24.5.

To use a sifter effectively, p should be tuned to match the number of processes that are likely to use it. This is because of the following lemma:

Lemma 24.7.1. *Fix p, and let X processes executed a sifter with parameter p. Let Y be the number of processes for which the sifter returns $\perp$. Then*

$$\mathrm{E}\left[X \mid Y\right] \leq pX + \frac{1}{p}. \tag{24.7.1}$$

```
1 procedure sifter(p, r)
2     with probability p do
3         r ← id
4         return ⊥
5     else
6         return r
```

Algorithm 24.5: A sifter

Proof. In order to return $\perp$, a process must either (a) write to r, which occurs with probability p, or (b) read r before any other process writes to it. The expected number of writers, conditioned on X, is exactly pX. The expected number of readers before the first write has a geometric distribution truncated by X. Removing the truncation gives exactly $\frac{1}{p}$ expected readers, which is an upper bound on the correct value. □

For n initial processes, the choice of p that minimizes the bound in (24.7.1) is $\frac{1}{\sqrt{n}}$, giving at most $2\sqrt{n}$ expected survivors. Iterating this process with optimal p at each step gives a sequence of at most n, $2\sqrt{n}$, $2\sqrt{2\sqrt{n}}$, etc., expected survivors after each sifter. The twos are a little annoying, but a straightforward induction bounds the expected survivors after i rounds by $4 \cdot n^{2^{-i}}$. In particular, we get at most 8 expected survivors after $\lceil \lg \lg n \rceil$ rounds.

At this point it makes sense to switch to a fixed p and a different analysis. For $p = 1/2$, the first process to access r always survives, and each subsequent process survives with probability at most $3/4$ (because it leaves if the first process writes and it reads). So the number of "excess" processes drops as $(3/4)^i$, and an additional $\lceil \log_{4/3}(7/\epsilon) \rceil$ rounds are enough to reduce the expected number of survivors from $1 + 7$ to $1 + \epsilon$ for any fixed ϵ.[4]

It follows that

Theorem 24.7.2. *An initial set of n processes can be reduced to 1 with probability at least $1 - \epsilon$ using $O(\log \log n + \log(1/\epsilon))$ rounds of sifters.*

Proof. Let X be the number of survivors after $\lceil \lg \lg n \rceil + \lceil \log_{4/3}(7/\epsilon) \rceil$ rounds of sifters, with probabilities tuned as described above. We've shown that $\mathrm{E}[X] \leq 1 + \epsilon$, so $\mathrm{E}[X - 1] \leq \epsilon$. Since $X - 1 \geq 0$, from Markov's inequality we have $\Pr[X \geq 2] = \Pr[X - 1 \geq 1] \leq \mathrm{E}[X - 1]/1 \leq \epsilon$. □

[4] This argument essentially follows the proof of [Asp12a, Theorem 2], which, because of neglecting to subtract off a 1 at one point, ends up with $8/\epsilon$ instead of $7/\epsilon$.

24.7.1 Test-and-set using sifters

Sifters were initially designed to be used for test-and-set. For this purpose, we treat a return value of $\perp$ as "keep going" and anything else as "leave with value 1." Using $O(\log \log n)$ rounds of sifters, we can get down to one process that hasn't left with probability at least $1 - \log^{-c} n$ for any fixed constant c. We then need a fall-back TAS to handle the $\log^{-c} n$ chance that we get more than one such survivor.

Alistarh and Aspnes [AA11] used the `RatRace` algorithm of Alistarh *et al.* [AAG+10] for this purpose. This is an adaptive randomized test-and-set built from splitters and two-process consensus objects that runs in $O(\log k)$ expected time, where k is the number of processes that access the test-and-set; a sketch of this algorithm is given in §25.5.2. If we want to avoid appealing to this algorithm, a somewhat simpler approach is to use an approach similar to the Lamport's fast-path mutual exclusion algorithm (described in §18.5.2): any process that survives the sifters tries to rush to a two-process TAS at the top of a tree of two-processes TASes by winning a splitter, and if it doesn't win the splitter, it enters at a leaf and pays $O(\log n)$ expected steps. By setting $\epsilon = 1/\log n$, the overall expected cost of this final stage is $O(1)$.

This algorithm does not guarantee linearizability. I might lose a sifter early on only to have a later process win all the sifters (say, by writing to each one) and return 0. A `gate` bit as in Algorithm 23.3 solves this problem. The full code is given in Algorithm 24.6.

24.7.2 Consensus using sifters

With some trickery, the sifter mechanism can be adapted to solve consensus, still in $O(\log \log n)$ expected individual work [Asp12a]. The main difficulty is that a process can no longer drop out as soon as it knows that it lost: it still needs to figure out who won, and possible help that winner over the finish line.

The basic idea is that when a process p loses a sifter to some other process q, p will act like a clone of q from that point on. In order to make this work, each process writes down at the start of the protocol all of the coin-flips it intends to use to decide whether to read or write at each round of sifting. Together with its input, these coin-flips make up the process's **persona**. In analyzing the progress of the sifter, we count surviving personae (with multiple copies of the same persona counting as one) instead of surviving processes.

Pseudocode for this algorithm is given in Algorithm 24.7. Note that the

```
1  if gate ≠ ⊥ then
2      return 1
3  else
4      gate ← myId
5      for i ← 1 . . . ⌈log log n⌉ + ⌈log_{4/3}(7 log n)⌉ do
6          with probability min(1/2, 2^{1−2^{−i+1}}) do
7              r_i ← myId
8          else
9              w ← r_i
10             if w ≠ ⊥ then
11                 return 1
12 if splitter() = stop then
13     return 0
14 else
15     return AWWTAS()
```

Algorithm 24.6: Test-and-set in $O(\log \log n)$ expected time

loop body is essentially the same as the code in Algorithm 24.5, except that the random choice is replaced by a lookup in persona.chooseWrite.

To show that this works, we need to argue that having multiple copies of a persona around doesn't change the behavior of the sifter. In each round, we will call the first process with a given persona p to access r_i the **representative** of p, and argue that a persona survives round i in this algorithm precisely when its representative would survive round i in a corresponding test-and-set sifter with the schedule restricted only to the representatives.

There are three cases:

1. The representative of p writes. Then at least one copy of p survives.

2. The representative of p reads a null value. Again at least one copy of p survives.

3. The representative of p reads a non-null value. Then no copy of p survives: all subsequent reads by processes carrying p also read a non-null value and discard p, and since no process with p writes, no other process adopts p.

```
1  procedure conciliator(input)
2      Let R = ⌈log log n⌉ + ⌈log_{4/3}(7/ε)⌉
3      Let chooseWrite be a vector of R independent random Boolean
       variables with Pr[chooseWrite[i] = 1] = p_i, where
       p_i = 2^{1−2^{−i+1}} (n)^{−2^{−i}} for i ≤ ⌈log log n⌉ and p_i = 1/2 for larger i.
4      persona ← ⟨input, chooseWrite, myId⟩
5      for i ← 1 . . . R do
6          if persona.chooseWrite[i] = 1 then
7              r_i ← persona
8          else
9              v ← r_i
10             if v ≠ ⊥ then
11                 persona ← v
12     return persona.input
```

Algorithm 24.7: Sifting conciliator (from [Asp12a])

From the preceding analysis for test-and-set, we have that after $O(\log\log n + \log 1/\epsilon)$ rounds with appropriate probabilities of writing, at most $1+\epsilon$ values survive on average. This gives a probability of at most ϵ of disagreement. By alternating these conciliators with adopt-commit objects, we get agreement in $O(\log\log n + \log m/\log\log m)$ expected time, where m is the number of possible input values.

I don't think the $O(\log\log n)$ part of this expression is optimal, but I don't know how to do better.

24.7.3 A better sifter for test-and-set

A more sophisticated sifter due to Giakkoupis and Woelfel [GW12a] removes all but $O(\log n)$ processes, on average, using two operations for each process. Iterating this sifter reduces the expected survivors to $O(1)$ in $O(\log^* n)$ rounds. A particularly nice feature of the Giakkoupis-Woelfel algorithm is that (if you don't care about space) it doesn't have any parameters that require tuning to n: this means that exactly the same structure can be used in each round. An unfortunate feature is that it's not possible to guarantee that every process that leaves learns the identity of a process that stays: this means that it can't adapted into a consensus protocol using the persona trick described in §24.7.2.

Pseudocode is given in Algorithm 24.8. In this simplified version, we assume an infinitely long array $A[1\dots]$, so that we don't need to worry about n. Truncating the array at $\log n$ also works, but the analysis requires handling the last position as a special case, which I am too lazy to do here.

```
1 Choose r ∈ ℤ⁺ such that Pr[r = i] = 2^{-i}
2 A[r] ← 1
3 if A[r + 1] = 0 then
4 |  stay
5 else
6 |  leave
```

Algorithm 24.8: Giakkoupis-Woelfel sifter [GW12a]

Lemma 24.7.3. *In any execution of Algorithm 24.8 with an oblivious adversary and n processes, at least one process stays, and the expected number of processes that stay is $O(\log n)$.*

Proof. For the first part, observe that any process that picks the largest value of r among all processes will survive; since the number of processes is finite, there is at least one such survivor.

For the second part, let X_i be the number of survivors with $r = i$. Then $\mathrm{E}[X_i]$ is bounded by $n \cdot 2^{-i}$, since no process survives with $r = i$ without first choosing $r = i$. But we can also argue that $\mathrm{E}[X_i] \leq 3$ for any value of n, by considering the sequence of write operations in the execution.

Because the adversary is oblivious, the location of these writes is uncorrelated with their ordering. If we assume that the adversary is trying to maximize the number of survivors, its best strategy is to allow each process to read immediately after writing, as delaying this read can only increase the probability that $A[r + 1]$ is nonzero. So in computing X_i, we are counting the number of writes to $A[i]$ before the first write to $A[i + 1]$. Let's ignore all writes to other registers; then the j-th write to either of $A[i]$ or $A[i + 1]$ has a conditional probability of $2/3$ of landing on $A[i]$ and $1/3$ on $A[i + 1]$. We are thus looking at a geometric distribution with parameter $1/3$, which has expectation 3.

Combining these two bounds gives $\mathrm{E}[X_i] \leq \min(3, 2^{-i})$. So then

$$\begin{aligned}\mathrm{E}[\text{survivors}] &\leq \sum_{i=1}^{\infty} \min(3, n \cdot 2^{-i}) \\ &= 3 \lg n + O(1),\end{aligned}$$

because once $n \cdot 2^{-i}$ drops below 3, the remaining terms form a geometric series. □

Like square root, logarithm is concave, so Jensen's inequality applies here as well. So $O(\log^* n)$ rounds of Algorithm 24.8 reduces us to an expected constant number of survivors, which can then be fed to RatRace.

With an adaptive adversary, all of the sifter-based test-and-sets fail badly: in this particular case, an adaptive adversary can sort the processes in order of increasing write location so that every process survives. The best known n-process test-and-set for an adaptive adversary is still a tree of 2-process randomized test-and-sets, as in the Afek *et al.* [AWW93] algorithm described in §23.2. Whether $O(\log n)$ expected steps is in fact necessary is still open (as is the exact complexity of test-and-set with an oblivious adversary).

24.8 Space bounds

A classic result of Fich, Herlihy, and Shavit [FHS98] showed that $\Omega(\sqrt{n})$ registers are needed to solve consensus even under the very weak requirement of **nondeterministic solo termination**, which says that for every reachable configuration and every process p, there exists some continuation of the execution in which the protocol terminates with only p running. The best known upper bound is the trivial bound of n—one single-writer register per process—since any algorithm that uses multi-writer registers can be translated into one that uses only single-writer registers, and (assuming wide enough registers) multiple registers of a single process can be combined into one.

For many years, there was very little progress in closing the gap between these two bounds. In 2013, we got a hint that FHS might be tight when Giakkoupis *et al.* [GHHW13] gave a surprising $O(\sqrt{n})$-space algorithm for the closely related problem of obstruction-free one-shot test-and-set.

But then Gelashvili [Gel15] showed an $n/20$ space lower bound for consensus for anonymous processes, and Zhu quickly followed this with a lower bound for non-anonymous processes [Zhu16], showing that at least $n - 1$ registers are required, using a clever combination of bivalence and covering arguments. Around the same time, Giakkoupis *et al.* [GHHW15] further improved the space complexity of obstruction-free test-and-set to $O(\log n)$, using a deterministic obstruction-free implementation of a sifter. So the brief coincidence of the $\Omega(\sqrt{n})$ lower bound on consensus and the $O(\sqrt{n})$ upper bound on test-and-set turned out to be an accident.

For consensus, there is still a gap, but it's a very small gap. Whether the actual space needed is $n - 1$ or n remains open.

Chapter 25

Renaming

Last updated 2022. Some material may be out of date.

We will start by following the presentation in [AW04, §16.3]. This mostly describes results of the original paper of Attiya *et al.* [ABND+90] that defined the renaming problem and gave a solution for message-passing; however, it's now more common to treat renaming in the context of shared-memory, so we will follow Attiya and Welch's translation of these results to a shared-memory setting.

25.1 Renaming

In the **renaming** problem, we have n processes, each starts with a name from some huge namespace, and we'd like to assign them each unique names from a much smaller namespace. The main application is allowing us to run algorithms that assume that the processes are given contiguous numbers, e.g., the various collect or atomic snapshot algorithms in which each process is assigned a unique register and we have to read all of the registers. With renaming, instead of reading a huge pile of registers in order to find the few that are actually used, we can map the processes down to a much smaller set.

Formally, we have a decision problem where each process has input x_i (its original name) and output y_i, with the requirements:

Termination Every nonfaulty process eventually decides.

Uniqueness If $p_i \neq p_j$, then $y_i \neq y_j$.

Anonymity The code executed by any process depends only on its input x_i: for any execution of processes $p_1 \dots p_n$ with inputs $x_1 \dots x_n$, and any permutation π of $[1 \dots n]$, there is a corresponding execution of processes $p_{\pi(1)} \dots p_{\pi(n)}$ with inputs $x_1 \dots x_n$ in which $p_{\pi(i)}$ performs exactly the same operations as p_i and obtains the same output y_i.

The last condition is like non-triviality for consensus: it excludes algorithms where p_i just returns i in all executions. Typically we do not have to do much to prove anonymity other than observing that all processes are running the same code.

We will be considering renaming in a shared-memory system, where we only have atomic registers to work with.

25.2 Performance

Conventions on counting processes:

- N = number of possible original names.
- n = maximum number of processes.
- k = number of processes that actually execute the algorithm.

Ideally, we'd like any performance measures we get to depend on k alone if possible (giving an **adaptive** algorithm). Next best would be something polynomial in n and k. Anything involving N is bad.

We'd also like to minimize the size of the output namespace. How well we can do this depends on what assumptions we make. For deterministic algorithms using only read-write registers, a lower bound due to Herlihy and Shavit [HS99] shows that we can't get fewer than $2n - 1$ names for general n.[1] Our target thus will be exactly $2n - 1$ output names if possible, or $2k - 1$ if we are trying to be adaptive. For randomized algorithms, it is possible to solve **strong** or **tight** renaming, where the size of the namespace is exactly k; we'll see how to do this in §25.5.

A small note on bounds: There is a lot of variation in the literature on how bounds on the size of the output namespace are stated. The original Herlihy-Shavit lower bound [HS99] says that there is no general renaming

[1] This lower bound was further refined by Castañeda and Rajsbaum [CR08], who show that $2n - 2$ (but no less!) is possible for certain special values of n; all of these lower bounds make extensive use of combinatorial topology, so we won't try to present them here.

algorithm that uses $2n$ names for $n+1$ processes; in other words, any n-process algorithm uses at least $2n-1$ names. Many subsequent papers discussing lower bounds on the namespace follow the approach of Herlihy and Shavit and quote lower bounds that are generally 2 higher than the minimum number of names needed for n processes. This requires a certain amount of translation when comparing these lower bounds with upper bounds, which use the more natural convention.

25.3 Order-preserving renaming

Before we jump into upper bounds, let's do an easy lower bound from the Attiya *et al.* paper [ABND+90]. This bound works on a variant of renaming called **order-preserving renaming**, where we require that $y_i < y_j$ whenever $x_i < x_j$. Unfortunately, this requires a very large output namespace: with t failures, any asynchronous algorithm for order-preserving renaming requires $2^t(n-t+1)-1$ possible output names. This lower bound applies regardless of the model, as long as some processes may start after other processes have already been assigned names.

For the wait-free case, we have $t = n-1$, and the bound becomes just $2^n - 1$. This is a simpler case than the general t-failure case, but the essential idea is the same: if I've only seen a few of the processes, I need to leave room for the others.

Theorem 25.3.1. *There is no order-preserving renaming algorithm for n processes using fewer than $2^n - 1$ names.*

Proof. By induction on n. For $n = 1$, we use $2^1 - 1 = 1$ names; this is the base case. For larger n, suppose we use m names, and consider an execution in which one process p_n runs to completion first. This consumes one name y_n and leaves k names less than y_n and $m-k-1$ names greater than y_n. By setting all the inputs x_i for $i < n$ either less than x_n or greater than x_n, we can force the remaining processes to choose from the remaining k or $m-k-1$ names. Applying the induction hypothesis, this gives $k \geq 2^{n-1} - 1$ and $m-k-1 \geq 2^{n-1}-1$, so $m = k+(m-k-1)+1 \geq 2(2^{n-1}-1)+1 = 2^n-1$. □

25.4 Deterministic renaming

In **deterministic renaming**, we can't use randomization, and may or may not have any primitives stronger than atomic registers. With just atomic registers, we can only solve loose renaming; with test-and-set, we can solve

tight renaming. In this section, we describe some basic algorithms for deterministic renaming.

25.4.1 Wait-free renaming with $2n - 1$ names

Here we use Algorithm 55 from [AW04], which is an adaptation to shared memory of the message-passing renaming algorithm of [ABND+90]. One odd feature of the algorithm is that, as written, it is not anonymous: processes communicate using an atomic snapshot object and use their process IDs to select which component of the snapshot array to write to. But if we think of the process IDs used in the algorithm as the inputs x_i rather than the actual process IDs i, then everything works. The version given in Algorithm 25.1 makes this substitution explicit, by treating the original name i as the input.

```
1  procedure getName()
2      s ← 1
3      while true do
4          a[i] ← s
5          view ← snapshot(a)
6          if view[j] = s for some j then
7              r ← |{j : view[j] ≠ ⊥ ∧ j ≤ i}|
8              s ← r-th positive integer not in
                   {view[j] : j ≠ i ∧ view[j] = ⊥}
9          else
10             return s
```

Algorithm 25.1: Wait-free deterministic renaming

The array a holds proposed names for each process (indexed by the original names), or $\perp$ for processes that have not proposed a name yet. If a process proposes a name and finds that no other process has proposed the same name, it takes it; otherwise it chooses a new name by first computing its rank r among the active processes and then choosing the r-th smallest name that hasn't been proposed by another process. Because the rank is at most n and there are at most $n - 1$ names proposed by the other processes, this always gives proposed names in the range $[1 \dots 2n - 1]$. But it remains to show that the algorithm satisfies uniqueness and termination.

For uniqueness, consider two process with original names i and j. Suppose that i and j both decide on s. Then i sees a view in which $a[i] = s$ and $a[j] \neq s$, after which it no longer updates $a[i]$. Similarly, j sees a view in

which $a[j] = s$ and $a[i] \neq s$, after which it no longer updates $a[j]$. If i's view is obtained first, then j can't see $a[i] \neq s$, but the same holds if j's view is obtained first. So in either case we get a contradiction, proving uniqueness.

Termination is a bit trickier. Here we argue that no process can run forever without picking a name, by showing that if we have a set of processes that are doing this, the one with smallest original name eventually picks a name. More formally, call a process *trying* if it runs for infinitely many steps without choosing a name. Then in any execution with at least one trying process, eventually we reach a configuration where all processes have either finished or are trying. In some subsequent configuration, all the processes have written to the a array at least once; from this point on, the set of non-null positions in a—and thus the rank each process computes for itself—is stable.

Starting from some such stable configuration, look at the trying process i with the smallest original name, and suppose it has rank r. Let $F = \{z_1 < z_2 \dots\}$ be the set of "free names" that are not proposed in a by any of the finished processes. Observe that no trying process $j \neq i$ ever proposes a name in $\{z_1 \dots z_r\}$, because any such process has rank greater than r. This leaves z_r open for i to claim, provided the other names in $\{z_1 \dots z_r\}$ eventually become free. But this will happen, because only trying processes may have proposed these names (early on in the execution, when the finished processes hadn't finished yet), and the trying processes eventually propose new names that are not in this range. So eventually process i proposes z_r, sees no conflict, and finishes, contradicting the assumption that it is trying.

Note that we haven't proved any complexity bounds on this algorithm at all, but we know that the snapshot alone takes at least $\Omega(N)$ time and space. Brodksy *et al.* [BEW11] cite a paper of Bar-Noy and Dolev [BND89] as giving a shared-memory version of [ABND+90] with complexity $O(n \cdot 4^n)$; they also give algorithms and pointers to algorithms with much better complexity.

25.4.2 Long-lived renaming

In **long-lived renaming** a process can release a name for later use by other processes (or the same process, if it happens to run choose-name again). Now the bound on the number of names needed is $2k - 1$, where k is the maximum number of concurrently active processes. Algorithm 25.1 can be converted to a long-lived renaming algorithm by adding the `releaseName` procedure given in Algorithm 25.2. This just erases the process's proposed name, so that some other process can claim it.

Here the termination requirement is weakened slightly, to say that some

```
1 procedure releaseName()
2   a[i] ← ⊥
```

Algorithm 25.2: Releasing a name

process always makes progress in `getName`. It may be, however, that there is some process that never successfully obtains a name, because it keeps getting stepped on by other processes zipping in and out of `getName` and `releaseName`.

25.4.3 Renaming without snapshots

Moir and Anderson [MA95] give a renaming protocol that is somewhat easier to understand and doesn't require taking snapshots over huge arrays. A downside is that the basic version requires $k(k+1)/2$ names to handle k active processes.

25.4.3.1 Splitters

The Moir-Anderson renaming protocol uses a network of **splitters**, which we last saw providing a fast path for mutual exclusion in §18.5.2. Each splitter is a widget, built from a pair of atomic registers, that assigns to each processes that arrives at it the value right, down, or stop. As discussed previously, the useful properties of splitters are that if at least one process arrives at a splitter, then (a) at least one process returns right or stop; and (b) at least one process returns down or stop; (c) at most one process returns stop; and (d) any process that runs by itself returns stop.

We proved the last two properties in §18.5.2; we'll prove the first two here. Another way of describing these properties is that of all the processes that arrive at a splitter, some process doesn't go down and some process doesn't go right. By arranging splitters in a grid, this property guarantees that every row or column that gets at least one process gets to keep it—which means that with k processes, no process reaches row $k+1$ or column $k+1$.

Algorithm 25.3 gives the implementation of a splitter (it's identical to Algorithm 18.6, but it will be convenient to have another copy here).

Lemma 25.4.1. *If at least one process completes the splitter, at least one process returns* stop *or* right*.*

Proof. Suppose no process returns right; then every process sees open in door, which means that every process writes its ID to race before any process

```
   shared data:
 1 atomic register race, big enough to hold an ID, initially ⊥
 2 atomic register door, big enough to hold a bit, initially open
 3 procedure splitter(id)
 4     race ← id
 5     if door = closed then
 6         return right
 7     door ← closed
 8     if race = id then
 9         return stop
10     else
11         return down
```

Algorithm 25.3: Implementation of a splitter

closes the door. Some process writes its ID last: this process will see its own ID in race and return stop. □

Lemma 25.4.2. *If at least one process completes the splitter, at least one process returns* stop *or* down*.*

Proof. First observe that if no process ever writes to door, then no process completes the splitter, because the only way a process can finish the splitter without writing to door is if it sees closed when it reads door (which must have been written by some other process). So if at least one process finishes, at least one process writes to door. Let p be any such process. From the code, having written door, it has already passed up the chance to return right; thus it either returns stop or down. □

25.4.3.2 Splitters in a grid

Now build an m-by-m triangular grid of splitters, arranged as rows $0 \ldots m-1$ and columns $0 \ldots m-1$, where a splitter appears in each position (r, c) with $r+c \leq m-1$ (see Figure 25.1 for an example; this figure is taken from [Asp10]). Assign a distinct name to each of the $\binom{m}{2}$ splitters in this grid. To obtain a name, a process starts at $(r, c) = (0, 0)$, and repeatedly executes the splitter at its current position (r, c). If the splitter returns right, it moves to $(r, c+1)$; if down, it moves to $(r+1, c)$; if stop, it stops, and returns the name of its current splitter. This gives each name to at most one process

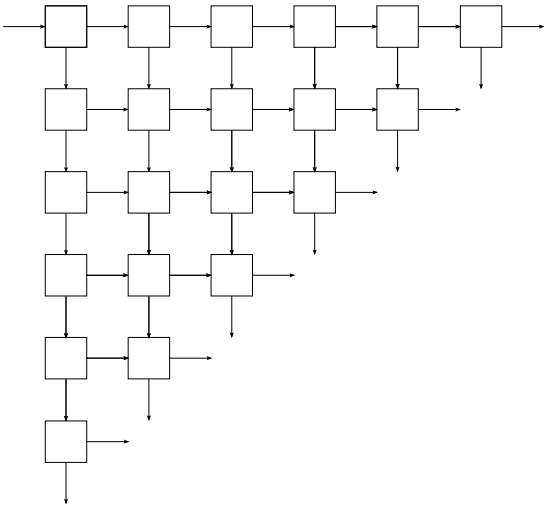


Figure 25.1: A 6×6 Moir-Anderson grid

(by Lemma 18.5.3); we also have to show that if at most m processes enter the grid, every process stops at some splitter.

The argument for this is simple. Suppose some process p leaves the grid on one of the $2m$ output wires. Look at the path it takes to get there (see Figure 25.2, also taken from [Asp10]). Each splitter on this path must handle at least two processes (or p would have stopped at that splitter, by Lemma 18.5.4). So some other process leaves on the other output wire, either right or down. If we draw a path from each of these wires that continues right or down to the end of the grid, then at every step along this path we either have a process stop or continue in this same direction as long as there is a process left to do so. This means that on each of these m disjoint paths, either some splitter stops a process, or some process reaches a final output wire, each of which is at a distinct splitter. But this gives m distinct processes in addition to p, for a total of $m + 1$ processes. It follows that:

Theorem 25.4.3. *An $m \times m$ Moir-Anderson grid solves renaming for up to m processes.*

The time complexity of the algorithm is $O(m)$: Each process spends at most 4 operations on each splitter, and no process goes through more than $2m$ splitters. In general, any splitter network will take at least n steps to stop n processes, because the adversary can run them all together in a horde that drops only one process at each splitter.

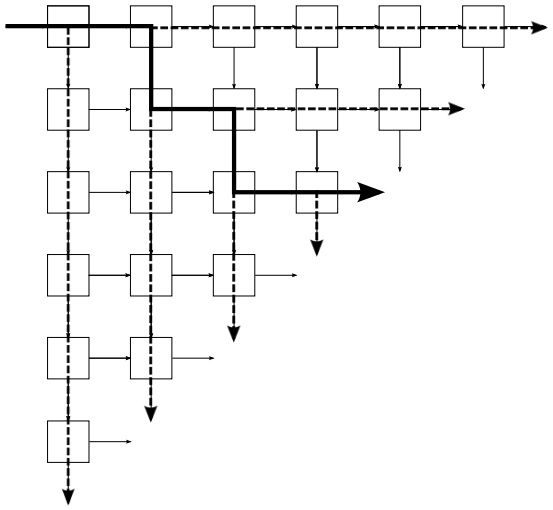


Figure 25.2: Path taken by a single process through a 6×6 Moir-Anderson grid (heavy path), and the 6 disjoint paths it spawns (dashed paths). (From [Asp10].)

If we don't know k in advance, we can still guarantee names of size $O(k^2)$ by carefully arranging them so that each k-by-k subgrid contains the first $\binom{k}{2}$ names. This gives an adaptive renaming algorithm (although the namespace size is pretty high). We still have to choose our grid to be large enough for the largest k we might actually encounter; the resulting space complexity is $O(n^2)$.

With a slightly more clever arrangement of the splitters, it is possible to reduce the space complexity to $O(n^{3/2})$ [Asp10]. Whether further reductions are possible is an open problem. Note however that linear time complexity makes splitter networks uncompetitive with much faster randomized algorithms (as we'll see in §25.5), so this may not be a very important open problem.

25.4.4 Getting to $2n - 1$ names in polynomial space

From before, we have an algorithm that will get $2n - 1$ names for n processes out of N possible processes when run using $O(N)$ space (for the enormous snapshots). To turn this into a bounded-space algorithm, run Moir-Anderson first to get down to $\Theta(k^2)$ names, then run the previous algorithm (in $\Theta(n^2)$ space) using these new names as the original names.

Since we didn't prove anything about time complexity of the humongous-

snapshot algorithm, we can't say much about the time complexity of this combined one. Moir and Anderson suggest instead using an $O(Nk^2)$ algorithm of Borowsky and Gafni to get $O(k^4)$ time for the combined algorithm.

This is close to the best known: a later paper by Afek and Merritt [AM99] holds the current record for deterministic adaptive renaming into $2k - 1$ names at $O(k^2)$ individual steps. On the lower bound side, it is known that $\Omega(k)$ is a lower bound on the individual steps of any renaming protocol with a polynomial output namespace [AAGG11].

25.4.5 Renaming with test-and-set

Moir and Anderson give a simple renaming algorithm based on test-and-set that is **strong** (k processes are assigned exactly the names $1 \dots k$), **adaptive** (the time complexity to acquire a name is $O(k)$), and **long-lived**, which means that a process can release its name and the name will be available to processes that arrive later. In fact, the resulting algorithm gives **long-lived strong renaming**, meaning that the set of names in use will always be no larger than the set of processes that have started to acquire a name and not yet finished releasing one; this is a little stronger than just saying that the algorithm is strong and that it is long-lived separately.

The algorithm is simple: we have a line of test-and-set bits $T[1] \dots T[n]$. To acquire a name, a process starts at $T[1]$ and attempts to win each test-and-set until it succeeds; whichever $T[i]$ it wins gives it name i. To release a name, a process releases the test-and-set.

Without the releases, the same mechanism gives fetch-and-increment [AWW93]. Fetch-and-increment by itself solves tight renaming (although not long-lived renaming, since there is no way to release a name).

25.5 Randomized renaming

With randomization, we can beat both the $2k - 1$ lower bound on the size of the output namespace from [HS99] and the $\Omega(k)$ lower bound on individual work from [AAGG11], achieving strong renaming with $O(\log k)$ expected individual work [AACH+11].

The basic idea is that we can use randomization for **load balancing**, where we avoid the problem of having an army of processes marching together with only a few peeling off at a time (as in splitter networks) by having the processes split up based on random choices. For example, if each process generates a random name consisting of $2\lceil \lg n \rceil$ bits, then it is reasonably likely that every process gets a unique name in a namespace of size $O(n^2)$

(we can't hope for less than $O(n^2)$ because of the **birthday paradox**). But we want all processes to be guaranteed to have unique names, so we need some more machinery.

We also need the processes to have initial names; if they don't, there is always some nonzero probability that two identical processes will flip their coins in exactly the same way and end up with the same name. This observation was formalized by Buhrman, Panconesi, Silvestri, and Vitányi [BPSV06].

25.5.1 Randomized splitters

Attiya *et al.* [AKP+06] suggested the use of **randomized splitters** in the context of another problem (**adaptive collect**) that is closely related to renaming.

A randomized splitter is just like a regular splitter, except that if a process doesn't stop it flips a coin to decide whether to go right or down. Randomized splitters are nice because they usually split better than deterministic splitters: if k processes reach a randomized splitter, with high probability no more than $k/2 + O(\sqrt{k \log k})$ will leave on either output wire.

It's not hard to show that a binary tree of these things of depth $2\lceil \lg n \rceil$ stops all but a constant expected number of processes on average;[2] processes that don't stop can be dropped into a backup renaming algorithm (Moir-Anderson, for example) with only a constant increase in expected individual work.

Furthermore, the binary tree of randomized splitters is adaptive; if only k processes show up, we only need $O(\log k)$ levels levels on average to split them up. This gives renaming into a namespace with expected size $O(k^2)$ in $O(\log k)$ expected individual steps.

25.5.2 Randomized test-and-set plus sampling

Subsequent work by Alistarh *et al.* [AAG+10] showed how some of the same ideas could be used to get strong renaming, where the output namespace has size exactly n (note this is not adaptive; another result in the same paper gives adaptive renaming, but it's not strong). There are two pieces to this result: an implementation of randomized test-and-set called `RatRace`, and a sampling procedure for getting names called `ReShuffle`.

[2] The proof is to consider the expected number of pairs of processes that flip their coins the same way for all $2\lceil \lg n \rceil$ steps. This is at most $\binom{n}{2} n^{-2} < 1/2$, so on average at most 1 process escapes the tree, giving (by symmetry) at most a $1/n$ chance that any particular process escapes. Making the tree deeper can give any polynomial fraction of escapees while still keeping $O(\log n)$ layers.

The `RatRace` protocol implements a randomized test-and-set with $O(\log k)$ expected individual work. The essential idea is to use a tree of randomized splitters to assign names, then have processes walk back up the same tree attempting to win a 3-process randomized test-and-set at each node (there are 3 processes, because in addition to the winners of each subtree, we may also have a process that stopped on that node in the renaming step); this test-and-set is just a very small binary tree of 2-process test-and-sets implemented using the algorithm of Tromp and Vitányi [TV02]. A `gate` bit is added at the top as in the test-and-set protocol of Afek *et al.* [AGTV92] to get linearizability.

Once we have test-and-set, we could get strong renaming using a linear array of test-and-sets as suggested by Moir and Anderson [MA95], but it's more efficient to use the randomization to spread the processes out. In the `ReShuffle` protocol, each process chooses a name in the range $[1 \dots n]$ uniformly at random, and attempts to win a test-and-set guarding that name. If it doesn't work, it tries again. Alistarh *et al.* show that this method produces unique names for everybody in $O(n \log^4 n)$ total steps with high probability. The individual step complexity of this algorithm, however, is not very good: there is likely to be some unlucky process that needs $\Omega(n)$ probes (at an expected cost of $\Theta(\log n)$ steps each) to find an empty slot.

25.5.3 Renaming with sorting networks

A later paper by Alistarh *et al.* [AACH+11] reduces the cost of renaming still further, getting $O(\log k)$ expected individual step complexity for acquiring a name. The resulting algorithm is both adaptive and strong: with k processes, only names 1 through k are used. We'll describe the non-adaptive version here.

The basic idea is to build a **sorting network** out of test-and-sets; the resulting structure, called a **renaming network**, routes each process through a sequence of test-and-sets to a unique output wire. Unlike a splitter network, a renaming network uses the stronger properties of test-and-set to guarantee that (once the dust settles) only the lowest-numbered output wires are chosen; this gives strong renaming.

25.5.3.1 Sorting networks

A sorting network is a kind of parallel sorting algorithm that proceeds in synchronous rounds, where in each round the elements of an array at certain fixed positions are paired off and swapped if they are out of order. The

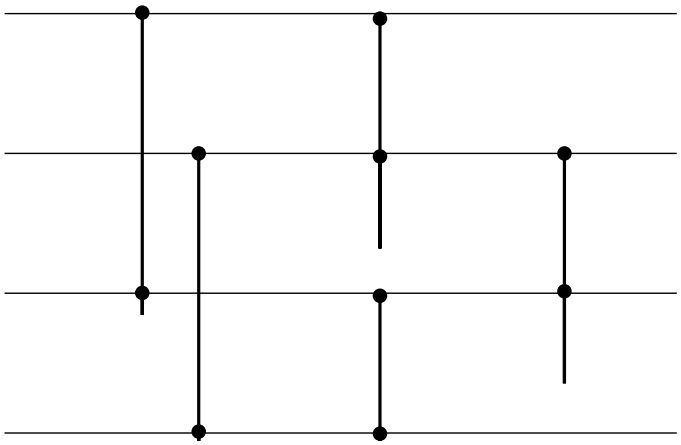


Figure 25.3: A sorting network

difference between a sorting network and a standard comparison-based sort is that the choice of which positions to compare at each step is static, and doesn't depend on the outcome of previous comparisons; also, the only effect of a comparison is possibly swapping the two values that were compared.

Sorting networks are drawn as in Figure 25.3. Each horizontal line or **wire** corresponds to a position in the array. The vertical lines are **comparators** that compare two values coming in from the left and swap the larger value to the bottom. A network of comparators is a sorting network if the sequences of output values is always sorted no matter what the order of values on the inputs is.

The **depth** of a sorting network is the maximum number of comparators on any path from an input to an output. The **width** is the number of wires; equivalently, the number of values the network can sort. The sorting network in Figure 25.3 has depth 3 and width 4.

Explicit constructions of sorting networks with width n and depth $O(\log^2 n)$ are known [Bat68]. It is also known that sorting networks with depth $O(\log n)$ exist [AKS83], but no explicit construction of such a network is known.

25.5.3.2 Renaming networks

To turn a sorting network into a renaming network, we replace the comparators with test-and-set bits, and allow processes to walk through the network asynchronously. This is similar to an earlier mechanism called a **counting**

network [AHS94], which used certain special classes of sorting networks as counters, but here any sorting network works.

Each process starts on a separate input wire, and we maintain the invariant that at most one process ever traverses a wire. It follows that each test-and-set bit is only used by two processes. The first process to reach the test-and-set bit is sent out the lower output, while the second is sent out the upper output. If we imagine each process that participates in the protocol as a one and each process that doesn't as a zero, the test-and-set bit acts as a comparator: if no processes show up on either input (two zeros), no processes leave (two zeros again); if processes show up on both inputs (two ones), processes leave on both (two ones again); and if only one process ever shows up (a zero and a one), it leaves on the bottom output (zero and one, sorted). Because the original sorting network sorts all the ones to the bottom output wires, the corresponding renaming network sorts all the processes that arrive to the bottom outputs. Label these outputs starting at 1 at the bottom to get renaming.

Since each test-and-set involves at most two processes, we can carry them out in $O(1)$ expected register operations using, for example, the protocol of Tromp and Vitányi [TV02]. The expected cost for a process to acquire a name is then $O(\log n)$ (using an AKS [AKS83] sorting network). A more complicated construction in the Alistarh *et al.* paper shows how to make this adaptive, giving an expected cost of $O(\log k)$ instead.

The problem with using an AKS network is that the AKS result is non-constructive: what Ajtai, Komlós, and Szemerédi show is that there is a particular randomized construction of candidate sorting networks that succeeds in producing a correct sorting network with nonzero (but very small) probability. Other disturbing features of this result are that we have no efficient way to test candidate sorting networks (determining if a network of comparators is in fact a sorting network is co-**NP**-hard), and the constant in the big-O for AKS is quite spectacularly huge. So it probably makes more sense to think of renaming networks as giving renaming in $O(\log^2 n)$ time, since this is the most efficient practical sorting network we currently know about. This has led to efforts to produce $O(\log k)$-work tight renaming algorithms that don't depend on AKS. So far this has not worked out in the standard shared-memory model, even allowing test-and-sets.[3]

[3]The closest to this so far is an algorithm of Berenbrink *et al.* [BBE+15], who use an extended model that incorporates an extra primitive called a τ**-register**, which is basically a collection of $2 \log n$ test-and-set objects that are restricted so that at most $\tau < 2 \log n$ of them can be set at a time. Adding this primitive to the model is not entirely cheating, as the authors make a case that it could be plausibly implemented in hardware. But it does

The use of test-and-sets to route processes to particular names is similar to the line of test-and-sets proposed by Moir and Anderson [MA95] as described in §25.4.5. Some differences between that protocol and renaming networks is that renaming networks do not by themselves give fetch-and-increment (although Alistarh *et al.* show how to build fetch-and-increment on top of renaming networks at a small additional cost), and renaming networks do not provide any mechanism for releasing names. The question of whether it is possible to get cheap long-lived strong renaming is still open.

25.5.4 Randomized loose renaming

Loose renaming should be easier than strong renaming, and using a randomized algorithm it essentially reduces to randomized load balancing. A basic approach is to use $2n$ names, and guard each with a test-and-set; because less than half of the names are taken at any given time, each process gets a name after $O(1)$ tries and the most expensive renaming operation over all n processes takes $O(\log n)$ expected steps.

A more sophisticated version of this strategy, which appears in [AAGW13], uses $n(1+\epsilon)$ output names to get $O(\log \log n)$ maximum steps. The intuition for why this works is if n processes independently choose one of cn names uniformly at random, then the expected number of collisions—pairs of processes that choose the same name—is $\binom{n}{2}/cn$, or about $n/2c$. This may seem like only a constant-factor improvement, but if we instead look at the ratio between the survivors $n/2c$ and the number of allocated names cn, we have now moved from $1/c$ to $1/2c^2$. The 2 gives us some room to reduce the number of names in the next round, to $cn/2$, say, while still keeping a $1/c^2$ ratio of survivors to names.

So the actual renaming algorithm consists of allocating $cn/2^i$ names to round i, and squaring the ratio of survivors to names in each rounds. It only takes $O(\log \log n)$ rounds to knock the ratio of survivors to names below $1/n$, so at this point it is likely that all processes will have finished. At the same time, the sum over all rounds of the allocated names forms a geometric series, so only $O(n)$ names are needed altogether.

Swept under the carpet here is a lot of careful analysis of the probabilities. Unlike what happens with sifters (see §24.7), Jensen's inequality goes the wrong way here, so some additional technical tricks are needed (see the paper for details). But the result is that only $O(\log \log n)$ rounds are to assign every process a name with high probability, which is the best value currently

mean that we don't know what happens if we don't have this additional primitive.

known.

There is a rather weak lower bound in the Alistarh *et al.* paper that shows that $\Omega(\log \log n)$ steps are needed for some process in the worst case, under the assumption that the renaming algorithm uses only test-and-set objects and that a process acquires a name as soon as it wins some test-and-set object. This does not give a lower bound on the problem in general, and indeed the renaming-network based algorithms discussed previously do not have this property. So the question of the exact complexity of randomized loose renaming is still open.

Chapter 26

Software transactional memory

Last updated 2011. Some material may be out of date. If you are interested in software transactional memory from a theoretical perspective, there is a more recent survey on this material by Attiya [Att14], available at http://www.eatcs.org/images/bulletin/beatcs112.pdf.

Software transactional memory, or **STM** for short, goes back to Shavit and Touitou [ST97] based on earlier proposals for hardware support for transactions by Herlihy and Moss [HM93]. Recently very popular in programming language circles. We'll give a high-level description of the Shavit and Touitou results; for full details see the actual paper.

We start with the basic idea of a **transaction**. In a transaction, I read a bunch of registers and update their values, and all of these operations appear to be **atomic**, in the sense that the transaction either happens completely or not at all, and serializes with other transactions as if each occurred instantaneously. Our goal is to implement this with minimal hardware support, and use it for everything.

Generally we only consider **static transactions** where the set of memory locations accessed is known in advance, as opposed to **dynamic transactions** where it may vary depending on what we read (for example, maybe we have to follow pointers through some data structure). Static transactions are easier because we can treat them as multi-word read-modify-write.

Implementations are usually **non-blocking**: some infinite stream of transactions succeed, but not necessarily yours. This excludes the simplest method based on acquiring locks, since we have to keep going even if a

lock-holder crashes, but is weaker than wait-freedom since we can have starvation.

26.1 Motivation

Some selling points for software transactional memory:

1. We get atomic operations without having to use our brains much. Unlike hand-coded atomic snapshots, counters, queues, etc., we have a universal construction that converts any sequential data structure built on top of ordinary memory into a concurrent data structure. This is useful since most programmers don't have very big brains. We also avoid burdening the programmer with having to remember to lock things.

2. We can build large shared data structures with the possibility of concurrent access. For example, we can implement atomic snapshots so that concurrent updates don't interfere with each other, or an atomic queue where enqueues and dequeues can happen concurrently so long as the queue always has a few elements in it to separate the enqueuers and dequeuers.

3. We can execute atomic operations that span multiple data structures, even if the data structures weren't originally designed to work together, provided they are all implemented using the STM mechanism. This is handy in classic database-like settings, as when we want to take $5 from my bank account and put it in yours.

On the other hand, we now have to deal with the possibility that operations may fail. There is a price to everything.

26.2 Basic approaches

- Locking (not non-blocking). Acquire either a single lock for all of memory (doesn't allow much concurrency) or a separate lock for each memory location accessed. The second approach can lead to deadlock if we aren't careful, but we can prove that if every transaction acquires locks in the same order (e.g., by increasing memory address), then we never get stuck: we can order the processes by the highest lock acquired, and somebody comes out on top. Note that acquiring locks in

increasing order means that I have to know which locks I want before I acquire any of them, which may rule out dynamic transactions.

- Single-pointer compare-and-swap (called "Herlihy's method" in [ST97], because of its earlier use for constructing concurrent data structures by Herlihy [Her93]). All access to the data structure goes through a pointer in a CAS. To execute a transaction, I make my own copy of the data structure, update it, and then attempt to redirect the pointer. Advantages: trivial to prove that the result is linearizable (the pointer swing is an atomic action) and non-blocking (somebody wins the CAS); also, the method allows dynamic transactions (since you can do anything you want to your copy). Disadvantages: There's a high overhead of the many copies,[1] and the single-pointer bottleneck limits concurrency even when two transactions use disjoint parts of memory.

- Multiword RMW: This is the approach suggested by Shavit and Touitou, which most subsequent work follows. As usually implemented, it only works for static transactions. The idea is that I write down what registers I plan to update and what I plan to do to them. I then attempt to acquire all the registers. If I succeed, I update all the values, store the old values, and go home. If I fail, it's because somebody else already acquired one of the registers. Since I need to make sure that somebody makes progress (I may be the only process left alive), I'll help that other process finish its transaction if possible. Advantages: allows concurrency between disjoint transactions. Disadvantages: requires implementing multi-word RMW—in particular, requires that any process be able to understand and simulate any other process's transactions. Subsequent work often simplifies this to implementing multi-word CAS, which is sufficient to do non-blocking multi-word RMW since I can read all the registers I need (without any locking) and then do a CAS to update them (which fails only if somebody else succeeded).

26.3 Implementing multi-word RMW

We'll give a sketchy description of Shavit and Touitou's method [ST97], which essentially follows the locking approach but allows other processes to help dead ones so that locks are always released.

[1] This overhead can be reduced in many cases by sharing components, a subject that has seen much work in the functional programming literature. See for example [Oka99].

The synchronization primitive used is **LL/SC**: LL (**load-linked**) reads a register and leaves our ID attached to it, SC (**store-conditional**) writes a register only if our ID is still attached, and clears any other IDs that might also be attached. It's easy to build a 1-register CAS (CAS1) out of this, though Shavit and Touitou exploit some additional power of LL/SC.

26.3.1 Overlapping LL/SC

The particular trick that gets used in the Shavit-Touitou protocol is to use two overlapping LL/SC pairs to do a CAS-like update on one memory location while checking that another memory location hasn't changed. The purpose of this is to allow multiple processes to work on the same transaction (which requires the first CAS to avoid conflicts with other transactions) while making sure that slow processes don't cause trouble by trying to complete transactions that have already finished (the second check).

To see this in action, suppose we have a register r that we want to do a CAS on, while checking that a second register status is $\perp$ (as opposed to success or failure). If we execute the code fragment in Algorithm 26.1, it will succeed only if nobody writes to status between its LL and SC and similarly for r; if this occurs, then at the time of LL(r), we know that status $= \perp$, and we can linearize the write to r at this time if we restrict all access to r to go through LL/SC.

```
1 if LL(status) = ⊥ then
2     if LL(r) = oldValue then
3         if SC(status, ⊥) = true then
4             SC(r, newValue)
```

Algorithm 26.1: Overlapping LL/SC

26.3.2 Representing a transaction

Transactions are represented by records rec. Each such record consists of a status component that describes how far the transaction has gotten (needed to coordinate cooperating processes), a version component that distinguishes between versions that may reuse the same space (and that is used to shut down the transaction when complete), a stable component that indicates when the initialization is complete, an Op component that describes the RMW to be performance, an array addresses[] of pointers to the arguments

to the RMW, and an array oldValues[] of old values at these addresses (for the R part of the RMW). These are all initialized by the initiator of the transaction, who will be the only process working on the transaction until it starts acquiring locks.

26.3.3 Executing a transaction

Here we give an overview of a transaction execution:

1. Initialize the record rec for the transaction. (Only the initiator does this.)

2. Attempt to acquire ownership of registers in addresses[]. See the `AcquireOwnerships` code in the paper for details. The essential idea is that we want to set the field owner[r] for each memory location r that we need to lock; this is done using an overlapping LL/SC as described above so that we only set owner[r] if (a) r is currently unowned, and (b) nothing has happened to rec.status or rec.version. Ownership is acquired in order of increasing memory address; if we fail to acquire ownership for some r, our transaction fails. In case of failure, we set rec.status to failure and release all the locks we've acquired (checking rec.version in the middle of each LL/SC so we don't release locks for a later version using the same record). If we are the initiator of this transaction, we will also go on to attempt to complete the transaction that got in our way.

3. Do a `LL` on rec.status to see if `AcquireOwnerships` succeeded. If so, update the memory, store the old results in oldValues, and release the ownerships. If it failed, release ownership and help the next transaction as described above.

Note that only an initiator helps; this avoids a long chain of helping and limits the cost of each attempted transaction to the cost of doing two full transactions, while (as shown below) still allowing some transaction to finish.

26.3.4 Proof of linearizability

Intuition is:

- Linearizability follows from the linearizability of the locking protocol: acquiring ownership is equivalent to grabbing a lock, and updates occur only when all registers are locked.

- Complications come from (a) two or more processes trying to complete the same transaction and (b) some process trying to complete an old transaction that has already terminated. For the first part we just make sure that the processes don't interfere with each other, e.g. I am happy when trying to acquire a location if somebody else acquires it for the same transaction. For the second part we have to check rec.status and rec.version before doing just about anything. See the pseudocode in the paper for details on how this is done.

26.3.5 Proof of non-blockingness

To show that the protocol is non-blocking we must show that if an unbounded number of transactions are attempted, one eventually succeeds. First observe that in order to fail, a transaction must be blocked by another transaction that acquired ownership of a higher-address location than it did; eventually we run out of higher-address locations, so there is some transaction that doesn't fail. Of course, this transaction may not succeed (e.g., if its initiator dies), but either (a) it blocks some other transaction, and that transaction's initiator will complete it or die trying, or (b) it blocks no future transactions. In the second case we can repeat the argument for the $n - 1$ surviving processes to show that some of them complete transactions, ignoring the stalled transaction from case (b).

26.4 Improvements

One downside of the Shavit and Touitou protocol is that it uses LL/SC very aggressively (e.g., with overlapping LL/SC operations) and uses non-trivial (though bounded, if you ignore the ever-increasing version numbers) amounts of extra space. Subsequent work has aimed at knocking these down; for example a paper by Harris, Fraser, and Pratt [HFP02] builds multi-register CAS out of single-register CAS with $O(1)$ extra bits per register. The proof of these later results can be quite involved; Harris *et al.*, for example, base their algorithm on an implementation of 2-register CAS whose correctness has been verified only by machine (which may be a plus in some views).

26.5 Limitations

There has been a lot of practical work on STM designed to reduce overhead on real hardware, but there's still a fair bit of overhead. On the theory side,

a lower bound of Attiya, Hillel, and Milani [AHM09] shows that any STM system that guarantees non-interference between non-overlapping RMW transactions has the undesirable property of making read-only transactions as expensive as RMW transactions: this conflicts with the stated goals of many practical STM implementations, where it is assumed that most transactions will be read-only (and hopefully cheap). So there is quite a bit of continuing research on finding the right trade-offs.

Chapter 27

Obstruction-freedom

Last updated 2011. Some material may be out of date. In particular: §27.3 has not been updated to include some more recent results [ACHS16, GHHW13]; and §27.4 mostly follows the conference version [FHS05] of the Ellen-Hendler-Shavit paper and omits stronger results from the journal version [EHS12].

The gold standard for shared-memory objects is **wait-freedom**: I can finish my operation in a bounded number of steps no matter what anybody else does. Like the gold standard in real life, this can be overly constraining. So researchers have developed several weaker progress guarantees that are nonetheless useful. The main ones are:

Lock-freedom An implementation is **lock-free** if infinitely many operations finish in any infinite execution. In simpler terms, somebody always makes progress, but maybe not you. (Also called **non-blocking**.)

Obstruction-freedom An implementation is **obstruction-free** if, starting from any reachable configuration, any process can finish in a bounded number of steps if all of the other processes stop. This definition was proposed in 2003 by Herlihy, Luchangco, and Moir [HLM03]. In lower bounds (e.g., the Jayanti-Tan-Toueg bound described in Chapter 21) essentially the same property is often called **solo-terminating**.

Both of these properties exclude traditional lock-based algorithms, where some process grabs a lock, updates the data structure, and then release the lock; if this process halts, no more operations finish. Both properties are also weaker than wait-freedom. It is not hard to show that lock-freedom is a

stronger condition that obstruction-freedom: given a lock-free implementation, if we can keep some single process running forever in isolation, we get an infinite execution with only finitely many completed operations. So we have a hierarchy: wait-free > lock-free > obstruction-free > locking.

27.1 Why build obstruction-free algorithms?

The pitch is similar to the pitch for building locking algorithms: an obstruction-free algorithm might be simpler to design, implement, and reason about than a more sophisticated algorithm with stronger properties. Unlike locking algorithms, an obstruction-free algorithm won't fail because some process dies holding the lock; instead, it fails if more than one process runs the algorithm at the same time. This possibility may be something we can avoid by building a **contention manager**, a high-level protocol that detects contention and delays some processes to avoid it (say, using randomized exponential back-off).

27.2 Examples

27.2.1 Lock-free implementations

Pretty much anything built using compare-and-swap or LL/SC ends up being lock-free. A simple example would be a counter, where an increment operation does

```
1 x ← LL(C)
2 SC(C, x + 1)
```

This is lock-free (the only way to prevent a store-conditional from succeeding is if some other store-conditional succeeds, giving infinitely many successful increments) but not wait-free (I can starve). It's also obstruction-free, but since it's already lock-free we don't care about that.

27.2.2 Double-collect snapshots

Similarly, suppose we are doing atomic snapshots. We know that there exist wait-free implementations of atomic snapshots, but they are subtle and confusing. So we want to do something simpler, and hope that we at least get obstruction-freedom.

If we do double-collects, that is, we have updates just write to a register and have snapshots repeatedly collect until they get two collects in a row with the same values, then any snapshot that finishes is correct (assuming no updaters ever write the same value twice, which we can enforce with nonces). This isn't wait-free, because we can keep a snapshot going forever by doing a lot of updates. It *is* lock-free, because we have to keep doing updates to make this happen.

We can make this merely obstruction-free if we work hard (there is no reason to do this, but it illustrates the difference between lock-freedom—good—and obstruction-freedom—not so good). Suppose that every process keeps a count of how many collects it has done in a register that is included in other process's collects (but not its own). Then two concurrent scans can stall each other forever (the implementation is not lock-free), but if only one is running it completes two collects in $O(n)$ operations without seeing any changes (it is obstruction-free).

27.2.3 Software transactional memory

Similar things happen with software transactional memory (see Chapter 26). Suppose that I have an implementation of multiword compare-and-swap, and I want to carry out a transaction. I read all the values I need, then execute an MCAS operation that only updates if these values have not changed. The resulting algorithm is lock-free (if my transaction fails, it's because some update succeeded). If however I am not very clever and allow some values to get written outside of transactions, then I might only be obstruction-free.

27.2.4 Obstruction-free test-and-set

Algorithm 27.1 gives an implementation of 2-process test-and-set from atomic registers that is obstruction-free; this demonstrates that obstruction-freedom lets us evade the wait-free impossibility results implied by the consensus hierarchy ([Her91b], discussed in Chapter 19).

The basic idea goes back to the **racing counters** technique used in consensus protocols starting with Chor, Israeli, and Li [CIL94], and there is some similarity to a classic randomized wait-free test-and-set due to Tromp and Vitányi [TV02]. Each process keeps a position x in memory that it also stores from time to time in its register $a[i]$. If a process gets 2 steps ahead of the other process (as observed by comparing x to $a[i-1]$, it wins the test-and-set; if a process falls one or more steps behind, it (eventually) loses. To keep space down and guarantee termination in bounded time, all values

are tracked modulo 5.

```
1  x ← 0
2  while true do
3  |  δ ← x − a[1 − i]
4  |  if δ = 2 (mod 5) then
5  |  |  return 0
6  |  else if δ = −1 (mod 5) do
7  |  |  return 1
8  |  else
9  |  |  x ← (x + 1) mod 5
10 |  |  a[i] ← x
```

Algorithm 27.1: Obstruction-free 2-process test-and-set

Why this works: observe that whenever a process computes δ, x is equal to $a[i]$; so δ is always an instantaneous snapshot of $a[i] - a[1-i]$. If I observe $\delta = 2$ and return 0, your next read will either show you $\delta = -2$ or $\delta = -1$ (depending on whether you increment $a[1-i]$ after my read). In the latter case, you return 1 immediately; in the former, you return after one more increment (and more importantly, you can't return 0). Alternatively, if I ever observe $\delta = -1$, your next read will show you either $\delta = 1$ or $\delta = 2$; in either case, you will eventually return 0. (We chose 5 as a modulus because this is the smallest value that makes the cases $\delta = 2$ and $\delta = -2$ distinguishable.)

We can even show that this is linearizable, by considering a solo execution in which the lone process takes two steps and returns 0 (with two processes, solo executions are the only interesting case for linearizability).

However, Algorithm 27.1 is not wait-free or even lock-free: if both processes run in lockstep, they will see $\delta = 0$ forever. But it is obstruction-free. If I run by myself, then whatever value of δ I start with, I will see -1 or 2 after at most 6 operations.[1]

This gives an **obstruction-free step complexity** of 6, where the obstruction-free step complexity is defined as the maximum number of operations any process can take after all other processes stop. Note that our usual wait-free measures of step complexity don't make a lot of sense for obstruction-free algorithms, as we can expect a sufficiently cruel adversary to be able to run them up to whatever value he likes.

[1] The worst case is where an increment by my fellow process leaves $\delta = -1$ just before my increment.

Building a tree of these objects as in §23.2 gives n-process test-and-set with obstruction-free step complexity $O(\log n)$.

27.2.5 An obstruction-free deque

(We probably aren't going to do this in class.)

So far we don't have any good examples of why we would want to be obstruction-free if our algorithm is based on CAS. So let's describe the case Herlihy *et al.* suggested.

A **deque** is a generalized queue that supports push and pop at both ends (thus it can be used as either a queue or a stack, or both). A classic problem in shared-memory objects is to build a deque where operations at one end of the deque don't interfere with operations at the other end. While there exist lock-free implementation with this property, there is a particularly simple implementation using CAS that is only obstruction-free.

Here's the idea: we represent the deque as an infinitely-long array of compare-and-swap registers (this is a simplification from the paper, which gives a bounded implementation of a bounded deque). The middle of the deque holds the actual contents. To the right of this region is an infinite sequence of **right null** (RN) values, which are assumed never to appear as a pushed value. To the left is a similar infinite sequence of **left null** (LN) values. Some magical external mechanism (called an **oracle** in the paper) allows processes to quickly find the first null value at either end of the non-null region; the correctness of the protocol does not depend on the properties of the oracle, except that it has to point to the right place at least some of the time in a solo execution. We also assume that each cell holds a version number whose only purpose is to detect when somebody has fiddled with the cell while we aren't looking (if we use LL/SC, we can drop this).

Code for `rightPush` and `rightPop` is given in Algorithm 27.2 (the code for `leftPush` and `leftPop` is symmetric).

It's easy to see that in a solo execution, if the oracle doesn't lie, either operation finishes and returns a plausible value after $O(1)$ operations. So the implementation is obstruction-free. But is it also correct?

To show that it is, we need to show that any execution leaves the deque in a sane state, in particular that it preserves the invariant that the deque consists of left-nulls followed by zero or more values followed by right-nulls, and that the sequence of values in the queue is what it should be.

This requires a detailed case analysis of which operations interfere with each other, which can be found in the original paper. But we can give some intuition here. The two CAS operations in `rightPush` or `rightPop` succeed

```
1  procedure rightPush(v)
2      while true do
3          k ← oracle(right)
4          prev ← a[k − 1]
5          next ← a[k]
6          if prev.value ≠ RN and next.value = RN then
7              if CAS(a[k − 1], prev, [prev.value, prev.version + 1]) then
8                  if CAS(a[k], next, [v, next.version + 1]) then
9                      we win, go home

10 procedure rightPop()
11     while true do
12         k ← oracle(right)
13         cur ← a[k − 1]
14         next ← a[k]
15         if cur.value ≠ RN and next.value = RN then
16             if cur.value = LN and A[k − 1] = cur then
17                 return empty
18             else if CAS(a[k], next, [RN, next.version + 1]) do
19                 if CAS(a[k − 1], cur, [RN, cur.version + 1]) then
20                     return cur.value
```

Algorithm 27.2: Obstruction-free deque

only if neither register was modified between the preceding read and the CAS. If both registers are unmodified at the time of the second CAS, then the two CAS operations act like a single two-word CAS, which replaces the previous values (top, RN) with (top, value) in `rightPush` or (top, value) with (top, RN) in `rightPop`; in either case the operation preserves the invariant. So the only way we get into trouble is if, for example, a `rightPush` does a CAS on $a[k-1]$ (verifying that it is unmodified and incrementing the version number), but then some other operation changes $a[k-1]$ before the CAS on $a[k]$. If this other operation is also a `rightPush`, we are happy, because it must have the same value for k (otherwise it would have failed when it saw a non-null in $a[k-1]$), and only one of the two right-pushes will succeed in applying the CAS to $a[k]$. If the other operation is a `rightPop`, then it can only change $a[k-1]$ after updating $a[k]$; but in this case the update to $a[k]$ prevents the original right-push from changing $a[k]$. With some more tedious effort we can similarly show that any interference from `leftPush` or `leftPop` either causes the interfering operation or the original operation to fail. This covers 4 of the 16 cases we need to consider. The remaining cases will be brushed under the carpet to avoid further suffering.

27.3 Boosting obstruction-freedom to wait-freedom

Naturally, having an obstruction-free implementation of some object is not very helpful if we can't guarantee that some process eventually gets its unobstructed solo execution. In general, we can't expect to be able to do this without additional assumptions; for example, if we could, we could solve consensus using a long sequence of adopt-commit objects with no randomization at all.[2] So we need to make some sort of assumption about timing, or find somebody else who has already figured out the right assumption to make.

Those somebodies turn out to be Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit, who give an algorithm for boosting obstruction-freedom to wait-freedom [FLMS05]. The timing assumption is **unknown-bound semisynchrony**, which means that in any execution there is some maximum ratio R between the shortest and longest time interval between any two consecutive steps of the same non-faulty process, but the processes

[2] This fact was observed by Herlihy *et al.* [HLM03] in their original obstruction-free paper; it also implies that there exists a universal obstruction-free implementation of anything based on Herlihy's universal construction.

don't know what this ratio is.[3] In particular, if I can execute more than R steps without you doing anything, I can reasonably conclude that you are dead—the semisynchrony assumption thus acts as a failure detector.

The fact that R is unknown might seem to be an impediment to using this failure detector, but we can get around this. The idea is to start with a small guess for R; if a process is suspected but then wakes up again, we increment the guess. Eventually, the guessed value is larger than the correct value, so no live process will be falsely suspected after this point. Formally, this gives an eventually perfect ($\Diamond P$) failure detector, although the algorithm does not specifically use the failure detector abstraction.

To arrange for a solo execution, when a process detects a conflict (because its operation didn't finish quickly), it enters into a "panic mode" where processes take turns trying to finish unmolested. A fetch-and-increment register is used as a timestamp generator, and only the process with the smallest timestamp gets to proceed. However, if this process is too sluggish, other processes may give up and overwrite its low timestamp with ∞, temporarily ending its turn. If the sluggish process is in fact alive, it can restore its low timestamp and kill everybody else, allowing it to make progress until some other process declares it dead again.

The simulation works because eventually the mechanism for detecting dead processes stops suspecting live ones (using the technique described above), so the live process with the winning timestamp finishes its operation without interference. This allows the next process to proceed, and eventually all live processes complete any operation they start, giving the wait-free property.

The actual code is in Algorithm 27.3. It's a rather long algorithm but most of the details are just bookkeeping.

The preamble before entering PANIC mode is a fast-path computation that allows a process that actually is running in isolation to skip testing any timestamps or doing any extra work (except for the one register read of PANIC). The assumption is that the constant B is set high enough that any process generally will finish its operation in B steps without interference. If there is interference, then the timestamp-based mechanism kicks in: we grab a timestamp out of the convenient fetch-and-add register and start slugging it out with the other processes.

(A side note: while the algorithm as presented in the paper assumes a fetch-and-add register, any timestamp generator that delivers increasing

[3] This is a much older model, which goes back to a famous paper of Dwork, Lynch, and Stockmeyer [DLS88].

```
1  if ¬PANIC then
2      execute up to B steps of the underlying algorithm
3      if we are done then return
4  PANIC ← true // enter panic mode
5  myTimestamp ← fetchAndIncrement()
6  A[i] ← 1 // reset my activity counter
7  while true do
8      T[i] ← myTimestamp
9      minTimestamp ← myTimestamp; winner ← i
10     for j ← 1...n, j ≠ i do
11         otherTimestamp ← T[j]
12         if otherTimestamp < minTimestamp then
13             T[winner] ← ∞ // not looking so winning any more
14             minTimestamp ← otherTimestamp; winner ← j
15         else if otherTimestamp < ∞ do
16             T[j] ← ∞
17     if i = winner then
18         repeat
19             execute up to B steps of the underlying algorithm
20             if we are done then
21                 T[i] ← ∞
22                 PANIC ← false
23                 return
24             else
25                 A[i] ← A[i] + 1
26                 PANIC ← true
27         until T[i] = ∞
28     repeat
29         a ← A[winner]
30         wait a steps
31         winnerTimestamp ← T[winner]
32     until a = A[winner] or winnerTimestamp ≠ minTimestamp
33     if winnerTimestamp = minTimestamp then
34         T[winner] ← ∞ // kill winner for inactivity
```

Algorithm 27.3: Obstruction-freedom booster from [FLMS05]

values over time will work. So if we want to limit ourselves to atomic registers, we could generate timestamps by taking snapshots of previous timestamps, adding 1, and appending process IDs for tie-breaking.)

Once I have a timestamp, I try to knock all the higher-timestamp processes out of the way (by writing ∞ to their timestamp registers). If I see a smaller timestamp than my own, I'll drop out myself ($T[i] \leftarrow \infty$), and fight on behalf of its owner instead. At the end of the j loop, either I've decided I am the winner, in which case I try to finish my operation (periodically checking $T[i]$ to see if I've been booted), or I've decided somebody else is the winner, in which case I watch them closely and try to shut them down if they are too slow ($T[\mathsf{winner}] \leftarrow \infty$). I detect slow processes by inactivity in $A[\mathsf{winner}]$; similarly, I signal my own activity by incrementing $A[i]$. The value in $A[i]$ is also used as an increasing guess for the time between increments of $A[i]$; eventually this exceeds the $R(B + O(1))$ operations that I execute between incrementing it.

We still need to prove that this all works. The essential idea is to show that whatever process has the lowest timestamp finishes in a bounded number of steps. To do so, we need to show that other processes won't be fighting it in the underlying algorithm. Call a process *active* if it is in the loop guarded by the "if $i = \mathsf{winner}$" statement. Lemma 1 from the paper states:

Lemma 27.3.1 ([FLMS05, Lemma 1])**.** *If processes i and j are both active, then $T[i] = \infty$ or $T[j] = \infty$.*

Proof. Assume without loss of generality that i last set $T[i]$ to $\mathsf{myTimestamp}$ in the main loop after j last set $T[j]$. In order to reach the active loop, i must read $T[j]$. Either $T[j] = \infty$ at this time (and we are done, since only j can set $T[j] < \infty$), or $T[j]$ is greater than i's timestamp (or else i wouldn't think it's the winner). In the second case, i sets $T[j] = \infty$ before entering the active loop, and again the claim holds. □

The next step is to show that if there is some process i with a minimum timestamp that executes infinitely many operations, it increments $A[i]$ infinitely often (thus eventually making the failure detector stop suspecting it). This gives us Lemma 2 from the paper:

Lemma 27.3.2 ([FLMS05, Lemma 2])**.** *Consider the set of all processes that execute infinitely many operations without completing an operation. Suppose this set is non-empty, and let i hold the minimum timestamp of all these processes. Then i is not active infinitely often.*

Proof. Suppose that from some time on, i is active forever, i.e., it never leaves the active loop. Then $T[i] < \infty$ throughout this interval (or else i leaves the loop), so for any active j, $T[j] = \infty$ by the preceding lemma. It follows that any active $T[j]$ leaves the active loop after $B + O(1)$ steps of j (and thus at most $R(B + O(1))$ steps of i). Can j re-enter? If j's timestamp is less than i's, then j will set $T[i] = \infty$, contradicting our assumption. But if j's timestamp is greater than i's, j will not decide it's the winner and will not re-enter the active loop. So now we have i alone in the active loop. It may still be fighting with processes in the initial fast path, but since i sets PANIC every time it goes through the loop, and no other process resets PANIC (since no other process is active), no process enters the fast path after some bounded number of i's steps, and every process in the fast path leaves after at most $R(B + O(1))$ of i's steps. So eventually i is in the loop alone forever—and obstruction-freedom means that it finishes its operation and leaves. This contradicts our initial assumption that i is active forever. □

So now we want to argue that our previous assumption that there exists a bad process that runs forever without winning leads to a contradiction, by showing that the particular i from Lemma 27.3.2 actually finishes (note that Lemma 27.3.2 doesn't quite do this—we only show that i finishes if it stays active long enough, but maybe it doesn't stay active).

Suppose i is as in Lemma 27.3.2. Then i leaves the active loop infinitely often. So in particular it increments $A[i]$ infinitely often. After some finite number of steps, $A[i]$ exceeds the limit $R(B+O(1))$ on how many steps some other process can take between increments of $A[i]$. For each other process j, either j has a lower timestamp than i, and thus finishes in a finite number of steps (from the premise of the choice of i), or j has a higher timestamp than i. Once we have cleared out all the lower-timestamp processes, we follow the same logic as in the proof of Lemma 27.3.2 to show that eventually (a) i sets $T[i] < \infty$ and PANIC = true, (b) each remaining j observes $T[i] < \infty$ and PANIC = true and reaches the waiting loop, (c) all such j wait long enough (since $A[i]$ is now very big) that i can finish its operation. This contradicts the assumption that i never finishes the operation and completes the proof.

27.3.1 Cost

If the parameters are badly tuned, the potential cost of this construction is quite bad. For example, the slow increment process for $A[i]$ means that the time a process spends in the active loop even after it has defeated all other processes can be as much as the square of the time it would normally take

to complete an operation alone—and every other process may pay R times this cost waiting. This can be mitigated to some extent by setting B high enough that a winning process is likely to finish in its first unmolested pass through the loop (recall that it doesn't detect that the other processes have reset $T[i]$ until after it makes its attempt to finish). An alternative might be to double $A[i]$ instead of incrementing it at each pass through the loop. However, it is worth noting (as the authors do in the paper) that nothing prevents the underlying algorithm from incorporating its own **contention management** scheme to ensure that most operations complete in B steps and PANIC mode is rarely entered. So we can think of the real function of the construction as serving as a backstop to some more efficient heuristic approach that doesn't necessarily guarantee wait-free behavior in the worst case.

27.4 Lower bounds for lock-free protocols

So far we have seen that obstruction-freedom buys us an escape from the impossibility results that plague wait-free constructions, while still allowing practical implementations of useful objects under plausible timing assumptions. Yet all is not perfect: it is still possible to show non-trivial lower bounds on the costs of these implementations in the right model. We will present one of these lower bounds, the linear-contention lower bound of Ellen, Hendler, and Shavit [EHS12].[4] First we have to define what is meant by contention.

27.4.1 Contention

A limitation of real shared-memory systems is that physics generally won't permit more than one process to do something useful to a shared object at a time. This limitation is often ignored in computing the complexity of a shared-memory distributed algorithm (and one can make arguments for ignoring it in systems where communication costs dominate update costs in the shared-memory implementation), but it is useful to recognize it if we can't prove lower bounds otherwise. Complexity measures that take the cost of simultaneous access into account go by the name of **contention**.

The particular notion of contention used in the Ellen *et al.* paper is an adaptation of the contention measure of Dwork, Herlihy, and Waarts [DHW97].

[4] The result first appeared in FOCS in 2005 [FHS05], with a small but easily fixed bug in the definition of the class of objects the proof applies to. We'll use the corrected definition from the journal version.

The idea is that if I access some shared object, I pay a price in **memory stalls** for all the other processes that are trying to access it at the same time but got in first. In the original definition, given an execution of the form $A\phi_1\phi_2 \ldots \phi_k\phi A'$, where all operations ϕ_i are applied to the same object as ϕ, and the last operation in A is not, then ϕ_k incurs k memory stalls. Ellen *et al.* modify this to only count sequences of *non-trivial* operations, where an operation is non-trivial if it changes the state of the object in some states (e.g., writes, increments, compare-and-swap—but not reads). Note that this change only strengthens the bound they eventually prove, which shows that in the worst case, obstruction-free implementations of operations on objects in a certain class incur a linear number of memory stalls (possibly spread across multiple base objects).

27.4.2 The class G

The Ellen *et al.* bound is designed to be as general as possible, so the authors define a class G of objects to which it applies. As is often the case in mathematics, the underlying meaning of G is "a reasonably large class of objects for which this particular proof works," but the formal definition is given in terms of when certain operations of the implemented object are affected by the presence or absence of other operations—or in other words, when those other operations need to act on some base object in order to let later operations know they occurred.

An object is in **class** G if it has some operation Op and initial state s such that for any two processes p and q and every sequence of operations $A\phi A'$, where

1. ϕ is an instance of Op executed by p,
2. no operation in A or A' is executed by p,
3. no operation in A' is executed by q, and
4. no two operations in A' are executed by the same process;

then there exists a sequence of operations Q by q such that for every sequence $H\phi H'$ where

1. HH' is an interleaving of Q and the sequences $AA'|r$ for each process r,
2. H' contains no operations of q, and

3. no two operations in H' are executed by the same process;

then the return value of ϕ to p changes depending on whether it occurs after $A\phi$ or $H\phi$.

This is where "makes the proof work" starts looking like a much simpler definition. The intuition is that deep in the guts of the proof, we are going to be injecting some operations of q into an existing execution (hence adding Q), and we want to do it in a way that forces q to operate on some object that p is looking at (hence the need for $A\phi$ to return a different value from $H\phi$), without breaking anything else that is going on (all the rest of the conditions). The reason for pulling all of these conditions out of the proof into a separate definition is that we also want to be able to show that particular classes of real objects satisfy the conditions required by the proof, without having to put a lot of special cases into the proof itself.

Lemma 27.4.1. *A mod-m fetch-and-increment object, with $m \geq n$, is in G.*

Proof. This is a classic proof-by-unpacking-the-definition. Pick some execution $A\phi A'$ satisfying all the conditions, and let a be the number of fetch-and-increments in A and a' the number in A'. Note $a' \leq n-2$, since all operations in A' are by different processes.

Now let Q be a sequence of $n - a' - 1$ fetch-and-increments by q, and let HH' be an interleaving of Q and the sequences $AA'|r$ for each r, where H' includes no two operation of the same process and no operations at all of q. Let h, h' be the number of fetch-and-increments in H, H', respectively. Then $h + h' = a + a' + (n - a' - 1) = n + a - 1$ and $h' \leq n - 2$ (since H' contains at most one fetch-and-increment for each process other than p and q). This gives $h \geq (n + a + 1) - (n - 2) = a + 1$ and $h \leq n + a - 1$, and the return value of ϕ after $H\phi$ is somewhere in this range mod m. But none of these values is equal to a mod m (that's why we specified $m \geq n$, although as it turns out $m \geq n - 1$ would have been enough), so we get a different return value from $H\phi$ than from $A\phi$. □

As a corollary, we also get stock fetch-and-increment registers, since we can build mod-m registers from them by taking the results mod m.

A second class of class-G objects is obtained from snapshot:

Lemma 27.4.2. *Single-writer snapshot objects are in G.*[5]

[5]For the purposes of this lemma, "single-writer" means that each segment can be written to by only one process, not that there is only one process that can execute update operations.

Proof. Let $A\phi A'$ be as in the definition, where ϕ is a scan operation. Let Q consist of a single update operation by q that changes its segment. Then in the interleaved sequence HH', this update doesn't appear in H' (it's forbidden), so it must be in H. Nobody can overwrite the result of the update (single-writer!), so it follows that $H\phi$ returns a different snapshot from $A\phi$. □

27.4.3 The lower bound proof

Theorem 27.4.3 ([EHS12, Theorem 5.2])**.** *For any obstruction-free implementation of some object in class G from RMW base objects, there is an execution in which some operation incurs $n-1$ stalls.*

We can't do better than $n-1$, because it is easy to come up with implementations of counters (for example) that incur at most $n-1$ stalls. Curiously, we can even spread the stalls out in a fairly arbitrary way over multiple objects, while still incurring at most $n-1$ stalls. For example, a counter implemented using a single counter (which is a RMW object) gets exactly $n-1$ stalls if $n-1$ processes try to increment it at the same time, delaying the remaining process. At the other extreme, a counter implemented by doing a collect over $n-1$ single-writer registers (also RMW objects) gets at least $n-1$ stalls—distributed as one per register—if each register has a write delivered to it while the reader waiting to read it during its collect. So we have to allow for the possibility that stalls are concentrated or scattered or something in between, as long as the total number adds up at least $n-1$.

The proof supposes that the theorem is not true and then shows how to boost an execution with a maximum number $k < n-1$ stalls to an execution with $k+1$ stalls, giving a contradiction. (Alternatively, we can read the proof as giving a mechanism for generating an $(n-1)$-stall execution by repeated boosting, starting from the empty execution.)

This is pretty much the usual trick: we assume that there is a class of bad executions, then look for an extreme member of this class, and show that it isn't as extreme as we thought. In doing so, we can restrict our attention to particularly convenient bad executions, so long as the existence of some bad execution implies the existence of a convenient bad execution.

Formally, the authors define a *k-stall execution* for process p as an execution $E\sigma_1 \dots \sigma_i$ where E and σ_i are sequence of operations such that:

1. p does nothing in E,
2. Sets of processes S_j, $j = 1 \dots i$, whose union $S = \bigcup_{j=1}^{i} S_j$ has size k, are each covering objects $\mathcal{O}_j$ after E with pending non-trivial operations,

3. Each σ_j consists of p applying events by itself until it is about to apply an event to $\mathcal{O}_j$, after which each process in S_j accesses $\mathcal{O}_j$, after which p accesses $\mathcal{O}_j$.

4. All processes not in S are idle after E,

5. p starts at most one operation of the implemented object in $\sigma_1 \dots \sigma_i$, and

6. In every extension of E in which p and the processes in S don't take steps, no process applies a non-trivial event to any base object accessed in $\sigma_1 \dots \sigma_i$. (We will call this the **weird condition** below.)

So this definition includes both the fact that p incurs k stalls and some other technical details that make the proof go through. The fact that p incurs k stalls follows from observing that it incurs $|S_j|$ stalls in each segment σ_j, since all processes in S_j access $\mathcal{O}_j$ just before p does.

Note that the empty execution is a 0-stall execution (with $i = 0$) by the definition. This shows that a k-stall execution exists for some k.

Note also that the weird condition is pretty strong: it claims not only that there are no non-trivial operation on $\mathcal{O}1 \dots \mathcal{O}i$ in τ, but also that there are no non-trivial operations on *any* objects accessed in $\sigma_1 \dots \sigma_i$, which may include many more objects accessed by p.[6]

We'll now show that if a k-stall execution exists, for $k \leq n-2$, then a $(k+k')$-stall execution exists for some $k' > 0$. Iterating this process eventually produces an $(n-1)$-stall execution.

Start with some k-stall execution $E\sigma_1 \dots \sigma_i$. Extend this execution by a sequence of operations σ in which p runs in isolation until it finishes its operation ϕ (which it may start in σ if it hasn't done so already), then each process in S runs in isolation until it completes its operation. Now linearize the high-level operations completed in $E\sigma_1 \dots \sigma_i\sigma$ and factor them as $A\phi A'$ as in the definition of class G.

Let q be some process not equal to p or contained in any S_j (this is where we use the assumption $k \leq n-2$). Then there is some sequence of high-level operations Q of q such that $H\phi$ does not return the same value as $A\phi$ for any interleaving HH' of Q with the sequences of operations in AA' satisfying the conditions in the definition. We want to use this fact to shove at least one more memory stall into $E\sigma_1 \dots \sigma_i\sigma$, without breaking any of the other conditions that would make the resulting execution a $(k+k')$-stall execution.

[6] And here is where I screwed up in class on 2011-11-14, by writing the condition as the weaker requirement that nobody touches $\mathcal{O}_1 \dots \mathcal{O}_i$.

Consider the extension τ of E where q runs alone until it finishes every operation in Q. Then τ applies no nontrivial events to any base object accessed in $\sigma_1 \dots \sigma_k$, (from the weird condition on k-stall executions) and the value of each of these base objects is the same after E and $E\tau$, and thus is also the same after $E\sigma_1 \dots \sigma_k$ and $E\tau\sigma_1 \dots \sigma_k$.

Now let σ' be the extension of $E\tau\sigma_1 \dots \sigma_k$ defined analogously to σ: p finishes, then each process in each S_j finishes. Let $H\phi H'$ factor the linearization of $E\tau\sigma_1 \dots \sigma_i\sigma'$. Observe that HH' is an interleaving of Q and the high-level operations in AA', that H' contains no operations by q (they all finished in τ, before ϕ started), and that H' contains no two operations by the same process (no new high-level operations start after ϕ finishes, so there is at most one pending operation per process in S that can be linearized after ϕ).

Now observe that q does some non-trivial operation in τ to some base object accessed by p in σ. If not, then p sees the same responses in σ' and in σ, and returns the same value, contradicting the definition of class G.

So does q's operation in τ cause a stall in σ? Not necessarily: there may be other operations in between. Instead, we'll use the existence of q's operation to demonstrate the existence of at least one operation, possibly by some other process we haven't even encountered yet, that does cause a stall. We do this by considering the set F of all finite extensions of E that are free of p and S operations, and look for an operation that stalls p somewhere in this infinitely large haystack.

Let $\mathcal{O}_{i+1}$ be the first base object accessed by p in σ that is also accessed by some non-trivial event in some sequence in F. We will show two things: first, that $\mathcal{O}_{i+1}$ exists, and second, that $\mathcal{O}_{i+1}$ is distinct from the objects $\mathcal{O}_1 \dots \mathcal{O}_i$. The first part follows from the fact that τ is in F, and we have just shown that τ contains a non-trivial operation (by q) on a base object accessed by p in σ. For the second part, we use the weird condition on k-stall executions again: since every extension of E in F is $(\{p\} \cup S)$-free, no process applies a non-trivial event to any base object accessed in $\sigma_1 \dots \sigma_i$, which includes all the objects $\mathcal{O}_1 \dots \mathcal{O}_i$.

You've probably guessed that we are going to put our stalls in on $\mathcal{O}_{i+1}$. We choose some extension X from F that maximizes the number of processes with simultaneous pending non-trivial operations on $\mathcal{O}_{i+1}$ (we'll call this set of processes S_{i+1} and let $|S_{i+1}|$ be the number $k' > 0$ we've been waiting for), and let E' be the minimum prefix of X such that these pending operations are still pending after EE'.

We now look at the properties of EE'. We have:

- EE' is p-free (follows from E being p-free and $E' \in F$, since everything in F is p-free).
- Each process in S_j has a pending operation on $\mathcal{O}_j$ after EE' (it did after E, and didn't do anything in E').

This means that we can construct an execution $EE'\sigma_1 \ldots \sigma_i\sigma_{i+1}$ that includes $k + k'$ memory stalls, by sending in the same sequences $\sigma_1 \ldots \sigma_i$ as before, then appending a new sequence of events where (a) p does all of its operations in σ up to its first operation on $\mathcal{O}_{i+1}$; then (b) all the processes in the set S_{i+1} of processes with pending events on $\mathcal{O}_{i+1}$ execute their pending events on $\mathcal{O}_{i+1}$; then (c) p does its first access to $\mathcal{O}_{i+1}$ from σ. Note that in addition to giving us $k + k'$ memory stalls, σ_{i+1} also has the right structure for a $(k + k')$-stall execution. But there is one thing missing: we have to show that the weird condition on further extensions still holds.

Specifically, letting $S' = S \cup S_{i+1}$, we need to show that any $(\{p\} \cup S')$-free extension α of EE' includes a non-trivial access to a base object accessed in $\sigma_1 \ldots \sigma_{i+1}$. Observe first that since α is $(\{p\} \cup S')$-free, then $E'\alpha$ is $(\{p\} \cup S)$-free, and so it's in F: so by the weird condition on $E\sigma_1 \ldots \sigma_i$, $E'\alpha$ doesn't have any non-trivial accesses to any object with a non-trivial access in $\sigma_1 \ldots \sigma_i$. So we only need to squint very closely at σ_{i+1} to make sure it doesn't get any objects in there either.

Recall that σ_{i+1} consists of (a) a sequence of accesses by p to objects already accessed in $\sigma_1 \ldots \sigma_i$ (already excluded); (b) an access of p to $\mathcal{O}_{i+1}$; and (c) a bunch of accesses by processes in S_{i+1} to $\mathcal{O}_{i+1}$. So we only need to show that α includes no non-trivial accesses to $\mathcal{O}_{i+1}$. Suppose that it does: then there is some process that eventually has a pending non-trivial operation on $\mathcal{O}_{i+1}$ somewhere in α. If we stop after this initial prefix α' of α, we get $k' + 1$ processes with pending operations on $\mathcal{O}_{i+1}$ in $EE'\alpha'$. But then $E'\alpha'$ is an extension of E with $k' + 1$ processes with a simultaneous pending operation on $\mathcal{O}_{i+1}$. This contradicts the choice of X to maximize k'. So if our previous choice was in fact maximal, the weird condition still holds, and we have just constructed a $(k + k')$-stall execution. This concludes the proof.

27.4.4 Consequences

We've just shown that counters and snapshots have $(n - 1)$-stall executions, because they are in the class G. A further, rather messy argument (given in the Ellen *et al.* paper) extends the result to stacks and queues, obtaining a slightly weaker bound of n total stalls and operations for some process in

the worst case.[7] In both cases, we can't expect to get a sublinear worst-case bound on time under the reasonable assumption that both a memory stall and an actual operation takes at least one time unit. This puts an inherent bound on how well we can handle hot spots for many practical objects, and means that in an asynchronous system, we can't solve contention at the object level in the worst case (though we may be able to avoid it in our applications).

But there might be a way out for some restricted classes of objects. We saw in Chapter 22 that we could escape from the Jayanti-Tan-Toueg [JTT00] lower bound by considering bounded objects. Something similar may happen here: the Fich-Herlihy-Shavit bound on fetch-and-increments requires executions with $n(n-1)^d + n$ increments to show $n-1$ stalls for some fetch-and-increment if each fetch-and-increment only touches d objects, and even for $d = \log n$ this is already superpolynomial. The max-register construction of a counter [AACH12] doesn't help here, since everybody hits the switch bit at the top of the max register, giving $n-1$ stalls if they all hit it at the same time. But there might be some better construction that avoids this.

27.4.5 More lower bounds

There are many more lower bounds one can prove on lock-free implementations, many of which are based on previous lower bounds for stronger models. We won't present these in class, but if you are interested, a good place to start is [AGHK06].

27.5 Practical considerations

Also beyond the scope of what we can do, there is a paper by Fraser and Harris [FH07] that gives some nice examples of the practical trade-offs in choosing between multi-register CAS and various forms of software transactional memory in implementing lock-free data structures.

[7] This is out of date: Theorem 6.2 of [EHS12] gives a stronger result than what's in [FHS05].

Chapter 28

BG simulation

The **Borowsky-Gafni simulation** [BG93], or **BG simulation** for short, is a deterministic, wait-free algorithm that allows $t+1$ processes to collectively construct a simulated execution of a system of $n > t$ processes of which t may crash. For both the simulating and simulated system, the underlying shared-memory primitives are atomic snapshots; these can be replaced by atomic registers using any standard snapshot algorithm. The main consequence of the BG simulation is that the question of what decision tasks can be computed deterministically by an asynchronous shared-memory system that tolerates t crash failures reduces to the question of what can be computed by a wait-free system with exactly $t+1$ processes. This is an easier problem, and in principle can be solved exactly using the topological approach described in Chapter 29.

The intuition for how this works is that the $t+1$ simulating processes solve a sequence of agreement problems to decide what the n simulated processes are doing; these agreement problems are structured so that the failure of a simulator stops at most one agreement. So if at most t of the simulating processes can fail, only t simulated processes get stuck as well.

We'll describe here a version of the BG simulation that appears in a follow-up paper by Borowsky, Gafni, Lynch, and Rajsbaum [BGLR01]. This gives a more rigorous presentation of the mechanisms of the original Borowsky-Gafni paper, and includes a few simplifications.

28.1 High-level strategy

To avoid having to simulate specific choices of operations, the BG simulation assumes that all simulated processes alternate between taking snapshots and

doing updates. This assumption is not very restrictive, because two snapshots with no intervening update are equivalent to two snapshots separated by an update that doesn't change anything, and two updates with no intervening snapshot can be replaced by just the second update, since the adversary could choose to schedule them back-to-back anyway.

This approach means that we can determine the actions of some simulated process by determining the sequence of snapshots that it receives. So the goal will be to allow any of the real processes to take a snapshot on behalf of any of the simulated processes, and then coordinate these snapshots via weak consensus objects to enforce consistency if more than one real process tries to simulate a step of the same simulated process. The key tool for doing this is a **safe agreement** object, described in §28.2.

28.2 Safe agreement

A naive approach to simulate n processes using $f + 1$ processes would be to lock each simulated process behind a mutex, and have the real processes take turns grabbing a lock, simulating a step, and releasing the lock. If we could somehow guarantee that processes never get stuck waiting for a particular mutex just because some process died holding the lock, then we could treat any blocked simulated process as dead, and charge its death to the dead process holding the lock. This would give the mapping of at most f simulated failures to f real failures we are hoping for. But this depends on a lot of subtleties in how we implement the mutexes, so the standard BG simulation goes through a weakening of consensus instead.

The **safe agreement** mechanism performs agreement without running into the FLP bound, by providing a weaker termination condition. It is guaranteed to terminate only if there are no failures by any process during an initial, bounded, **unsafe** section of its execution, but if a process fails later, it can prevent termination. Processes can detect when they leave the unsafe section and have to wait for other processes only in the safe section. This means that they can dovetail spinning in the safe sections of multiple safe agreement objects without getting stuck entirely, even if dead processes in the unsafe sections are blocking some of the objects.

Each process i starts the agreement protocol with a $\texttt{propose}_i(v)$ event for its input value v. At some point during the execution of the protocol, the process receives a notification $\texttt{safe}_i$, followed later (if the protocol finishes) by a second notification $\texttt{agree}_i(v')$ for some output value v'. It is guaranteed that the protocol terminates as long as all processes continue to take steps

until they receive the `safe` notification, and that the usual validity (all outputs equal some input) and agreement (all outputs equal each other) conditions hold. There is also a wait-free progress condition that the $\texttt{safe}_i$ notices do eventually arrive for any process that doesn't fail, no matter what the other processes do (so nobody gets stuck in their unsafe section).

Pseudocode for a safe agreement object is given in Algorithm 28.1. This is a translation of the description of the algorithm in [BGLR01], which is specified at a lower level using I/O automata.[1]

```
   // propose_i(v)
1  A[i] ← ⟨v, 1⟩
2  if snapshot(A) contains ⟨j, 2⟩ for some j ≠ i then
       // Back off
3      A[i] ← ⟨v, 0⟩
4  else
       // Advance
5      A[i] ← ⟨v, 2⟩
   // safe_i
6  repeat
7      s ← snapshot(A)
8  until s does not contain ⟨j, 1⟩ for any j
   // agree_i
9  return s[j].value where j is smallest index with s[j].level = 2
```

Algorithm 28.1: Safe agreement (adapted from [BGLR01])

The communication mechanism is a snapshot object containing a pair $A[i] = \langle \mathsf{value}_i, \mathsf{level}_i \rangle$ for each process i, initially $\langle \perp, 0 \rangle$. When a process carries out $\texttt{propose}_i(v)$, it sets $A[i]$ to $\langle v, 1 \rangle$, advancing to level 1. It then looks around to see if anybody else is at level 2; if so, it backs off to 0, and if not, it advances to 2. In either case it then spins until it sees a snapshot with nobody at level 1, and agrees on the level-2 value with the smallest index i.

The $\texttt{safe}_i$ transition occurs when the process leaves level 1 (no matter which way it goes). This satisfies the progress condition, since there is no loop before this, and guarantees termination if all processes leave their unsafe interval, because no process can then wait forever for the last 1 to disappear.

To show agreement, observe that at least one process advances to level 2 (because the only way a process doesn't is if some other process has already

[1] The I/O automaton model is described in Appendix J.

advanced to level 2), so any process i that terminates observes a snapshot s that contains at least one level-2 tuple and no level-1 tuples. This means that any process j whose value is not already at level 2 in s can at worst reach level 1 after s is taken. But then j sees a level-2 tuples and backs off. It follows that any other process i' that takes a later snapshot s' that includes no level-1 tuples sees the same level-2 tuples as i, and computes the same return value. (Validity also holds, for the usual trivial reasons.)

28.3 The basic simulation algorithm

The basic BG simulation uses a single snapshot object A with $t+1$ components and an infinite array of safe agreement objects S_{jr}.

Each component $A[i]$ of A belongs to one of the $t+1$ simulating processes, and is a vector of values $A[i][j]$ that process i believes process j will have written at some point during the simulated execution. These values are tagged with round numbers: each $A[i][j]$ holds a tuple $\langle v, r\rangle$ representing the value v that process i determines process j would have written after taking r snapshots.

The contents of these snapshots are obtained from the S_{jr} objects. The inputs to S_{jr} are simulated snapshots, and the output s_{jr} of S_{jr} represents the value of the r-th snapshot performed by simulated process j.

Each simulating process i cycles through all simulated processes j. Simulating one round of a particular process j involves four phases:

1. Make an initial guess for s_{jr} by taking a snapshot of A and taking the value with the largest round number for each component $A[-][k]$.

2. Initiate the safe agreement protocol S_{jr} using this guess. It continues to run S_{jr} until it leaves the unsafe interval.

3. Attempt to finish S_{jr}, by performing one iteration of the loop from Algorithm 28.1. If this iteration doesn't succeed, move on to simulating $j+1$ (but come back to this phase for j eventually).

4. If S_{jr} terminates, compute a new value v_{jr} for j to write based on the simulated snapshot returned by S_{jr}, and update $A[i][j]$ with $\langle v_{jr}, r\rangle$.

Actually implementing this while maintaining an abstraction barrier around safe agreement is tricky. One approach might be to have each process i manage a separate thread for each simulated process j, and wrap the unsafe part of the safe agreement protocol inside a mutex just for threads of i. This

guarantees that i enters the unsafe part of any safe agreement object on behalf of only one simulated j at a time, while preventing delays in the safe part of S_{jr} from blocking it from finishing some other $S_{j'r'}$.

28.4 Effect of failures

So now what happens if a simulating process i fails? This won't stop any other process i' from taking snapshots on behalf of j, or from generating its own values to put in $A[i'][j]$. What it may do is prevent some safe agreement object S_{jr} from terminating. The termination property of S_{jr} means that this can only occur if the failure occurs while i is in the unsafe interval for S_{jr}—but since i is only in the unsafe interval for at most one S_{jr} at a time, this stalls only one simulated process j. It doesn't block any i', because any other i' is guaranteed to leave its own unsafe interval for S_{jr} after finitely many steps, and though it may waste some effort waiting for S_{jr} to finish, once it is in the safe interval it doesn't actually wait for it before moving on to other simulated j'.

It follows that each failure of a simulating process knocks out at most one simulated process. So a wait-free system with $t+1$ processes—and thus at most t failures in the executions we care about—will produces at most t failures inside the simulation.

28.5 Inputs and outputs

Two details not specified in the description above are how i determines j's initial input and how i determines its own outputs from the outputs of the simulated processes. For the basic BG simulation, this is pretty straightforward: we use the safe agreement objects S_{j0} to agree on j's input, after each i proposes its own input vector for all j based on its own input to the simulator protocol. For outputs, i waits for at least $n-t$ of the simulated processes to finish, and computes its own output based on what it sees.

One issue that arises here is that we can only use the simulation to solve **colorless tasks**, which are decision problems where any process can return the output of any other process without causing trouble.[2] This works for consensus or k-set agreement, but fails pretty badly for renaming. The **extended BG simulation**, due to Gafni [Gaf09], solves this problem by

[2]The term "colorless" here comes from use of colors to represent process IDs in the topological approach described in Chapter 29. These colors aren't really colors, but topologists like coloring nodes better than assigning them IDs.

mapping each simulating process p to a specific simulated process q_p, and using a more sophisticated simulation algorithm to guarantee that q_p doesn't crash unless p does; details can be found in Gafni's paper. There is also a later paper by Imbs and Raynal [IR09] that simplifies some details of the construction. Here, we will limit ourselves to the basic BG simulation.

28.6 Correctness of the simulation

To show that the simulation works, observe that we can extract a simulated execution by applying the following rules:

1. The round-r write operation of j is represented by the first write tagged with round r performed for j.

2. The round-r snapshot operation of j is represented by whichever snapshot operation wins S_{jr}.

The simulated execution then consists of a sequence of write and snapshot operations, with order of the operations determined by the order of their representatives in the simulating execution, and the return values of the snapshots determined by the return values of their representatives.

Because all processes that simulate a write for j in round r use the same snapshots to compute the state of j, they all write the same value. So the only way we get into trouble is if the writes included in our simulated snapshots are inconsistent with the ordering of the simulated operations defined above. Here the fact that each simulated snapshot corresponds to a real snapshot makes everything work: when a process performs a snapshot for S_{jr}, then it includes all the simulated write operations that happen before this snapshot, since the s-th write operation by k will be represented in the snapshot if and only if the first instance of the s-th write operation by k occurs before it. The only tricky bit is that process i's snapshot for S_{jr} might include some operations that can't possibly be included in S_{jr}, like j's round-r write or some other operation that depends on it. But this can only occur if some other process finished S_{jr} before process i takes its snapshot, in which case i's snapshot will not win S_{jr} and will be discarded.

28.7 BG simulation and consensus

BG simulation was originally developed to attack k-set agreement, but (as pointed out by Gafni [Gaf09]) it gives a particularly simple proof of the

impossibility of consensus with one faulty process. Suppose that we had a consensus protocol that solved consensus for $n > 1$ processes with one crash failure, using only atomic registers. Then we could use BG simulation to get a wait-free consensus protocol for two processes. But it's easy to show that atomic registers can't solve wait-free consensus, because (following [LAA87]), we only need to do the last step of FLP that gets a contradiction when moving from a bivalent C to 0-valent Cx or 1-valent Cy. We thus avoid the complications that arise in the original FLP proof from having to deal with fairness.

More generally, BG simulation means that increasing the number of processes while keeping the same number of crash failures doesn't let us compute anything we couldn't before. This gives a formal justification for the slogan that the difference between distributed computing and parallel computing is that in a distributed system, more processes can only make things worse.

Chapter 29

Topological methods

Here we'll describe some results applying topology to distributed computing, mostly following a classic paper of Herlihy and Shavit [HS99]. This was one of several papers [BG93, SZ00] that independently proved lower bounds on **k-set agreement** [Cha93], which is a relaxation of consensus where we require only that there are at most k distinct output values (consensus is 1-set agreement). These lower bounds had failed to succumb to simpler techniques.

29.1 Basic idea

The basic idea is to use tools from combinatorial topology to represent indistinguishability proofs. We've seen a lot of indistinguishability proofs that involving showing that particular pairs of executions are indistinguishable to some process, which means that that process produces the same output in both executions. In a typical proof of this kind, we then construct a chain of executions $\Xi_1, \ldots, \Xi_k$ such that for each i, there is some p with $\Xi_i|p = \Xi_{i+1}|p$. We've generally been drawing these with the executions as points and the indistinguishability relation as an edge between two executions. In the topological method, we use the dual of this picture: each process's view (the restriction of some execution to events visible to that process) is represented as a point, and an execution Ξ is represented as a **simplex** connecting all of the points corresponding to views of Ξ by particular processes.

A simplex is a generalization to arbitrary dimension of the sequence that starts with a point (a 0-simplex), an edge (a 1-simplex), a triangle (a 2-simplex), or a tetrahedron (a 3-simplex). In general, an n-simplex is a solid n-dimensional object with $n + 1$ vertices and $n + 1$ faces that are

$(n-1)$-simplexes. As a combinatorial object, this is a fancy way of depicting the power set of the set of vertices: each subset corresponds to a facet of the original simplex. A simplicial complex consists of a bunch of simplexes pasted together by identifying vertices: this is similar to the technique in graphics of representing the surface of a three-dimensional object by decomposing it into triangles. Topologists use these to model continuous surfaces, and have many tools for deriving interesting properties of those surfaces from a description of the simplicial complex.

For distributed computing, the idea is that some of these topological properties, when computed for the simplicial complex resulting from some protocol or problem specification may sometimes useful to determine properties of the underlying protocol or problem.

29.2 k-set agreement

The motivating problem for much of this work was getting impossibility results for **k-set agreement**, proposed by Chaudhuri [Cha93]. The k-set agreement problem is similar to consensus, where each process starts with an input and eventually returns a decision value that must be equal to some process's input, but the agreement condition is relaxed to require only that the set of decision values include at most k values.

With $k-1$ crash failures, it's easy to build a k-set agreement algorithm: wait until you have seen $n-k+1$ input values, then choose the smallest one you see. This works because any value a process returns is necessarily among the k smallest input values (including the $k-1$ it didn't see).

Chaudhuri conjectured that k-set agreement was not solvable with k failures. Proving this is surprisingly difficult. Being able to solve the problem with $k-1$ failures knocks out many standard indistinguishability arguments that use only 1 failure, and it is now known that a large class of bivalence-like arguments where the adversary probes the future looking for a bad execution also can't work for this problem [AAE+23]. So the k-set agreement problem quickly became a central test case for more general impossibility results for computations with crash failures.

In her original paper, Chaudhuri gave a proof of a partial result (analogous to the existence of an initial bivalent configuration for consensus) based on Sperner's Lemma [Spe28]. This is a classic result in topology that says that certain colorings of the vertices of a graph in the form of a triangle that has been divided into smaller triangles necessarily contain a small triangle with three different colors on its corners. This connection between k-set

agreement and Sperner's Lemma became the basic idea behind each the three independent proofs of the conjecture that appeared shortly thereafter [HS99, BG93, SZ00], all of which adopted an approach that reduces decision problems in distributed systems to the existence of certain structures in combinatorial topology.

Our plan is to give a sufficient high-level description of the topological approach that the connection between k-set agreement and Sperner's Lemma becomes obvious. It is possible to avoid this by approaching the problem purely combinatorially, as is done, for example, in Section 16.3 of [AW04]. The presentation there is obtained by starting with a topological argument and getting rid of the topology (in fact, the proof in [AW04] contains a proof of Sperner's Lemma with the serial numbers filed off). The disadvantage of this approach is that it obscures what is really going in and makes it harder to obtain insight into how topological techniques might help for other problems. The advantage is that (unlike these notes) the resulting text includes actual proofs instead of handwaving.

29.3 Representing distributed computations using topology

Topology is the study of properties of shapes that are preserved by continuous functions between their points that have continuous inverses, which get the rather fancy name of **homeomorphisms**. A continuous function[1] is one that maps nearby points to nearby points. A homeomorphism is continuous in both directions: this basically means that you can stretch and twist and otherwise deform your object however you like, as long as you don't tear it (which would map nearby points on opposite sides of the tear to distant points) or glue bits of it together (which turns into tearing when we look at the inverse function). Topologists are particularly interested in showing when there is no homeomorphism between two objects; the classic example is that you can't turn a sphere into a donut without damaging it, but you can turn a donut into a coffee mug (with a handle).

Working with arbitrary objects embedded in umpteen-dimensional spaces is messy, so topologists invented a finite way of describing certain well-behaved objects combinatorially, by replacing ugly continuous objects like spheres and coffee mugs with simpler objects pasted together in complex ways. The

[1] Strictly speaking, this is the definition a continuous function between metric spaces, which are spaces that have a consistent notion of distance. There is an even more general definition of continuity that holds for spaces that are too strange for this.

simpler objects are **simplexes**, and the more complicated pasted-together objects are called **simplicial complexes**. The nifty thing about simplicial complexes is that they give a convenient tool for describing what states or outputs of processes in a distributed algorithm are "compatible" in some sense, and because topologists know a lot about simplicial complexes, we can steal their tools to describe distributed algorithms.

29.3.1 Simplicial complexes and process states

The formal definition of a k-dimensional **simplex** is the convex closure of $(k+1)$ points $\{x_1 \dots x_{k+1}\}$ in general position; the convex closure part means the set of all points $\sum a_i x_i$ where $\sum a_i = 1$ and each $a_i \geq 0$, and the general position part means that the x_i are not all contained in some subspace of dimension $(k-1)$ or smaller (so that the simplex isn't squashed flat somehow). What this gives us is a body with $(k+1)$ corners and $(k+1)$ faces, each of which is a $(k-1)$-dimensional simplex (the base case is that a 0-dimensional simplex is a point). Each face includes all but one of the corners, and each corner is on all but one of the faces. So we have:

- 0-dimensional simplex: point.[2]
- 1-dimensional simplex: line segment with 2 endpoints (which are both corners and faces).
- 2-dimensional simplex: triangle (3 corners with 3 1-dimensional simplexes for sides).
- 3-dimensional simplex: tetrahedron (4 corners, 4 triangular faces).
- 4-dimensional simplex: 5 corners, 5 tetrahedral faces. It's probably best not to try to visualize this.

A simplicial complex is a bunch of simplexes stuck together; formally, this means that we pretend that some of the corners (and any faces that include them) of different simplexes are identical points. There are ways to do this right using equivalence relations. But it's easier to abstract out the actual geometry and go straight to a combinatorial structure.

An (abstract) simplicial complex is just a collection of sets with the property that if A is a subset of B, and B is in the complex, then A is also

[2]For consistency, it's sometimes convenient to define a point as having a single (-1)-dimensional face defined to be the empty set. We won't need to bother with this, since 0-dimensional simplicial complexes correspond to 1-process distributed systems, which are amply covered in almost every other Computer Science class you have ever taken.

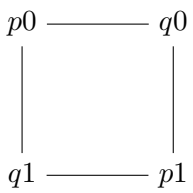
in the complex (this means that if some simplex is included, so are all of its faces, their faces, etc.). This combinatorial version is nice for reasoning about simplicial complexes, but is not so good for drawing pictures.

The trick to using this for distributed computing problems is that we are going to build simplicial complexes by letting points be process states (or sometimes process inputs or outputs), each labeled with a process ID, and letting the sets that appear in the complex be those collections of states/inputs/outputs that are compatible with each other in some sense. For states, this means that they all appear in some global configuration in some admissible execution of some system; for inputs and outputs, this means that they are permitted combinations of inputs or outputs in the specification of some problem.

Example: For 2-process binary consensus with processes 0 and 1, the **input complex**, which describes all possible combinations of inputs, consists of the sets

$$\{\{\}, \{p0\}, \{q0\}, \{p1\}, \{q1\}, \{p0, q0\}, \{p0, q1\}, \{p1, q0\}, \{p1, q1\}\},$$

which we might draw like this:

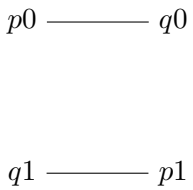


Note that there are no edges from $p0$ to $p1$ or $q0$ to $q1$: we can't have two different states of the same process in the same global configuration.

The **output complex**, which describes the permitted outputs, is

$$\{\{\}, \{p0\}, \{q0\}, \{p1\}, \{q1\}, \{p0, q0\}, \{p1, q1\}\}.$$

As a picture, this omits two of the edges (1-dimensional simplexes) from the input complex:



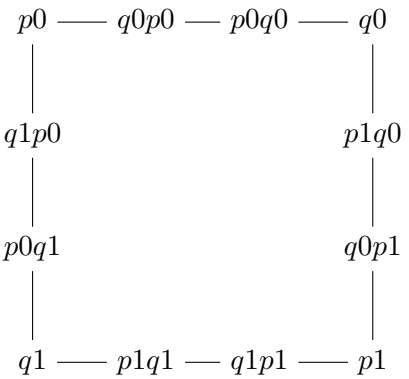
One thing to notice about this output complex is that it is not **connected**: there is no path from the $p0$–$q0$ component to the $q1$–$p1$ component.

Here is a simplicial complex describing the possible states of two processes p and q, after each writes 1 to its own bit then reads the other process's bit. Each node in the picture is labeled by a sequence of process IDs. The first ID in the sequence is the process whose view this node represents; any other process IDs are processes this first process sees (by seeing a 1 in the other process's register). So p is the view of process p running by itself, while pq is the view of process p running in an execution where it reads q's register after q writes it.

$$p \text{ ——— } qp \text{ ——— } pq \text{ ——— } q$$

The edges express the constraint that if we both write before we read, then if I don't see your value you must see mine (which is why there is no p–q edge), but all other combinations are possible. Note that this complex *is* connected: there is a path between any two points.

Here's a fancier version in which each process writes its input (and remembers it), then reads the other process's register (i.e., a one-round full-information protocol). We now have final states that include the process's own ID and input first, then the other process's ID and input if it is visible. For example, $p1$ means p starts with 1 but sees a null and $q0p1$ means q starts with 0 but sees p's 1. The general rule is that two states are compatible if p either sees nothing or q's actual input and similarly for q, and that at least one of p or q must see the other's input. This gives the following simplicial complex:



Again, the complex is connected.

The fact that this looks like four copies of the p–qp–pq–q complex pasted into each edge of the input complex is not an accident: if we fix a pair of inputs i and j, we get pi–$qjpi$–$piqj$–qj, and the corners are pasted together because if p sees only $p0$ (say), it can't tell if it's in the $p0/q0$ execution or the $p0/q1$ execution.

The same process occurs if we run a two-round protocol of this form, where the input in the second round is the output from the first round. Each round subdivides one edge from the previous round into three edges:

$$p - q$$

$$p - qp - pq - q$$

$$p - (qp)p - p(qp) - qp - (pq)(qp) - (qp)(pq) - pq - q(pq) - (pq)q - q$$

Here $(pq)(qp)$ is the view of p after seeing pq in the first round and seeing that q saw qp in the first round.

29.3.2 Subdivisions

In the simple write-then-read protocol above, we saw a single input edge turn into 3 edges. Topologically, this is an example of a **subdivision**, where we represent a simplex using several new simplexes pasted together that cover exactly the same points.

Certain classes of protocols naturally yield subdivisions of the input complex. The **iterated immediate snapshot** (IIS) model, defined by Borowsky and Gafni [BG97], considers executions made up of a sequence of rounds (the iterated part) where each round is made up of one or more mini-rounds in which some subset of the processes all write out their current views to their own registers and then take snapshots of all the registers (the immediate snapshot part). The two-process protocols of the previous section are special cases of this model.

Within each round, each process p obtains a view v_p that contains the previous-round views of some subset of the processes. We can represent the views as a subset of the processes, which we will abbreviate in pictures by putting the view owner first: pqr will be the view $\{p, q, r\}$ as seen by p, while qpr will be the same view as seen by q. The requirements on these views are that (a) every process sees its own previous view: $p \in v_p$ for all p; (b)

all views are comparable: $v_p \subseteq v_q$ or $v_q \subseteq v_p$; and (c) if I see you, then I see everything you see: $q \in v_p$ implies $v_q \subseteq v_p$. This last requirement is called **immediacy** and follows from the assumption that writes and snapshots are done in the same mini-round: if I see your write, then I see all the values you do, because your snapshot is either in an earlier mini-round than mine or in the same mini-round. Note this depends on the peculiar structure of the mini-rounds, where all the writes precede all the snapshots.

The IIS model does not correspond exactly to a standard shared-memory model (or even a standard shared-memory model augmented with cheap snapshots). There are two reasons for this: standard snapshots don't provide immediacy, and standard snapshots allow processes to go back and perform more than one snapshot on the same object. The first issue goes away if we are looking at impossibility proofs, because the adversary can restrict itself only to those executions that satisfy immediacy; alternatively, we can get immediacy from the **participating set** protocol of [BG97], which we will describe in §29.6.1. The second issue is more delicate, but Borowsky and Gafni demonstrate that any decision protocol that runs in the standard model can be simulated in the IIS model, using a variant of the BG simulation algorithm described in Chapter 28.

For three processes, one round of immediate snapshots gives rise to the simplicial complex depicted in Figure 29.1. The corners of the big triangle are the solo views of processes that do their snapshots before anybody else shows up. Along the edges of the big triangle are views corresponding to 2-process executions, while in the middle are complete views of processes that run late enough to see everything. Each little triangle corresponds to some execution. For example, the triangle with corners p, qp, rpq corresponds to a sequential execution where p sees nobody, q sees p, and r sees both p and q. The triangle with corners pqr, qpr, and rpq is the maximally-concurrent execution where all three processes write before all doing their snapshots: here everybody sees everybody. It is not terribly hard to enumerate all possible executions and verify that the picture includes all of them. In higher dimension, the picture is more complicated, but we still get a subdivision that preserves the original topological structure [BG97].

Figure 29.2 shows (part of) the next step of this process: here we have done two iterations of immediate snapshot, and filled in the second-round subdivisions for the p–qpr–rpq and pqr–qpr–rpq triangles. (Please imagine similar subdivisions of all the other triangles that I was too lazy to fill in by hand.) The structure is recursive, with each first-level triangle mapping to an image of the entire first-level complex. As in the two-process case, adjacent triangles overlap because the relevant processes don't have enough

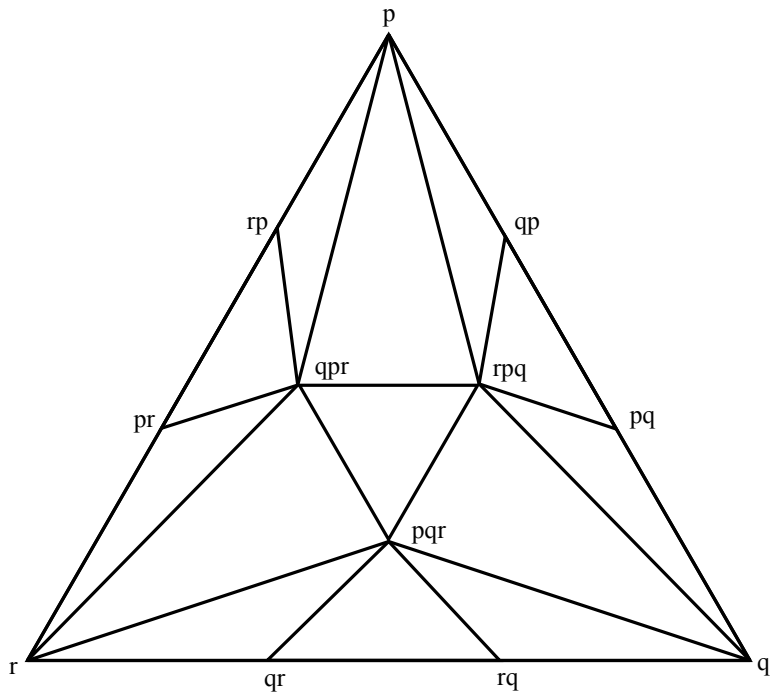


Figure 29.1: Subdivision corresponding to one round of immediate snapshot

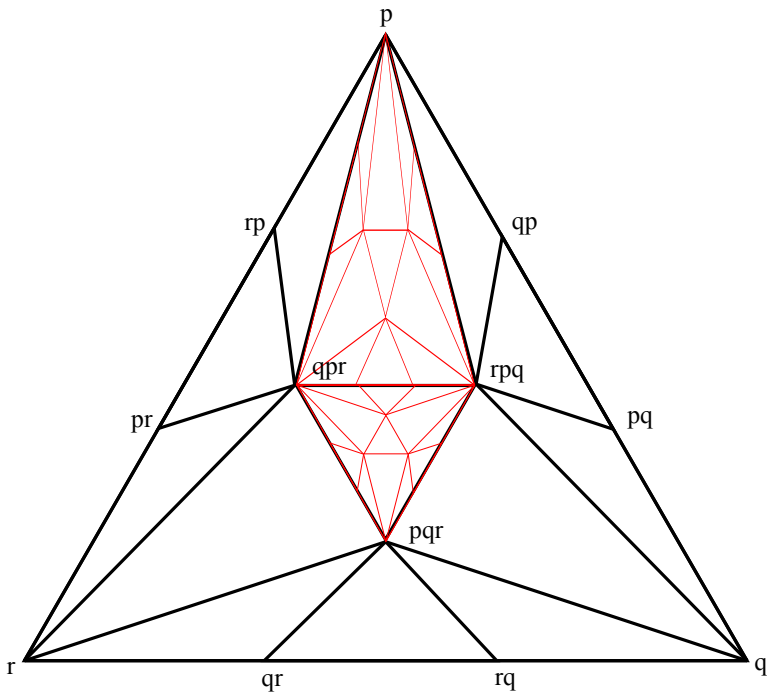


Figure 29.2: Subdivision corresponding to two rounds of immediate snapshot

information; for example, the points on the *qpr*–*rpq* edge correspond to views of q or r that don't include p in round 2 and so can't tell whether p saw p or *pqr* in round 1.

The important feature of the round-2 complex (and the round-k complex in general) is that it's a **triangulation** of the original outer triangle: a partition into little triangles where each corner aligns with corners of other little triangles.

(Better pictures of this process in action can be found in Figures 25 and 26 of [HS99].)

29.4 Impossibility of k-set agreement

Now let's show that there is no way to do k-set agreement with $n = k + 1$ processes in the IIS model.

Suppose that after some fixed number of rounds, each process chooses an output value. This output can only depend on the view of the process, so is fixed for each vertex in the subdivision. Also, the validity condition means

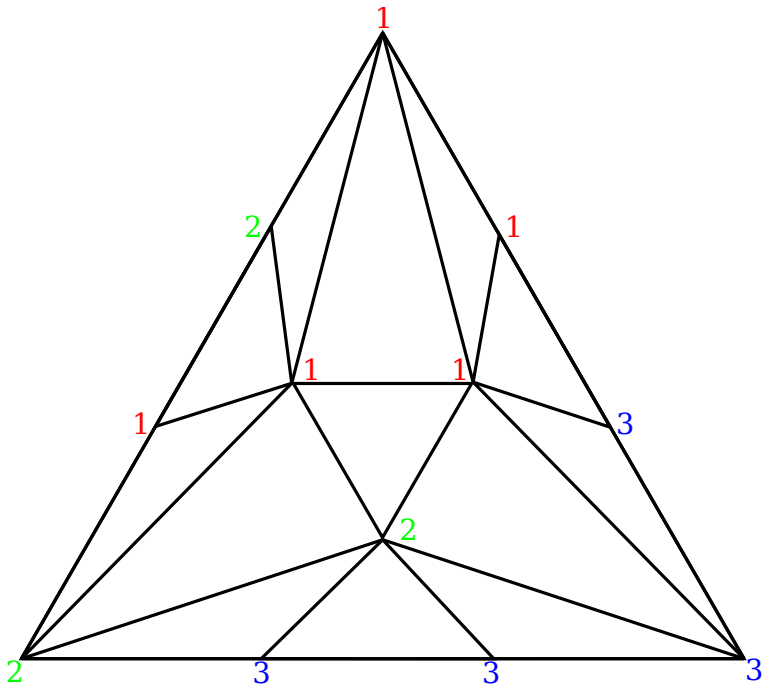


Figure 29.3: An attempt at 2-set agreement

that a process can only choose an output that it can see among the inputs in its view. This means that at the corners of the outer triangle (corresponding to views where the process thinks it's alone), a process must return its input, while along the outer edges (corresponding to views where two processes may see each other but not the third), a process must return one of the two inputs that appear in the corners incident to the edge. Internal corners correspond to views that include—directly or indirectly—the inputs of all processes, so these can be labeled arbitrarily. An example is given in Figure 29.3, for a one-round protocol with three processes.

We now run into Sperner's Lemma [Spe28], which says that, for any subdivision of a simplex into smaller simplexes, if each corner of the original simplex has a different color, and each corner that appears on some face of the original simplex has a color equal to the color of one of the corners of that face, then within the subdivision there are an odd number of simplexes whose corners are all colored differently.[3]

[3]The proof of Sperner's Lemma is not hard, and is done by induction on the dimension k. For $k = 0$, any subdivision consists of exactly one zero-dimensional simplex whose single

How this applies to k-set agreement: Suppose we have $n = k+1$ processes in a wait-free system (corresponding to allowing up to k failures). With the cooperation of the adversary, we can restrict ourselves to executions consisting of ℓ rounds of iterated immediate snapshot for some ℓ (termination comes in here to show that ℓ is finite). This gives a subdivision of a simplex, where each little simplex corresponds to some particular execution and each corner some process's view. Color all the corners of the little simplexes in this subdivision with the output of the process holding the corresponding view. Validity means that these colors satisfy the requirements of Sperner's Lemma. Sperner's Lemma then says that some little simplex has all $k+1$ colors, giving us a bad execution with more than k distinct output values.

The general result says that we can't do k-set agreement with k failures for any $n > k$. This follows immediately from the $n = k+1$ version using BG simulation (Chapter 28).

29.5 Simplicial maps and specifications

Let's step back and look at consensus again.

One thing we could conclude from the fact that the output complex for consensus was not connected but the ones describing our simple protocols were was that we can't solve consensus (non-trivially) using these protocols. The reason is that to solve consensus using such a protocol, we would need to have a mapping from states to outputs (this is just whatever rule tells each process what to decide in each state) with the property that if some collection of states are consistent, then the outputs they are mapped to are

corner covers all $k+1 = 1$ colors. For $k+1$, suppose that the colors are $\{1, \ldots, k+1\}$, and construct a graph with a vertex for each little simplex in the subdivision and an extra vertex for the region outside the big simplex. Put an edge in this graph between each pair of regions that share a k-dimensional face with colors $\{1, \ldots, k\}$. The induction hypothesis tells us that there are an odd number of edges between the outer-region vertex and simplexes on the $\{1, \ldots, k\}$-colored face of the big simplex. The Handshaking Lemma from graph theory says that the sum of the degrees of all the nodes in the graph is even. But this can only happen if there are an even number of nodes with odd degree, implying that the are are an odd number of simplexes in the subdivision with an odd number of faces colored $\{1, \ldots, k\}$, because the extra node for the outside region has exactly one face colored $\{1, \ldots, k\}$. Since zero is even, this means there is at least one simplex in the subdivision with an odd number of faces colored $\{1, \ldots, k\}$.

Now suppose we have a simplex with an odd number of faces colored $\{1, \ldots, k\}$. Let f be one such face. If the corner v not contained in f is colored $c \neq k+1$, then our simplex has exactly two faces colored $\{1, \ldots, k\}$: f, and the face that replaces f's c-colored corner with v. So the only way to get an odd number of $\{1, \ldots, k\}$-colored faces is to have all $k+1$ colors. It follows that there are an odd number of $(k+1)$-colored simplexes.

consistent.

In simplicial complex terms, this means that the mapping from states to outputs is a **simplicial map**, a function f from points in one simplicial complex C to points in another simplicial complex D such that for any simplex $A \in C$, $f(A) = \{f(x)|x \in A\}$ gives a simplex in D. (Recall that consistency is represented by including a simplex, in both the state complex and the output complex.) A mapping from states to outputs that satisfies the consistency requirements encoded in the output complex s always a simplicial map, with the additional requirement that it preserves process IDs (we don't want process p to decide the output for process q). Conversely, any id-preserving simplicial map gives an output function that satisfies the consistency requirements.

Simplicial maps are examples of **continuous functions**, which have all sorts of nice topological properties. One nice property is that a continuous function can't separate a path-connected space (one in which there is a path between any two points) into path-disconnected components. We can prove this directly for simplicial maps: if there is a path of 1-simplexes $\{x_1, x_2\}, \{x_2, x_3\}, \ldots \{x_{k-1}, x_k\}$ from x_1 to x_k in C, and $f : C \to D$ is a simplicial map, then there is a path of 1-simplexes $\{f(x_1), f(x_2)\}, \ldots$ from $f(x_1)$ to $f(x_k)$. Since being path-connected just means that there is a path between any two points, if C is connected we've just shown that $f(C)$ is as well.

Getting back to our consensus example, it doesn't matter what simplicial map f you pick to map process states to outputs; since the state complex C is connected, so is $f(C)$, so it lies entirely within one of the two connected components of the output complex. This means in particular that everybody always outputs 0 or 1: the protocol is trivial.

29.5.1 Mapping inputs to outputs

For general decision tasks, it's not enough for the outputs to be consistent with each other. They also have to be consistent with the inputs. This can be expressed by a relation Δ between input simplexes and output simplexes.

Formally, a decision task is modeled by a triple (I, O, Δ), where I is the input complex, O is the output complex, and $(A, B) \in \Delta$ if and only if B is a permissible output given input A. Here there are no particular restrictions on Δ (for example, it doesn't have to be a simplicial map or even a function), but it probably doesn't make sense to look at decision tasks unless there is at least one permitted output simplex for each input simplex.

29.6 The asynchronous computability theorem

Given a decision task specified in this way, there is a topological characterization of when it has a wait-free solution. This is given by the **Asynchronous Computability Theorem** (Theorem 3.1 in [HS99]), which says:

Theorem 29.6.1. *A decision task (I, O, Δ) has a wait-free protocol using shared memory if and only if there exists a chromatic subdivision σ of I and a color-preserving simplicial map $\mu : \sigma(I) \to O$ such that for each simplex S in $\sigma(I)$, $\mu(S) \in \Delta(\text{carrier}(S, I))$.*

To unpack this slightly, a **chromatic subdivision** is a subdivision where each vertex is labeled by a process ID (a color), and no simplex has two vertices with the same color. A color-preserving simplicial map is a simplicial map that preserves IDs. The carrier of a simplex in a subdivision is whatever original simplex it is part of. So the theorem says that I can only solve a task if I can find a simplicial map from a subdivision of the input complex to the output complex that doesn't do anything strange to process IDs and that is consistent with Δ.

Looking just at the theorem, one might imagine that the proof consists of showing that the **protocol complex** defined by the state complex after running the protocol to completion is a subdivision of the input complex, followed by the same argument we've seen already about mapping the state complex to the output complex. This is almost right, but it's complicated by two inconvenient facts: (a) the state complex generally isn't a subdivision of the input complex, and (b) if we have a map from an arbitrary subdivision of the input complex, it is not clear that there is a corresponding protocol that produces this particular subdivision.

So instead the proof works like this:

Protocol implies map Even though we don't get a subdivision with the full protocol, there is a restricted set of executions that does give a subdivision. So if the protocol works on this restricted set of executions, an appropriate map exists. There are two ways to prove this: Herlihy and Shavit do so directly, by showing that this restricted set of executions exists, and Borowksy and Gafni [BG97] do so indirectly, by showing that the IIS model (which produces exactly the standard chromatic subdivision used in the ACT proof) can simulate an ordinary snapshot model. Both methods are a bit involved, so we will skip over this part.

Map implies protocol This requires an algorithm. The idea here is that that **participating set** algorithm, originally developed to solve k-set agreement [BG93], produces precisely the standard chromatic subdivision used in the ACT proof. In particular, it can be used to solve the problem of **simplex agreement**, the problem of getting the processes to agree on a particular simplex contained within the subdivision of their original common input simplex. This is a little easier to explain, so we'll do it.

29.6.1 The participating set protocol

Algorithm 29.1 depicts the participating set protocol; this first appeared in [BG93], although the presentation here is heavily influenced by the version in Elizabeth Borowsky's dissertation [Bor95]. The shared data consists of a snapshot object level, and processes start at a high level and float down until they reach a level i such that there are already i processes at this level or below. The set returned by a process consists of all processes it sees at its own level or below, and it can be shown that this in fact implements a one-shot immediate snapshot. Since immediate snapshots yield a standard subdivision, this gives us what we want for converting a color-preserving simplicial map to an actual protocol.

```
1 Initially, level[i] = n + 2 for all i.
2 repeat
3 |   level[i] ← level[i] − 1
4 |   v ← snapshot(level)
5 |   S ← {j | v[j] ≤ level[i]}
6 until |S| ≥ level[i]
7 return S
```

Algorithm 29.1: Participating set

The following theorem shows that the return values from participating set have all the properties we want for iterated immediate snapshot:

Theorem 29.6.2. *Let S_i be the output of the participating set algorithm for process i. Then all of the following conditions hold:*

1. *For all i, $i \in S_i$. (Self-containment.)*
2. *For all i, j, $S_i \subseteq S_j$ or $S_j \subseteq S_i$. (Atomic snapshot.)*

3. *For all i, j, if $i \in S_j$, then $S_i \subseteq S_j$. (Immediacy.)*

Proof. Self-inclusion is trivial, but we will have to do some work for the other two properties.

We will show that Algorithm 29.1 neatly sorts the processes out into levels, where each process that returns at level ℓ returns precisely the set of processes at level ℓ and below.

For each process i, let S_i be the set of process IDs that i returns, let ℓ_i be the final value of $\mathsf{level}[i]$ when i returns, and let $S'_i = \{j \mid \ell_j \leq \ell_i\}$. Our goal is to show that $S'_i = S_i$, justifying the above claim.

Because no process ever increases its level, if process i observes $\mathsf{level}[j] \leq \ell_i$ in its last snapshot, then $\ell_j \leq \mathsf{level}[j] \leq \ell_i$. So S'_i is a superset of S_i. We thus need to show only that no extra processes sneak in; in particular, we will to show that $|S_i| = |S'_i|$, by showing that both equal ℓ_i.

The first step is to show that $|S'_i| \geq |S_i| \geq \ell_i$. The first inequality follows from the fact that $S'_i \supseteq S_i$; the second follows from the code (if not, i would have stayed in the loop).

The second step is to show that $|S'_i| \leq \ell_i$. Suppose not; that is, suppose that $|S'_i| > \ell_i$. Then there are at least $\ell_i + 1$ processes with level ℓ_i or less, all of which take a snapshot on level $\ell_i + 1$. Let i' be the last of these processes to take a snapshot while on level $\ell_i + 1$. Then i' sees at least $\ell_i + 1$ processes at level $\ell_i + 1$ or less and exits, contradicting the assumption that it reaches level ℓ_i. So $|S'_i| \leq \ell_i$.

The atomic snapshot property follows immediately from the fact that if $\ell_i \leq \ell_j$, then $\ell_k \leq \ell_i$ implies $\ell_k \leq \ell_j$, giving $S_i = S'_i \subseteq S'_j = S_j$. Similarly, for immediacy we have that if $i \in S_j$, then $\ell_i \leq \ell_j$, giving $S_i \leq S_j$ by the same argument. □

The missing piece for turning this into IIS is that in Algorithm 29.1, I only learn the identities of the processes I am supposed to include but not their input values. This is easily dealt with by the usual trick of adding an extra register for each process, to which it writes its input before executing participating set.

29.7 Proving impossibility results

To show something is impossible using the ACT, we need to show that there is no color-preserving simplicial map from a subdivision of I to O satisfying the conditions in Δ. This turns out to be equivalent to showing that there is no continuous function from I to O with the same properties,

because any such simplicial map can be turned into a continuous function (on the geometric version of I, which includes the intermediate points in addition to the corners). Fortunately, topologists have many tools for proving non-existence of continuous functions.

29.7.1 k-connectivity

Define the m-dimensional **disk** to be the set of all points at most 1 unit away from the origin in $\mathbb{R}^m$, and the m-dimensional **sphere** to be the surface of the $(m+1)$-dimensional disk (i.e., all points exactly 1 unit away from the origin in $\mathbb{R}^{m+1}$). Note that what we usually think of as a sphere (a solid body), topologists call a disk, leaving the term sphere for just the outside part.

An object is **k-connected** if any continuous image of an m-dimensional sphere can be extended to a continuous image of an $(m+1)$-dimensional disk, for all $m \leq k$.[4] This is a roundabout way of saying that if we can draw something that looks like a deformed sphere inside our object, we can always include the inside as well: there are no holes that get in the way. The punch line is that continuous functions preserve k-connectivity: if we want to map an object with no holes continuously into some other object, the image had better not have any holes either.

Ordinary path-connectivity is the special case when $k = 0$; here, the 0-sphere consists of two points and the 1-disk is the path between them. So 0-connectivity says that for any two points, there is a path between them.

For 1-connectivity, if we draw a loop (a path that returns to its origin), we can include the interior of the loop somewhere. One way to thinking about this is to say that we can shrink the loop to a point without leaving the object (the technical term for this is that the path is **null-homotopic**, where a **homotopy** is a way to transform one thing continuously into another thing over time and the **null path** sits on a single point). An object that is 1-connected is also called **simply connected**.

For 2-connectivity, we can't contract a sphere (or box, or the surface of a 2-simplex, or anything else that looks like a sphere) to a point.

The important thing about k-connectivity is that it is possible to prove that any subdivision of a k-connected simplicial complex is also k-connected (sort of obvious if you think about the pictures, but it can also be proved formally), and that k-connectivity is preserved by simplicial maps (if not,

[4] This definition is for the topological version of k-connectivity. It is not related in any way to the definition of k-connectivity in graph theory, where a graph is k-connected if there are k disjoint paths between any two points.

somewhere in the middle of all the k-simplexes representing our surface is a $(k+1)$-simplex in the domain that maps to a hole in the range, violating the rule that simplicial maps map simplexes to simplexes). So a quick way to show that the Asynchronous Computability Theorem implies that something is not asynchronously computable is to show that the input complex is k-connected and the output complex isn't.

29.7.2 Impossibility proofs for specific problems

Here are some applications of the Asynchronous Computability Theorem and k-connectivity:

Consensus There is no nontrivial wait-free consensus protocol for $n \geq 2$ processes. Proof: The input complex is 1-connected, but the output complex is not, and we need a map that covers the entire output complex (by nontriviality).

k-set agreement There is no wait-free k-set agreement for $n \geq k+1$ processes. Proof: The output complex for k-set agreement is not k-connected, because buried inside it are lots of $(k+1)$-dimensional holes corresponding to missing simplexes where all $k+1$ processes choose different values. But these holes aren't present in the input complex—it's OK if everybody starts with different inputs—and the validity requirements for k-set agreement force us to map the surfaces of these non-holes around holes in the output complex. (This proof actually turns into the Sperner's Lemma proof if we fully expand the claim about having to map the input complex around the hole.)

Renaming There is no wait-free renaming protocol with less than $2n-1$ output names for all n. The general proof of this requires showing that with fewer names we get holes that are too big (and ultimately reduces to Sperner's Lemma); for the special case of $n = 3$ and $m = 4$, see Figure 29.4, which shows how the output complex of renaming folds up into the surface of a torus. This means that renaming for $n = 3$ and $m = 4$ is *exactly the same* as trying to stretch a basketball into an inner tube.

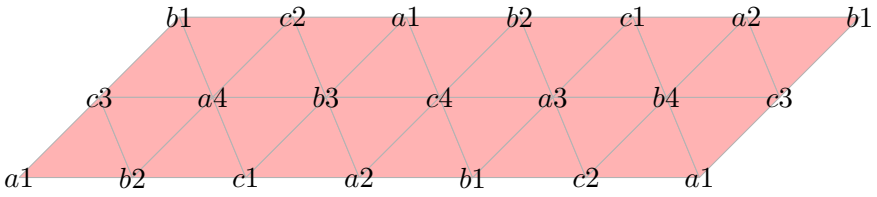


Figure 29.4: Output complex for renaming with $n = 3$, $m = 4$. Each vertex is labeled by a process ID (a, b, c) and a name $(1, 2, 3, 4)$. Observe that the left and right edges of the complex have the same sequence of labels, as do the top and bottom edges; the complex thus folds up into a (twisted) torus. (This is a poor imitation of part of [HS99, Figure 9].)

Chapter 30

Approximate agreement

Last updated 2011. Some material may be out of date.

The **approximate agreement** [DLP+86] or **ϵ-agreement** problem is another relaxation of consensus where input and output values are real numbers, and a protocol is required to satisfy modified validity and agreement conditions.

Let x_i be the input of process i and y_i its output. Then a protocol satisfies approximate agreement if it satisfies:

Termination Every nonfaulty process eventually decides.

Validity Every process returns an output within the range of inputs. Formally, for all i, it holds that $(\min_j x_j) \leq y_i \leq (\max_j x_j)$.

ϵ-agreement For all i and j, $|i - j| \leq \epsilon$.

Unlike consensus, approximate agreement has wait-free algorithms for asynchronous shared memory, which we'll see in §30.1). But a curious property of approximate agreement is that it has no **bounded wait-free** algorithms, even for two processes (see §30.2)

30.1 Algorithms for approximate agreement

Not only is approximate agreement solvable, it's actually easily solvable, to the point that there are many known algorithms for solving it.

We'll use the algorithm of Moran [Mor95], mostly as presented in [AW04, Algorithm 54] but with a slight bug fix;[1] pseudocode appears in Algorithm 30.1.[2]

The algorithm carries out a sequence of asynchronous rounds in which processes adopt new values, such that the **spread** of the vector of all values V_r in round r, defined as $\text{spread}\, V_r = \max V_r - \min V_r$, drops by a factor of 2 per round. This is done by having each process choose a new value in each round by taking the midpoint (average of min and max) of all the values it sees in the previous round. Slow processes will jump to the maximum round they see rather than propagating old values up from ancient rounds; this is enough to guarantee that latecomer values that arrive after some process writes in round 2 are ignored.

The algorithm uses a single snapshot object A to communicate, and each process stores its initial input and a round number along with its current preference. We assume that the initial values in this object all have round number 0, and that $\log_2 0 = -\infty$ (which avoids a special case in the termination test).

```
1 A[i] ← ⟨x_i, 1, x_i⟩
2 repeat
3 |   ⟨x'_1, r_1, v_1⟩ ... ⟨x'_n, r_n, v_n⟩ ← snapshot(A)
4 |   r_max ← max_j r_j
5 |   v ← midpoint{v_j | r_j = r_max}
6 |   A[i] ← ⟨x_i, r_max + 1, v⟩
7 until r_max ≥ 2 and r_max ≥ log_2(spread({x'_j})/ε)
8 return v
```

Algorithm 30.1: Approximate agreement

To show this works, we want to show that the midpoint operation guarantees that the spread shrinks by a factor of 2 in each round. Let V_r

[1] The original algorithm from [AW04] does not include the test $r_{\max} \geq 2$. This allows for bad executions in which process 1 writes its input of 0 in round 1 and takes a snapshot that includes only its own input, after which process 2 runs the algorithm to completion with input 1. Here process 2 will see 0 and 1 in round 1, and will write $(1/2, 2, 1)$ to $A[2]$; on subsequent iterations, it will see only the value 1/2 in the maximum round, and after $\lceil \log_2(1/\epsilon) \rceil$ rounds it will decide on 1/2. But if we now wake process 1 up, it will decided 0 immediately based on its snapshot, which includes only its own input and gives $\text{spread}(x) = 0$. Adding the extra test prevents this from happening, as new values that arrive after somebody writes round 2 will be ignored.

[2] Showing that this particular algorithm works takes a lot of effort. If I were to do this over, I'd probably go with a different algorithm due to Schenk [Sch95].

be the set of all values v that are ever written to the snapshot object with round number r. Let $U_r \subseteq V_r$ be the set of values that are ever written to the snapshot object with round number r before some process writes a value with round number $r+1$ or greater; the intuition here is that U_r includes only those values that might contribute to the computation of some round-$(r+1)$ value.

Lemma 30.1.1. *For all r for which V_{r+1} is nonempty,*

$$\text{spread}(V_{r+1}) \leq \text{spread}(U_r)/2.$$

Proof. Let U_r^i be the set of round-r values observed by a process i in the iteration in which it sees $r_{\max} = r$ in some iteration, if such an iteration exists. Note that $U_r^i \subseteq U_r$, because if some value with round $r+1$ or greater is written before i's snapshot, then i will compute a larger value for $r_{\max}$.

Given two processes i and j, we can argue from the properties of snapshot that either $U_r^i \subseteq U_r^j$ or $U_r^j \subseteq U_r^i$. The reason is that if i's snapshot comes first, then j sees at least as many round-r values as i does, because the only way for a round-r value to disappear is if it is replaced by a value in a later round. But in this case, process j will compute a larger value for $r_{\max}$ and will not get a view for round r. The same holds in reverse if j's snapshot comes first.

Observe that if $U_r^i \subseteq U_r^j$, then

$$\left|\text{midpoint}(U_r^i) - \text{midpoint}(U_r^j)\right| \leq \text{spread}(U_r^j)/2.$$

This holds because $\text{midpoint}(U_r^i)$ lies within the interval $\left[\min U_r^j, \max U_r^j\right]$, and every point in this interval is within $\text{spread}(U_r^j)/2$ of $\text{midpoint}(U_r^j)$. The same holds if $U_r^j \subseteq U_r^i$. So any two values written in round $r+1$ are within $\text{spread}(U_r)/2$ of each other.

In particular, the minimum and maximum values in V_{r+1} are within $\text{spread}(U_r)/2$ of each other, so $\text{spread}(V_{r+1}) \leq \text{spread}(U_r)/2$. □

Corollary 30.1.2. *For all $r \geq 2$ for which V_r is nonempty,*

$$\text{spread}(V_r) \leq \text{spread}(U_1)/2^{r-1}.$$

Proof. By induction on r. For $r = 2$, this is just Lemma 30.1.1. For larger r, use the fact that $U_{r-1} \subseteq V_{r-1}$ and thus $\text{spread}(U_{r-1}) \leq \text{spread}(V_{r-1})$ to

compute

$$\begin{aligned}\text{spread}(V_r) &\leq \text{spread}(U_{r-1})/2 \\ &\leq \text{spread}(V_{r-1})/2 \\ &\leq (\text{spread}(U_1)/2^{r-2})/2 \\ &= \text{spread}(U_1)/2^{r-1}.\end{aligned}$$

□

Let i be some process that finishes in the fewest number of rounds. Process i can't finish until it reaches round $r_{\max}+1$, where $r_{\max} \geq \log_2(\text{spread}(\{x'_j\})/\epsilon)$ for a vector of input values x' that it reads after some process writes round 2 or greater. We have $\text{spread}(\{x'_j\}) \geq \text{spread}(U_1)$, because every value in U_1 is included in x'. So $r_{\max} \geq \log_2(\text{spread}(U_1)/\epsilon)$ and $\text{spread}(V_{r_{\max}+1}) \leq \text{spread}(U_1)/2^{r_{\max}} \leq \text{spread}(U_1)/(\text{spread}(U_1)/\epsilon) = \epsilon$. Since any value returned is either included in $V_{r_{\max}+1}$ or some later $V_{r'} \subseteq V_{r_{\max}+1}$, this gives us that the spread of all the outputs is less than ϵ: Algorithm 30.1 solves approximate agreement.

The cost of Algorithm 30.1 depends on the cost of the snapshot operations, on ϵ, and on the initial input spread D. For linear-cost snapshots, this works out to $O(n \log(D/\epsilon))$.

30.2 Lower bound on step complexity

The dependence on D/ϵ is necessary, at least for deterministic algorithms. Here we give a lower bound due to Herlihy [Her91a], which shows that any deterministic approximate agreement algorithm takes at least $\log_3(D/\epsilon)$ total steps even with just two processes.

Define the **preference** of a process in some configuration as the value it will choose if it runs alone starting from this configuration. The preference of a process p is well-defined because the process is deterministic; it also can only change as a result of a write operation by another process q (because no other operations are visible to p, and p's own operations can't change its preference). The validity condition means that in an initial state, each process's preference is equal to its input.

Consider an execution with two processes p and q, where p starts with preference p_0 and q starts with preference q_0. Run p until it is about to perform a write that would change q's preference. Now run q until it is about to change p's preference. If p's write no longer changes q's preference, start p

again and repeat until both p and q have pending writes that will change the other process's preference. Let p_1 and q_1 be the new preferences that result from these operations. The adversary can now choose between running P only and getting to a configuration with preferences p_0 and q_1, Q only and getting p_1 and q_0, or both and getting p_1 and q_1; each of these choices incurs at least one step. By the triangle inequality, $|p_0 - q_0| \leq |p_0 - q_1| + |q_1 - p_1| + |p_1 - q_0|$, so at least on of these configurations has a spread between preferences that is at least $1/3$ of the initial spread. It follows that after k steps the best spread we can get is $D/3^k$, requiring $k \geq \log_3(D/\epsilon)$ steps to get ϵ-agreement.

Herlihy uses this result to show that there are decisions problems that have wait-free but not bounded wait-free deterministic solutions using registers. Curiously, the lower bound says nothing about the dependence on the number of processes; it is conceivable that there is an approximate agreement protocol with running time that depends only on D/ϵ and not n.

Part III

Other communication models

Chapter 31

Overview

In this part, we consider models that don't fit well into the standard message-passing or shared-memory models. These includes models where processes can directly observe the states of nearby processes (Chapter 32); where computation is inherently local and the emphasis is on computing information about the communication graph (Chapter 33); where processes wander about and exchange information only with processes they physically encounter (Chapter 34); where processes (in the form of robots) communicate only by observing each others' locations and movements (Chapter 35); and where processes can transmit only beeps, and are able to observe only whether at least one nearby process beeped (Chapter 36).

Despite the varying communication mechanisms, these models all share the usual features of distributed systems, where processes must contend with nondeterminism and incomplete local information.

Chapter 32

Self-stabilization

A **self-stabilizing** algorithm has the property that, starting from any arbitrary configuration, it eventually reaches a **legal** configuration, and this property is **stable** in the sense that it remains in a legal configuration thereafter. The notion of which configurations are legal depends on what problem we are trying to solve, but the overall intuition is that an algorithm is self-stabilizing if it can recover from arbitrarily horrible errors, and will stay recovered as long as no new errors occur.

It's generally not possible to detect whether the algorithm is in a legal configuration from the inside: if a process has a bit that says that everything is OK, the adversary can set that bit in the initial configuration, even if everything is in fact broken. So self-stabilizing algorithms don't actually terminate: at best, they eventually converge to a configuration where the necessary ongoing paranoid consistency checks produce no further changes to the configuration (a property called **silent self-stabilization**.

The idea of self-stabilization first appeared in a paper by Edsger Dijkstra [Dij74], where he considered the problem of building robust token-ring networks. In a token-ring network, there are n nodes arranged in a directed cycle, and we want a single token to circulate through the nodes, as a mechanism for enforcing mutual exclusion: only the node currently possessing the token can access the shared resource.

The problem is: how do you get the token started? Dijkstra worried both about the possibility of starting with no tokens or with more than one token, and he wanted an algorithm that would guarantee that, from any starting state, eventually we would end up with exactly one token that would circulate as desired. He called such an algorithm **self-stabilizing**, and gave three examples, the simplest of which we will discuss in §32.2 below. These became

the foundation for the huge field of self-stabilization, which spans thousands of papers, at least one textbook [Dol00], a specialized conference (SSS, the *International Symposium on Stabilization, Safety, and Security in Distributed Systems)*, and its own domain name http://www.selfstabilization.org/. We won't attempt to summarize all of this, but will highlight a few results to give a sampling of what self-stabilizing algorithms look like.

32.1 Model

Much of the work in this area, dating back to Dijkstra's original paper, does not fit well in either the message-passing or shared-memory models that we have been considering in this class, both of which were standardized much later. Instead, Dijkstra assumed that processes could, in effect, directly observe the states of their neighbors. A self-stabilizing program would consist of a collection of what he later called **guarded commands** [Dij75], statements of the form "if [some condition is true] then [update my state in this way]." In any configuration of the system, one or more of these guarded commands might have the if part (the **guard**) be true; these commands are said to be **enabled**.

A step consists of one or more of these enabled commands being executed simultaneously, as chosen by an adversary scheduler, called the distributed daemon. The usual fairness condition applies: any process that has an enabled command eventually gets to execute it. If no commands are enabled, nothing happens. With the **central daemon** variant of the model, only one step can happen at a time. With the **synchronous daemon**, every enabled step happens at each time. Note that both the central and synchronous daemons are special cases of the distributed daemon.

More recent work has tended to assume a distinction between the part of a process's state that is visible to its neighbors and the part that isn't. This usually takes the form of explicit **communication registers** or **link registers**, which allow a process to write a specific message for a specific neighbor. This is still not quite the same as standard message-passing or shared-memory, because a process is often allowed to read and write multiple link registers atomically.

32.2 Token ring circulation

For example, let us consider Dijkstra's token ring circulation algorithm. There are several versions of this in Dijkstra's paper [Dij74]. We will do the

unidirectional $(n+1)$-state version, which is the simplest to describe and analyze.

For this algorithm, the processes are numbered as elements $0 \ldots n-1$, with all arithmetic on process IDs being done modulo n.[1] Each process i can observe both its own state and that of its predecessor at $(i-1) \bmod n$.

Process 0 has a special role and has different code from the others, but the rest of the processes are symmetric. Each process i has a variable ℓ_i that takes on values in the range $0 \ldots n$, interpreted as elements of $\mathbb{Z}_{n+1}$. The algorithm is given in Algorithm 32.1.

```
1 Code for process 0:
2 if ℓ_0 = ℓ_{n-1} then ℓ'_0 ← (ℓ_{n-1} + 1) mod (n + 1)
3 Code for process i ≠ 0:
4 if ℓ_i ≠ ℓ_{i-1} then ℓ'_i ← ℓ_{i-1}
```

Algorithm 32.1: Dijkstra's large-state token ring algorithm [Dij74]

In this algorithm, the nonzero processes just copy the state of the process to their left. The zero process increments its state if it sees the same state to its left. Note that the nonzero processes have guards on their commands that might appear useless at first glance, but these are there ensure that the adversary can't waste steps by getting nonzero processes to carry out operations that have no effect.

What does this have to with tokens? The algorithm includes an additional interpretation of the state, which says that:

1. If $\ell_0 = \ell_{n-1}$, then 0 has a token, and
2. If $\ell_i \neq \ell_{i-1}$, for $i \neq 0$, then i has a token.

Like the update rule, the token rule can be evaluated by a node that can only see its predecessor. This allows it to do detect when it acquires the token and do whatever leaderly things it needs to before applying an update to pass it on to the next process.

Using the token rule instantly guarantees that there is at least one token: if none of the nonzero processes have a token, then all the ℓ_i variables are equal. But then 0 has a token. It remains though to show that we eventually converge to a configuration where at most one process has a token.

[1] In Dijkstra's paper, there are $n+1$ processes numbered $0 \ldots n$, but this doesn't really make any difference.

Define a configuration ℓ as legal if there is some value j such that $\ell_i = \ell_j$ for all $i \leq j$ and $\ell_i = \ell_j - 1 \pmod{n+1}$ for all $i > j$. When $j = n-1$, this makes all ℓ_i equal, and 0 has the only token. When $j < n-1$, then $\ell_0 \neq \ell_{n-1}$ (so 0 does not have a token), $\ell_j \neq \ell_{j+1}$ (so $j+1$ has a token), and $\ell_i = \ell_{i+1}$ for all $i \notin j, n-1$ (so nobody else has a token). That each legal configuration has exactly one token partially justifies our definition of legal configurations.

If a configuration ℓ is legal, then when $j = n-1$, the only enabled step is $\ell'_0 \leftarrow (\ell_{n-1} + 1) \bmod (n+1)$; when $j < n-1$, the only enabled step is $\ell'_{j+1} \leftarrow \ell_j$. In either case, we get a new legal configuration ℓ'. So the property of being a legal configuration is stable, which is the other half of justifying our definition.

Now we want to show that we eventually converge to a legal configuration. Fix some initial configuration ℓ^0, and let c be some value such that $\ell^0_i \neq c$ for all i. (There is at least one such c by the Pigeonhole Principle.) We will argue that there is a sequence of configurations with c as a prefix of the values that forms a bottleneck forcing us into a legal configuration.

Lemma 32.2.1. *Let $\ell^0, \ell^1, \ldots$ be the sequence of configurations in some execution of Dijkstra's token ring circulation algorithm. Let $0 \leq c \leq n$ be such that $\ell^0_i \neq c$ for all i. Then for any configuration ℓ^t, either t is legal, or there is some $0 \leq j < n$ such that $\ell^t_i = c$ if and only if $i < j$.*

Proof. By induction on t. For the base case, ℓ^0 satisfies $\ell^0_i = c$ if and only if $i < j$ when $j = 0$.

If ℓ^t is legal, ℓ^{t+1} is also legal. So the interesting case is when ℓ^t is not legal. In this case, there is some $0 \leq j < n$ such that $\ell^t_i = c$ if and only if $i < j$.

If $j = 0$, then $\ell^t_i \neq c$ for all i. Then the only way to get $\ell^{t+1}_i = c$ is if $i = 0$. But then ℓ^{t+1} satisfies the condition with $j' = 1$.

If $0 < j < n$, then $\ell^t_i = c$ for at least one $i < j$, and $\ell^t_{n-1} \neq c$ since $n - 1 \not< j$. So we may get a transition that sets $\ell^{t+1}_j = \ell^t_{j-1} = c$, giving a new configuration ℓ^{t+1} that satisfies the induction hypothesis with $j' = j + 1$, or we may get a transition that does not create or remove any copies of c. In either case the induction goes through. □

To show that we eventually hit this bottleneck, we use a potential function. Starting in some initial configuration ℓ^0, let c be some missing value in ℓ^0 as defined above. For any configuration ℓ, define $g(\ell) = (c - \ell_0) \bmod (n+1)$ to be the gap between ℓ_0 and c. For each $i \in \{0, \ldots, n-2\}$, define $u_i(\ell) = [\ell_i \neq \ell_{i+1}]$ to be the indicator variable for whether i is *unhappy* with its

successor, because its successor has not yet agreed to adopt its value.[2] The idea is that unhappiness moves right when some $i \neq 0$ copies its predecessor and that the gap drops when 0 increments its value. By weighting these values appropriately, we can arrange for a function that always drops.

Let

$$\Phi(\ell) = ng(\ell) + \sum_{i=0}^{n-2}(n-1-i)u_i(\ell). \tag{32.2.1}$$

Most of the work here is being done by the first two terms. The g term tracks the gap between ℓ_0 and c, weighted by n. The sum tracks unhappiness, weighted by distance to position $n-1$.

In the initial configuration ℓ^0, g is at most n, and each u_i is at most 1, so $\Phi(\ell^0) = O(n^2)$. We also have that $\Phi \geq 0$ always; if $\Phi = 0$, then $g = 0$ and $u_i = 0$ for all i implies we are in an all-c configuration, which is legal. So we'd like to argue that every step of the algorithm in a non-legal configuration reachable from ℓ^0 reduces Φ by at least 1, forcing us into a legal configuration after $O(n^2)$ steps.

Consider any step of the algorithm starting from a non-legal configuration ℓ^t with $\Phi(\ell^t) > 0$ that satisfies the condition in Lemma 32.2.1:

- If it is a step by $i \neq 0$, then u_{i-1} changes from 1 to 0, reducing Φ by $(n-1-(i-1)) = n-i$. It may be that u_i changes from 0 to 1, increasing Φ by $n-i-1$, but the sum of these changes is at most -1.
- If it is a step by 0, then u_0 may increase from 0 to 1, increasing Φ by $n-1$. But g drops by 1 as long as $\ell_0^t \neq c$, reducing Φ by n, for a total change of at most -1. The case $\ell^t = c$ is excluded by the assumption that ℓ^t is non-legal and satisfies the conditions of Lemma 32.2.1, as the only way for 0 to change its value away from c is if ℓ_{n-1}^t is also c.

Since the condition of Lemma 32.2.1 holds for any reachable ℓ^t, as long as we are in a non-legal configuration, Φ drops by at least 1 per step. If we do not reach a legal configuration otherwise, Φ can only drop $O(n^2)$ times before hitting 0, giving us a legal configuration. Either way, the configuration stabilizes in $O(n^2)$ steps.

32.3 Synchronizers

Self-stabilization has a curious relationship with failures: the arbitrary initial state corresponds to an arbitrarily bad initial disruption of the system, but

[2] The notation $[P]$, where P is some logical predicate, is called an **Iverson bracket** and means the function that is 1 when P is true and 0 when P is false.

once we get past this there are no further failures. So it is not surprising that many of the things we can do in a failure-free distributed system we can also do in a self-stabilizing system. One of these is to implement a synchronizer, which will allow us to pretend that our system is synchronous even if it isn't.

The synchronizer we will describe here, due to Awerbuch *et al.* [AKM+93, AKM+07], is a variant of the alpha synchronizer. It assumes that each process can observe the states of its neighbors and that we have a central daemon (meaning that one process takes a step at a time).

To implement this synchronizer in a self-stabilizing system, each process v has a variable $P(v)$, its current pulse. We also give each process a rule for adjusting $P(v)$ when it takes a step. Our goal is to arrange for every v to increment its pulse infinitely often while staying at most one ahead of its neighbors $N(v)$. Awerbuch *et al.* give several possible rules for achieving this goal, and consider the effectiveness of each.

The simplest rule is taken directly from the alpha synchronizer. When activated, v sets

$$P(v) \leftarrow \min_{u \in N(v)} (P(u) + 1)$$

This rule works find as long as every process starts synchronized. But it's not self-stabilizing. A counterexample, given in the paper, assumes we have 10 processes organized in a ring. By carefully choosing which processes are activated at each step, we can go through the following sequence of configurations, where in each configuration the updated node is shown in boldface:

1234312343
1234**2**12343
12342**3**2343
1234234343
123423**4**343
1234234**5**43
12342345**4**2
3234234542
3**4**34234542
34**5**4234542

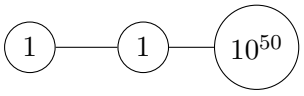
Here the final configuration is identical to the original if we increment each value by one and shift the values to the left one position. So we can continue this process indefinitely. But at the same time, each configuration has at least one pair of adjacent nodes whose values differ by more than one.

The problem that arises in the counterexample is that sometimes values can go backwards. A second rule proposed by Awerbuch *et al.* avoids this

problem by preventing this, using the rule:

$$P(v) \leftarrow \max\left(P(v), \min_{u \in N(v)} (P(u) + 1)\right)$$

This turns out to be self-stabilizing, but the time to stabilize is unbounded even in small networks. One counterexample is a network consisting of just three nodes:



If we run the nodes in round-robin order, the left two nodes will eventually catch up to the rightmost, but it will take a while.

After some further tinkering, the authors present their optimal rule, which they call **max minus one**:

$$P(v) \leftarrow \begin{cases} \min_{u \in N(v)}(P(u) + 1) & \text{if } P(v) \text{ looks legal,} \\ \max_{u \in N(v)}(P(u) - 1) & \text{otherwise.} \end{cases}$$

Here $P(v)$ looks legal if it is within ± 1 of all of its neighbors.

The intuition for why this works is that the most backward node pulls the rest down to its level in $O(D)$ time[3] using the max-minus-one rule, after which we just get the alpha synchronizer since everybody's local values look legal.

The actual proof uses a potential function at each node v given by

$$\phi(v) = \max_u (P(u) - P(v) - d(u, v)),$$

where $d(u, v)$ is the distance between u and v in the graph. This is zero if the skew between any pair of nodes is equal to the distance, which is the most we can expect from a synchronizer. The proof shows that applying the max-minus-one rule never increases $\phi(v)$, and decreases it by at least 1 whenever a node v with positive $\phi(v)$ changes $P(v)$. Because this only gives a bound of $\sum \phi(v)$, which can be arbitrarily big, the rest of the proof uses a second potential function

$$\Phi(v) = \min_u \left\{ d(u, v) \mid P(u) - P(v) - d(u, v) = \phi(v) \right\},$$

[3] Defining a time unit as a minimum interval in which every process takes at least one step.

which measures the distance from v to the nearest node u that supplies the maximum in $\phi(v)$. It is shown that $\Phi(v)$ drops by 1 per time unit. When it reaches 0, then $\phi(v) = P(v) - P(v) - d(v, v) = 0$. Since $\Phi(v)$ can never start at more than the diameter D, this implies convergence in D time units.

The intuition for why this works is that if the closest node u to v with $P(u)$ too high is at distance d, then max-minus-one will pull $P(w)$ up for some node w at distance $d - 1$ the next time w takes a step. The full set of cases is more complicated, and we'll skip over the details of the argument here. If you are interested, the presentation in the paper is not too hard to follow.

The important part is that once we have a synchronizer, we can effectively assume synchrony in other self-stabilizing algorithms. We just run the synchronizer underneath our main protocol, and when the synchronizer stabilizes, that gives us the initial starting point for the main protocol. Because the main protocol itself should stabilize starting from an arbitrary configuration, any insanity produced while waiting for the synchronizer to converge is eventually overcome.

32.4 Spanning trees

The straightforward way to construct a spanning tree in a graph is to use Bellman-Ford [Bel58, For56] to compute a breadth-first search tree rooted at the node with lowest ID. This has a natural implementation in the self-stabilizing model: each process maintains root and dist, and when a process takes a step, it sets root to the minimum of its own ID and the minimum root among its neighbors, and sets dist to 0 if it has the minimum ID, or to one plus the minimum distance to the root among its neighbors otherwise. It is not hard to show that in the absence of errors, this converges to a configuration where every node knows the ID of the root and its distance to the root in $O(D)$ time, where D is the diameter of the network. A spanning tree can then be extracted by the usual trick of having each node select as parent some neighbor closer to the root.

But it fails badly if the system starts in an arbitrary state, because of the **ghost root** problem. Suppose that some process wakes up believing in the existence of a distant, fake root with lower ID than any real process. This fake root will rapidly propagate through the other nodes, with distances increasing without bound over time. For most graphs, the algorithm will never converge to a single spanning tree.

Awerbuch *et al.* [AKM+93] solve this problem by assuming a known

upper bound on the maximum diameter of the graph. Because the distance to a ghost root will steadily increase over time, eventually only real roots will remain, and the system will converge to a correct BFS tree.

It's easiest to show this if we assume synchrony, or at least some sort of asynchronous round structure. Define a round as the minimum time for every node to update at least once. Then the minimum distance for any ghost root rises by at least one per round, since any node with the minimum distance either has no neighbor with the ghost root (in which case it picks a different root), or any neighbor that has the ghost root has at least the same distance (in which case it increases its distance) Once the minimum distance exceeds the upper bound D', all the ghost roots will have been eliminated, and only real distances will remain. This gives a stabilization time (in rounds) linear in the upper bound on the diameter.

32.5 Self-stabilization and local algorithms

In Chapter 33, we will look at algorithms in the **LOCAL** model, where named processes in a synchronous network, organized as an unknown graph, can send a polynomial-sized message to each neighbor in each round and perform arbitrary computation locally. The goal is usually to compute some property of the graph quickly, often in significantly fewer rounds than the diameter of the graph.

There is a close connection between self-stabilizing algorithms and the LOCAL model. The idea is that if we have a local algorithm that runs in $f(n)$ rounds, each process can propagate its information in a self-stabilizing way to all nodes at distance at most $f(n)$, and we can reconstruct the output of the local algorithm whenever this information changes.

For each node u, let x_u be its input value; we assume that this is fixed once the system stabilizes. The state of u will be a table T_u, where T_u is a partial function from sequences of nodes of length at most $f(n)$ to input values. We can represent this partial function as a set of ordered pairs $T_u = \langle w, x\rangle$, where we write $T_u(w) = x$ if x is the unique value such that $\langle w, x\rangle \in T_u$, or $T_u(w) = \perp$ if there is no such value. We have one rule at each node u, which we can imagine is guarded so that it fires only if it changes T_u:

$$T_u \leftarrow \{\langle u, x_u\rangle\} \cup \bigcup_{v\in\delta(u)} \{\langle uw, x\rangle \mid |uw| \le f(n), T_v(w) = x\} \tag{32.5.1}$$

We can now argue that, after stabilization, this process eventually converges to T_u consisting precisely of the set of all pairs $\langle w, x_v\rangle$ where w is a

u–v path of length at most $f(n)$ and x_v is the input to v. Indeed, this works under almost any reasonable assumption about scheduling. The relevant lemma:

Lemma 32.5.1. *Starting from any initial configuration, for any sequence w of at most $f(n)$ vertices starting at u and ending at v, if (32.5.1) fires for each node in w in reverse order, then $T_u(w) = x_v$ if w is a u–v path, and $T_u(w) = \perp$ otherwise.*

Proof. The proof is by induction on the length of w. The base case is when $|w| = 1$, implying $w = u = v$. Here rule (32.5.1) writes $\langle u, x_u \rangle$ to T_u, giving $T_u(u) = x_u$ as claimed.

For a sequence $w = uw'$ where w' is a nonempty path from some node u' to v, if u' is a neighbor of u, then firing rule (32.5.1) at u after firing the rule for each node in w' has $T_u(uw') \leftarrow T_{u'}(w) = x_v$ by the induction hypothesis. If uw' is not a path from u to v, then either u' is not a neighbor of u, or w' is not a path from u' to v and $T_{u'}(w') = \perp$ by the induction hypothesis. In either case, $T_u(uw') \leftarrow \perp$. □

What does this buy us? Suppose we have a deterministic synchronous algorithm that runs in $f(n)$ rounds. Starting from a stable configuration, Lemma 32.5.1 tells us that any fair daemon will eventually leave us in a configuration where each node u stores in T_u both the inputs of all nodes within distance $f(n)$ and enough information to reconstruct how they are connected. So u can simulate the execution of any node at distance d for up to $f(n) - d$ rounds. In particular, it can simulate its own execution for $f(n)$ rounds, computing the same output as it would produce in the LOCAL model.

Chapter 33

Distributed graph algorithms

In Chapter 32, we saw that certain classes of "local" algorithms have a straightforward conversion to self-stabilizing algorithms. In this chapter, we'll look more closely at what kinds of problems can be solved with this kind of locality.

33.1 The LOCAL and CONGEST models

The LOCAL and CONGEST models were defined by Peleg [Pel00] to formalize the idea of local distributed computation. Similar models had been considered previously without being specifically named [Lin92], but these names are now standard.

The LOCAL model is a synchronous message-passing model where the processes are organized into a graph, all run the same code, and can communicate only with their neighbors in the graph. To break symmetry, each process starts with a unique ID that is polynomial in the number of processes n. The processes may also start with local inputs, but often we are interested simply in computing some property of the graph itself. There is no bound on the size of messages.

The CONGEST model is like the local model, but messages are limited to $O(\log n)$ bits. More generally, the CONGEST(b) model allows messages of size b, making LOCAL = CONGEST(∞) and CONGEST = CONGEST($O(\log n)$).

In both models, we usually assume that the processes do *not* know the structure of the graph or their place in it. But for specific problems, we might require the graph to be from some restricted class (e.g., rings, trees, cliques).

We'll mostly focus on the LOCAL model in this chapter, studying it through the problem of graph coloring.

33.2 Local graph coloring

One of the first problems studied in the LOCAL model is local graph coloring [Lin92], where we wish to assign each node in the graph a small label distinct from its neighbors. Because the nodes initially start with large distinct labels, graph coloring in the LOCAL model shares some similarities with renaming (Chapter 25), since we will use the unique IDs as a starting point for generating the colors.

33.2.1 Coloring graphs with out-degree 1

Let us start by describing a classic local algorithm for 3-coloring a directed graph with maximum out-degree 1, a class of graphs that includes both cycles and rooted trees. The algorithm we will use is ultimately due to Cole and Vishkin [CV86], although the application to local graph coloring was given by Linial [Lin92], and the version given here incorporates some additional features from Peleg's textbook [Pel00].

The core idea from the Cole and Vishkin algorithm is to treat each identity x as a long bit-string $x_k x_{k-1} \dots x_0$, where $k = \lfloor \lg N \rfloor$ and $x = \sum 2^i x_i$ and repeatedly apply an operation that to shorten these IDs while maintaining the property that neighbors have distinct IDs.

At each synchronous round, each process adopts a new identity based on its old identity x and the identity y of its successor. We look for the smallest index i for which $x_i \neq y_i$. We then generate a new identity $2i + x_i$; this is the same as taking the bit-vector representation of i and shifting it one position to the left so we can append x_i to the end of it.

In the case of a node with no successor, we pretend that it has a successor with $y_0 \neq x_0$. This will knock x down to just its last bit x_0.

We now argue that this never produces two adjacent identities that are the same. Consider three consecutive identities x, y, and z. Let i be the smallest index with $x_i \neq y_i$, and let j be the smallest index with $x_j \neq y_j$. If $j \neq i$, then my successor's new identity $2j + y_j$ will not equal my new identity $2i + x_i$, because the initial bits will be different. But if $j = i$, then my successor's new identity is $2i + y_i \neq 2i + x_i$ because $y_i \neq x_i$.

Assuming that the largest initial color is N, the largest possible value for i is $\lfloor \lg N \rfloor$, and so the largest possible value for $2i + x_i$ is $2\lfloor \lg N \rfloor + 1$. Iterating the function $2\lfloor \lg N \rfloor + 1$ converges to at most 5 after $O(\log^* N)$

rounds, which gives us six colors $0, \ldots, 5$, where no two adjacent processes have the same color.

To reduce this to three colors, add a phase for each $c \in \{3, 4, 5\}$ to eliminate c. In each phase, we carry out a two-stage process. The first stage cleans up the neighborhood around each node, and the second stage replaces all copies of c with some color in $\{0, 1, 2\}$.

In the first stage, we shift all colors down, by having each node switch its color to that of its successor (or some new color chosen from $\{0, 1, 2\}$ if it doesn't have a successor). The reason for doing this is that it guarantees that each node's predecessors will all share the same color, meaning that that node now has at most two colors represented among its predecessors and successor. At the same time, it doesn't create any new pair of adjacent nodes with the same color.

For the second stage, each node v that currently has color c chooses a new color from $\{0, 1, 2\}$ that is the smallest color that doesn't appear in its neighborhood. Since none of v's neighbors change color during this stage (they don't have color c), this replaces all instances of c with a color from $\{0, 1, 2\}$ while keeping all edges two-colored. After doing this for all $c \in \{3, 4, 5\}$, the only colors left are in $\{0, 1, 2\}$.

Doing the 6 to 3 reduction in the obvious way takes an additional 6 rounds, which is (asymptotically) dominated by the $O(\log^* N)$ rounds of reducing from initial IDs with values up to N.

Because the reduction to 6 colors technically requires more than constant time, it's theoretically necessary for the nodes to have an upper bound on $O(\log^* N)$ to know when to switch to the $6 \to 3$ step. In practice, $\log^* N \leq 7$ for any N that can be represented by bits encoded using subatomic particles contained in the visible universe, so we may be able to get away with fixing a constant. Despite this useful property of $\log^*$ in practice, we can't get rid of it in theory, because of an $\Omega(\log^* n)$ bound on coloring rings shown in the next section.

33.2.2 Lower bound for rings

Using a Ramsey-theoretic argument, Linial [Lin92] showed that $\Omega(\log^* n)$ is a lower bound on the time to color a directed ring with n nodes in the LOCAL model, which implies that the algorithm of the previous section is optimal up to constants, since a directed ring is a special case of a graph with out-degree 1. We'll describe here a simplified version of Linial's original proof given by Laurinharju and Suomela [LS14]. (The Laurinharju and Suomela paper is only two pages long, so it may be worth skipping the rest of this

section and just reading it in the original.)

The idea is that any coloring algorithm in the local model that runs in time T assigns a color to each node based only on the initial IDs of the $2T+1$ nodes that are within T hops. So we can represent any possible deterministic coloring algorithm by specifying the mapping from these $2T+1$ IDs to colors.

Define a **k-ary c-coloring function** as a function $A : [n]^k \to [c]$ where $[n] = \{0, \ldots, n-1\}$ is the ID space and $[c] = \{0, \ldots, c-1\}$ is a set of c colors, with the property that

$$A(x_1, x_2, \ldots, x_k) \neq A(x_2, \ldots, x_k, x_{k+1}) \tag{33.2.1}$$

for any $0 \leq x_1 < x_2 < \cdots < x_{k+1} \leq n-1$.

The restriction to increasing sequences and values in $[n]$ rather than $[N]$ is more restrictive that a general c-coloring algorithm, but if we have a successful 3-coloring algorithm that runs in time T, we can extract from it a $(2T+1)$-ary 3-coloring function, and condition (33.2.1) will hold given that the original algorithm never assigns the same color to adjacent nodes. Taking the contrapositive, if condition (33.2.1) fails for some sequence $(x_1, x_2, \ldots x_{k+1})$, then we can supply this sequence as the IDs for the first $k+1$ nodes in the ring and show the algorithm fails. This implies that a 3-coloring algorithm that runs in time T can exist only if there is a $(2T+1)$-ary 3-coloring function. The lower bound proof works by showing that T needs to be $\Omega(\log^* n)$ for this to be possible.

It holds trivially that any 1-ary c-coloring function requires $c \geq n$. The proof works by showing how to transform any k-ary c-coloring function into a $(k-1)$-ary 2^c-coloring function, which hits the trivial bound after $k-1$ steps.

Lemma 33.2.1 ([LS14, Lemma 2])**.** *For $k > 1$, given a k-ary c-coloring function A, it is possible to construct a $(k-1)$-ary 2^c-coloring function B.*

Proof. Let $B'(x_1, \ldots, x_{k-1}) = \{A(x_1, x_2, \ldots, x_{k-1}, x_k) \mid x_k > x_{k-1}\}$. In other words, we fill in the missing parameter x_k with all possible values $x_k > x_{k-1}$, and return the set of colors that we obtain from A. Since there are exactly 2^c possible sets, we can obtain $B : [N]^{k-1} \to [2^c]$ by encoding each set as a distinct number in $[2^c] = \{1, \ldots, 2^c\}$.

We will now prove that B satisfies (33.2.1) whenever A does, by showing the contraposition that if B does not satisfy (33.2.1), then A doesn't either.

Suppose now that (33.2.1) does not hold for B, that is, there is some increasing sequence $(x_1, \ldots, x_k)$ such that $B(x_1, \ldots, x_{k-1}) = B(x_2, \ldots, x_k)$, or equivalently $B'(x_1, \ldots, x_{k-1}) = B'(x_2, \ldots, x_k)$.

We will feed this bad sequence to A and see what happens. Let $\alpha = A(x_1, \ldots, x_k)$. Since x_k is one of the possible extensions of $(x_1, \ldots, x_{k-1})$ used to generate $B'(x_1, \ldots, x_{k-1})$, we get $\alpha \in B'(x_1, \ldots, x_{k-1})$. But then α is also contained in $B'(x_2 \ldots, x_k) = B'(x_1, \ldots, x_{k-1})$. From the definition of $B'(x_2, \ldots, x_k)$, this implies that there is some $x_{k+1} > x_k$ such that $\alpha = A(x_2, \ldots, x_k, x_{k+1}) = A(x_1, x_2, \ldots, x_k)$. But then A is not a k-ary c-coloring function. □

To get the $\Omega(\log^* n)$ lower bound, start with a k-ary 3-coloring function and iterate Lemma 33.2.1 to get a 1-ary $f(k-1)$-coloring function where $f(k)$ is the result of iteratively applying the function 2^x to 3, $k-1$ times. Then $f(k-1) \geq n$, which implies $k = \Omega(\log^* n)$.

33.2.3 Coloring bounded-degree graphs

The $O(\log^* n)$-time 3-coloring algorithm for out-degree 1 digraphs can be used to get a simple $O(\Delta^2 + \log^* n)$ time algorithm for $(\Delta+1)$-coloring any graph with maximum degree Δ, using an algorithm of Panconesi and Rizzi [PR01].

This algorithm has three steps:

1. First, partition the original graph G into Δ directed graphs $G_1, \ldots, G_\Delta$, each with maximum out-degree 1. We can do this in $O(1)$ rounds: each process collects the IDs of its neighbors, and assigns each a **port number** in $\{1, \ldots, \Delta\}$ in increasing order of ID, while also orienting each edge to point to the neighbor with larger ID. Each directed graph G_i then consists of all edges for which the source node assigns port number 1.

2. Next, use Cole-Vishkin (§33.2.1) to 3-color each G_i.

3. To color the original graph G, start with $H_1 = G_1$ and repeatedly merge each H_i with the next unmerged G_{i+1} to get H_{i+1}. Each H_i will have at most $\Delta + 1$ colors, which we will show by induction on i.[1]

   The merging process consists of assigning each node a color in $[3(\Delta+1)]$ by taking an ordered pair of its color in G_{i+1} (3 choices) and its color in H_i. Then for each $c \in \{\Delta+2, \ldots, 3(\Delta+1)\}$, have each node with color c choose the smallest color not represented among its neighbors. This is the same color-reduction scheme used to go from six to three

[1] We are assuming here that $\Delta \geq 2$ to get the induction going, but the cases $\Delta = 0$ and $\Delta = 1$ are easy to handle using a simpler algorithm.

colors in §33.2.1, except without the shifting, and just like there we don't create any new conflicts because no two nodes with the same color c are adjacent to begin with.

Each merging step costs $O(\Delta)$ rounds (mostly for polling the neighbors to see what colors they currently have). There are $O(\Delta)$ total merges, so it takes $O(\Delta^2)$ rounds to complete them all and get $\Delta + 1$ colors.

Since we are using Cole-Vishkin as a subroutine, we do need an upper bound on $\log^* n$, but this shouldn't be too hard to obtain in practice.

The Panconesi and Rizzi algorithm has the advantage of simplicity, but there are faster algorithms. An algorithm of Ghaffari and Kuhn [GK22] obtains a $(\Delta+1)$-coloring of a graph with maximum degree Δ in $O(\log^2 \Delta \log n)$ rounds.

Chapter 34

Population protocols

Here are four mostly-equivalent models:

Population protocols A **population protocol** [AAD+06] consists of a collection of agents with states in some state space Q. At each step, the adversary picks two of the agents to interact, and both get to update their state according to a joint transition function $\delta : Q \times Q \to Q \times Q$. A **global fairness** condition requires that if some global configuration C of the system occurs infinitely often, and there is a step that transforms C to C', then this step eventually occurs.

Computation by population protocols usually consists of computing some function of the initial states of the population and propagating the output of this function to all agents. As in a self-stabilizing system, termination is not detected; instead, we hope to converge to the correct answer eventually.

In some versions of the model, interactions between agents are limited by an interaction graph (only adjacent agents can interact), or are assumed to be chosen randomly instead of adversarially. These assumptions may in some cases increase the power of the model.

Chemical reaction networks In a (**CRN** for short), we have a collection of molecules representing different **species**. These molecules may undergo chemical reactions that transform one or more inputs into one or more outputs, as in this bit of classic rocketry:

$$H_2 + O_2 \to H_2O + O$$

Computation by a chemical reaction network consists of putting some

appropriate mix of molecules into a test tube, stirring it up, and hoping to learn something from the final product.

Unlike population protocols, chemical reaction networks do not necessarily conserve the number of molecules in each interaction, and (in reality at least) require some source of energy to keep the reactions going.

Petri nets A **Petri net** [Pet62] is a collection of of **places** and **transitions**, in the form of a bipartite graph, with **tokens** wandering around through the places. A transition **fires** by consuming one token from each place in its in-neighborhood and adding one token to each place in its out-neighborhood, assuming there is at least one token on each place in its in-neighborhood. Various conditions are assumed on which transitions fire in which order.

Petri nets were invented to model chemical reaction networks, so it's not surprising that they do so. Pretty much any result in population protocols or CRNs can be translated to Petri nets or vice versa, by the mapping:

	agent	molecule	token
mapping:	state	species	place
	transition	reaction	transition

Following a long-standing and probably unjustified American prejudice, we will not talk much about Petri nets, but there has been a fair bit of cross-pollenization between the population protocol, CRN, and Petri net literature.

Vector addition systems You have a non-negative integer vector x. There is a set of rules $-a+b$, where a and b are both non-negative integer vectors, and you are allowed to replace x by $x-a+b$ if $x-a \geq 0$. These are basically Petri nets without jargon.

Of these models, population protocols are currently the most popular in the theory of distributed computing community, with chemical reaction networks moving up fast. So we'll talk about population protocols.

34.1 Definition of a population protocol

Let us begin by giving a formal definition of a population protocol, following the original definition of Angluin *et al.* [AAD+06].

A **population protocol** is a tuple $\langle X, Y, Q, I, O, \delta \rangle$, where X is the input alphabet, Y is the output alphabet, Q is the state space, $I : X \to Q$

maps inputs to initial states, $O : Q \to Y$ maps states to outputs, and $\delta : Q \times Q \to Q \times Q$ is the transition function.

A **population** consists of n agents, taken to be the vertices of a directed graph called the interaction graph. Most of the time we will assume the interaction graph is a complete graph, but the model allows for more restrictive assumptions. A **configuration** is a mapping C from agents to Q. A **transition** involves choosing two agents x and y such that xy is an edge in the interaction graph, and updating the configuration C to a new configuration C' with $\left\langle C'_x, C'_y \right\rangle = \delta(\langle C_x, C_y \rangle)$ and $C'_z = C_z$ for all $z \notin \{x, y\}$.

The first agent in an interaction is called the **initiator** and the second agent the **responder**. Note that this means that the model breaks symmetry for us.

With a complete interaction graph, we can will often not bother with the identities of specific agents and just treat the configuration C as a multiset of states.

The main difference between population protocols and similar models is the input and output mappings, and the notion of stable computation, which gets its own section.

34.2 Stably computable predicates

A predicate P on a vector of initial inputs is **stably computable** if there exists a population protocol such that it eventually holds forever that every agent correctly outputs whether P is true or false. Stably computable functions are defined similarly.

One of the big early results on population protocols was an exact characterization of stably computable predicates for the complete interaction graph. We will give a sketch of this result below, after giving some examples of protocols that compute particular predicates.

34.2.1 Time complexity

The original population protocol did not define a notion of time, since the fairness condition allows arbitrarily many junk transitions before the system makes progress. More recent work has tended to compute time complexity by assuming random scheduling, where the pair of agents to interact is determined by choosing an edge uniformly from the interaction graph (which means uniformly from all possible pairs when the interaction graph is complete).

Assuming random scheduling (and allowing for a small probability of error) greatly increases the power of population protocols. So when using this time measure we have to be careful to mention whether we are also assuming random scheduling to improve our capabilities. Most of the protocols in this section are designed to work as long as the scheduling satisfies global fairness—they don't exploit random scheduling—but we will discuss running time in the random-scheduling case as well.

34.2.2 Examples

These examples are mostly taken from the original paper of Angluin *et al.* [AAD+06].

34.2.2.1 Leader election

Most stably computable predicates can be computed as a side-effect of **leader election**, so we'll start with a leader election protocol. The state space consists of L (leader) and F (follower); the input map makes every process a leader initially. Omitting transitions that have no effect, the transition relation is given by

$$L, L \to L, F.$$

It is easy to see that in any configuration with more than one leader, there exists a transition that eventually reduces the number of leaders. So global fairness says this happens eventually, which causes us to converge to a single leader after some finite number of interactions.

If we assume random scheduling, the expected number of transitions to get down to one leader is exactly

$$\begin{aligned}
\sum_{k=2}^{n} \frac{n(n-1)}{k(k-1)} &= n(n-1) \sum_{k=2}^{n} \frac{1}{k(k-1)} \\
&= n(n-1) \sum_{k=2}^{n} \left(\frac{1}{k-1} - \frac{1}{k} \right) \\
&= n(n-1) \left(1 - \frac{1}{n} \right) \\
&= n^2.
\end{aligned}$$

34.2.2.2 Distributing the output

The usual convention in a population protocol is that we want every process to report the output. It turns out that this is equivalent to the leader reporting the output.

Given a protocol A with states of the form $\langle \ell, x \rangle$ where $\ell \in \{L, F\}$ is the leader bit and x is whatever the protocol is computing, define a new protocol A' with states $\langle \ell, x, y \rangle$ where $y = O(x)$ when $\ell = L$ and y is the output of the last leader the agent met when $\ell = F$.

Now as soon as the leader has converged on an output, it only needs to meet each other agent once to spread it to them. This takes an additional $nH_{n-1}/2 = O(n^2 \log n)$ interactions on average.

34.2.2.3 Remainder mod m

We can now give an example of a protocol that stably computes a function: we will count the number of agents in some special initial state A, modulo a constant m. (We can't count the exact total because the agents are finite-state.)

Each agent has a state $\langle \ell, x \rangle$, where $\ell \in \{L, F\}$ as in the leader election protocol, and $x \in \mathbb{Z}_m$. The input mapping sends A to $\langle L, 1 \rangle$ and everything else to $\langle L, 0 \rangle$. The non-trivial transitions are given by

$$\langle L, x \rangle, \langle L, y \rangle \to \langle L, (x + y) \bmod m \rangle, \langle F, 0 \rangle$$

This protocol satisfies the invariant that the sum over all agents of the second component, mod m, is unchanged by any transition. Since the components for any is follower is zero, this means that when we converge to a unique leader, it will contain the count of initial A's mod m.

34.2.2.4 Linear threshold functions

Remainder mod m was one of two tools in [AAD+06] that form the foundation for computing all stably computable predicates. The second computes linear threshold predicates, of the form

$$\sum a_i x_i \geq b, \tag{34.2.1}$$

where the x_i are the counts of various possible inputs and the a_i and b are integer constants. This includes comparisons like $x_1 > x_2$ as a special case.

The idea is to compute a truncated version of the left-hand side of (34.2.1) as a side-effect of leader election.

Fix some $k > \max(|b|, \max_i |a_i|)$. In addition to the leader bit, each agent stores an integer in the range $-k$ through k. The input map sends each x_i to the corresponding coefficient a_i, and the transition rules cancel out positive and negative a_i, and push any remaining weight to the leader as much as possible subject to the limitation that values lie within $[-k, k]$.

Formally, define a truncation function $t(x) = \max(-k, \min(k, r))$, and a remainder function $r(x) = x - t(x)$. These have the property that if $|x| \leq 2k$, then $t(x)$ and $r(x)$ both have their absolute value bounded by k. If we have the stronger condition $|x| \leq k$, then $t(x) = x$ and $r(x) = 0$.

We can now define the transition rules:

$$\langle L, x\rangle, \langle -, y\rangle \to \langle L, t(x+y)\rangle, \langle F, r(x+y)\rangle$$
$$\langle F, x\rangle, \langle F, y\rangle \to \langle F, t(x+y)\rangle, \langle F, r(x+y)\rangle$$

These have the property that the sum of the second components is preserved by all transitions. Formally, if we write y_i for the second component of agent i, then $\sum y_i$ does not change through the execution of the protocol.

When agents with positive and negative values meet, we get cancellation. This reduces the quantity $\sum |y_i|$. Global fairness implies that this quantity will continue to drop until eventually all nonzero y_i have the same sign.

Once this occurs, and there is a unique leader, then the leader will eventually absorb as much of the total as it can. This will leave the leader with $y = \min(k, \max(-k, \sum y_i))$. By comparing this quantity with b, the leader can compute the threshold predicate.

34.2.3 Presburger arithmetic and semilinear sets

Presburger arithmetic [Pre29] is the first-order theory (in the logic sense) of the natural numbers with addition, equality, 0, and 1. This allows expressing ideas like "x is even:"

$$\exists y : x = y + y$$

or "x is greater than y":

$$\exists z : x = y + z + 1$$

but not "x is prime" or even $x = y \cdot z$."

Presburger arithmetic has various amazing properties, including **decidability**–there is an algorithm that will tell you if any statement in Presburger arithmetic is true or not (in doubly-exponential time [FR98])—and **quantifier elimination**—a formula using any combination of $\forall$ and $\exists$ quantifiers

can be converted to a formula with no quantifiers, using the predicates $<$ and $\equiv_k$ for constant values of k, where $x \equiv_k y$ if x and y have the same remainder mod k.

There is also a one-to-one correspondence between predicates in Presburger arithmetic and **semilinear sets**, which are finite unions of **linear sets** of the form $\{b + \sum a_i x_i\}$ where b is a non-negative integer vector, the a_i are non-negative integer coefficients, and the x_i are non-negative integer vectors, and there are only finitely many terms.

(We will not attempt to prove any of this.)

It turns out that Presburger arithmetic (alternatively, semilinear sets) captures exactly what can and can't be stably computed by a population protocol. For example, no semilinear set contains all and only primes (because any infinite semilinear set on one variable is an arithmetic progression), and primes aren't recognizable by a population protocol. An intuitive and not entirely incorrect explanation is that in both cases we can't do multiplication because we can't do nested loops. In population protocols this is because even though we can do a single addition that turns exactly A many blank tokens into B's, using the rule

$$A, - \to A', B$$

we can't multiply by repeated addition, because we can't detect that the first addition step addition has ended to start the next iteration of the outer loop.

Below we'll describe the correspondence between semilinear sets and stably-computable predicates. For full details see [AAD+06, AAE06].

34.2.3.1 Semilinear predicates are stably computable

This part is easy. We have that any Presburger formula can be represented as a logical combination of $<$, $+$, and $\equiv_k$ operators. We can implement any formula of the form $\sum a_i x_i < b$, where a_i and b are integer constants, using the linear threshold function from §34.2.2.4. We can implement any formula of the form $\sum a_i x_i \equiv_k b$ using a straightforward extension of the mod-k counter from §34.2.2.3. If we run these in parallel for each predicate in our formula, we can then apply any logical connectives to the result.

For example, if we want to express the statement that "x is an odd number greater than 5", we have out agents compute separately $x \equiv_2 1$ and $x > 5$; if the leader computes true for both of these, it assigns true to its real output.

34.2.3.2 Stably computable predicates are semilinear

This is the hard direction, because we have to exclude any possible algorithm for computing a non-semilinear set. The full proof is pretty involved, and can be found in [AAE06]. A survey paper of Aspnes and Ruppert [AR09] gives a simplified proof of the weaker result (modeled on an introductory argument in [AAE06]) that any stably-computable set is a finite union of **monoids**. Like linear sets, monoids are of the form $\{b + \sum a_i x_i\}$, but the number of terms in the sum might be infinite.

We won't do either of these proofs.

34.3 Random interactions

An alternative to assuming worst-case scheduling is to assume random scheduling: at each step, a pair of distinct agents is chosen uniformly at random to interact. This gives the population protocol substantially more power, and (with some tweaks to allow for different reactions to occur at different rates) is the standard assumption in chemical reaction networks.

An example of an algorithm that exploits the assumption of random scheduling is the **approximate majority** protocol of Angluin, Aspnes, and Eisenstat [AAE08b], which was also independently discovered by Perron, Vasudevan, and Vojnovic [PVV09]. This protocol starts with a mix of agents in states x and y, and uses a third state b (for blank) to allow the initial majority value to quickly take over the entire population. The non-trivial transition rules are:

$$xy \to xb$$
$$yx \to yb$$
$$xb \to xx$$
$$bx \to xx$$
$$yb \to yy$$
$$by \to yy$$

If two opposite agents meet, one becomes blank, depending on which initiates the reaction (this is equivalent to flipping a coin under the random-scheduling assumption). These reactions produce a supply of blank agents, drawing equally from both sides. But when a blank agent meets a non-blank agent, it adopts the non-blank agent's state. This is more likely to be the majority state, since there are more agents to meet in the majority state.

So if we consider only transitions that change the net number of x agents minus y agents, we get a random walk biased toward the majority value with an absorbing barrier in the state where all agents are equal. However, the rate at which these special transitions occur depends on how quickly blank agents are created, which in turn depends on the relative numbers of x and y agents.

Analysis of the full process is difficult, but Angluin *et al.* show that with high probability all agents end up in the initial majority state in $O(n \log n)$ interactions, provided the initial majority is large enough ($\Omega(\sqrt{n} \log n)$, later improved to $\Omega(\sqrt{n \log n}$ by Condon *et al.* [CHKM19]). Curiously, a later paper by Cardelli and Csikász-Nagy [CCN12] showed that the cell cycle controlling mitosis in all living organisms uses a chemical switch that looks suspiciously like the approximate majority algorithm, making this algorithm roughly three billion years old.

But we can do better than this. With random scheduling, we have much more control over how a computation evolves, and this can be used to simulate (with high probability) a register machine, which in turn can be used to simulate a Turing machine. The catch is that the state of a population protocol with n agents can be described using $O(\log n)$ bits, by counting the number of agents in each state. So the most we can simulate is a machine that has $O(\log n)$ space.

The original population protocol paper included a simulation of an $O(\log n)$-space Turing machine, but the time overhead per operation was very bad, since most operations involved a controller agent personally adjusting the state of some other agent, which requires $\Theta(n)$ time on average before the controller meets its target.

A better construction was given by Angluin *et al.* [AAE08a], under the assumption that the population starts with a single agent in a special leader state. The main technique used in this paper it to propagate a message m using an epidemic protocol $mb \to mm$. The time for an epidemic to spread through a population of n individuals through random pairwise interactions is well-understood, and has the property that (a) the time to infect everybody is $\Theta(\log n)$ with high probability, and (b) it's still $\Theta(\log n)$ with high probability if we just want to infect a polynomial fraction n^ϵ of the agents.

So now the idea is that if the leader, for example, wants to test if there is a particular state x in the population, it can spread a message $x?$ using an epidemic, and any agent with x can respond by starting a counter-epidemic $x!$. So if there is an x, the leader finds out about it in $O(\log n)$ time, the time for the first epidemic to go out plus the time for the second epidemic to come back.

What if there is no x agent? Then the query goes out but nothing comes back. If the leader can count off $\Theta(\log n)$ time units (with an appropriate constant, it can detect this. But it does not have enough states by itself to count to $\Theta(\log n)$.

The solution is to take advantage of the known spreading time for epidemics to build a **phase clock** out of epidemics. The idea here is that the leader will always be in some **phase** $0 \dots m-1$. Non-leader agents try to catch up with the leader by picking up on the latest rumor of the leader's phase, which is implemented formally by transitions of the form $\langle x, i\rangle \langle F, j\rangle \to \langle x, i\rangle \langle F, i\rangle$ when $0 < i - j < m/2 \pmod m$. The leader on the other hand is a hipster and doesn't like it when everybody catches up; if it sees a follower in the same phase, it advances to the next phase to maintain its uniqueness: $\langle L, i\rangle \langle F, i\rangle \to \langle L, i+1\rangle \langle F, i\rangle$.

Because the current phase spreads like an epidemic, when the leader advances to $i+1$, every agent catches up in $a \log n$ time w.h.p. This means both that the leader doesn't spend too much time in $i+1$ before meeting a same-phase follower and that followers don't get too far behind. (In particular, followers don't get so far behind that they start pulling other followers forward.) But we also have that it takes at least $b \log n$ time w.h.p. before more than n^ϵ followers catch up. This gives at most an $n^{\epsilon-1} \ll 1$ probability that the leader advances twice in $b \log n$ time. By making m large enough, the chances that this happens enough to get all the way around the clock in less than, say $b(m/2) \log n)$ time can be made at most n^{-c} for any fixed c. So the leader can now count of $\Theta(\log n)$ time w.h.p., and in particular can use this to time any other epidemics that are propagating around in parallel with the phase clock.

Angluin *et al.* use these techniques to implement various basic arithmetic operations such as addition, multiplication, division, etc., on the counts of agents in various states, which gives the register machine simulation. The simulation can fail with nonzero probability, which is necessary because otherwise it would allow implementing non-semilinear operations in the adversarial scheduling model.

The assumption of an initial leader can be replaced by a leader election algorithm, but at the time of the Angluin *et al.* paper, no leader election algorithm better than the $\Theta(n)$-time fratricide protocol described §34.2.2.1 was known, and even using this protocol requires an additional polynomial-time cleaning step before we can run the main algorithm, to be sure that there are no leftover phase clock remnants from deposed leaders to cause trouble. So the question of whether this could be done faster remained open.

Hopes of finding a better leader election protocol without changing the

model ended when Doty and Soloveichek [DS15] proved a matching $\Omega(n)$ lower bound on the expected time to convergence for any leader election algorithm in the more general model of chemical reaction networks. This results holds assuming constant states and a **dense** initial population where any state that appears is represented by a constant fraction of the agents.

Because of this and related lower bounds, recent work on fast population protocols has tended to assume more states. This is a fast-moving area of research, so I will omit trying to summarize the current state of the art here. For an introduction to this work see [AG18, ER+18].

Chapter 35

Mobile robots

Last updated 2016. Some material may be out of date.

Mobile robots are a model of distributed computation where the agents (robots) are located in a plane, and have no ability to communicate except by observing each others' positions. Typical problems are to get a group of robots to gather on a single point, or more generally to arrange themselves in some specified pattern. This is complicated by the usual issues of asynchrony and failures.

35.1 Model

We will start by describing the **Suzuki-Yamashita model** [SY99], the **CORDA** model [Pri01], and some variants. We'll follow the naming conventions used by Agmon and Peleg [AP06].

Basic idea:

- We have a bunch of robots represented by points in the plane $\mathbb{R}^2$.
- Each robot executes a **look-compute-move** cycle:
  - Look phase: obtain snapshot of relative positions of all the other robots.
  - Compute phase: pick a new point to move to.
  - Move phase: move to that point.
- Robots are dumb. Various possible limitations that might apply:

  - **Anonymity**: any two robots that see the same view take the same action.
  - **Oblivious**: The output of the compute phase is base *only* on results of last look phase, and not on any previous observations. Robots have no memory!
  - No **absolute coordinates**: Translations of the space don't change the behavior of the robots.
  - No **sense of direction**: robots don't know which way is north. More formally, if view v can be rotated to get view v', then a robot that sees v' will make the same move as in v, subject to the same rotation.
  - No **sense of scale**: robots don't have a consistent linear measure. If view v can be scaled to get view v', then a robot that sees v' will move to the same point as in v, after applying the scaling.
  - No **sense of chirality**: robots can't tell counter-clockwise from clockwise. Flipping a view flips the chosen move but has no other effect.
  - No ability to detect **multiplicities**: the view of other robots is a set of points (rather than a multiset), so if two robots are on the same point, they look like one robot.
  - **Fat robots**: robots block the view of the robots behind them.

- Adversary can interfere in various ways:
  - During move phase, robot is guaranteed to either move some minimum distance $\delta > 0$ or reach its target, but adversary can stop a robot after it has moved δ.
  - Look-compute-move phases are asynchronous, and adversary can schedule robots subject to various constraints.
    * **Asynchronous** model (**ASYNC**): The adversary can delay a robot between look and move phases, so that robots might be moving based on out-of-date information.
    * **Semi-synchronous** model (**SSYNC**): Each look-compute-move cycle is an atomic action, so moves are always based on current information. The adversary may schedule more one or more robots to do their look-compute-move in each round. Also known as the **ATOM** model. This was the model given by Suzuki and Yamashita [SY99].

    - ∗ **Fully synchronous** model (**FSYNC**): Like SSYNC, but every robot is active in every round.
  - But we do have fairness: the adversary must activate each (non-faulty) robot infinitely often.

- We may also have faults:
  - **Byzantine faults**: Byzantine robots can move anywhere they like.
  - **Crash faults**: crashed robots don't move even when they are supposed to.

The simplest goal is to gather the non-faulty robots together on a single point despite all these possible disasters. Other goals might be formation of particular shapes. An additional source of variation here is whether we want exact gathering (every robot eventually gets to exactly where it should be) or just convergence (over time, robots get closer and closer to where they should be).

Below, we will mostly be looking at the semi-synchronous model, with the assumption that robots are anonymous and oblivious, and have no absolute coordinates, sense of direction, or sense of scale. However, we will generally let robots detect multiplicity. Depending on the results we are describing, we may or may not assume chirality.

35.2 Two robots, no faults

Suzuki and Yamashita [SY99] showed that it's impossible to get two deterministic, oblivious robots to the same point in the semi-synchronous model assuming no absolute coordinates and no sense of direction, although they can converge. The convergence algorithm is simple: have each robot move to the midpoint of the two robots whenever it is activated. This always reduces the distance between the robots by $\min(\delta, d/2)$. But it doesn't solve gathering if only one robot moves at a time.

This turns out to be true in general [SY99, Theorem 3.1]. The idea is this: Suppose we have an oblivious algorithm for gathering. Consider two robots at distinct points p and q, and suppose that after one round they both move to r. There are two cases:

1. Both robots move. By symmetry, $r = (p + q)/2$. So now construct a different execution in which only one robot moves (say, the one that moved least recently, to avoid running into fairness).

2. Only one robot moves. Without loss of generality, suppose the robot at p moves to q. Then there is a different execution where q also moves to p and the robots switch places.

In either case the distance between the two robots in the modified execution is at least half the original distance. In particular, it's not zero. Note that this works even if the adversary can't stop a robot in mid-move.

Both obliviousness and the lack of coordinates and sense of direction are necessary. If the robots are not oblivious, then they can try moving to the midpoint, and if only one of them moves then it stays put until the other one catches up. If the robots have absolute coordinates or a sense of direction, then we can deterministically choose one of the two initial positions as the ultimate gathering point (say, the northmost position, or the westmost position if both are equally far north). But if we don't have any of this we are in trouble.

Like the 3-process impossibility result for Byzantine agreement, the 2-process impossibility result for robot gathering extends to any even number of robots where half of them are on one point and half on the other. Anonymity then means that each group of robots acts the same way a single robot would if we activate them all together. Later work (e.g., [BDT12]) refers to this as **bivalent** configuration, and it turns out to be the only initial configuration for which it is not possible to solve gathering absent Byzantine faults.

35.3 Three robots

Agmon and Peleg [AP06] show that with three robots, it is possible to solve gathering in the SSYNC model with one crash fault but not with one Byzantine fault. We'll start with the crash-fault algorithm. Given a view $v = \{p_1, p_2, p_3\}$, this sends each robot to the "goal" point p_G determined according to the following rules:

1. If v has a point p with more than one robot, set $p_G = p$.
2. If p_1, p_2, and p_3 are collinear, set p_G to the middle point.
3. If p_1, p_2, and p_3 form an obtuse triangle (one with a corner whose angle is $\geq \pi/2$, set p_G to the obtuse corner.
4. If p_1, p_2, and p_3 form an acute triangle (one with no angles $\geq \pi/2$), set p_G to the intersection of the angle bisectors.

Here is a sketch of why this works. For the real proof see [AP06].

1. If we are in a configuration with multiplicity > 1, any non-faulty robot not on the multiplicity point eventually gets there.

2. If we are in a collinear configuration, we stay collinear until eventually one of the outer robots gets to the middle point, giving a configuration with multiplicity > 1.

3. If we are in an obtuse-triangle configuration, we stay in an obtuse-triangle configuration until eventually one of the acute-corner robots gets to the obtuse corner, again giving a configuration with multiplicity > 1.

4. If we are in an acute-triangle configuration, then a somewhat messy geometric argument shows that if at least one robot moves at least δ toward the intersection of the angle bisectors, then the circumference of the triangle drops by $c\delta$ for some constant $c > 0$. This eventually leads either to the obtuse-triangle case (if we happen to open up one of the angles enough) or the multiplicity > 1 case (if the circumference drops to zero).

However, once we have a Byzantine fault, this blows up. This is shown by considering a lot of cases, and giving a strategy for the adversary and the Byzantine robot to cooperate to prevent the other two robots from gathering in each case. This applies to both algorithms for gathering and convergence: the bad guys can arrange so that the algorithm eventually makes no progress at all.

The first trick is to observe that any working algorithm for the $n = 3, f = 1$ case must be **hyperactive**: every robot attempts to move in every configuration with multiplicity 1. If not, the adversary can (a) activate the non-moving robot (which has no effect); (b) stall the moving non-faulty robot if any, and (c) move the Byzantine robot to a symmetric position relative to the first two so that the non-moving robot become the moving robot in the next round and vice versa. This gives an infinite execution with no progress.

The second trick is to observe that if we can ever reach a configuration where two robots move in a way that places them further away from each other (a **diverging** configuration), then we can keep those two robots at the same or greater distance forever. This depends on the adversary being able to stop a robot in the middle of its move, which in turn depends on the robot moving at least δ before the adversary stops it. But if the robots have no sense of scale, then we can scale the initial configuration so that this is not a problem.

Here is the actual argument: Suppose that from positions p_0 and q_0 there is a step in which the non-faulty robots move to p_1 and q_1 with $d(p_1, q_1) > d(p, q)$. Starting from p_1 and q_1, run both robots until they are heading for states p_2 and q_2 with $d(p_2, q_2) \leq d(p_0, q_0)$. By continuity, somewhere along the paths p_1p_2 and q_1q_2 there are intermediate points p_2' and q_2' with $d(p_2', q_2') = d(p_0, q_0)$. Stop the robots at these points, move the Byzantine robot r to the appropriate location to make everything look like the initial p_0, q_0 configuration, and we are back where we started.

So now we know that (a) we have to move every robot, and (b) we can't move any two robots away from each other. In the full version of the proof, this is used to show by an extensive case analysis that as long as we start with no two robots on the same point, this always either makes no progress or reaches three distinct points on the same line. We'll skip over this part and just argue that once we have three hyperactive collinear robots, that two of them are diverging. This will show that in the worst case we can't win, because the adversary could start everybody out on the same line, but it is not quite as general as the full result of Agmon and Peleg.

Suppose the robots are at positions $p_1 < p_2 < p_3$. Then p_2 has to move either left or right, which means moving away from either p_3 or p_1. In either case we have a diverging pair (because the adversary can elect not to move the robots on the end). So now the divergence argument goes through, and we are done.

35.4 Many robots, with crash failures

It turns out that we can solve the gathering problem even if we have many robots and some of them can crash, as long as the robots do not start on the same line. The reason for this is that any set of non-collinear points $x_1, \ldots, x_n$ in $\mathbb{R}^2$ has a unique **geometric median**, defined as the point m that minimizes $\sum_{i=1}^{n} d(m, x_i)$, and the geometric median is unchanged if we move any of the points towards it.

So the algorithm is for all the robots to walk toward this point. It doesn't matter if some of the robots don't move, or don't move at the same speed, because the median doesn't change. Eventually, all the non-faulty processes will reach it.

There is one drawback to this approach, which is that even though very good approximation algorithms exist [CLM+16], the geometric median appears to be difficult to compute exactly. We could declare that we are willing to assume that our robots have infinite computational power, but this

is not an easy assumption to justify in this case. An alternative is to build an algorithm that marches toward the geometric median in certain cases where it is straightforward to compute, and does something more sophisticated otherwise. This approach was taken by Bramas and Tixeuil [BT15], who also supplied the idea of using the geometric median in the first place. We will not go into detail about their algorithm.

Chapter 36

Beeping

Last updated 2016. Some material may be out of date.

The (discrete) **beeping model** was introduced by Cornejo and Kuhn [CK10] to study what can be computed in a wireless network where communication is limited to nothing but carrier sensing. According to the authors, the model is inspired in part by some earlier work on specific algorithms based on carrier sensing due to Scheideler *et al.* [SRS08] and Flury and Wattenhofer [FW10]. It has in turn spawned a significant literature, not only in its original domain of wireless networking, but also in analysis of biological systems, which often rely on very limited signaling mechanisms. Some of this work extends or adjusts the capabilities of the processes in various ways, but the essential idea of tightly limited communication remains.

In its simplest form, the model consists of synchronous processes organized in an undirected graph. Processes wake up at arbitrary rounds chosen by the adversary, and do not know which round they are in except by counting the number of rounds since they woke. Once awake, a process chooses in each round to either send (beep) or listen. A process that sends learns nothing in that round. A process that listens learns whether any of its neighbors sends, but not how many or which one(s).

From a practical perspective, the justification for the model is that carrier sensing is cheap and widely available in radio networks. From a theoretical perspective, the idea is to make the communication mechanism as restrictive as possible while still allowing some sort of distributed computing. The assumption of synchrony both adds to and limits the power of the model. With no synchrony at all, it's difficult to see how to communicate anything with beeps, since each process will just see either a finite or infinite sequence of beeps with not much correlation to its own actions. With continuous

time, subtle changes in timing can be used to transmit arbitrarily detailed information. So the assumption of a small number of synchronous rounds is a compromise between these two extremes. The assumption that processes wake at different times and have no common sense of time prevents synchronization on rounds, for example by reserving certain rounds for beeps by processes with particular IDs. It is up to the protocol to work around these limitations.

36.1 Interval coloring

One way to get around the problem of not having a common global clock is to solve **interval coloring**, the main problem considered by Cornejo and Kuhn. This is related to TDMA multiplexing in cell phone networks, and involves partitioning a repeating interval of T rounds in a network with maximum degree Δ into subintervals of length $\Omega(T/\Delta)$ such that each process is assigned a subinterval and no two adjacent processes are assigned overlapping subintervals. The idea is that these intervals can then be used to decide when each process is allowed to use its main radio to communicate.[1]

Cornejo and Kuhn give an algorithm for interval coloring that assigned a subinterval of length $\Omega(T/\Delta)$ to each process assuming that the size of the interval T is known to all processes and that T is at least a constant multiple of Δ. However, the processes do not know anything about the structure of the graph, and in particular do not know Δ. This requires each process to get an estimate of the size of its neighborhood (so that it knows how large a subinterval to try to acquire) and to have a mechanism for collision detection that keeps it from grabbing an interval that overlaps with a neighbor's interval. The process is complicated by the fact that my length-T intervals and your length-T intervals may be offset from each other, and that I can't detect that you are beeping if you and I are beeping at the same time.

To simplify things a bit, the presentation below will assume that the graph is regular, so that $d(v)$ equals the maximum degree Δ for all nodes in the graph. The paper [CK10] gives an analysis that does not need this assumption. We'll also wave our hands around a lot instead of doing actual algebra in many places.

[1] We may really want **2-hop coloring** here, where no two of my neighbors get the same color, because this is what (a) allows me to tell my neighbors apart, and (b) allows my neighbors not to interfere with each other, but that is a subject for later papers (see, for example, [MRZ15].

36.1.1 Estimating the degree

The basic idea is to have each process beep once in every T consecutive slots. Each process divides time into **periods** of length T, starting when it wakes up. Because processes wake up at different times, my period might overlap with up to two of yours. This means that if S is the set of times during my period where I hear beeps, then S includes at most two beeps per process, so $|S|$ is at most twice my actual degree. This gives an upper bound on $d(v)$, and indeed each process will just use the maximum number of beeps it heard in the last period as the basis for its estimate of $d(v)$.

For the lower bound side, we want to argue that if processes choose slots at random in a way that is more likely to avoid collisions than create them, and there are enough slots, then we expect to get few enough collisions that $|S|$ is also $\Omega(\Delta)$. The details of this depend on the mechanism for choosing slots, but if we imagine slots are chosen uniformly, then $\mathrm{E}\left[|S|\right] \geq \Delta(1-\Delta/T)$, which is $\Omega(\Delta)$ under our assumption that $T \geq c\Delta$ for some sufficiently large c. We can compensate by the error by inserting a fudge factor η, chosen so that $(1/\eta)|S|$ is very likely to be an upper bound on the degree.

36.1.2 Picking slots

Each process will try to grab a subinterval of size $b = \eta\frac{T}{|S|+1}$, where η is the fudge factor mentioned above. If it has not already picked a position p, then it chooses one uniformly at random from the set of all positions p such that $S[p-b-2, p+b+1]$ from the most recent period includes no beeps. Because this selection criterion knocks out up to $(2b+4)|S|$ possible choices, it does tend to concentrate uncolored processes on a smaller range of positions than a uniform pick from the entire interval, increasing the likelihood of collisions. But we can choose η to make $2(b+4)|S| = 2\eta T\frac{|S|}{|S|+1}$ a small enough fraction of T that this is not a problem.

36.1.3 Detecting collisions

The basic idea for detecting a collision is that I will abandon my color p if I hear any beeps in $[p-b-2, p+b+1]$ during the next period. This works great as long as nobody chooses exactly the same p as me. To avoid this, each process flips a coin and beeps at either p or $p+1$. So even if I choose the same slot as one or more of my neighbors, there is a $1/2$ chance per period that I detect this and pick a new color next time around.

What this means is that in each round (a) I have a constant probability of getting a good estimate of my degree (which means I set b correctly); (b)

I have a constant probability of detecting a collision with a neighbor if there is one (which means I pick a new position if I have to); and (c) I have a constant probability that if I pick a new position it is a good one. If we repeat all these constant-probability wins for $O(\log n)$ periods, then all n processes win, and we are done.

36.2 Maximal independent set

A high-impact early result in the beeping model was a paper by Afek *et al.*. [AAB+11] that showed that a biological mechanism used in fruit-fly sensory organ development to choose a subset of cells that are not too close to each other can be viewed as an implementation of maximal independent set using beeps. As a distributed algorithm, this algorithm is not so good, so instead we will talk about a follow-up paper [AABJ+11] by some of the same authors on more effective beeping algorithms for MIS.

Recall that a subset of the vertices of a graph is **indepedent** if no two vertices in the set are adjacent. A **maximal independent set** (**MIS**) is an independent set of vertices that can't be increased without including adjacent vertices. Equivalently, it's an independent set where every non-member is adjacent to some member.

Afek *et al.* give a couple of algorithms for beeping MIS that require either special knowledge of the graph or extensions to the beeping model. The justification for this is a lower bound, which they also give, that shows that without any knowledge of the graph, computing an MIS in the standard beeping model takes $\Omega(\sqrt{n/\log n})$ time with constant probaiblity. We'll describe the lower bound and then show how to compute MIS in $O(\log^3 n)$ time given a polynomial upper bound on n.

36.2.1 Lower bound

For the lower bound, the idea is to exploit the fact that the adversary can wake up nodes over time. To avoid allowing the adversary to delay the algorithm from finishing indefinitely by just not waking up some nodes for a while, the running time is computed as the maximum time from when any particular node p wakes up to when p converges to being in the MIS or not.

The cornerstone of the proof is the observation that if a process doesn't know the size of the graph, then it has to decide whether to beep or not within a constant number of rounds. Specifically, for any fixed sequence of beeps $b_0, b_1, \ldots$, where b_i is an indicator variable for whether the process hears a beep in round i after it wakes up, either the process never beeps

or there are constant ℓ and p such that the process beeps in round ℓ with probability p. This follows because if the process is ever going to beep, there is some first round ℓ where it might beep, and the probability that it does so is constant because it depends only on the algorithm and the sequence b, and not on n.

If an algorithm that hears only silence remains silent, then nobody ever beeps, and nobody learns anything about the graph. Without knowing anything, it's impossible to correctly compute an MIS (consider a graph with only two nodes that might or might not have an edge between them). This means that in any working algorithm, there is some round ℓ and probability p such that each process beeps with probability p after ℓ rounds of silence.

We can now beep the heck out of everybody by assembling groups of $\Theta(\frac{1}{p}\log n)$ processes and waking up each one ℓ rounds before we want them to deliver their beeps. But we need to be a little bit careful to keep the graph from being so connected that the algorithm finds an MIS despite this.

There are two cases, depending on what a process that hears only beeps does:

1. If a process that hears only beeps stays silent forever, then we build a graph with $k-1$ cliques $C_1, \ldots, C_{k-1}$ of size $\Theta(\frac{k}{p}\log n)$ each, and a set of k cliques $U_1, \ldots, U_k$ of size $\Theta(\log n)$ each. Here $k \gg \ell$ is a placeholder that will be filled in later (foreshadowing: it's the biggest value that doesn't give us more than n processes). Each C_i clique is further partitioned into subcliques $C_{i1}, \ldots, C_{ik}$ of size $\Theta(\frac{1}{p}\log n)$ each. Each C_{ij} is attached to U_j by a complete bipartite graph.

   We wake up each clique C_i in round i, and wake up all the U cliques in round ℓ. We can prove by induction on rounds that with high probability, at least one process in each C_{ij} beeps in round $i+\ell$, which means that every process in every U_i hears a beep in the first $k-1$ rounds that it is awake, and remains silent, causing the later C cliques to continue to beep.

   Because each C_i is a clique, each contains at most one element of the MIS, and so between them they contain at most $k-1$ elements of the MIS. But there are k U cliques, so one of them is not adjacent to any MIS element in a C clique. This means that one of the U_j must contain an MIS element.

   So now we ask when this extra U_j picks an MIS element. If it's in the first $k-1$ rounds after it wakes up, then all elements have seen the same history, so if any of them attempt to join the MIS then all of

them do with independent constant probability each. This implies that we can't converge to the MIS until at least k rounds.

Now we pick k to be as large as possible so that the total number of processes $\Theta(k^2 \log n) = n$. This gives $k = \Omega(\sqrt{n/\log n})$ as claimed.

2. If a process starts beeping with probability p' after hearing beeps for m rounds, then we can't apply the silent-beeper strategy because the C cliques will stop hearing silence. Instead, we replace the C cliques with new cliques $S_1, \ldots, S_{m-1}$ of size $\Theta(\frac{1}{p} \log n)$ each. We start the process by having the S cliques shout at the U cliques for the first $m - 1$ rounds. After this, we can start having the U cliques shout at each other: each clique U_j is connected to q earlier cliques, consisting of up to q $U_{j'}$ for $j < j$ and enough S_i to fill out the remainder.

   We now argue that if a process that hears only beeps chooses to join the MIS with constant probability after q rounds, then every U clique gets at least two processes joining with high probability, which is trouble. Alternatively, if no process in a U clique tries to join the MIS for at least q rounds, then for $q = O(n/\log n)$, there are U cliques that are connected only to other U cliques, which again means we don't get an MIS. So in this case we get a lower bound of $\Omega(n/\log n)$ on the time for each node to converge.

The lower bound in the paper is actually a bit stronger than this, since it allows the processes to send more detailed messages than beeps as long as there are no collisions. Reducing this back to beeping means tuning the constants so we get at least two messages out of every clique.

36.2.2 Upper bound with known bound on n

Algorithm 36.1 [AABJ+11] converges to a maximal independent set in $O(\log^2 N \log n)$ rounds, from any initial state, given an upper bound N on the number of processes n.

The proof that this works is a bit involved, so if you want to see all the details, you should look at the paper. The intuition runs like this:

1. At least one of any two adjacent processes that both think they are in the MIS will eventually notice the other during the final phase, causing it to restart.

2. If I start at the beginning of the protocol, and I have a neighbor already in the MIS, then I will hear them during my initial listening phase and restart.

```
1  Leave MIS and restart the algorithm here upon hearing a beep
2  for c lg^2 N rounds do
3      listen
4  for i ← 1 to lg N do
5      for c lg N rounds do
6          with probability 2^i/(8N) do
7              beep
8          else
9              listen
10 Join MIS
11 while I don't hear any beeps do
12     with probability 1/2 do
13         beep
14         listen
15     else
16         listen
17         beep;
```

Algorithm 36.1: Beeping a maximal independent set (from [AABJ+11])

3. If two adjacent nodes both execute the middle phase of increasing-probability beeps, then one of them will go through a phase where it listens with probability at least 1/2 while the other beeps with probability at least 1/2 (note that this might not be the same phase for both, because the nodes might not start at the same time). This gives at least a 1/4 chance per round that the likely listener drops out, for at least a $1 - n^{-c/2}$ chance that it drops out during the $c \lg n$ rounds that it listens with this probability, assuming its neighbor does not drop out. This means that by tuning c large enough, we can make it highly improbable that any pair of neighbors both enter the MIS (and if they do, eventually at least one drops out). So we eventually get a set that is independent, but maybe not maximal.

4. The hard part: After $O(\log^2 N \log n)$ rounds, it holds with high probability that every node is either in the MIS or has a neighbor in the MIS. This will give us that the alleged MIS is in fact maximal.

   The bad case for termination is when some node u hears a neighbor v that is then knocked out by one of its neighbors w. So now u is not in the MIS, but neither is its (possibly only) neighbor v. The paper gives a rather detailed argument that this can't happen too often, which we will not attempt to reproduce here. The basic idea is that if one of v's neighbors were going to knock v shortly after v first beeps, then the sum of the probabilities of those neighbors beeping must be pretty high (because at least one of them has to be beeping instead of listening when v beeps). But they don't increase their beeping probabilities very fast, so if this is the case, then with high probability one of them would have beeped in the previous $c \log N$ rounds before v does. So the most likely scenario is that v knocks out u and knocks out the rest of its neighbors at the same time, causing it to enter the MIS and remain there forever. This doesn't happen always, so we might have to have some processes go through the whole $O(\log^2 N)$ initial rounds of the algorithm more than once before the MIS converges. But $O(\log n)$ attempts turn out to be enough to make it work in the end.

Appendix

Appendix A

Assignments

Assignments should be uploaded to Canvas in PDF format.

Do not include any identifying information in your submissions. This will allow grading to be done anonymously.

Make sure that your submissions are readable. You are strongly advised to use LaTeX, Microsoft Word, Google Docs, or similar software to generate typeset solutions. Scanned or photographed handwritten submissions often come out badly, and submissions that are difficult for the grader to read will be penalized.

Sample solutions will appear in this appendix after the assignment is due.

Questions about assignments can be sent to the instructor directly at james.aspnes@gmail.com, or posted to the course Discord.

A.1 Assignment 1: due Thursday 2025-02-06, at 23:59 Eastern US time

[[[**To be announced.**]]]

A.2 Assignment 2: due Thursday 2025-02-20, at 23:59 Eastern US time

[[[**To be announced.**]]]

A.3 Assignment 3: due Thursday 2025-03-06, at 23:59 Eastern US time

[[[**To be announced.**]]]

A.4 Assignment 4: due Thursday 2025-04-03, at 23:59 Eastern US time

[[[**To be announced.**]]]

A.5 Assignment 5: due Thursday 2025-04-17, at 23:59 Eastern US time

[[[**To be announced.**]]]

Appendix B

Sample assignments from Fall 2023

B.1 Assignment 1: due Thursday 2023-09-21, at 23:59 Eastern US time

B.1.1 Maximal independent set in a ring

Given a graph, a **maximal independent set** (MIS) is a subset S of the vertices that is an **independent set** (no two vertices in S have an edge between them) that is **maximal** (no superset of S is also an independent set). We will say that a distributed algorithm computes a maximal independent set if every process eventually returns 0 or 1, and the set of processes that return 1 form an MIS.

Let's suppose we have an asynchronous bidirectional ring of unknown size with deterministic processes. For each of the following assumptions, show either (a) no algorithm correctly computes an MIS in the worst case, or (b) there is an algorithm that computes an MIS in $O(f(n))$ time in the worst case, *and* there is a matching lower bound showing no algorithm can do better than $\Omega(f(n))$ in the worst case.

1. The network is anonymous.

2. All processes have distinct ids, but the algorithm is comparison-based.

Solution

1. This case is impossible. The proof is the same as for leader election in an anonymous ring. In a synchronous execution, symmetry is never

broken, and so if any process returns, all processes return the same value. This either yields $S = \emptyset$ (not maximal) or $S = V$ (not independent).

2. Possible, with $\Theta(n)$ time both necessary and sufficient.

   - For the algorithm, elect a leader using a comparison-based $O(n)$-time leader election algorithm (LCR works). Relay a message clockwise from the leader to count off the position of each node (see Algorithm B.1). Send a single message counterclockwise from the leader to notify node $n-1$ of its special position. The time to complete both of these steps is at most $O(n)$.

     Now have each node return 1 if and only if (a) it has an even position and (b) it is not in position $n-1$.

     This is an independent set since no two even-position nodes other than $n-1$ are adjacent. It is maximal because adding any other node i creates two adjacent nodes (i and $i-1$ in the case of an odd node, $n-1$ and 0 in the case of $n-1$). So we get an MIS in $O(n)$ time.

   - For the lower bound, adapt the Frederickson-Lynch lower bound for leader election. As for the upper bound we need to be a little careful about odd vs even rings.

     Consider a synchronous execution, then after k rounds, two nodes will return the same value (if any) if their k-neighborhoods are order-equivalent. Now observe that in a ring of size n i with $\mathsf{id}_i = i$ for all i, nodes $\lfloor \frac{n}{2} \rfloor - 1$, $\lfloor \frac{n}{2} \rfloor$, and $\lfloor \frac{n}{2} \rfloor + 1$ have order-equivalent $(\lfloor \frac{n}{2} \rfloor - 1)$-neighborhoods, as in both cases the ids in these neighborhoods are strictly increasing. It follows that if any of of these nodes returns a value after $\lfloor \frac{n}{2} \rfloor - 1$ or fewer rounds, either all three return 0 (meaning that the computed independent set is not maximal, since node $\lfloor \frac{n}{2} \rfloor$ can be added without creating two adjacent nodes); or all three return 1 (meaning that the computed set is not independent). This gives the desired matching $\Omega(n)$ worst-case lower bound.

B.1.2 Deanonymization

Suppose you have an asynchronous bidirectional message-passing network in the form of an arbitrary connected graph, which is mostly anonymous in the sense that every node but one runs the same code, and that each node can

```
1 initially do
2     if I am the leader then
3         position ← 0
4         send 0 clockwise
5 upon receiving m do
6     if I am not the leader then
7         position ← m + 1
8         send m + 1 clockwise
```

Algorithm B.1: Counting off nodes in a ring

only identify its neighbors by a local **port number**, an element of $\mathbb{N}$, that is only meaningful to that node and is not correlated with any other node's port numbers. This means that when a message is delivered from a node i to a node j, j sees that the message came from a particular port p that it uniquely associates with i; it can similarly send messages to port p that will be delivered to i. You may assume that each node has a complete list of its neighbors' port numbers, so it can tell, for example, if it has a neighbor that it hasn't received any messages from.

The one non-anonymous node is marked as the initiator and can run special code, but is subject to the same port-number limitations as all the other nodes. None of the nodes know the size of the graph n or its diameter D.

We would like to assign a unique id to every node in the system in the range 1 through n. Prove or disprove: There exists an algorithm that does this in time $O(D)$.

Solution

We'll show that it is possible by constructing an algorithm.

First observe that we can apply the alpha synchronizer to this this system, since the alpha synchronizer only requires that a node be able to detect when it has received a message (or noMsg from each of its neighbors, and the assumptions on port numbers are sufficient to do this. We also don't care about message complexity. So we can simplify our life by assuming that the model is synchronous. (Alternatively we can replace the synchronous breadth-first search protocol in the first step below with an asynchronous breadth-first search protocol, but the end result is pretty much the same

either way.)

Run a synchronous breadth-first search protocol to construct a shortest-path tree rooted at the initiator. This takes $O(D)$ time and yields a tree with depth at most D. Note that the parent pointers in the usual protocol will now have port numbers rather than ids, but this doesn't affect the algorithm.

Using convergecast, compute the size of every subtree and have each node pass this information on to its parent. This takes an additional $O(D)$ time.

We can now recursively assign ids through the tree. The initiator starts the process by sending itself a message containing the id range $\{1 \dots n\}$. Each node that receives an id range $\{i \dots j\}$ assigns i to itself and then partitions the remaining range $\{i+1 \dots j\}$ into subranges $\{i_1 \dots j_1\}, \dots \{i_k \dots j_k\}$, where k is the number of children it has and each range has length equal to the number of nodes at the subtree rooted at the corresponding child when sorted by port number. Now send each child its range. A straightforward induction argument shows that this assigns a unique identifier to every node. The time to perform this broadcast-like operation is proportional to the depth of the tree, giving another $O(D)$ time. So the total time for all steps is $O(D)$.

B.2 Assignment 2: due Thursday 2023-10-05, at 23:59 Eastern US time

B.2.1 Synchronous agreement in a bipartite network

Let $n \geq 2$, and suppose you have a synchronous network with $2n$ processes $p_1, \dots, p_n$ and $q_1, \dots, q_n$. The network is bipartite: each p_i can send and receive messages from each q_j, but no pair of processes p_i and p_j or q_i and q_j can communicate directly. The processes are subject to crash failures, where as usual a process that crashes in a particular round may send any subset of the messages it intended to send in that round. Our goal is to solve synchronous agreement, as defined in §9.1.

1. As a function of n, what is the largest number of potential crash failures t for which it is possible to solve agreement? (Give an exact value.)

2. As a function of n and t, what is the best possible asymptotic worst-case running time for synchronous agreement, assuming t is small enough to make synchronous agreement possible. (Give an asymptotic expression in t and/or n.)

You should justify your answers with matching upper and lower bounds. For the upper bound side, you may find it helpful to give a single algorithm that applies to both cases.

Solution

For our algorithm, we'll run Dolev-Strong (see §9.2) largely unmodified, except that we will run for $2t+2$ rounds and each process will send messages only to its neighbors.

Observe that (a) if $t \leq n-1$, there is at least one non-faulty p_i and at least one non-faulty q_j; and (b) since we can divide the rounds into $t+1$ phases of two rounds each, there are at least two consecutive rounds $2s$ and $2s+1$ with no new crash failures in either round. Let $\langle k, v\rangle$ appear in S_{p_i} at the beginning of round $2s$, where p_i has not yet crashed. Then $\langle k, v\rangle$ is transmitted to all surviving q_j in round $2s$, and at least one such q_j forwards $\langle k, v\rangle$ to all surviving p_i in round $2s+1$. Similarly, any $\langle k, v\rangle$ that appears in S_{q_j} at the beginning of round $2s$ is transmitted to all p_i and $q_{j'}$ by the end of round $2s+1$. It follows that $S_{p_i}^{2s+1} = S_{q_j}^{2s+1}$ for all p_i and q_j that do not crash in round $2s+1$ or earlier. The same argument as used for the original algorithm shows that this continues to hold for all subsequent rounds, and so all processes choose the same value from the same set at the end of the protocol, giving agreement. Termination is trivial as usual, and validity follows from the same argument as for the original algorithm.

This shows that consensus is possible in $O(t)$ time when $t \leq n-1$. Now we just need the corresponding lower bounds.

1. To show that $t \leq n-1$ is necessary, suppose $t \geq n$. Then the adversary can crash all processes q_j immediately, leaving each p_i with no live neighbors. If some p_i decides on a value that is not its input, it violates validity in the execution where all other processes have the same input. But if each p_i decides its own input and $n \geq 2$, then we violate agreement if the inputs don't all agree.

   (There is an annoying special case when $n = 1$. In this case the protocol can tolerate one crash failure, because the survivor will agree with itself. Fortunately the problem statement excludes this case.)

2. For the time bound, suppose that we can solve the problem in t rounds. Then in a complete network we can also solve synchronous agreement with t failures in t rounds, since nothing prevents the processes in the complete network from choosing only to communicate using a

bipartite subnetwork. But this violates the Dolev-Strong lower bound (see §9.3). So we get matching upper and lower bounds of $\Theta(t)$ on the time complexity for this problem.

B.2.2 Leader rotation

We are given an asynchronous bidirectional network of n processes in the form of an arbitrary connected graph with diameter D. Each process has access to a **special action**, a local operation it uses to claim temporary leadership of the network. We'd like this leader role to repeatedly rotate through the n processes, in the sense that there is an assignment $0, \dots n-1$ of positions to the processes such that the i-th special action is always carried out by the process in position $i \bmod n$. (Note that these positions can be chosen by the protocol and do not necessarily have any meaning outside of showing that the protocol satisfies this requirement.)

We assume that the processes are not anonymous and that every process in the network knows the entire structure of the graph, including all process identities.

Since we are considering infinite executions, we can't talk about the time complexity of the execution has a whole, so instead we will define the **responsiveness** of an execution of the protocol as the maximum time between any two consecutive special actions.

Show that there is a function $f(n, D)$ such that any protocol for this problem has responsiveness $\Omega(f(n, D))$ in the worst case, and that some such protocol has responsiveness $O(f(n, D))$ in the worst case.

Solution

It turns out that the diameter is not important. There exists a protocol with responsiveness $\Theta(1)$, which is also the lower bound.

We'll start by showing that no protocol with $n \geq 2$ can have responsiveness less than 1.

Consider a synchronous execution Ξ, and suppose that there are two consecutive special actions s_i and s_{i+1} such that the time between s_i and s_{i+1} is less than 1. Then s_i and s_{i+1} are not causally ordered, and there is a causal shuffle Ξ' of Ξ in which s_{i+1} occurs before s_i but the other special actions occur in the same order as before. Let p_i and p_{i+1} be the processes execution s_i and s_{i+1}. Then in Ξ', p_i executes both the $(i+1)$-th special action and the $(i+n)$-th special action, which is requires p_i to have both

positions $(i+1) \bmod n$ and $(i+n) \bmod n = i \bmod n$, which is inconsistent withe requirement of distinction positions when $n \geq 2$.

For the upper bound, we need to show that for any graph G there is a protocol that rotates through the special actions as described above with a gap of at most $O(1)$ time between any consecutive special actions. We can do this by adapting a depth-first traversal of a spanning tree T of G, circulating a token along the $2n$ edges of the tour so that it reaches every node at least once every $2n$ steps. A node will execute its special action on exactly one of these occasions where it receives the token, carefully chosen so that the token doesn't travel too far without triggering a special action.

```
1  for ever do
2      if I am not the root then
3          wait to receive token from my parent
4      if My depth is even then
5          perform special action
6      for each child c in increasing order by id do
7          send token to c
8          wait to receive token from c
9      if My depth is odd then
10         perform special action
```

Algorithm B.2: Leader rotation algorithm

The algorithm is given as Algorithm B.2. We assume that a rooted spanning tree has already been constructed and that each node knows its parent (if any), its children, and its depth in the tree. (Each process can easily compute this at the start of the tree based on its knowledge of the graph; so long as the processes use the same algorithm to construct the tree, they will all behave consistently.)

This protocol repeatedly carries out a depth-first traversal of the tree by passing a single token along the edges of the tree. Each even-depth node (including the root, at depth 0) performs its special action when the token enters its subtree; each odd-depth node performs its special action when the token leaves. Since exactly one of these events occurs during each traversal and each traversal visits the nodes in the same fixed order, we satisfy the requirements that the special actions rotate among the nodes.

To show a bound on the gap between special actions, consider four consecutive events in the execution, and look at the messages sent by the

first three events along edges of the tree. Classify these messages as D or U depending on whether the message goes down the tree (is sent to a child) or goes up (is sent to a parent). There are eight possible patterns DDD, DDU, DUD, ..., UUU for the three messages.

1. If DD appears in the pattern, then one of the receivers of these messages has even depth and performs the special action. This covers DDD, DDU, and UDD.

2. Similarly, if UU appears in the pattern, then one of the senders of these messages has odd depth and performs the special action. This covers DUU, UUD, and UUU.

3. The remaining patterns are DUD and UDU. In both cases, the process in the middle of DU that receives the D message and sends the U message is a leaf. No matter what its depth, it performs the special action.

It follows that any sequence of four consecutive send events involves at least one special action. So the maximum time between special actions is 4.

This gives a matching $\Omega(1)$ lower bound and $O(1)$ upper bound on the responsiveness of this protocol.

B.3 Assignment 3: due Thursday 2023-10-26, at 23:59 Eastern US time

B.3.1 Evil twins

Suppose we have a system where a process p can be paired with an **evil twin**, a Byzantine process $\check{p}$ that can send messages that appear to come from p. Messages from $\check{p}$ enter the same buffer as messages from p, and cannot be distinguished by the recipient from legitimate messages from p. The existence of the evil twin does not otherwise affect the execution of p, which continues to behave normally.

Prove or disprove: There exists a constant $c > 0$ such that it is possible to solve binary consensus in an asynchronous message-passing system with deterministic processes, as long as the number of evil twins t is less than cn.

Here binary consensus is defined as a protocol that satisfies the usual requirements of agreement (all processes decide on the same value), termination (all processes eventually decide), and validity (if all processes start with the same input, they all decide on this input)?

Solution

We can solve the problem for $t < n/3$, by simulating any standard synchronous Byzantine agreement algorithm with optimal fault tolerance, with an extra round at the end to handle processes that have evil twins but still need to decide on the common value.

To enforce synchrony, we use the alpha synchronizer. Since every good process sends a message to every other process in every simulated round, nobody gets stuck waiting for messages from all other processes, and the worst that happens is that some process might receive a round-r message from $\check{p}_i$ instead of p_i. In this case we treat p_i as Byzantine for the simulated execution.

We will also assume that any p_i with an evil twin behaves arbitrarily during the main protocol. This absolves us from worrying about good processes sending bad messages, and again bad messages from a twinned p_i are indistinguishable from bad messages from a Byzantine p_i in the simulated execution.

Running EIG or a similar algorithm then gives agreement among all the processes that do not have evil twins. We add one more round where each process announces its decision value, and all good processes (including twinned processes) wait to receive decision values from all n processes and decide on the majority. Since at least $\frac{2}{3}n$ processes agree coming out of the Byzantine agreement protocol, all good processes will see the same majority value and reach the same decision.

B.3.2 Crash failures with recovery

Consider an asynchronous message-passing model with deterministic processes, where a process can crash, losing all of its state (including its input), but then recovers to a default state from which it can continue its execution. We would like to solve binary consensus in this model, characterized by agreement (all processes eventually decide the same value), validity (if all processes start with the same input, they all decide this input), and termination (every process eventually decides on some value). Note that while defining the problem in this model we do not necessarily think of processes as being faulty or non-faulty; any process can crash, possibly more than once, but we still require that it eventually makes a decision on the same value as all the others.

As a function of n, what the largest number of possible crash failures t for which it is consensus as defined above can be solved in this model?

Solution

The largest number of crash failures we can tolerate is $t = n-1$. At $t = n$, it is possible for every process to crash immediately, erasing all inputs. Since this gives the same configuration in both an all-0-input and all-1-input execution, whatever the processes decide will validity in one of these executions.

To solve consensus with $t = n - 1$, we'll first show how to simulate a system with standard crash failures and a perfect failure detector, then adapt the consensus protocol from Chandra and Toueg [CT96] for the strong failure detector (see Algorithm 13.2). to solve the problem in the crash-with-recovery model.

The idea is that whenever a process recovers, it will send a message failed to all other processes, and otherwise act like a crashed process by no longer participating in the simulated consensus protocol. A never-crashed process that receives a failed message from some process p will (a) add p to its list of suspect processes; and (b) send p its decision value, if it has already decided, or add p to a list of processes to be notified of its decision value when it decides, if it has not already decided. A previously-crashed process that is notified of a decision value decides on that value. Other than these changes, the never-crashed processes run Algorithm 13.2 essentially unmodified.

Agreement follows from the fact that all never-crashed processes agree in Algorithm 13.2 and all crashed processes that decide choose a value sent to them by a never-crashed process. Validity follows from validity of Algorithm 13.2 and the same argument.

Termination is a bit trickier since we have to allow for the possibility that a process might crash more than once. Any process that doesn't crash decides at the end of Algorithm 13.2 (but note that it may still need to respond to failed messages). For a process p that does crash, consider what happens when it recovers for the last time. At this point the process sends failed to all processes, including at least one process q that does not crash. Eventually q sends a value to p (either immediately in response to p's message or eventually when it decides). This value is sent after p's last crash, so eventually p receives it and decides.

B.4 Assignment 4: due Thursday 2023-11-09, at 23:59 Eastern US time

B.4.1 A one-object mutex

The Deadlock-Free Lock Company has hired you as a consultant for its project to built a new fetch-and-add-based mutex that works for any number of processes and uses no extra registers. Their starting point is the ticket algorithm for simulating a queue using a RMW object as described in §18.3.2.1, but rather than use a general RMW object, they wish to use a fetch-and-add object that supports a single operation $\texttt{FAA}(r, v)$ that adds v to the current value of r and returns the old value. Both v and the contents of the register may be arbitrary integers (including negative integers) of any size.

The intern who previously worked on the project suggested the implementation in Algorithm B.3. Here K is a large constant. The intuition is that r mod K is used to track which tickets will be given out next and $\lfloor r/K \rfloor$ stores which ticket can be used to enter the critical section. Each process calls $\texttt{acquire}(r)$ in its entry section and $\texttt{release}(r)$ in its exit section. The fetch-and-add register starts with value 0.

```
  // acquire the lock
1 procedure acquire(r)
      // take a ticket
2     t ← FAA(r, 1) mod K
      // spin until I am at the front of the line
3     while ⌊FAA(r, 0)/K⌋ ≠ t do
4         spin
  // release the lock
5 procedure release(r)
      // advance the front of the line
6     FAA(r, K)
```

Algorithm B.3: Candidate fetch-and-add mutex

1. Show that Algorithm B.3 can violate both mutual exclusion and deadlock-freedom.

2. Prove or disprove: For *any* algorithm, if (a) it uses only one fetch-and-add object and no other objects and (b) it works for an arbitrarily

large unknown number of processes, then there exists an execution in which it eventually violates at least one of mutual exclusion or deadlock-freedom.

Solution

1. Since I am lazy I will give a single execution that violates both mutual exclusion and deadlock-freedom.

   Send in $K+2$ processes $p_0 \dots p_{K+1}$, and have all of them execute Line 2 in order. Then each process p_i gets ticket $i \bmod K$, and in particular p_1 and p_{K+1} both get 1. This is unfortunate, because $\lfloor r/K \rfloor = 1$, so both of these processes leave the loop in Line 3 and enter the critical section together. Mutex is violated!

   Even worse, since r never decreases, poor process p_0 can never see $\lfloor r/K \rfloor = 0$ and thus remains stuck at Line 3 forever. This is true even if every other process runs to completion and makes no attempt to re-enter the critical section. We haven't actually shown that every other process *can* run to completion, but we eventually reach some configuration where either (a) every remaining process is stuck, or (b) p_0 is alone and stuck. In either case, deadlock-freedom is violated.

2. We'll disprove the claim by showing that a working mutex is possible.

   Here condition (b) makes things difficult, because even if we could tweak the calculation of $\lfloor r/K \rfloor$ to make it wrap around like $r \bmod K$, for $n > K$ we still have the issue of two processes getting the same ticket. So we will need to abandon Algorithm B.3 and do something else.

   Algorithm B.4 gives a mutex algorithm using a single fetch-and-add object, which we assume is initialized to 0. The idea is similar to the mutex using test-and-set given in Algorithm 18.1. Each process will attempt to acquire the lock by incrementing the fetch-and-add object, and only a process that sees 0 will win. But since we can't reset the object we'll have the winner decrement the object on its way out, and have each loser decrement the object once to remove its excess increment and then spin until it sees a 0 before attempting to increment again.

   Let's prove that Algorithm B.4 works. We'll write that a process p is in the critical section if it has escaped the loop by seeing 0 in Line 2 and has not yet performed the decrement in Line 7.

```
  // acquire the lock
1 procedure acquire(r)
2     while FAA(r, 1) ≠ 0 do
3         FAA(r, −1)
4         while FAA(r, 0) ≠ 0 do
5             spin
  // release the lock
6 procedure release(r)
7     FAA(r, −1)
```

Algorithm B.4: Improved fetch-and-add mutex

We can now state an invariant: The value of r is equal to the number of processes c in the critical section plus the number of processes d at Line 3. To prove this, start by noting that in the initial configuration, $r = c + d = 0$. The value of r changes only when a process executes a fetch-and-add in Line 2, Line 3, or Line 7, so we need to show that $r = c + d$ continues to hold in each of these cases:

- In Line 2, r increasing by 1 and exactly one of c or d increases by 1, depending on whether the process sees 0 and enters the critical section or sees 1 and moves to Line 3.
- In Line 3, r and d both drop by 1.
- In Line 7, r and c both drop by 1.

Conversely, these three lines are also the only places where c or d change. Since we have already shown that they preserve $r = c + d$, the invariant holds throughout any execution of the algorithm.

The invariant directly gives mutual exclusion: If in some configuration there is already a process in the critical section, then $r = c + d \geq c \geq 1$ and so no process can observe $r = 0$ in Line 2 and enter the critical section.

For deadlock-freedom we want to show that if there is at least one process in the entry section, r eventually reaches 0 and stays there long enough for some process to see it in Line 2. Start in any reachable configuration. If $c = 1$, then we can run until the process in the critical section leaves, reducing c to 0. Suppose that c remains 0 forever (if not, some process entered the critical section and we are done). If r

never reaches 0, every process in the entry section eventually gets stuck at Line 4. But then $d = 0$ implies $r = 0$, a contradiction. If instead r reaches 0, then in that configuration no process is in Line 3, so every process is either at Line 4 or Line 2. Processes in Line 4 see $r = 0$ and move to Line 2; this does not change r. So eventually some process executes Line 2, sees $r - 0$, and enters the critical section.

A more general solution. Here's an alternative approach that is a bit more general. Let $r = \sum_{i=0}^{\infty} 2^i r_i$ be the value of the fetch-and-add register. Assign a countably infinite sequence $b_{p0}, b_{p1}, \ldots$ of bit positions to each process p, so that no two processes' bits overlap. (We can do this for countably many processes using Cantor's pairing function.) Observe that (a) any process can take a snapshot of all the bits of all processes using $\texttt{FAA}(r, 0)$, and (b) any process p can update its own bits atomically by doing $\texttt{FAA}(r, \delta)$ where $\delta = \sum 2^{b_{pj}} \delta_j$ with $\delta_j \in \{-1, 0, 1\}$ being the desired change in p's j-th bit. This gives an implementation of snapshot over single-writer registers of unbounded size using a single `FAA`.

Since unbounded single-writer registers are enough to implement Lamport's bakery algorithm for starvation-free mutex (see §18.5.3), we are done.

A curious feature of this construction is that we don't actually need full-blown fetch-and-add, since we are effective only doing reads and generalized increments. So an unbounded generalized counter by itself is enough to simulate unbounded single-writer snapshot for any finite number of processes.

B.4.2 A locker object

The Wait-Free Locker Company has hired you as a consultant to evaluate the strength of its new locker object. This object, intended for delivery of licensed digital content to subscribing consumer processes, stores at most one value. It guarantees that data is not lost by ignoring writes to a non-empty locker, and preserves the licensor's valuable intellectual property rights by emptying the locker when it is read.

Specifically, a `write` operation inserts a value into the locker if none is present already; otherwise it discards the new value. A `read` operation removes and returns any value in the locker, returning $\perp$ if the locker is empty. Pseudocode describing these operations is given in Algorithm B.5.

```
1 procedure write(ℓ, v)
2     atomically do
3         if ℓ = ⊥ then ℓ ← v
4 procedure read(ℓ)
5     atomically do
6         v ← ℓ
7         ℓ ← ⊥
8         return v
```

Algorithm B.5: Locker operations

What is the consensus number of this object?

Solution

The consensus number of this object is 2.

To solve consensus for $n = 2$, initialize the locker with some non-null default value, say 1, and have each process attempt to read the locker after writing its input to a register. Then whichever process gets 1 has won and can return its own input, while the other process can read the winning input from the winner's register as usual.

To show we can do consensus for $n = 3$, we'll use an argument similar to that for queues without peek. Consider an alleged three-consensus protocol using locker objects and atomic registers. Do the usual thing to get to a bivalent configuration C with pending operations x and y on the same locker object ℓ by processes p and q such that Cx is 0-valent and Cy is 1-valent. Let z be a pending operation by the third process r. We have that Cr is univalent but we don't care about this for the purpose of the argument.

We want to show that for any choice of x and y, we can construct an execution in which r can't tell which of x and y went first. As usual we know that x and y must be operations on the same object and that this object must be a locker.

If x and y are both `read` operations, then Cxy and Cyx both leave an empty locker and are indistinguishable to r.

If x is a `write` and y is a `read`, then we need to consider two cases depending on whether the locker is empty in C or not. If the locker is empty, then $Cyx \sim_r Cx$, since in either case only q knows if y occurred or not. If the locker is not empty, then $Cxy \sim_r Cy$ since x has no effect on a non-empty

locker and only p knows whether it occurred or not.

If x and y are both `write`s, then we have to put in some effort to destroy the evidence of which went first. We can assume that the locker is empty in C, because otherwise x and y are both no-ops. Configurations Cxy and Cyx now differ in the value in the locker. Run p solo starting from either of these configurations. To decide, it must be able to distinguish between them, which requires reading the locker. Let α be the sequence of operations done by p up to and including its first read of the locker. Then $Cxy\alpha \sim_r Cyx\alpha$ since the locker is now empty and only p knows its value.

B.5 Assignment 5: due Thursday 2023-11-30, at 23:59 Eastern US time

B.5.1 Writable max registers

Consider a **writable max register** object r that supports operations `read`(r), `write`(r, v) and `writeMax`(r, v), where `read`(r) returns the current value of r, `write`(r, v) replaces the value of r with v, and `writeMax`(r, v) replaces the value of r with v only if v is larger than the current value.

Since this object implements an unbounded max register (just don't do any `write` operations), the Jayanti-Tan-Toueg bound shows that any possible solo-terminating linearizable implementation of a writable max register from atomic registers requires at least $\Omega(n)$ steps for some operation in the worst case. So let us consider a writable max register restricted by the following constraints:

1. The register holds only m possible values $0 \ldots m-1$, where m is polynomial in n.

2. At most w `write` operations can safely be applied to the register, where w is also polynomial in n. Any additional `write` operations have an unpredictable effect.

Note that the limited-use restriction only applies to `write` operations. There is no limit on the number of `read` or `writeMax` operations.

Prove or disprove: There exists a wait-free linearizable implementation of a restricted writable max register as defined above from atomic registers that uses $o(n)$ steps for any operation in the worst case.

Solution

To implement a writable max register r, we'll use a standard bounded max register m_r to store lexicographically-ordered tuples $\langle g, i, v\rangle$ where g is a generation number in $\{0 \dots w\}$, i is a process id in the range $0 \dots n-1$, and v is a value in $\{0 \dots m-1\}$. We can do this by encoding $\langle g, i, v\rangle$ as $mn \cdot g + m \cdot i + v$, which is both bijective and order-preserving. To simplify the presentation of the algorithm, we will treat this encoding as happening implicitly. We assume that m_r starts with its minimum value 0, corresponding to the tuple $\langle 0, 0, 0\rangle$.

We can then increment the generation to reset the register in response to `write` operations, and use the max-register property within a generation to implement `writeMax`. Pseudocode for the resulting algorithm is given in Algorithm B.6.

```
1 procedure read(r)
2     ⟨−, −, v⟩ ← read(m_r)
3     return v
4 procedure write(r, v)
5     ⟨g, −, −⟩ ← read(m_r)
6     writeMax(m_r, ⟨g + 1, myId, v⟩)
7 procedure writeMax(r, v)
8     ⟨g, i, −⟩ ← read(m_r)
9     writeMax(m_r, ⟨g, i, v⟩)
```

Algorithm B.6: Writable max register

We assume that the number of calls to `write` is bounded by w; this avoids overflow in Line 6. Under this assumption, the embedded max register m_r takes on values in the range $\{0 \dots mnw + m(n-1) + (m-1)\}$. So we can implement it with the standard construction of [AACH12] (see §22.2) using $O(\log mnw) = O(\log n) = o(n)$ steps per operation. This gives a wait-free implementation that uses $o(n)$ steps per operation, since Algorithm B.6 uses only a constant number of operations on m_r for each operation of r.

To linearize a concurrent execution, the intuition is that the generation and id (used as a tie-breaker) gives an increasing sequence of intervals, each consisting of a `write` operation, followed by zero or more `read` and `writeMax` operations. But we need to be a little careful to deal with out-of-date `write` and `writeMax` operations that have no effect on m_r.

Call a `write` or `writeMax` operation **punctual** if it writes a $\langle g, i, v\rangle$ where

$\langle g, i\rangle$ is at least as big as the corresponding components of m_r, and **delayed** otherwise. Assign linearization points as follows:

1. A `read` operation is linearized at the time of its `read`(m_r) operation in Line 2.
2. A punctual `write` operation is linearized at the time of its `writeMax` operation in Line 6.
3. A punctual `write` operation is linearized at the time of its `writeMax` operation in Line 9.
4. A late `write` or `writeMax` operation that writes $\langle g, v\rangle$ is linearized just before the first `write` operation that writes $\langle g', i', v'\rangle$ where $\langle g', i'\rangle > \langle g, i\rangle$. Ties between such late operations are broken arbitrarily.

First let us show that each linearization point lies within the interval of its operation. For the first three cases, this is trivial. For the last case, in order for an operation π to be late, it must read a pair $\langle g, i\rangle$ from m_r and then write m_r while m_r holds some pair $\langle g', i'\rangle > \langle g, i\rangle$. Since the only operation that changes this pair in m_r is a `write`, the first such `write` writes to m_r between π's `read` and `writeMax` operations, and thus within the interval of π. So the sequential execution order is consistent with the observed execution order.

To show that this gives a correct sequential execution S, observe that we can organize S as a sequence of intervals. The first interval consists only of zero or more `read` and `writeMax` operations with initial pair $\langle 0, 0\rangle$, followed by zero or more delayed operations; subsequent intervals are similar but start with a `write` that writes some $\langle g, i, v_0\rangle$, Within each such interval, m_r starts with some value $\langle g, i, v_0\rangle$, and all operations that precede a `read` operation have the same initial pair $\langle g, i\rangle$. So a `read` operation within the interval returns the largest of the value v_0 supplied by the most recent write `write` or any value written in the same interval by a `writeMax`. This matches the specification of the writable max register, so we are done.

B.5.2 Approximate vector agreement

Given two vectors x and y, the **Hamming distance** between x and y is the number of positions i such that $x_i \neq y_i$.

Consider the following vector agreement problem. Each process p has an input vector x^p with m components, where m is typically much larger than n. We would like a protocol that gives to each process p an output y^p, satisfying the following conditions, for some choice of k:

Wait-free termination Each process obtains an output after a finite number of its own steps.

Validity For each position i and process p, y_i^p is equal to some x_i^q.

Maximum distance The Hamming distance between any two outputs y^p and y^q is at most k.

For example, the following might be an example of inputs and outputs that satisfy these constraints for $n = 3$ and $k = 3$:

```
saffron  sanding
evening  winding
windows  winning
```

Show that there is wait-free deterministic solution to this problem using atomic registers for some $k = O(n)$, where n is the number of processes.

Solution

We'll use a safe agreement object [BGLR01] (see §28.2) for each position i. Since it takes a distinct failure to knock out each safe agreement object, at most $n - 1$ of these objects will get stuck. So when a process p sees return values from $m - (n - 1)$ objects, it will combine these with its own inputs for the missing positions to produce its output y^p.

To avoid a lot of handwaving about how the safe agreement objects interact, we'll break the abstraction barriers around their implementations and build an explicit loop for managing the unsafe phases. This also allows us to skip looping in the safe phase. Pseudocode is given in Algorithm 15.

We claim that this satisfies all three requirements for $k = 2n - 3$.

Validity is easy. Any y_i^p is either x_i^p or a proposal derived from some x_i^q.

Termination is also easy, since the algorithm contains no unbounded loops.

For maximum distance, observe that the final snapshot s always contains at least one level 2 proposal for each position, since every process that reaches this line either observes a level 2 proposal in Line 5 or writes one in Line 8. We can argue that any two such level 2 proposals that are used in Line 14 are equal, because if I take a snapshot that includes a level 2 proposal in position 1 and no level 1 proposal, any process working on position i that has not yet written a level 1 proposal will see the level 2 proposal and back off instead of writing a new one. So the only places where y^p and y^q can

```
1  procedure vectorAgreement(x)
       // unsafe phase of safe agreement for each i
2      for i ← 1 to m do
           // propose x_i at level 1 as in safe agreement
3          a[p]_i ← ⟨1, x_i⟩
4          s ← snapshot(a)
5          if s contains a[q]_i with level 2 then
               // back off
6              a[p]_i ← ⟨0, x_i⟩
7          else
               // advance
8              a[p]_i ← ⟨2, x_i⟩
       // safe phase of safe agreement
9      s ← snapshot(a)
10     for i ← 1 to m do
11         if s contains a proposal at level 1 for i then
12             y_i ← x_i
13         else
14             y_i ← some level 2 proposal for i
15     return y
```

Algorithm B.7: Solution to vector agreement problem

differ are locations where at least one of p or q sees a level 1 proposal in its last snapshot. Suppose p does the last snapshot first. Then there are at most $n-1$ level 1 proposals in p's snapshot, since each process has at most one level 1 proposal at a time, and p has already removed any of its level 1 proposals. For q, there are at most $n-2$ level 1 proposals, since both p and q have left the unsafe phase when q does its snapshot. This gives the claimed bound of $k \leq (n-1) + (n-2) = 2n-3 = O(n)$.

There is a much simpler solution that I did not come up with myself, but which was suggested by several people during office hours. Construct a multi-writer snapshot array A with m entries, initially blank. Have each process repeatedly take a snapshot, and if the snapshot contains a blank position $A[i]$, write the process's value x_i to $A[i]$. If not, return the snapshot.

When some process sees a full snapshot and returns, there are at most $n-1$ pending write operations that together can change at most $n-1$ positions in A before all processes see a full snapshot and return. Since any two return values can disagree only in one of these $n-1$ positions, this gives $k = n-1 = O(n)$.

Appendix C

Sample assignments from Fall 2022

C.1 Assignment 1: due Thursday 2022-09-22, at 23:59 Eastern US time

C.1.1 Leader election using broadcast

In the usual asynchronous message-passing model, each process can choose to send a message to any of its neighbors. To make our system super-anonymous, suppose that we eliminate the need for a process to know what neighbors it has by replacing these point-to-point channels with a **broadcast channel** where any message that is sent is eventually delivered to every process (including the sender). This is equivalent to requiring in the standard model that whenever a process sends a message, it sends n copies of the message, one to each possible recipient. As in the standard model, we assume that every copy of a message is delivered after at most 1 time unit, but by default impose no other constraints on the time at which each copy of a message is delivered.

We would like to solve leader election in this model, under various assumptions. By leader election, we mean a protocol in which exactly one process eventually sets its `leader` bit to 1. For each of the conditions below, give an algorithm for solving leader election, prove its correctness, and compute its message complexity and running time; or prove that no such algorithm is possible.

1. An anonymous system in which all processes run the same code and do not have unique IDs.

2. A uniform system with IDs, where uniformity means that the code for each process depends only on its ID and not on the size of the system.

3. A non-uniform system with IDs, where the processes know n.

4. A uniform system with IDs, but where the broadcast channel is replaced by an **ordered broadcast** channel that guarantees for each pair of messages m_1 and m_2, that if m_1 is sent before m_2, each process receives m_1 before it receives m_2.

Solution

For computing message complexity, there is an ambiguity in the problem description: does sending a single broadcast count as n messages or one message? Below, we assume a broadcast counts as n messages, but one message is also a reasonable interpretation, so either assumption is acceptable as long as it is clear.

1. Not possible. Construct a synchronous execution in which we alternate between having all n processes take steps until each sends a message then having all n^2 messages delivered. The usual symmetry argument shows that each process updates to the same state and sends the same messages in each round, so either no process ever declares itself the leader, or they all do.

2. Not possible. Consider a system with two processes p_1 and p_2. Run p_1 but do not deliver any of its messages to p_2. Since this execution is indistinguishable from an execution in which p_1 is the only process, it must eventually set its leader bit. Now run p_2 without delivering any of its messages to or from p_1. It also must eventually set its leader bit. We can now satisfy admissibility by delivering all the undelivered messages, but it's too late: we already have two leaders.

3. Possible. Have each process broadcast its ID then wait to collect n IDs. The process with the smallest ID among these n IDs sets its leader bit. Message complexity is n^2 and time complexity is 1.

4. Possible. Have each process broadcast its ID. If a process receives its own ID before any others, it sets its leader bit. Since the broadcast channel is ordered, only the first process to do a broadcast wins. Message complexity is n^2 and time complexity is 1.

C.1.2 Discovery by flooding

In the usual message-passing model, it is assumed that every process has the ability to communicate directly only with its immediate neighbors in the communication graph. For this problem we will consider model closer to the current Internet, where (in principle) any machine in the network can send a message to any other machine, provided it knows the other machine's IP address.

For each process p_i, let S_i be the set of processes p_j such that p_i knows p_j's address, and let $G = (V, E)$ be the directed graph whose vertices V are all processes and which contains an edge $ij \in E$ for each pair p_i, p_j such that $p_j \in S_i$ in the initial configuration. Assume that p_i knows about itself, so that G includes all the self-loops ii.

We'd like the processes to exchange messages until this graph is complete, with an edge for every pair of processes. The protocol is simple: In each (synchronous) round, every process p_i sends its current list S_i to every process in S_i, then updates S_i to be the union of every message it receives.

Show that if the initial graph G is weakly-connected, then after at most $O(\log n)$ rounds, this protocol reaches a configuration where $S_i = V$ for all i.

Solution

For each r, let S_i^r be the value of S_i after r rounds of messages. Define $G^r = (V, E^r)$ as the graph where V is the set of processes and $ij \in E^r$ if and only if $p_j \in S_i$. From the definition we have $G^0 = G$.

It is convenient to work with undirected graphs. Let H^r be the *undirected* graph that contains an edge ij if and only if ij and ji are both edges in G^r. Note that H^r is always a subgraph of G^r.

Claim: H^1 is connected. Proof: For each edge $ij \in G^0$, p_i sends $p_i \in S_i$ to p_j, so p_j updates S_j^1 to include ji. So H^1 contains the undirected version of G^0 as a subgraph. Since G^0 is weakly connected, H^0 is connected.

Because H^1 is connected, there is a path in H^1 between any two nodes, and the diameter $d(H^1)$ of H^1 is at most $n - 1$. We now show that each round of the protocol reduces the diameter of H by roughly half.

Claim: If uv and vw are both edges in H^r, then uw is an edge in H^{r+1}. Proof: From the definition of H^r, we have $\{u, w\} \subseteq S_v^r$. So both of u and w add the other upon receiving S_v^r from v.

Now consider arbitrary $u, v \in H^r$ with $d(u, v) = m$. This means that there is a path $u = u_0 u_1 \dots u_m = v$ in H^r. From the claim, we have that $u = u_0 u_2 u_4 \dots u_m = v$ is a path in H^{r+1} if m is even, and $u =$

$u_0u_2u_4\dots u_{m-1}u_m = v$ is a path in H^{r+1} if m is odd. In either case we have $d_{H_{r+1}}(u,v) \le \lceil m/2 \rceil$. It follows that $d(H^{r+1}) = \max_{u,v} d_{H_{r+1}}(u,v) \le \max_{u,v} \lceil d_{H_r}(u,v)/2 \rceil \le \lceil d(H^r)/2 \rceil$.

A simple induction on r shows that if $d(H^1) \le 2^k$, then $d(H^r) \le \min(1, 2^{k-r+1})$. In particular for $r = \lceil \lg n \rceil + 1$ we have $d(H^r) \le 1$, which shows that there is an edge between every pair of nodes in H^r. Since H^r is defined to contain ij if and only if ij and ji are edges in G^r, it follows that G^r is complete for $r = \lceil \lg n \rceil + 1 = O(\log n)$.

C.2 Assignment 2: due Thursday 2022-10-06, at 23:59 Eastern US time

C.2.1 Maximum consensus

Suppose you have a synchronous message-passing system with n processes that may experience up to f crash failures. Each process p_i starts with an input x_i that is an arbitrarily-large natural number. What is the minimum number of rounds needed to solve each of the following problems in the worst case as a function of f? In each case, provide matching upper and lower bounds for sufficiently large n.

1. Each non-faulty process p_i outputs a value y_i such that (a) $y_i = x_j$ for some process p_j, and (b) $y_i \ge x_j$ for all non-faulty processes p_j.

2. As above, but in addition $y_i = y_j$ for all non-faulty processes i and j.

Solution

1. One round is enough. Each process sends x_i to all processes (including itself), and each process returns y_i equal to the largest of all x_j it received.

   Condition (a) follows immediately from y_i being equal to some x_j. For (b), if p_j is non-faulty, p_i receives x_j from p_j, so it returns either x_j or some larger $x_{j'}$.

   For the lower bound, if a protocol uses zero rounds, then no messages are sent. If process p_i decides x_i in some execution, then for $n \ge 2$ there exists an execution indistinguishable to p_i from this one, where some non-faulty p_j with $j \ne i$ has $x_j > x_i$, violating (b). Similarly, if p_i decides a value $y_i \ne x_i$, there exists an indistinguishable execution where no process has y_i as its input value, violating (a).

2. Here we need $f + 1$ rounds. For the lower bound we can reduce from synchronous consensus and apply Dolev-Strong ([DS83]; see also §9.3). To solve consensus using this problem, have each process p_i decide on y_i. This satisfies validity from (a) and agreement from the added condition that $y_i = y_j$ for all non-faulty i and j. So if we have an algorithm that uses less than $f + 1$ rounds, we get an algorithm for consensus that also uses less than $f + 1$ rounds, contradicting the known lower bound for consensus.

   For the upper bound, we can use the flooding mechanism from Dolev-Strong ([DS83]; see also §9.2). This guarantees that after $f + 1$ rounds, every non-faulty process obtains the same set S of input values, which includes the inputs of all non-faulty processes. So taking $\max S$ gives a common return value for all non-faulty processes that satisfies both (a) and (b).

C.2.2 Colorful Byzantine agreement

Consider a synchronous system with n processes, each of which is labeled with one of four colors: red, green, blue, or yellow. The processes have unique IDs that are known to all the other processes, and all processes know which processes have which color.

The adversary can turn as many processes as it likes Byzantine, provided that all the processes corrupted by the adversary are of the same color.

Prove or disprove: It is possible to solve Byzantine agreement in this system for any number of processes $n \geq 4$ using any assignment of colors that gives at least one process of each color.

Solution

Possible. The idea is to reduce the problem to four processes of which at most one is Byzantine, then use any Byzantine agreement algorithm that tolerates $f < n/3$ Byzantine faults to solve agreement. One possibility would be exponential information gathering [PSL80] (see §10.2.1), since we don't particularly care about anything but fault tolerance and 4 is a constant anyway.

For each color group, let the process with maximum ID represent the group (this does not require any rounds of communication under the assumption that all IDs are known to all processes). We then have four representatives that can execute EIG in $f + 1 = 2$ rounds to solve Byzantine agreement among themselves. Each representative then broadcasts its decision value to

all n processes, and each non-faulty process decides on the value broadcast by the majority of representatives. (Note that it is not enough for a process to follow its own representative, because there may be non-faulty processes within the faulty group.)

We would like to show that this algorithm solves Byzantine agreement for all n processes. Termination is immediate. For validity, if all non-faulty processes have the same input v, then so do the three non-faulty representatives; validity in the four-process protocols implies that all three non-faulty representatives broadcast this value and thus all non-faulty processes decide it. Agreement is similar: because all three non-faulty representatives agree on the same value v, each non-faulty process will see a majority for v and decide on v.

C.3 Assignment 3: due Thursday 2022-10-27, at 23:59 Eastern US time

C.3.1 A census of failure

Suppose we have an asynchronous message passing system with crash failures, and we want to implement an oracle that returns a count of the number of processes that haven't crashed yet. Define a **census protocol** to be a protocol that stores at every point in the execution a value c_i at each process p_i, such that (a) $c_i \geq n - f$ always, where n is the number of processes in the system and f is the number of processes that have crashed so far, and (b) once f converges to a fixed value, c_i eventually converges to $n - f$. These properties should hold for every non-crashed process p_i.

Prove or disprove each of the following statements. In each case assume that we have an asynchronous message-passing system with a complete communication graph, deterministic processes, and crash failures modeled as explicit crash events, and that any implementation must work for arbitrarily large n (which is known to the processes).

1. It is possible to implement a census protocol without using a failure detector.

2. It is possible to implement a census protocol using an eventually perfect ($\Diamond P$) failure detector.

3. It is possible to implement a census protocol using a perfect (P) failure detector.

Solution

1. Disproof: With no failure detector, consider two executions of a two-process system. In one execution, process p_1 takes no steps because it crashes immediately. In the other, p_1 takes no steps for a very long time.

   If p_2 eventually sets c_2 to 1, this violations $c_2 \geq n - f$ in the execution where p_1 has not crashed.

   If p_2 does not eventually set c_2 to 1, this violations c_2 converging to $n - f$ in the execution where p_1 has crashed.

2. Disproof: Consider the two executions in the previous case, and suppose that $\Diamond P$ correctly suspects p_1 throughout the crash execution and incorrectly suspects p_1 in the no-crash execution.

   If p_2 sets c_2 to 1, it violates (a) again in the no-crash execution, and afterwards we can both wake up p_1 and have $\Diamond P$ stop suspecting p_1.

   If p_2 doesn't set c_2 to 1, it violates (b) in the crash execution.

3. Proof: Recall that P eventually permanently suspects every crashed process and never suspects a process before it crashes. So have each process p_i set c_i to $n - f_i$, where f_i is the number of processes that p_i's instance of P currently suspects. Because P only suspects crashed processes, $f_i \leq f$ and thus $c_i = n - f_i \geq n - f$, satisfying (a). Because P eventually permanently suspects all crashed processes, once every process that will crash has crashed, P will eventually suspect all of them at each p_i. This gives $f_i = f$ and $c_i = n - f_i = n - f$.

C.3.2 Distributed shared memory with Byzantine servers

Consider the following modification to the usual asynchronous message-passing model:

1. There are m clients, and any of them may crash at any time.
2. There are n servers. These do not crash, but up to f of them may be Byzantine.

We would like to have a linearizable implementation of a single-writer multi-reader register in this model, where the single writer and multiple readers are all clients, and any operation by a non-faulty client eventually finishes. Show that there is a constant c such that this is possible for $n \geq cf + 1$.

Solution

We can do this when $n \geq 4f + 1$ by modifying ABD (see §17.2).

To make things easier, we will assume that the honest servers keep track of every timestamp-value pair $\langle t, v \rangle$ they have every received, instead of just the one with the maximum timestamp. Upon receiving a `read`(u) message, the server responds with its entire list (including $\langle t, v \rangle$ if it wasn't there already).

To perform a write operation with value v, the writer increments its local timestamp t, sends `write`(t, v) to all servers, and waits for $n - f$ acknowledgments.

To perform a read operation, a reader sends `read`(u) to all servers, waits for $n - f$ replies, and then chooses a pair $\langle t, v \rangle$ that (a) is sent by at least $f + 1$ servers, and (b) has the largest t out of all such pairs. If there is no pair sent by $f + 1$ servers, the reader returns the default initial register value $\perp$. Otherwise, it sends `write`(t, v) to all servers, waits for $n - f$ acknowledgments, then returns v.

To show this gives a linearizable implementation of a single-writer multi-reader register, we will largely follow the original proof for ABD, constructing an explicit linearization of any complete execution. We start with a simple invariant:

Lemma C.3.1. *Let $\langle t, v \rangle$ be a pair that is (a) in some honest server's list, (b) in a* `write`(t, v) *message, or (c) adopted by a reader. Then $\langle t, v \rangle$ was previously sent by the writer.*

Proof. It is easy to see that if (b) and (c) hold in some configuration, then (a) and (b) hold in any successor configuration, since we can only add a tuple to an honest server if it was in a `write`(t, v) message and we can only generate a `write`(t, v) message if $\langle t, v \rangle$ is sent by the writer or was previously adopted by a reader. To show that (c) holds, observe that if a reader adopts $\langle t, v \rangle$, it must first receive it from $f + 1$ servers. At least one of these servers is honest, so (a) applies. □

For any operation a, let $t(a)$ be the timestamp of the pair $\langle t, v \rangle$ that a sends in its `write`(t, v) messages. Observe that if a finishes, then it receives acknowledgements from $n - f$ servers of which at least $n - 2f$ are not faulty: this implies that by the time a finishes, at least $n - f$ servers have $\langle t, v \rangle$ in their lists. If b is a read operation with $a <_H b$, then b receives responses from at least $n - 3f$ of these servers. With $n \geq 4f + 1$, this is at least $f + 1$. So b either adopts $\langle t, v \rangle$ or adopts some other $\langle t', v' \rangle$ with $t' > t$. So whenever $a <_H b$, $t(a) \leq t(b)$.

To define $<_S$, a before b if (1) $t(a) < t(b)$ (which we've just shown is consistent with $<_H$); or (2) $t(a) = t(b)$, a is a write, and b is a read (which is consistent with $<_H$ by Lemma C.3.1); or (3) $t(a) = t(b)$, both operations are reads, and $a <_H b$ (definitely consistent with $<_H$!). Then extend the resulting partial order to a total order. As in the original ABD algorithm, we get a sequence of blocks of operations where all operations in a block have the same $\langle t, v\rangle$ pair, and the first operation in each block (except possibly the first block) is a write of v and the rest are reads that return v. So the resulting sequential execution is consistent both with H and the specification of a register, and we have shown that the implementation is linearizable.

C.4 Assignment 4: due Thursday 2022-11-10, at 23:59 Eastern US time

C.4.1 Arithmetic registers

An **arithmetic register** holds an integer value and supports operations `read`(), `add`(x), and `multiply`(x), where `read`() returns the current value of the register; `add`(x) updates the current value by adding x to it; and `multiply`(x) updates the current value by multiplying it by x. The `add` and `multiply` operations do not return a value.

Suppose that arithmetic registers come in two flavors: a **signed** arithmetic register can hold any integer value and allows any integer argument to `add` or `multiply`, while an **unsigned** arithmetic register holds only non-negative integer values and allows only non-negative integer arguments.

Prove or disprove: There exists a deterministic, wait-free, linearizable implementation of a signed arithmetic register from unsigned arithmetic registers and ordinary atomic registers.

Solution

Proof: We'll show that an unsigned arithmetic register implements consensus for any fixed number of processes n, then use universality of consensus to get an implementation of a signed arithmetic register.

The consensus construction follows a similar argument of Ellen *et al.* [EGSZ20] for registers supporting multiplication and decrement, but we have to be a little careful to only use non-negative values. Start with a single unsigned arithmetic register r initialized to 1. A process with input 0 applies `add`(1) to r. A process with input 1 applies `multiply`($n + 2$) to r, where n is the number of processes.

Consider some sequence of operations s, and let a_i be the number of calls to `add`(1) in s that are followed by exactly i calls to `multiply`$(n+2)$ in s. Let k be the total number of calls to `multiply`$(n+2)$ in s. Then it is easily shown by induction on the length of s that the value of the register at the end of s is given by $r = (a_k + 1)(n+2)^k + \sum_{i=0}^{k-1} a_i(n+2)^i$.

Since each coefficient in this expansion is at most $n+1$, we can recover the expansion uniquely from r. The value a_k will be nonzero if and only if the first operation on the register was `add`(1). Since this holds for any sequence of operations, any process reading the register can determine whether an adder or multiplier went first, and so all processes can return 0 in the first case and 1 in the second.

Now apply Herlihy's universal construction to implement a signed arithmetic register.

(With some tinkering, we can even drop the requirement for atomic registers by showing that they can be implemented from unsigned arithmetic registers, but this is not required by the problem.)

C.4.2 Counting to two

Let us say that we can count to k with m registers for n processes if there is a deterministic, wait-free, linearizable, one-shot implementation of a k-bounded counter from m registers that works for n processes. A k-bounded register starts at 0, has a read operation that returns its current value, and has an increment operation that increases the value by 1 unless it is already k. It is one-shot if each process is only allowed to call the increment operation at most once.

It is easy to show that we can count to 1 for any number of processes using 1 register: start with a 0 in the register, and implement an increment by writing 1. It is also straightforward to count to any value k for n processes using n registers: give a register to each process; implement increment by writing 1 to my register; and sum over a collect to get a number of increments s, returning $\min(k, s)$ to enforce k-boundedness.

Prove or disprove: We can count to 2 with 3 registers for 4 processes.

Solution

Proof: In fact, we can do this for any n, not just $n = 4$.

Use two of the three registers to build a splitter (Algorithm 18.6). The third register, initially 0, will be a flag indicating at least two increments.

To do an increment: Try to win the splitter. If I win, I am done. If not, write 1 to the flag.

To do a read: Check door. If it's open, assume no increments have finished yet and return 0. If it's closed, use the flag to decide whether to return 1 or 2.

Code is given in Algorithm C.1.

```
   shared data:
 1 atomic register race, big enough to hold an ID, initially ⊥
 2 atomic register door, big enough to hold a bit, initially open
 3 atomic register flag, big enough to hold a bit, initially 0
 4 procedure increment(id)
 5     race ← id
 6     if door = closed then
 7         flag ← 1
 8     door ← closed
 9     if race ≠ id then
10         flag ← 1
11 procedure read
12     if door = open then
13         return 0
14     else if flag = 0 do
15         return 1
16     else
17         return 2
```

Algorithm C.1: Counting to 2 with a splitter

Since each operation does at most a constant number of steps, this is clearly wait-free. But we need to show that it is linearizable. We'll use linearization points.

For a read that returns 0: linearize it at the point where it reads door and sees open.

For any other read: linearize it at the point where it reads flag.

This orders all reads that return 0 (when the door is still open) before all reads that return 1 or 2; and orders all reads that return 1 (when the flag is not yet set) before all reads that return 2. So now we just need to fit in some increments to justify the changes.

If there is an increment I_1 that wins the splitter and does not set the

flag, assign its linearization point to the step where the door closes (whether I_1 closes the door or not). Then I_1 linearizes between all reads that return 0 and all reads that return 1 or 2. Because every other increment loses the splitter, every other increment sets the flag; make each such increment's linearization point be the step where it sets the flag. The first such increment I_2 linearizes between all reads that return 0 or 1 and all reads that return 2.

If no increment wins the splitter, then no increment finishes before setting the flag, at the point where the flag is first set there are at least two increments in progress and none have already finished. Let I_1 be one of these increments that starts before the door closes, and assign its linearization point to the step where the door closes. Let I_2 be any other increment in progress when the flag is first set, and assign its linearization point to when the flag is first set. Assign the linearization points of any other increment anywhere during its execution interval that is after I_2's. Again we get one increment linearized between the 0 and 1 reads, and at least one between the 1 and 2 reads. We are done.

C.5 Assignment 5: due Monday 2022-12-05, at 23:59 Eastern US time

C.5.1 A hidden counter

Consider a system with n processes; n single-writer multi-reader atomic registers, one for each process; and a counter that can be incremented by any process but that can be read by nobody. We would like a wait-free protocol that results in the counter being incremented by at least $f(n)$ using as few total operations, across all processes, as possible, counting both increment operations on the counter and read and write operations on the registers.

In this context, wait-freedom means that a process can only return when it is sure that $f(n)$ increments have been done, which may, in the worst case, require it to do all $f(n)$ increments by itself. A process that returns is scheduled for no more operations.

1. Show that $O(n^2)$ total operations are sufficient to increment the counter at least n^2 times.

2. Show that $T(n)$ total operations are sufficient to increment the counter at least n times, for some $T(n) = o(n^2)$.

Solution

1. We'll have each process i alternate between incrementing the counter and writing out the total number of increments it has done so far to its register r_i.

   After n increments, the process will read all the registers r_j, and if $\sum r_j \geq n^2$, return.

   This gives an amortized cost of 3 operations per increment, so as long as we only do $O(n^2)$ increments, we are fine. To show this, observe that once the total value in the registers exceeds n^2, each process does at most n increments before it re-reads the registers, for at most n^2 extra increments.

2. There are a number of ways to do this. One simple approach is to divide the processes into groups of size $k = \sqrt{n}$, and have each group independently do at least $n = k^2$ increments using the algorithm from the previous case. This costs $O(n)$ operations per group, or $O(n^{3/2})$ operations total.

C.5.2 One register to rule them all

This problem was nearly identical to Problem D.5.1 from 2020 and has been withdrawn. Any submission for this assignment will be graded as if a complete solution to this problem had been provided.

Appendix D

Sample assignments from Spring 2020

D.1 Assignment 1: due Wednesday, 2020-09-23, at 5:00pm Eastern US time

D.1.1 A token-passing game

Suppose we have an asynchronous bidirectional message-passing network in the form of a connected graph, where initially m of the n nodes possess a token, represented by a local variable hasToken being set to true. We'd like to be able to move the tokens around, while preserving the total number of tokens.

1. Show that no algorithm that allows tokens to move can guarantee that there are exactly m tokens in any reachable configuration.

2. Give an algorithm that satisfies the following two properties, starting with a configuration with m tokens:

   (a) *Safety:* In any reachable configuration, there are at most m tokens. You should give an explicit invariant that implies this, and show that any transition of your algorithm preserves the invariant.

   (b) *Liveness:* From any reachable configuration C_0, for any subset S of the processes with $|S| = m$, there exists an execution starting in C_0 that ends with a configuration in which every process in S has a token.[1]

[1] Strictly speaking, this is a lot weaker than the usual definition of liveness, because it

To keep things simple, you may assume that the processes can make non-deterministic choices. For example, a process p might choose arbitrarily between sending a message to a neighbor q or to a different neighbor r, and each choice leads to a different possible execution.

Solution

1. Suppose that we are preserving total tokens. Consider some transition between configurations C_1 and C_2. If some process switches hasToken from 1 to 0 between these configurations, then some other process must switch hasToken from 0 to 1. But the definition of delivery events in the asynchronous message-passing model only allows one process at a time to change its state. It follows that no process can change hasToken from 1 to 0 in any transition, so tokens can't move.

2. Consider the following algorithm:

   - At any time, a process with hasToken $= 1$ may send a message takeThis to any of its neighbors and set hasToken $= 0$.
   - A process that receives takeThis when hasToken $= 1$ sends takeThis to any of its neighbors. A process that receives takeThis when hasToken $= 0$ may either set hasToken $= 1$ or send takeThis to any of its neighbors.

   Let us show that this has the desired properties:

   (a) *Safety:* Our invariant will be that the sum of the number of processes with hasToken $= 1$ plus the number of takeThis messages in transit will be m.
   The invariant holds in the initial configuration because there are exactly m processes with hasToken $= 1$ and no message in transit.
   It is preserved by transitions, because in each possible transition, either:
      i. Some process changes hasToken $= 1$ to hasToken $= 0$ and generates a takeThis message;
      ii. Some process changes hasToken $= 0$ to hasToken $= 1$ while consuming a takeThis message; or

effectively assumes that the adversary is cooperating with us. In retrospect I should have written this as "for any admissible adversary strategy, there is a sequence of nondeterministic choices by the algorithm that causes the execution to reach a desired configuration." But I didn't write this, and so it's fine to answer the problem I did write.

iii. Some process consumes a takeThis message but generates a new takeThis message.

In each case, the total number of tokens plus messages is preserved.

(b) *Liveness:* For any configuration C, let $T(C)$ be the set of processes with hasToken $= 1$. We will argue that if $T(C) \neq S$, there exists a partial execution that increases $|T(C) \cap S|$ by 1. First pick some $p \in S \setminus T(C)$. Now consider two cases:

i. If there is at least one takeThis message t in transit, apply the following strategy. Deliver t. If the recipient of t is p, set p.hasToken $= 1$. If not, have the recipient send takeThis to some neighbor that is closer to p than it is. Repeat this process until a takeThis message reaches p.

ii. If there is no takeThis message in transit, generate a takeThis message at some $q \in T(C) \setminus S$ while setting q.hasToken to 0. Then apply the previous case.

For any configuration with $T(C) \neq S$, at least one of these two conditions will hold because of the safety property.
Each partial execution defined above increases $|T(C) \cap S|$ by one, and we can only increase $|T(C) \cap S|$ at most m times because $|S| = m$, so after at most m of these partial executions we reach a configuration with $T(C) = S$.

D.1.2 A load-balancing problem

Consider a two-way message-passing ring with $n = mk$ nodes, where $m > 1$ and k is odd. Nodes at positions $0, k, 2k, \ldots, (m-1)k$ are initially marked as leaders, while nodes at other positions are followers. All nodes have a sense of direction, and can distinguish their left neighbor from their right, but they do not have any other ID information.

Algorithm D.1 is intended to allow the leaders to recruit followers. It is not hard to show that every follower eventually adds itself to a tree of parent pointers rooted at some leader. We would like all of these trees to contain roughly the same number of nodes.

1. Suppose we run this algorithm in a synchronous system. What is the minimum and maximum possible size of a tree?

2. Suppose instead we run the algorithm in an asynchronous system. Now what is the minimum and maximum possible size of a tree?

```
1  initially do
2      if I am a leader then
3          parent ← id
4          send recruit to both neighbors
5      else
6          parent ← ⊥
7  upon receiving recruit from p do
8      if parent = ⊥ then
9          parent ← p
10         send recruit to my neighbor who is not p
```

Algorithm D.1: Recruiting algorithm for Problem D.1.2.

3. Give an algorithm for the asynchronous version of this model that guarantees that all trees are the same size.

Solution

1. In a synchronous execution, we can prove by induction that for each t with $0 \leq t \leq \frac{k-1}{2}$, and each $0 \leq i \leq m-1$, each node at position $ik \pm t$ joins the tree rooted at ik at time t. This puts exactly k nodes in each tree.

2. In an asynchronous execution, by rushing messages from 0, we can recruit all nodes in the range $[-k+1, k-1]$ to the 0 tree before any other messages are delivered. Conversely, each leader ik can't recruit nodes $(i-1)k$ or $(i+1)k$, because these are leaders. So the maximum size of any tree is $2k-1$.

   For the minimum size, suppose we rush all messages from nodes k and $(m-1)k$. Then nodes 1 and $m-1$ are recruited into the trees rooted at these nodes before either message from 0 is delivered. This shows that there are executions with a minimum tree size of 1.

3. The easiest fix may be to have each leader initially send just one recruit message to the right. For each i, this recruits all agents $ik, \ldots, ik+(k-1)$ to a tree of size k rooted at ik.

D.2 Assignment 2: due Wednesday, 2020-10-07, at 5:00pm Eastern US time

D.2.1 Synchronous agreement with limited broadcast

Suppose that we are given a synchronous message passing system on a complete network in which messages are replaced by k-way broadcasts, where the recipient is replaced by a recipient list of up to k processes. Suppose further that when a process crashes in round r, each of its round-r messages is either delivered to all of the processes on the message's recipient list or to none of them. A process can send as many messages as it likes to as many groups as it likes, but if it crashes in some round, any subset of the messages sent in that round may be lost.

Show that for it is possible to solve agreement in this model in $O(f/k)$ rounds, assuming $n > f$.

Solution

We'll use the flooding algorithm of Dolev and Strong [DS83] (see §9.2), but replace sending S to all n processes in each round with sending S to all $\binom{n}{k}$ possible recipient lists. As in the original algorithm, we want to prove that after some round with few enough failures, all the non-faulty processes have the same set.

Let S_i^r be the set stored by process i after r rounds. Suppose there is some round $r+1$ in which fewer than k processes fail. Then every recipient list in round r includes a process that does not fail in round $r+1$. Let L be the set of processes that successfully deliver a message to at least one recipient list in round r, and let $S = \cup_{i \in L} S_i^r$. Then for each value $v \in S$, there is some process that receives v during round r, does not crash in round $r+1$, and so retransmits v to all processes in round $r+1$, causing it to be added to S_i^{r+2}. On the other hand, for any $v \notin S$, v is not transmitted to any recipient list in round r, which means that no non-faulty process i includes v in S_i^{r+1}. So $S \subseteq S_i^{r+2} \subseteq \cup_j S_j^{r+1} \subseteq S$ for all i, and the usual induction argument shows that $S_i^{r'}$ continues to equal S for all non-faulty i and all $r' \geq r+2$.

We can have at most $\lfloor f/k \rfloor$ rounds with $\geq k$ crashes before we run out, so the latest possible round in which we have fewer than k crashes is is $r = \lfloor f/k \rfloor + 1$, giving agreement after $\lfloor f/k \rfloor + 2$ rounds (since we don't need to send any messages in round $r+2$).

(With some tinkering, it is not too hard to adapt the Dolev-Strong lower

bound to get a $\lfloor f/k \rfloor + 1$ lower bound for this model. The main issue is now we have to crash k processes fully in round $r + 1$ before we can remove one outgoing broadcast from a process in round r, which means we need to budget tk failures to break a t-round protocol. The details are otherwise pretty much the same as described in §9.3.)

D.2.2 Asynchronous agreement with limited failures

Algorithm D.2 describes an algorithm for asynchronous agreement with f crash failures in a fully-connected message-passing network. The idea is to collect values from $n - f$ other processes in each of m rounds, and then decide on the smallest value collected.

```
1 preference ← input
2 for i ← 1 to m do
3 |   send ⟨i, preference⟩ to all processes
4 |   wait to receive ⟨i, v⟩ from n − f processes
5 |   for each ⟨i, v⟩ received do
6 |   |   preference ← min(preference, v)
7 decide preference
```

Algorithm D.2: Candidate algorithm for asynchronous agreement

The value m is a parameter of the algorithm and may depend on n and f.

As usual, when waiting for messages from round i, any messages delivered for with other round numbers will be buffered internally and processed when the algorithm is ready for them.

Note that when a process sends a message to all process, that includes itself.

Show that, for any n and $0 < f < n/2$, there exists a value of m such that Algorithm D.2 satisfies agreement, termination, and validity; or show how to construct an execution for any n, $0 < f < n/2$, and m that causes Algorithm D.2 to fail at least one of these requirements.

Solution

We know from the FLP bound ([FLP85], Chapter 11) that Algorithm D.2 can't work. So the only question is how to find an execution that shows it doesn't work.

It's not too hard to see that Algorithm D.2 satisfies both termination and validity. So we need to find a problem with agreement.

The easiest way I can see to do this is to pick a patsy process p and give it input 0, while giving all the other processes input 1. Now run Algorithm D.2 while delaying all outgoing messages $\langle i, v \rangle$ from p until after the receiver has finished the protocol. Because each other process is waiting for $n - f \leq n - 1$ messages, this will not prevent the other processes from finishing. But all the other processes have input 1, so we have an invariant that messages in transit from processes other than p and preferences of processes other than p will be 1 that holds as long as no messages from p are delivered. This results in the non-p processes all deciding 1. We can then run p to completion, at which point it will decide 0.

D.3 Assignment 3: due Wednesday, 2020-10-21, at 5:00pm Eastern US time

D.3.1 Too many Byzantine processes

The phase king algorithm (Algorithm 10.2) described in §10.2.2 solves Byzantine agreement for $f < n/4$ processes. For larger values of f, it may fail by violating one or more of the properties of termination, validity, or agreement.

For this algorithm:

1. How big does f need to be to prevent termination?
2. How big does f need to be to prevent validity?
3. How big does f need to be to prevent agreement?

Assume that the processes know the new bound on f, and any thresholds in the algorithm that use f are adjusted to correspond to this new bound.

Solution

1. Termination: The algorithm always terminates in $f + 1$ synchronous rounds, so f doesn't matter.
2. Validity: To violate validity, we need to convince some non-faulty process to decide on the wrong value when all non-faulty processes have the same input.

   Suppose all the non-faulty processes have input 0, and we want to introduce a 1 somewhere. Each process updates its preference in

each round to be either the majority value it sees, if this value has multiplicity greater than $n/2 + f$, or the kingMajority broadcast by the phase king otherwise.

If $f < n/2$, it's going to be hard to show a process a bogus majority. But a Byzantine phase king gives us more options. Suppose that all the f Byzantine processes send out 1 in all rounds. Then for $f \geq n/4$, the multiplicity of the correct value 0 will be $n - f \leq (3/4)n$, while the required multiplicity to ignore the phase king will be strictly greater than $n/2 + f \geq (3/4)n$. So at $f = n/4$, all non-faulty processes adopt the phase king's bad value 1. In any subsequent round, we can just run the algorithm with the Byzantine agents pretending to be non-faulty processes with preference 1, and eventually all processes incorrectly decide 1.

3. Agreement: Now we need to get two non-faulty processes to decide different values. Wait to the last round, and use $f = n/4$ Byzantine processes to prevent the non-faulty processes from seeing a high enough multiplicity on any majority value to accept it, and use a Byzantine phase king to transmit different kingMajority values to different non-faulty processes. So again, the algorithm fails at $f = n/4$.

D.3.2 Committee election

Consider the following **committee election** problem in an asynchronous message-passing system with $f < n/2$ crash failures. Each process runs a committee election protocol, at the end of which it receives a value 1 (elected) or 0 (not elected). The requirements of the protocol are:

1. Nonempty committee: If no processes fail, at least one process receives 1.

2. No latecomers: In any execution, if some process p finishes the protocol before another process q starts the protocol, then q receives 0.

Give an algorithm that solves this problem, and show that it satisfies these requirements.

(For the purpose of defining when a process starts or ends the protocol, imagine that it uses explicit invoke and respond events. Your protocol should have the property that all non-faulty processes eventually terminate.)

Solution

The easiest way to do this may be to use ABD (see §17.2). Algorithm D.3 has each process read the simulated register, which we assume is initialized to 1, then write a 0 before returning the value it read.

```
1 Let r be an ABD register initialized to 1.
2 procedure elect
3     onCommittee ← r
4     r ← 0
5     return onCommittee
```

Algorithm D.3: Committee election using ABD

This satisfies nonempty committee, because the first operation in the linearization of the register must be a read operation that returns 1. It satisfies no latecomers, because if p finishes before q starts, then p's write finishes before q's read starts, and linearizability of ABD implies q reads a 0.

This takes 3 round-trips to finish (2 for the ABD read and 1 for the ABD write). It is not too hard to reduce this to 2 round-trips by replacing the embedded write in the ABD read operation with a write of 1, but this requires a more detailed correctness argument.

D.4 Assignment 4: due Wednesday, 2020-11-04, at 5:00pm Eastern US time

D.4.1 Counting without snapshots

Algorithm D.4 gives a wait-free implementation of a generalized counter using a collect. The `inc`(v) procedure adjusts the value of the counter by v: if it was x before `inc`(v), it should be $x + v$ after. The `read` procedure returns the current value of the counter. Assume that the initial value of the counter is 0, as are the initial values of the registers $A[i]$ that implement it.

This counter implementation is not linearizable in all executions, but it may be linearizable if we restrict the allowed values v that can be supplied as arguments to an `inc` operations. For each of the following sets V, show that any execution in which all increments are elements of V is linearizable, or show that there exists an execution with increments in V that is not.

1. $V = \{0, 1\}$.

```
1 procedure inc(v)
2     A[i] ← A[i] + v
3 procedure read()
4     s ← 0
5     for j ← 1 to n do
6         s ← s + A[j]
7     return s
```

Algorithm D.4: An alleged counter. Code for process i.

2. $V = \{-1, 1\}$.

3. $V = \{1, 2\}$.

Solution

1. The $\{0, 1\}$ case is linearizable. Given an execution S of Algorithm D.4, we assign to a linearization point to each `inc` operation at the step where it writes to A, and assign a linearization point to each `read` operation ρ that returns s at the later of the first step that leaves $\sum_j A[j] = s$ or the first step of ρ. Since this may assign the same linearization point to some write operation π and one or more read operations $\rho_1, \ldots, \rho_k$, when this occurs, we order the write before the reads and the reads arbitrarily.

   Observe that:

   (a) The value of each $A[j]$ individually is non-decreasing over time, and increases by at most one at each step.

   (b) The same holds for $\sum_{j=1}^{n} A[j]$.

   These are easily shown by induction on the steps of the execution, since each `inc` operation only changes at most one $A[j]$ and only changes it by increasing it by 1.

   The first condition implies that the value v_j of $A[j]$ used by a particular `read` operation ρ lies somewhere between the minimum and maximum values of $A[j]$ during the operation's interval, which implies the same about the total $\sum_j A[j]$. In particular, if ρ returns s the value of $\sum_j A[j]$ is no greater than s, and it reaches s no later than the end of ρ.

Because $\sum_j A[j]$ increases by at most one per step, this means that either $\sum_j A[j] = s$ at the first step of ρ, or $\sum_j A[j] = s$ at some step within the execution interval of ρ. In either case, ρ is assigned an execution point within its interval that follows exactly s non-trivial increments. This means that the return values of all `read` operations are consistent with a sequential generalized counter execution, and because both `read` and `inc` operations are ordered consistently with the execution ordering in S, we have a linearization of S.

2. For increments in $\{-1, 1\}$, there are executions of Algorithm D.4 that are not linearizable. We will construct a specific bad execution for $n = 3$. Let p_1 perform `inc`(1) and p_2 perform `inc`(2), where p_1 finishes its operation before p_2 starts. Because the `inc`(1) must be linearized before the `inc`(-1), the values of the counter in any linearization will be $0, 1, 0$ in this order.

   Now add a `read` operation by p_3 that is concurrent with both `inc` operations. Suppose that in the execution, the follow operations are performed on the registers $A[1]$ through $A[3]$:

   (a) p_3 reads 0 from $A[1]$.
   (b) p_1 writes 1 to $A[1]$.
   (c) p_2 writes -1 to $A[2]$.
   (d) p_3 reads -1 from $A[2]$.
   (e) p_3 reads 0 from $A[3]$.

   Now p_3 returns -1. There is no point in the sequential execution at which this is the correct return value, so there is no linearization of this execution.

3. For increments in $\{1, 2\}$, essentially the same counterexample works. Here we let p_1 do `inc`(1) and p_2 do `inc`(2), while p_3 again does a concurrent read. The bad execution is:

   (a) p_3 reads 0 from $A[1]$.
   (b) p_1 writes 1 to $A[1]$.
   (c) p_2 writes 2 to $A[2]$.
   (d) p_3 reads 2 from $A[2]$.
   (e) p_3 reads 0 from $A[3]$.

   Now p_3 returns 2, but in any linearization of the two write operations, the values in the counter are $0, 1, 3$.

D.4.2 Rock-paper-scissors

Define a **rock-paper-scissors object** as having three states 0 (rock), 1 (paper), and 2 (scissors), with a `read` operation that returns the current state and a `play`(v) operation for $v \in \{0, 1, 2\}$ that changes the state from s to v if $v = (s + 1) \pmod 3$ and has no effect otherwise.

Prove or disprove: There exists a deterministic wait-free linearizable implementation of a rock-paper-scissors object from atomic registers.

Solution

Proof: We will show how to implement a rock-paper-scissors object using an unbounded max register, which can be built from atomic registers using snapshots. The idea is to store a value v such that $v \bmod 3$ gives the value of the rock-paper-scissors object. Pseudocode for both operations is given in Algorithm D.5.

```
1 Let m be a shared max register.
2 procedure play(v)
3     s ← m
4     if v = ((s + 1) mod 3) then
5         m ← s + 1
6 procedure read()
7     return (m mod 3)
```

Algorithm D.5: Implementation of a rock-paper-scissors object

Linearize each `play` operation that does not write m at the step at which it reads m.

Linearize each `play` operation that writes $s + 1$ to m at the first step at which $m \geq s + 1$. If this produces ties, break first in order of increasing $s + 1$ and then arbitrarily. Since each such operation has $m \leq s$ when the operation starts and $m \geq s + 1$ when it finishes, these linearization points fit within the intervals of their operations.

Linearize each `read`() operation at the step where it reads m.

Since each of these linearization points is within the corresponding operation's interval, this preserves the observed execution ordering.

Observe that the `play` operations that write are linearized in order of increasing values written, and there are no gaps in this sequence because no process writes $s + 1$ without first seeing s. (This actually shows there is

no to break ties by value.) So the sequence of values in the max register, taken mod 3, iterates through the values $0, 1, 2, 0, \ldots$ in sequence, with each value equal mod 3 to some argument to a `play` operation. So we can take these values mod 3 as the actual value of the register for the purposes of `read` operations, meaning the `read` operations all return correct values. The `play` operations that don't write are linearized at a point where they would have no effect on the state of the rock-paper-scissors object, which is also consistent with the sequential specification.

It follows that Algorithm D.5 is a linearizable implementation of a rock-paper-scissors object from max registers. It is also wait-free, since each operation is implemented using a constant number of max-register operations. By implementing max registers using snapshots, we get a wait-free linearizable implementation from atomic registers.

D.5 Assignment 5: due Wednesday, 2020-11-18, at 5:00pm Eastern US time

D.5.1 Randomized consensus with one max register

Prove or disprove: A single max register, with no other objects, is sufficient to solve randomized wait-free binary consensus for two processes against an oblivious adversary.

Solution

We'll disprove it.

Let p_0 and p_1 be the two processes. The idea is to consider, for each $i \in \{0, 1\}$ some nonzero-probability solo terminating execution ξ_i of p_i with input i, then show that ξ_0 and ξ_1 can be interleaved to form a two-process execution ξ that is indistinguishable by each p_i from ξ_i.

The oblivious adversary will simply choose to schedule the processes for ξ. Since the processes flip a finite number of coins in this execution, there is a nonzero chance that the adversary gets lucky and they flip their coins exactly the right way.

Fix ξ_0 and ξ_1 as above. Partition each ξ_i as $\alpha_i\beta_{i1}\beta_{i2}\ldots\beta_{ik_i}$ where α_i contains only read operations and each β_{ij} starts with a write operation of a value v_{ij} strictly larger than any previous write operation.

Let $\xi = \alpha_0\alpha_1\beta_{i_1j_1}\beta_{i_2j_2}\ldots\beta_{i_kj_k}$ where $k = k_0 + k_1$ and the blocks $\beta_{i_\ell j_\ell}$ are the blocks $\{\beta_{0j}\}$ and $\{\beta_{1j}\}$ sorted in order of non-decreasing v_{ij}. Then each block $\beta_{i_\ell j_\ell}$ in ξ starts with a write of a value no smaller than the previous

value in the max register, causing each read operation within the block to return the value of this write, just as in the solo execution ξ_{i_ℓ}. Assuming both processes flip their coins as in the solo executions, they both perform the same operations and return the same values. These values will either violate agreement in ξ or validity in at least one of ξ_0 or ξ_1.

D.5.2 A plurality object

Consider a shared-memory object with operations `vote`(v) and `winner`(), where `winner`() returns the value v that appeared in the largest number of previous `vote` operations, or $\perp$ if there is no such unique v. For example, in a sequential execution with votes $a, b, b, c, c, c, a, a, a$, the value returned by a `winner` operation following each vote will be $a, \perp, b, b, \perp, c, c, \perp, a$.

Pick one of these statements, and show that it is true:

1. There is a deterministic wait-free linearizable implementation of this object for n processes that uses $o(n)$ registers.

2. There is such an implementation that uses $O(n)$ registers, but not $o(n)$ registers.

3. There is no such implementation using $O(n)$ registers.

Solution

Case (2) holds.

To implement the object, use a snapshot array to hold the total votes from each process, and have the `winner` operation take a snapshot, add up all the votes and return the correct result. This can be done using n registers.

To show that it can't be done with $o(n)$ registers, use the JTT bound (see Chapter 21). We need to argue that the object is perturbable. Let $\Lambda\Sigma\pi$ be an execution that needs to be perturbed, and let m be the maximum number of `vote`(v) operations that start in Λ for any value v. Then a sequence γ of $m+1$ votes for some v' that does not appear in Λ will leave the object with v' as the plurality value, no matter how the remaining operations are linearized. Since v' did not previously appear in Λ, this gives a different return value for π in $\Lambda\gamma\Sigma\pi$ from $\Lambda\Sigma\pi$ as required. The JTT bound now implies that any implementation of the object requires at least $n-1$ registers.

Appendix E

Sample assignments from Spring 2019

E.1 Assignment 1: due Wednesday, 2019-02-13, at 5:00pm

E.1.1 A message-passing bureaucracy

Alice and Bob are communicating with each other by alternately exchanging messages. But Bob finds Alice's messages alarming, and whenever he responds to Alice, he also forwards a copy of Alice's message to his good friend Charlie 1, a secret policeman. Charlie 1 reports to Charlie 2, but following the rule that "once is happenstance, twice is coincidence, the third time it's enemy action," [Fle59] Charlie 1 only sends a report to Charlie 2 after receiving three messages from Bob. Similarly, Charlie 2 only sends a message to his supervisor Charlie 3 after receiving three messages from Charlie 2, and so on up until the ultimate boss Charlie n. Pseudocode for each participant is given in Algorithm E.1.

Assuming we are in a standard asynchronous message-passing system, that Alice sends her first message at time 0, and that the protocol finishes as soon as Charlie n receives a message, what is the worst-case time and message complexity of this protocol as a function of n?

Solution

Time complexity Observe that Alice sends at least k messages by time $2k - 2$. This is easily shown by induction on k, because Alice sends at least 1 message by time 0, and if Alice has sent at least $k - 1$ message by time

```
1  Alice:
2  initially do
3      send message to Bob
4  upon receiving message from Bob do
5      send message to Bob
6  Bob:
7  upon receiving message from Alice do
8      send message to Alice
9      send message to Charlie 1
10 Charlie i, for i < n:
11 initially do
12     c ← 0
13 upon receiving message from Bob or Charlie i − 1 do
14     c ← c + 1
15     if c = 3 then
16         c ← 0
17         send message to Charlie i + 1
```

Algorithm E.1: Reporting Alice's alarming messages

$2k-4$, the last of these is received by Bob no later than time $2k-3$, and Bob's response is received by Alice no later than time $2k-2$.

Because each message from Alice prompts a message from Bob at most one time unit later, this implies that Bob sends at least k messages by time $2k-1$.

Write $T_0(k) = 2k-1$ for the maximum time for Bob to send k messages. Write $T_i(k)$ for the maximum time for Charlie i to send k messages, for each $0 < i < n$. In order for Charlie i to send k messages, it must receive $3k$ messages from Bob or Charlie $i-1$ as appropriate. These messages are sent no later than $T_{i-1}(3k)$, and the last of them is received no later than $T_{i-1}(3k)+1$. So we have the recurrence

$$T_i(k) = T_{i-1}(3k) + 1$$
$$T_0(k) = 2k - 1$$

with the exact solution

$$T_i(k) = (2 \cdot 3^i \cdot k - 1) + k.$$

For $i = n-1$ and $k = 1$, this is $2 \cdot 3^{n-1} - 1 + n - 1 = 2 \cdot 3^{n-1} + n = O(3^n)$. We can get the exact time to finish by adding one more unit to account for the delay in delivering the message from Charlie $n-1$ to Charlie n. This gives $2 \cdot 3^{n-1} + n + 1$ time exactly in the worst case, or $O(3^n)$ if we want an asymptotic bound.

Message complexity Message complexity is easier: there is no bound on the number of messages that may be sent before Charlie n receives his first message. This is because in an asynchronous system, Alice and Bob can send an unbounded (though finite) number of messages to each other even before Bob's first message to Charlie 0 is delivered, without violating fairness.

E.1.2 Algorithms on rings

In Chapter 5, we saw several leader election algorithms for rings. But nobody builds rings. However, it may be that an algorithm for a ring can be adapted to other network structures.

1. Suppose you have a network in the form of a d-dimensional hypercube Q^d. This means we have $n = 2^d$ nodes, where each node is labeled by a d-bit coordinate vector, and two nodes are adjacent if their vectors

differ in exactly one coordinate. We also assume that each node knows its own coordinate vector and those of its neighbors.

Show that any algorithm for an asynchronous ring can be adapted to an asynchronous d-dimensional hypercube with no increase in its time or message complexity.

2. What difficulties arise if we try to generalize this to an arbitrary graph G?

Solution

1. The idea is to embed the ring in the hypercube, so that each node is given a clockwise and counterclockwise neighbors, and any time the ring algorithm asks to send a message clockwise or counterclockwise, we send to the appropriate neighbor in the hypercube. We can then argue that for any execution of the hypercube algorithm there is a corresponding execution of the ring algorithm and vice versa; this implies that the worst-case time and message-complexity in the hypercube is the same as in the ring.

   It remains only to construct an embedding. For $d = 0$, $d = 1$, and $d = 2$, the ring and hypercube are the same graph, so it's easy. For larger d, split the hypercube into two subcubes Q^{d-1}, consisting of nodes with coordinate vectors of the form $0x$ and $1x$. Use the previously constructed embedding for $d - 1$ to embed a ring on each subcube, using the same embedding for both. Pick a pair of matching edges $(0x, 0y)$ and $(1x, 1y)$ and remove them, replacing them with $(0x, 1x)$ and $(0y, 1y)$. We have now constructed an undirected Hamiltonian cycle on Q^d. Orient the edges to get a directed cycle, and we're done.

2. There are a several problems that may come up:

   (a) Maybe G is not Hamiltonian.

   (b) Even if G is Hamiltonian, finding an Hamiltonian cycle in an arbitrary graph is **NP**-hard. This could be trouble for a practical algorithm.

   (c) Even if we can find a Hamiltonian cycle for G (maybe because G is a nice graph of some kind, or maybe by taking advantage of the unbounded computational power of processes assumed in the standard message-passing model), the processes don't necessarily know what G looks like at the start. So they would need some

initial start-up cost to map the graph, adding to the time and message complexity of the ring algorithm.

E.1.3 Shutting down

Suppose we want to be able to stop a running protocol in an asynchronous message-passing system prematurely. Define a **shutdown mechanism** as a modification to an existing protocol in which any process can nondeterministically issue a `stop` order that eventually causes all processes to stop sending messages. We would like such a shutdown mechanism to satisfy two properties:

1. **Termination.** If some process issues a `stop` order at time t, no process sends a message at time $t + \Delta$ or later, for some finite bound Δ that may depend on the structure of the network.

2. **Non-interference.** If no process issues a `stop` order, the protocol carries out an execution identical to some execution of the underlying protocol without a shutdown mechanism.

Show how to implement a shutdown mechanism, and prove tight upper and lower bounds on Δ as a function of the structure of the network.

Solution

This is pretty much the same as a Chandy-Lamport snapshot [CL85], as described in §6.3. The main difference is that instead of recording its state upon receiving a `stop` message, a process shuts down the underlying protocol. Pseudocode is given in Algorithm E.2. We assume that the initial `stop` order takes the form of a `stop` message delivered by a process to itself.

```
1 initially do
2     stopped ← false
3 upon receiving stop do
4     if ¬stopped then
5         stopped ← true
6         send stop to all neighbors
7         replace all events in underlying protocol with no-ops
```

Algorithm E.2: Shutdown mechanism based on Chandy-Lamport

An easy induction argument shows that if p receives a `stop` message at time t, then any process q at distance d from p receives a `stop` message no later than time $t + d$. It may be that q sends `stop` messages in responds to this `stop` message, but these are the last messages q ever sends. It follows that no process sends a message later than time $t + D$, where D is the diameter of the graph. This gives an upper bound on Δ.

For the lower bound, we can apply an indistinguishability argument. Let p and q be processes at distance D from each other, and suppose that the underlying protocol involves processes sending messages to their neighbors at every opportunity. Consider two synchronous executions: X, an execution in which no `stop` order is ever issued, and X_t, an execution in which p delivers a `stop` message to itself at time t.

We can show by induction on d that any process r at distance d from p carries out the same steps in both X and X_t up until time $t+d-1$. The base case is when $d = 0$, and we are simply restating that p runs the underlying protocol before time t. For the induction step, we observe for any time $t' < t + d - 1$, any message sent to r from some neighbor s was sent at time $t' - 1 < t + d - 2$, and since $d(p, s) \geq d - 1$, the induction hypothesis gives that s sends the same messages at $t' - 1$ in both X and X_t.

It follows that q sends the same message in X and X_t at time $t + D - 1$. If it sends a message, then we have $\Delta > D - 1$. If it does not send a message, then the mechanism violates the non-interference condition. So any correct shutdown mechanism requires exactly $\Delta = D$ time to finish in the worst case.

E.2 Assignment 2: due Wednesday, 2019-03-06, at 5:00pm

E.2.1 A non-failure detector

Consider the following vaguely monarchist leader election mechanism for an asynchronous message-passing system with crash failures. Each process has access to an oracle that starts with the value 0 and may increase over time. The oracle guarantees:

1. No two processes ever see the same nonzero value.

2. Eventually some non-faulty process is given a fixed value that is larger than the values for all other processes for the rest of the execution.

As a function of the number of processes n, what is the largest number of crash failures f for which it is possible to solve consensus using this oracle?

Solution

We need $f < n/2$.

To show that $f < n/2$ is sufficient, observe that we can use the oracle to construct an eventually strong ($\Diamond S$) failure detector.

Recall that $\Diamond S$ has the property that there is some non-faulty process that is eventually never suspected, and every fault process is eventually permanently suspected. Have each process broadcast the current value of its leader oracle whenever it increases; when a process p receives i from some process q, it stops suspecting q if i is greater than any value p has previously seen, and starts suspecting all other processes. The guarantee that eventually some non-faulty q gets a maximum value that never changes ensures that eventually q is never suspected, and all other processes (including faulty processes) are suspected. We can now use Algorithm 13.2 to solve consensus.

To show that $f < n/2$ is necessary, apply a partition argument. In execution Ξ_0, processes $n/2+1$ through n crash, and processes 1 through $n/2$ run with input 0 and with the oracle assigning value 1 to process 1 (and no others). In execution Ξ_1, processes 1 through $n/2$ crashes, and processes $n/2+1$ through n run with input 1 and with the oracle assigning value 2 to process n (and no others). In each of these executions, termination and validity require that eventually the processes all decide on their respective input values 0 and 1.

Now construct an execution Ξ_2, in which both groups of processes run as in Ξ_0 and Ξ_1, but no messages are exchanged between the groups until after both have decided (which must occur after a finite prefix because this execution is indistinguishable to the processes from Ξ_0 or Ξ_1). We now violate agreement.

E.2.2 Ordered partial broadcast

Define **ordered partial broadcast** as a protocol that allows any process to broadcast a message, with the guarantees that, for messages sent through the broadcast mechanism:

1. Any message sent by a non-faulty process is received by at least one process;
2. Any message that is received by at least one process is received by at least k processes; and

3. If two processes p and q both receive messages m_1 and m_2 from the protocol, then either p and q both receive m_1 before m_2, or they both receive m_2 before m_1.

Give an implementation of ordered partial broadcast with $k = 3n/4$ that works for sufficiently large n in a fully-connected asynchronous message-passing system with up to $f = n/6$ crash failures, or show that no such implementation is possible.

Solution

No such implementation is possible. The proof is by showing that if some such implementation could work, we could solve asynchronous consensus with 1 crash failure, contradicting the Fischer-Lynch-Patterson bound [FLP85] (see Chapter 11).

An implementation of consensus based on totally-ordered partial broadcast for $k = 3n/4$ is given in Algorithm E.3. In fact, $k = 3n/4$ is overkill when $f = 1$; $k > n/2 + f$ is enough.

```
1  first ← ⊥
2  for i ← 1 to n do
3      count[i] ← 0
4      value[i] ← ⊥
5  broadcast ⟨i, input⟩
6  upon receiving ⟨j, v⟩ do
7      if first = ⊥ then
8          first ← ⟨j, v⟩
9          send received(⟨j, v⟩) to all processes
10 upon receiving received(⟨j, v⟩) do
11     count[j] ← count[j] + 1
12     value[j] ← v
13     if count[j] = k − f then
14         decide value[j]
```

Algorithm E.3: Consensus from totally-ordered partial broadcast. Code for process i.

The idea of the algorithm is to use the broadcast mechanism to chose a decision value, by looking at which value is delivered first. Since not every process will see the same value delivered first, this requires a second round

of communication in which processes retransmit their first incoming message. The following lemma shows that this is enough to get agreement:

Lemma E.2.1. *In any execution of Algorithm E.3 with $k > n/2 + f$. there is a unique pair $\langle j, v\rangle$ such that at least $k - f$ non-faulty processes resend* `received`$(\langle j, v\rangle)$.

Proof. Because all processes that receive messages m_1 and m_2 through the broadcast mechanism receive them in the same order, we can define a partial order on messages by letting $m_1 < m_2$ if any process receives m_1 before m_2.

There are only finitely many messages, so there is at least one pair $\langle j, v\rangle$ that is minimal in this partial order. This message is received by at least k processes, of which at least $k - f$ are non-faulty. Each such process receives $\langle j, v\rangle$ before any other broadcast messages, so it sets first to $\langle j, v\rangle$ and resends `received`$(\langle j, v\rangle)$.

To show that $\langle j, v\rangle$ is unique, observe that $k - f > n/2$ implies that if there is some other pair $\langle j', v'\rangle$ that is resent by $k - f$ non-faulty processes, then there is some process that resends both $\langle j, v\rangle$ and $\langle j', v'\rangle$. But each process resends at most one pair. □

Lemma E.2.1 immediately gives agreement, because a process only decides on a value v after receiving `received`$(\langle j, v\rangle)$ from $k - f$ processes, and only one such pair is sent by so many. Termination follows from the existence of such a pair: eventually every non-faulty process receives $\langle j, v\rangle$ from $k - f$ processes. Validity is immediate from the fact that v is j's input.

It follows that Algorithm E.3 solves consensus whenever $k > n/2 + f$, which includes the case $k = 3n/4$ and $f = n/6$. If an implementation of ordered partial broadcast with these parameters exists in the standard message-passing model, this would give a protocol for asynchronous consensus with $f = n/6 \geq 1$ when $n \geq 6$. This contradicts FLP, showing that such an implementation is impossible.

E.2.3 Mutual exclusion using a counter

Algorithm E.4 gives a modified version of Peterson's two-process mutual exclusion algorithm (§18.5.1) that replaces the present bits with an atomic counter count. This object supports read, increment, and decrement operations, where increment and decrement increase and decrease the value in the counter by one, respectively. Unlike the present array, count doesn't depend on the number of processes n. So in principle this algorithm might work for arbitrary n.

```
   shared data:
 1 waiting, atomic register, initially arbitrary
 2 count, atomic counter, initially 0
 3 Code for process i:
 4 while true do
       // trying
 5     increment count
 6     waiting ← i
 7     while true do
 8         if count = 1 then
 9             break
10         if waiting = i + 1 (mod n) then
11             break
       // critical
12     (do critical section stuff)
       // exiting
13     decrement count
       // remainder
14     (do remainder stuff)
```

Algorithm E.4: Peterson's mutual exclusion algorithm using a counter

Show that Algorithm E.4 provides starvation-free mutual exclusion for two processes, but not for three processes.

Solution

The proof that this works for two processes is essentially the same as in the original algorithm. The easiest way to see this is to observe that process p_i sees count $= 1$ in Line 8 under exactly the same circumstances as it sees present$[\neg i] = 0$ in Line 8 in the original algorithm; and similarly with two processes waiting is always set to the same value as waiting in the original algorithm. So we can map any execution of Algorithm E.4 for two processes to an execution of Algorithm 18.5, and all of the properties of the original algorithm carry over to the modified version.

To show that the algorithm doesn't work for three processes, we construct an explicit bad execution:

1. p_0 increments count
2. p_1 increments count
3. p_2 increments count
4. p_0 writes 0 to waiting
5. p_1 writes 1 to waiting
6. p_2 writes 2 to waiting
7. p_0 reads 3 from count
8. p_1 reads 3 from count
9. p_2 reads 3 from count
10. p_1 reads 2 from waiting and enters the critical section.
11. p_1 leaves the critical section and decrements count.

At this point we have count $= 2$ and waiting $= 2$, with both p_0 and p_2 at the start of the loop to check these variables. Suppose that p_1 doesn't come back. Because neither p_0 nor p_2 changes count or waiting, both variables remain at 2 forever. But then neither p_0 nor p_2 enters the critical section, because count is never 1 and waiting is never equal to $0+1$ or $2+1 \pmod 3$.

E.3 Assignment 3: due Wednesday, 2019-04-17, at 5:00pm

E.3.1 Zero, one, many

Consider a counter supporting `inc` and `read` operations that is capped at 2. This means that after the first two calls to `inc`, any further calls to `inc` have no effect: a `read` operation will return 0 if it follows no calls to `inc`, 1 if it follows exactly one call to `inc`, and 2 if it follows two or more calls to `inc`.

There is a straightforward implementation of this object using snapshot. This requires $O(n)$ space and $O(n)$ steps per operation in the worst case.

Is it possible to do better? That is, can one give a deterministic, wait-free, linearizable implementation of a 2-bounded counter from atomic registers that uses $o(n)$ space and $o(n)$ steps per operation in the worst case?

Solution

One possible implementation is given in Algorithm E.5. This requires $O(1)$ space and $O(1)$ steps per call to `inc` or `read`.

```
1  procedure inc
2      if c[1] = 1 then
           // somebody already did inc
3          c[2] ← 1
4      else
5          c[1] ← 1
           // maybe somebody else is doing inc
6          if splitter returns right or down then
7              c[2] ← 1
8  procedure read
9      if c[2] = 1 then
10         return 2
11     else if c[1] = 1 do
12         return 1
13     else
14         return 0
```

Algorithm E.5: A 2-bounded counter

The implementation uses two registers $c[1]$ and $c[2]$ to represent the value of the counter. Two additional registers implement a splitter object as in Algorithm 18.6.[1]

Claim: For any two calls to `inc`, at least one sets $c[2]$ to 1. Proof: Suppose otherwise. Then both calls are by different processes p and q (or else the second call would see $c[1] = 1$) and both execute the splitter. Since a splitter returns stop to at most one process, one of the two processes gets right or down, and sets $c[2]$.

It is also straightforward to show that a single `inc` running alone will set $c[1]$ but not $c[2]$, since in this case the splitter will return stop.

Now we need to argue linearizability. We will do so by assigning linearization points to each operation.

If some `inc` does not set $c[2]$, assign it the step at which it sets $c[1]$. Assign each other `inc` the step at which it first sets $c[2]$.

If every `inc` sets $c[2]$, assign the first `inc` to set $c[1]$ the step at which it does so, and assign all others the first point during its execution interval at which $c[2]$ is nonzero.

For a `read` operation that returns 2, assign the step at which it reads $c[2]$. For a `read` operation that returns 1, assign the first point in the execution interval after it reads $c[2]$ at which $c[1] = 1$. For a `read` operation that returns 0, assign the step at which it reads $c[1]$.

This will assign the same linearization point to some operations; in this case, put `inc`s before `read`s and otherwise break ties arbitrarily.

These choices create a linearization which consists of (a) a sequence of `read` operations that return 0, all of which are assigned linearization points before the first step at which $c[1] = 1$; (b) the first `inc` operation that sets $c[1]$; (c) a sequence of `read` operations that return 1, all of which are linearized after $c[1] = 1$ but before $c[2] = 1$; (c) some `inc` that is either the first to set $c[2]$ or spans the step that sets $c[2]$; and (d) additional `inc` operations together with `read` operations that all return 2. Since each `read` returns the minimum of 2 and the number of `inc`s that precede it, this is a correct linearization.

E.3.2 A very slow counter

Consider a **slow counter** object with operations `inc` and `read`, where the value v of a counter starts at 0 and increases by 1 as the result of each call

[1] It may be possible shave off a register by breaking the splitter abstraction and using the race or door register in place of $c[1]$, but I haven't worked out the linearizability for this case.

to `inc`, but `read` returns $\log^* v$ instead of v.

Suppose we want a deterministic, wait-free, linearizable implementation of a slow counter as defined above from atomic registers. Give tight bounds on the worst-case step complexity of operations on such an implementation.

Solution

The worst-case step complexity of an operation is $\Theta(n)$.

For the upper bound, implement a counter on top of snapshots (or just collect), and have `read` compute $\log^*$ of whatever value is read.

For the lower bound, observe that a slow counter has the perturbability property needed for the JTT proof. Given an execution of the form $\Lambda_k \Sigma_k \pi$ as described in Chapter 21, we can always insert some sequence of `inc` operations between Λ_k and Σ_k that will change the return value of π. The number of `inc`s needed will be the number needed to raise $\log^* v$, plus an extra n to overcome the possibility of pending `inc`s in Σ_k being linearized before or after π. Since this object is perturbable, and the atomic registers we are implementing it from are historyless, JTT applies and gives an $\Omega(n)$ lower bound on the cost of `read` in the worst case.

E.3.3 Double-entry bookkeeping

Consider an object that implements an unbounded collection of accounts $A_1, A_2, \ldots$, each of which holds an integer value, and that provides three operations:

- The operation `read`(i) returns the current value of A_i.
- The operation `transfer`(i, j, n) moves n units from A_i to A_j; if A'_i and A'_j are the new values, then $A'_i = A_i - n$ and $A'_j = A_j + n$.
- The operation `close`(i, j) sets A_i to zero and adds the previous value to A_j. It is equivalent to atomically executing `transfer`$(i, j,$ `read`$(i))$.

1. What is the consensus number of this object?
2. What is the consensus number of a restricted version of this object that provides only the `read` and `transfer` operations?

Solution

1. The consensus number of the object is infinite. Initialize A_0 to 1 and the remaining A_i to 0. We can solve ID consensus by having process i (where $i > 0$ execute `close`$(0, i)$ and then applying `read` to scan all the A_j values for itself and other processes. Whichever process gets the 1 wins.

2. The consensus number without `close` is 1. Proof: Observe that `transfer` operations commute.

E.4 CS465/CS565 Final Exam, May 7th, 2019

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are four problems on this exam, each worth 20 points, for a total of 80 points. You have approximately three hours to complete this exam.

E.4.1 A roster (20 points)

A **roster** object has operations `announce` and `read`, where `read` returns a list of the identities of all processes that have previously called `announce` at least once.

Suppose you want a wait-free, linearizable implementation of this object from multiwriter atomic registers. As a function of the number of processes n, give the best upper and lower bound you can on the number of registers you will need.

You may assume that the set of process identities is fixed for each n and that each process knows its own identity.

Solution

You will need exactly n registers ($\Theta(n)$ is also an acceptable answer).

For the upper bound, have each process write its ID to its own register, and use a double-collect snapshot to read all of them. This uses exactly n registers. The double-collect snapshot is wait-free because after each process has called `announce` once, the contents of the registers never change, so `read` finishes after $O(n)$ collects or $O(n^2)$ register reads. It's linearizable because double-collect snapshot returns the exact contents of the registers at some

point during its execution.[2]

For the lower bound, use a covering argument.[3]

Have the processes $p_1, \ldots, p_n$ run `announce` in order, stopping each process when it covers a new register. This will give sequence of partial executions Ξ_i where at the end of Ξ_i, there is a set of i registers $r_1 \ldots r_i$ that are covered by $p_1 \ldots p_i$, and no other operations are in progress.

To show this works, we need to argue that each p_{i+1} does in fact cover a register $r_{i+1} \notin \{r_1, \ldots, r_i\}$. If not, then we can extend Ξ_i by running p_{i+1}'s `announce` operation to completion, then delivering all the covering writes by $p_1 \ldots p_i$. Now any subsequent `read` will fail to return p_{i+1}, violating the specification. (If we have a spare process, we can have it do the bad `read`; otherwise we can run p_1 to completion and let it do it.)

At the end of Ξ_n, we have covered n distinct registers, proving the lower bound.

E.4.2 Self-stabilizing consensus (20 points)

Consider a model with n processes $p_0, \ldots, p_{n-1}$ organized in a ring, where p_i can directly observe both its own state and that of $p_{(i-1) \bmod n}$. Suppose that the state x_i of each process p_i is always a natural number.

At each step, an adversary chooses a process p_i to run. This process then updates its own state based on its previous state x_i and the state $x_{(i-1) \bmod n}$ of its counterclockwise neighbor $p_{(i-1) \bmod n}$. The adversary is required to run every process infinitely often but is not otherwise constrained.

Is there a protocol for the processes that guarantees that, starting from an arbitrary initial configuration, they eventually reach a configuration where (a) all processes have the same state $x \in \mathbb{N}$; and (b) no process ever changes its state as the result of taking additional steps?

---

[2] If we only do a single collect, the implementation is not linearizable. An example of a bad execution is one where a reader reads r_1, then p_1 writes to r_1 (starting and finishing its `announce` operation), then p_2 writes to r_2 (starting an finishing its `announce` operation), and finally the reader reads r_2. In this execution the reader will return $\{p_2\}$ only, which is inconsistent with the observed ordering that puts `announce`(p_1) before `announce`(p_2).

[3] It's tempting to use JTT [JTT00] here, but the roster object is not perturbable. Once all IDs of p_1 through p_{n-1} have been registered, subsequent `announce` operations by p_1 through p_{n-1} have no effect.

The subtlety here is that in the JTT argument, we probably won't choose γ when perturbing $\Lambda_k \Sigma_k$ to include more than one new `announce`, but replacing γ by γ' to hit the first possible uncovered register in π might involve an arbitrary sequence of operations by p_1 through p_{n-1}, including a sequence where all of them call `announce` at least once. Once this happens, we get a $\Lambda_{k+1} \Sigma_{k+1}$ execution that can no longer be perturbed.

Give such a protocol and prove that it works, or show that no such protocol is possible.

Solution

It turns out that this problem is a good example of what happens if you don't remember to include some sort of validity condition. As pointed in several student solutions, having each process pick a fixed constant x_i the first time it updates works.

Here is a protocol that also works, and satisfies the validity condition that the common output was some process's input (which was not required in the problem statement). When p_i takes a step, it sets x_i to $\max(x_i, x_{(i-1) \bmod n})$.

To show that this works, we argue by induction that the maximum value eventually propagates to all processes. Let $x = x_i$ be the initial maximum value. The induction hypothesis is that for each $j \in \{0, \ldots, n-1\}$, eventually all processes in the range i through $i + j \pmod n$ hold value x forever.

Suppose that the hypothesis holds for j; to show that it holds for $j + 1$, start in a configuration where x_i through x_{i+j} are all x. No transition can change any of these values, because taking the max of x and any other value yields x. Because each process is scheduled infinitely often, eventually p_{i+j+1} takes a step; when this happens, x_{i+j+1} is set to $\max(x, x_{i+j+1}) = x$.

Since the hypothesis holds for all $j \in \{0, \ldots, n-1\}$, it holds for $j = n-1$; but this just says that eventually all n processes hold x forever.

E.4.3 All-or-nothing intermittent faults (20 points)

Recall that in the standard synchronous message-passing model with crash failures, a faulty process runs correctly up until the round in which it crashes, during which it sends out some subset of the correct messages, and after which it sends out no messages at all.

Suppose instead we have intermittent faults, where any process may fail to send outgoing messages in a particular round, but these are all-or-nothing faults in the sense that a process either sends all of its messages in a given round or no messages in that round. To avoid shutting down a protocol completely, we require that in every round, there is at least one process that sends all of its messages. We also allow a process to send a message to itself.

If we wish to solve agreement (that is, get agreement, termination, and validity) in this model, what is the minimum number of rounds we need in the worst case?

Solution

We need one round. Every process transmits its input to all processes, including itself. From the all-or-nothing property, all processes receive the same set of messages. From the assumption that some process is not faulty in this round, this set is nonempty. So the processes can reach agreement by applying any consistent rule to choose an input from the set.

E.4.4 A tamper-proof register (20 points)

Consider a **tamper-proof register**, which is a modified version of a standard multiwriter atomic register for which the `read` operation returns $\perp$ if no `write` operation has occurred, v if exactly one `write`(v) operation has occurred, and fail if two or more `write` operations have occurred.

What is the consensus number of this object?

Solution

The consensus number is 1.

Proof: We can implement it from atomic snapshot, which can be implemented from atomic registers, which have consensus number 1.

For my first `write`(v) operation, write v to my component of the snapshot; for subsequent `write`(v) operations, write fail. For a `read` operation, take a snapshot and return (a) $\perp$ if all components are empty; (b) v if exactly one component is non-empty and has value v; and (c) fail if more than one component is non-empty or any component contains fail.

Appendix F

Sample assignments from Spring 2016

F.1 Assignment 1: due Wednesday, 2016-02-17, at 5:00pm

Bureaucratic part

Send me email! My address is james.aspnes@gmail.com.

In your message, include:

1. Your name.
2. Your status: whether you are an undergraduate, grad student, auditor, etc.
3. Whether you are taking the course as CPSC 465 or CPSC 565.
4. Anything else you'd like to say.

(You will not be graded on the bureaucratic part, but you should do it anyway.)

F.1.1 Sharing the wealth

A kindergarten consists of n children in a ring, numbered 0 through $n-1$, with all arithmetic on positions taken mod n.

In the initial configuration, child 0 possesses n cookies. The children take steps asynchronously, and whenever child i takes a step in a configuration

where they have a cookie but child $i+1$ does not, child i gives one cookie to child $i+1$. If child $i+1$ already has a cookie, or child i has none, nothing happens. We assume that a fairness condition guarantees that even though some children are fast, and some are slow, each of them takes a step infinitely often.

1. Show that after some finite number of steps, every child has exactly one cookie.

2. Suppose that we define a measure of time in the usual way by assigning each step the largest possible time consistent with the assumption that that no child ever waits more than one time unit to take a step. Show the best asymptotic upper bound you can, as a function of n, on the time until every child has one cookie.

3. Show the best asymptotic lower bound you can, as a function of n, on the worst-case time until every child has one cookie.

Solution

1. First observe that in any configuration reachable from the initial configuration, child 0 has k cookies, $n-k$ of the remaining children have one cookie each, and the rest have zero cookies. Proof: Suppose we are in a configuration with this property, and consider some possible step that changes the configuration. Let i be the child that takes the step. If $i=0$, then child i goes from k to $k-1$ cookies, and child 1 goes from 0 to 1 cookies, increasing the number of children with one cookie to $n-k+1$. If $i>0$, then child i goes from 1 to 0 cookies and child $i+1$ from 0 to 1 cookies, with k unchanged. In either case, the invariant is preserved.

   Now let us show that k must eventually drop as long as some cookie-less child remains. Let i be the smallest index such that the i-th child has no cookie. Then after finitely many steps, child $i-1$ takes a step and gives child i a cookie. If $i-1=0$, k drops. If $i-1>0$, then the leftmost 0 moves one place to the left. It can do so only finitely many times until $i=1$ and k drops the next time child 0 takes a step. It follows that after finitely many steps, $k=1$, and by the invariant all $n-1$ remaining children also have one cookie each.

2. Number the cookies 0 through $n-1$. When child 0 takes a step, have it give the largest-numbered cookie it still possesses to child 1. For each

cookie i, let x_i^t be the position of the i-th cookie after t asynchronous rounds, where an asynchronous round is the shortest interval in which each child takes at least one step.

Observe that no child $j > 0$ ever gets more than one cookie, since no step adds a cookie to a child that already has one. It follows that cookie 0 never moves, because if child 0 has one cookie, so does everybody else (including child 1). We can thus ignore the fact that the children are in a cycle and treat them as being in a line $0 \ldots n-1$.

We will show by induction on t that, for all i and t, $x_i^t \geq y_i^t = \max(0, \min(i, z_i^t))$ where $z_i^t = t + 2(i - n + 1)$.

Proof: The base case is when $t = 0$. Here $x_i^t = 0$ for all i. We also have $z_i^t = 2(i - n + 1) \leq 0$ so $y_i^t = \max(0, \min(i, z_i^t)) = \max(0, z_i^t) = 0$. So the induction hypothesis holds with $x_i^t = y_i^t = 0$.

Now suppose that the induction hypothesis holds for t. For each i, there are several cases to consider:

(a) $x_i^t = x_{i+1}^t = 0$. In this case cookie i will not move, because it's not at the top of child 0's stack. But from the induction hypothesis we have that $x_{i+1}^t = 0$ implies $z_{i+1}^t = t + 2(i + 1 - n + 1) \leq 0$, which gives $z_i^t = z_{i+1}^t - 2 \leq -2$. So $z_i^{t+1} \leq z_{i+1}^t + 1 \leq -1$ and $y_i^{t+1} = 0$, and the induction hypothesis holds for x_i^{t+1}.

(b) $x_i^t = i$. Then even if cookie i doesn't move (and it doesn't), we have $x_i^{t+1} \geq x_i^t \geq \min(i, z_i^t)$.

(c) $x_i^t < i$ and $x_{i+1}^t = x_i^t + 1$. Again, even if cookie i doesn't move, we still have $x_i^{t+1} \geq x_i^t = x_{i+1}^t - 1 \geq y_{i+1}^t - 1 \geq t + 2(i+1-n+1) - 1 = t + 2(i - n + 1) + 1 > y_i^t$.

(d) $x_i^t < i$ and $x_{i+1}^t > x_i^t + 1$. Nothing is blocking cookie i, so it moves: $x_i^{t+1} = x_i^t + 1 \geq t + 2(i - n + 1) + 1 = (t + 1) + 2(i - n + 1) = y_i^{t+1}$.

It follows that our induction hypothesis holds for all t. In particular, at $t = 2n - 2$ we have $z_i^t = 2n - 2 + 2(i - n + 1) = 2i - 1 \geq i$ for all $i > 0$. So at time $2n - 2$, $x_i^t \geq y_i^t = i$ for all i and every child has one cookie. This gives an asymptotic upper bound of $O(n)$.

3. There is an easy lower bound of $n - 1$ time. Suppose we run the processes in round-robin order, i.e., the i-th step is taken by process $i \bmod n$. Then one time unit goes by for every n steps, during which each process takes exactly one step. Since process 0 reduces its count

by at most 1 per step, it takes at least $n-1$ time to get it to 1. This gives an asymptotic lower bound of $\Omega(n)$, which is tight.

I believe it should be possible to show an exact lower bound of $2n-2$ time by considering a schedule that runs in reverse round-robin order $n-1, n-2, \ldots, 0, n-1, n-2, \ldots$, but this is more work and gets the same bound up to constants.

F.1.2 Eccentricity

Given a graph $G = (V, E)$, the **eccentricity** $\epsilon(v)$ of a vertex v is the maximum distance $\max_{v'} d(v, v')$ from v to any vertex in the graph.

Suppose that you have an anonymous[1] asynchronous message-passing system with no failures whose network forms a tree.

1. Give an algorithm that allows each node in the network to compute its eccentricity.

2. Safety: Prove using an invariant that any value computed by a node using your algorithm is in fact equal to its eccentricity. (You should probably have an explicit invariant for this part.)

3. Liveness: Show that every node eventually computes its eccentricity in your algorithm, and that the worst-case message complexity and time complexity are both within a constant factor of optimal for sufficiently large networks.

Solution

1. Pseudocode is given in Algorithm F.1. For each edge vu, the algorithm sends a message d from v to u, where d is the maximum length of any simple path starting with uv. This can be computed as soon as v knows the maximum distances from all of its other neighbors $u' \neq u$.

2. We now show correctness of the values computed by Algorithm F.1. Let $d_v[u]$ be the value of $d[u]$ at v. Let $\ell_v[u]$ be the maximum length of any simple path starting with the edge vu. To show that the

[1] Clarification added 2016-02-13: Anonymous means that processes don't have global IDs, but they can still tell their neighbors apart. If you want to think of this formally, imagine that each process has a **local identifier** for each of its neighbors: a process with three neighbors might number them $1, 2, 3$ and when it receives or sends a message one of these identifiers is attached. But the local identifiers are arbitrary, and what I call you has no relation to what you call me or where either of us is positioned in the network.

```
   local data: d[u] for each neighbor u, initially ⊥
   notified[u] for each neighbor u, initially false

 1 initially do
 2     notify ()
 3 upon receiving d from u do
 4     d[u] ← d
 5     notify ()
 6 procedure notify ()
 7     foreach neighbor u do
 8         if ¬notified[u] and d[u'] ≠ ⊥ for all u' ≠ u then
 9             Send 1 + max_{u'≠u} d[u'] to u
10             notified[u] ← true
11     if notified[u] = true for all neighbors u then
12         ε ← max_u d[u]
```

Algorithm F.1: Computing eccentricity in a tree

algorithm computes the correct values, we will prove the invariant that $d_v[u] \in \{\bot, \ell_v[u]\}$ always, and for any message d in transit from u to v, $d = \ell_v[u]$.

In the initial configuration, $d_v[u] = \bot$ for all v and u, and there are no messages in transit. So the invariant holds.

Now let us show that calling `notify` at some process v preserves the invariant. Because `notify`() does not change d_v, we need only show that the messages it sends contain the correct distances.

Suppose `notify`() causes v to send a message d to u. Then $d = 1 + \max_{u' \neq u} d_v[u'] = 1 + \max_{u' \neq u} \ell_v[u']$, because $d_v[u'] \neq \bot$ for all neighbors $u' \neq u$ by the condition on the if statement and thus $d_v[u'] = \ell_v[u']$ for all $u' \neq u$ by the invariant.

So the invariant will continue to hold in this case provided $\ell_u[v] = 1 + \max_{u' \neq u} \ell_v[u']$. The longest simple path starting with uv either consists of uv alone, or is of the form $uvw\ldots$ for some neighbor w of v with $w \neq u$. In the former case, v has no other neighbors u', in which case $d = 1 + \max_{u' \neq u} \ell_v[u'] = 1 + 0 = 1$, the correct answer. In the latter case, $d = 1 + \max_{u' \neq u} \ell_v[u'] = 1 + \ell_v[w]$, again the length of the longest path starting with uv.

This shows that `notify` preserves the invariant. We must also show that assigning $d_v[u] \leftarrow d$ upon receiving d from u does so. But in this case we know from the invariant that $d = \ell_v[u]$, so assigning this value to $d_v[u]$ leaves $d_v[u] \in \{\perp, \ell_v[u]\}$ as required.

3. First let's observe that at most one message is sent in each direction across each edge, for a total of $2|E| = 2(n-1)$ messages. This is optimal, because if in some execution we do not send a message across some edge uv, then we can replace the subtree rooted at u with an arbitrarily deep path, and obtain an execution indistinguishable to v in which its eccentricity is different from whatever it computed.

   For time complexity (and completion!) we'll argue by induction on $\ell_v[u]$ that we send a message across uv by time $\ell_v[u] - 1$.

   If $\ell_v[u] = 1$, then u is a leaf; as soon as `notify` is called in its initial computation event (which we take as occurring at time 0), u notices it has no neighbors other than v and sends a message to v.

   If $\ell_v[u] > 1$, then since $\ell_v[u] = 1 + \max_{v' \neq v} \ell_u[v']$, we have $\ell_u[v]' \leq \ell_v[u] - 1$ for all neighbors $v' \neq v$ of u, which by the induction hypothesis means that each such neighbor v' sends a message to u no later than time $\ell_v[u] - 2$. These messages all arrive at u no later than time $\ell_v[u] - 1$; when the last one is delivered, u sends a message to v.

   It follows that the last time a message is sent is no later than time $\max_{uv}(\ell_v[u] - 1)$, and so the last delivery event occurs no later than time $\max_{uv} \ell_v[u]$. This is just the diameter D of the tree, giving a worst-case time complexity of exactly D.

   To show that this is optimal, consider an execution of some hypothetical algorithm that terminates by time $D - 1$ in the worst case. Let u and v be nodes such that $d(u, v) = D$. Then there is an execution of this algorithm in no chain of messages passes from u to v, meaning that no event of u is causally related to any event of v. So we can replace u with a pair uw of adjacent nodes with $d(w, v) = d(u, v) + 1$, which changes $\epsilon(v)$ but leaves an execution that is indistinguishable to v from the original. It follows that v returns an incorrect value in some executions, and this hypothetical algorithm is not correct. So time complexity D is the best possible in the worst case.

F.1.3 Leader election on an augmented ring

Suppose that we have an asynchronous ring where each process has a distinct identity, but the processes do not know the size n of the ring. Suppose also that each process i can send messages not only to its immediate neighbors, but also to the processes at positions at positions $i-3$ and $i+3 \pmod n$ in the ring.

Show that $\Theta(n \log n)$ messages are both necessary and sufficient in the worst case to elect a unique leader in this system.

Solution

For sufficiency, ignore the extra edges and use Hirschberg-Sinclair [HS80] (see §5.2.2).

For necessity, we'll show that an algorithm that solves leader election in this system using at most $T(n)$ messages can be modified to solve leader election in a standard ring without the extra edges using at most $3T(n)$ messages. The idea is that whenever a process i attempts to send to $i+3$, we replace the message with a sequence of three messages relayed from i to $i+1$, $i+2$, and then $i+3$, and similarly for messages sent in the other direction. Otherwise the original algorithm is unmodified. Because both systems are asynchronous, any admissible execution in the simulated system has a corresponding admissible execution in the simulating system (replace each delivery event by three delivery events in a row for the relay messages) and vice versa (remove the initial two relay delivery events for each message and replace the third delivery event with a direct delivery event). So in particular if there exists an execution in the simulating system that requires $\Omega(n \log n)$ messages, then there is a corresponding execution in the simulated system that requires at least $\Omega(n \log n/3) = \Omega(n \log n)$ messages as well.

F.2 Assignment 2: due Wednesday, 2016-03-09, at 5:00pm

F.2.1 A rotor array

Suppose that you are given an object that acts as described in Algorithm F.2. A `write` operation on this object writes to location $A[r]$ and increments r mod n. A `read` operation by process i (where $i \in \{0 \dots n-1\}$) returns $A[i]$. Initially, $r = 0$ and $A[i] = \perp$ for all i.

What is the consensus number of this object?

```
1 procedure write(A, v)
2     atomically do
3         A[r] ← v; r ← (r + 1) mod n
4 procedure read(A)
5     return A[i]
```

Algorithm F.2: Rotor array: code for process i

Solution

First let's show that it is at least 2, by exhibiting an algorithm that uses a single rotor array plus two atomic registers to solve 2-process wait-free consensus.

```
1 procedure consensus(v)
2     input[i] ← v
3     write(A, i)
4     i′ ← read(A)
5     if i′ = i then
          // Process 0 wrote first
6         return input[0]
7     else
          // Process 1 wrote first
8         return input[1]
```

Algorithm F.3: Two-process consensus using a rotor array

The algorithm is given as Algorithm F.3. Each process i first writes its input value to a single-writer register $\mathsf{input}[i]$. The process then writes its ID to the rotor array. There are two cases:

1. If process 0 writes first, then process 0 reads 0 and process 1 reads 1. Thus both processes see $i' = i$ and return $\mathsf{input}[0]$, which gives agreement, and validity because $\mathsf{input}[0]$ is then equal to 0's input.

2. If process 1 writes first, then process 0 reads 1 and process 1 reads either 0 (if 0 wrote quickly enough) or $\perp$ (if it didn't). In either case, both processes see $i' \neq i$ and return $\mathsf{input}[1]$.

Now let us show that a rotor array can't be used to solve wait-free consensus with three processes. We will do the usual bivalence argument,

and concentrate on some bivalent configuration C and pair of operations π_0 and π_1 such that $C\pi_i$ is i-valent for each i.

If π_0 and π_1 are operations on different objects or operations on an atomic register, then they either commute or the usual analysis for atomic registers gives a contradiction. So the interesting case is when π_0 and π_1 are both operations on a single rotor array object A.

If either operation is a `read`, then only the process that carries out the `read` knows whether it occurred. The same argument as for atomic registers applies in this case. So the only remaining case is when both operations are `write`s.

Consider the configurations $C\pi_0\pi_1$ (which is 0-valent) and $C\pi_1\pi_0$ (which is 1-valent). These differ in that there are two locations j and $(j+1) \bmod n$ (which we will just write as $j+1$) that contain values v_0 and v_1 in the first configuration and v_1 and v_0 in the second. Suppose that we stop processes j and $j+1$, and let some other process run alone until it decides. Because this third process can't observe either locations j or $j+1$, it can't distinguish between $C\pi_0\pi_1$ and $C\pi_1\pi_0$, and thus decides the same value starting from either configuration. But this contradicts the assumption that $C\pi_i$ is i-valent. It follows that there is no escape from bivalence with three processes, and the rotor array plus atomic registers cannot be used to solve three-process wait-free consensus.

The consensus number of this object is 2.

F.2.2 Set registers

Suppose we want to implement a **set register** S in a message-passing system, where a set register provides operations `insert`(S, v), which inserts a new element v in S, and `read`(S), which returns the set of all elements previously inserted into S. So, for example, after executing `insert`$(S, 3)$, `insert`$(S, 1)$, and `insert`$(S, 1)$; `read`(S) would return $\{1, 3\}$.

1. Give an algorithm for implementing a linearizable set register where all operations terminate in finite time, in a deterministic asynchronous message-passing system with $f < n/2$ crash failures and no failure detectors, or show that no such algorithm is possible.

2. Suppose that we change the `read`(S) operation to return a list of all the elements of S in the order they were first inserted (e.g., $[3, 1]$ in the example above). Call the resulting object an **ordered set register**.

   Give an algorithm for implementing a linearizable ordered set register

under the same conditions as above, or show that no such algorithm is possible.

Solution

1. It's probably possible to do this with some variant of ABD, but getting linearizability when there are multiple concurrent `insert` operations will be tricky.

   Instead, we'll observe that it is straightforward to implement a set register using a shared-memory snapshot: each process writes to $A[i]$ the set of all values it has ever inserted, and a `read` consists of taking a snapshot and then taking the union of the values. Because we can implement snapshots using atomic registers, and we can implement atomic registers in a message-passing system with $f < n/2$ crash failures using ABD, we can implement this construction in a message-passing system with $f < n/2$ failures.

2. This we can't do. The problem is that an ordered set register can solve agreement: each process inserts its input, and the first input wins. But FLP says we can't solve agreement in an asynchronous message-passing system with one crash failure.

F.2.3 Bounded failure detectors

Suppose you have a deterministic asynchronous message-passing system equipped with a failure detector that is eventually weakly accurate and k-bounded strongly complete, meaning that at least $\min(k, f)$ faulty processes are eventually permanently suspected by all processes, where f is the number of faulty processes.

For what values of k, f, and n can this system solve agreement?

Solution

We can solve agreement using the k-bounded failure detector for $n \geq 2$ processes if and only if $f \leq k$ and $f < n/2$.

Proof:

If $k \geq f$, then every faulty process is eventually permanently suspected, and the k-bounded failure detector is equivalent to the $\Diamond S$ failure detector. The Chandra-Toueg protocol [CT96] then solves consensus for us provided $f < n/2$.

If $f \geq n/2$, the same partitioning argument used to show impossibility with $\Diamond P$ applies to the k-bounded detector as well (as indeed it applies to any failure detector that is only eventually accurate).

If $k < f$, then if we have an algorithm that solves agreement for n processes, then we can turn it into an algorithm that solves agreement for $n - k$ processes with $f - k$ failures, using no failure detector at all. The idea is that the $n - k$ processes can pretend that there are an extra k faulty processes that send no messages and that are permanently suspected. But this algorithm runs in a standard asynchronous system with $f - k$ failures, and FLP says we can't solve agreement in such a system with $n \geq 2$ and $f \geq 1$. So this rules out solving agreement in the original system if $k < f$ and $k \leq n - 2$.

There is one remaining case, where $k = n - 1$ and $f = n$. Here we can actually solve consensus when $n = 1$ (because we can always solve consensus when $n = 1$). For larger n, we have $f \geq n/2$. So there is only one exception to the general rule that we need $f \leq k$ and $f < n/2$.

F.3 Assignment 3: due Wednesday, 2016-04-20, at 5:00pm

F.3.1 Fetch-and-max

```
1  procedure fetchAndMax(r, 0 : x)
2      if switch = 0 then
3          return 0 : fetchAndMax(left, x)
4      else
5          return 1 : fetchAndMax(right, 0)
6  procedure fetchAndMax(r, 1 : x)
7      v ← fetchAndMax(right, x)
8      if TAS(switch) = 0 then
9          return 0 : fetchAndMax(left, 0)
10     else
11         return 1 : v
```

Algorithm F.4: Max register modified to use a test-and-set bit

Algorithm F.4 replaces the switch bit in the max register implementation from Algorithm 22.2 with a test-and-set, and adds some extra machinery to

return the old value of the register before the write.

Define a fetch-and-max register as a RMW object that supports a single operation `fetchAndMax`(x) that, as an atomic action, (a) replaces the old value v in the register with the maximum of x and v; and (b) returns the old value v.

Suppose that left and right are both linearizable wait-free k-bit fetch-and-max registers. Show that Algorithm F.4 implements a linearizable wait-free $(k+1)$-bit fetch-and-max register, or give an example of an execution that violates linearizability.

Solution

Here is a bad execution (there are others). Let $k = 1$, and let π_1 do `fetchAndMax`(01) and π_2 do `fetchAndMax`(10). Run these operations concurrently as follows:

1. π_1 reads switch and sees 0.
2. π_2 does `fetchAndMax`$(\mathsf{right}, 0)$.
3. π_2 does `TAS`(switch) and sees 0.
4. π_2 does `fetchAndMax`$(\mathsf{left}, 0)$ and sees 0.
5. π_1 does `fetchAndMax`$(\mathsf{left}, 1)$ and sees 0.

Now both π_1 and π_2 return 00. But in the sequential execution $\pi_1\pi_2$, π_2 returns 01; and in the sequential execution $\pi_2\pi_1$, π_1 returns 10. Since π_1 and π_2 return the values they return in the concurrent execution in neither sequential execution, the concurrent execution is not linearizable.

F.3.2 Median

Define a **median register** as an object r with two operations `addSample`(r, v), where v is any integer, and `computeMedian`(r). The `addSample` operation adds a sample to the multiset M of integers stored in the register, which is initially empty. The `computeMedian` operations returns a median of this multiset, defined as a value x with the property that (a) x is in the multiset; (b) at least $|M|/2$ values v in the multiset are less than or equal to x; (c) at least $|M|/2$ values v in the multiset are greater than or equal to x.

For example, if we add the samples $1, 1, 3, 5, 5, 6$, in any order, then a subsequent `computeMedian` can return either 3 or 5.

Suppose that you wish to implement a linearizable wait-free median register using standard atomic registers and resettable test-and-set bits. Give tight (up to constants) asymptotic upper and lower bounds on the number of such objects you would need. You may assume that the atomic registers may hold arbitrarily large values.

Solution

For the upper bound, we can do it with $O(n)$ registers using any linear-space snapshot algorithm (for example, Afek *et al.* [AAD+93]). Each process stores in its own segment of the snapshot object the multiset of all samples added by that process; `addSample` just adds a new sample to the process's segment. For `computeMedian`, take a snapshot, then take the union of all the multisets, then compute the median of this union. Linearizability and wait-freedom of both operations are immediate from the corresponding properties of the snapshot object.

For the lower bound, use JTT [JTT00]. Observe that both atomic registers and resettable test-and-sets are historyless: for both types, the new state after an operation doesn't depend on the old state. So JTT applies if we can show that the median register is perturbable.

Suppose that we have a schedule $\Lambda_k \Sigma_k \pi$ in which Λ_k consists of an arbitrary number of median-register operations of which at most k are incomplete, Σ_k consists of k pending base object operations (writes, test-and-sets, or test-and-set resets) covering k distinct base objects, and π is a read operation by a process not represented in $\Lambda_k \Sigma_k$. We need to find a sequence of operations γ that can be inserted between Λ_k and Σ_k that changes the outcome of π.

Let S be the multiset of all values appearing as arguments to `addSample` operations that start in Λ_k or Σ_k. Let $x = \max S$ (or 0 if S is empty), and let γ consist of $|S| + 1$ `addSample`$(r, x + 1)$ operations. Write T for the multiset of $|S| + 1$ copies of $x + 1$. Then in any linearization of $\Lambda_k \gamma \Sigma_k \pi$, the multiset U of samples contained in r when π executes includes at least all of T and at most all of S; this means that a majority of values in U are equal to $x + 1$, and so the median is $x + 1$. But $x + 1$ does not appear in S, so π can't return it in $\Lambda_k \Sigma_k \pi$. It follows that a median register is in fact perturbable, and JTT applies, which means that we need at least $\Omega(n)$ base objects to implement a median register.

F.3.3 Randomized two-process test-and-set with small registers

Algorithm F.5 gives an implementation of a randomized one-shot test-and-set for two processes, each of which may call the procedure at most once, with its process ID (0 or 1) as an argument.

The algorithm uses two registers, a_0 and a_1, that are both initialized to 0 and hold values in the range $0 \ldots m-1$, where m is a positive integer. Unfortunately, whoever wrote it forgot to specify the value of m.

```
1  procedure TAS(i)
2      myPosition ← 0
3      while true do
4          otherPosition ← read(a_¬i)
5          x ← myPosition − otherPosition
6          if x ≡ 2 (mod m) then
7              return 0
8          else if x ≡ −1 (mod m) do
9              return 1
10         else if fair coin comes up heads do
11             myPosition ← (myPosition + 1) mod m
12         write(a_i, myPosition)
```

Algorithm F.5: Randomized one-shot test-and-set for two processes

For what values of m does Algorithm F.5 correctly implement a one-shot, probabilistic wait-free, linearizable test-and-set, assuming:

1. An oblivious adversary?
2. An adaptive adversary?

Solution

For the oblivious adversary, we can quickly rule out $m < 5$, by showing that there is an execution in each case where both processes return 0:

- When $m = 1$ or $m = 2$, both processes immediately return 0, because the initial difference 0 is congruent to 2 mod m.
- When $m = 3$, there is an execution in which p_0 writes 1 to a_0, p_1 reads this 1 and computes $x = -1 \equiv 2 \pmod 3$ and returns 0, then p_0 reads

0 from a_0, computes $x = 1$, advances a_0 to 2, then re-reads 0 from a_0, computes $x = 2$, and returns 0.

- When $m = 4$, run p_0 until it writes 2 to a_0. It then computes $x = 2$ and returns 0. If we now wake up p_1, it computes $x = -2 \equiv 2 \pmod 4$ and also returns 0.

When $m \geq 5$ and the adversary is oblivious, the implementation works. We need to show both linearizability and termination. We'll start with linearizability.

Observe that in a sequential execution, first process to perform `TAS` returns 0 and the second 1. So we need to show (a) that the processes between them return both values, and (b) that if one process finishes before the other starts, the first process returns 0.

It is immediate from Algorithm F.5 that in any reachable configuration, $\mathsf{myPosition}_i \in \{a_i, a_i + 1\}$, because process i can only increment myPosition at most once before writing its value to a_i.

Below we will assume without loss of generality that p_0 is the first process to perform its last read before returning.

Suppose that p_0 returns 0. This means that p_0 observed $a_1 \equiv a_0 - 2 \pmod m$. So at the time p_0 last read a_1, $\mathsf{myPosition}_1$ was congruent to either $a_0 - 1$ or $a_0 - 2$. This means that on its next read of a_0, p_1 will compute x congruent to either -1 or -2. Because m is at least 5, in neither case will it mistake this difference for 2. If it computed $x \equiv -1$, it returns 1; if it computed $x \equiv -2$, it does not return immediately, but eventually it will flip its coin heads, increment $\mathsf{myPosition}_1$, and return 1. In either case we have that exactly one process returns each value.

Alternatively, suppose that p_0 returns 1. Then p_0 reads $a_1 \equiv a_0 + 1$, and at the time of this read, $\mathsf{myPosition}_1$ is either congruent to $a_0 + 1$ or $a_0 + 2$. In the latter case, p_1 returns 0 after its next read; in the former, p_1 eventually increments $\mathsf{myPosition}_1$ and then returns 0. In either case we again have that exactly one process returns each value.

Now suppose that p_0 runs to completion before p_1 starts. Initially, p_0 sees $a_0 \equiv a_1$, but eventually p_0 increments a_0 enough times that $a_0 - a_1 \equiv 2$; p_0 returns 0.

To show termination (with probability 1), consider any configuration in which neither process has returned. During the next $2k$ steps, at least one process takes k steps. Suppose that during this interval, this fast process increments myPosition at every opportunity, while the other process does not increment myPosition at all (this event occurs with nonzero probability

for any fixed k, because the coin-flips are uncorrelated with the oblivious adversary's choice of which process is fast). Then for k sufficiently large, the fast process eventually sees $a_0 - a_1$ congruent to either 2 or -1 and returns. Since this event occurs with independent nonzero probability in each interval of length $2k$, eventually it occurs.[2]

Once one process has terminated, the other increments myPosition infinitely often, so it too eventually sees a gap of 2 or -1.

For the adaptive adversary, the adversary can prevent the algorithm from terminating. Starting from a state in which both processes are about to read and $a_0 = a_1 = k$, run p_0 until it is about to write $(k+1) \bmod m$ to a_0 (unlike the oblivious adversary, the adaptive adversary can see when this will happen). Then run p_1 until it is about to write $(k+1) \bmod m$ to a_1. Let both writes go through. We are now in a state in which both processes are about to read, and $a_0 = a_1 = (k+1) \bmod m$. So we can repeat this strategy forever.

F.4 Presentation (for students taking CPSC 565): due Wednesday, 2016-04-27

Students taking CPSC 565, the graduate version of the class, are expected to give a 15-minute presentation on a recent paper in the theory of distributed computing.

The choice of paper to present should be made in consultation with the instructor. To a first approximation, any paper from PODC, DISC, or a similar conference in the last two or three years (that is not otherwise covered in class) should work.

Because of the limited presentation time, you are not required to get into all of the technical details of the paper, but your presentation should include[3]

1. Title, authors, and date and venue of publication of the paper.

2. A high-level description of the main result. Unlike a talk for a general

[2] The fancy way to prove this is to invoke the second Borel-Cantelli lemma of probability theory. Or we can just argue that the probability that we don't terminate in the first ℓ intervals is at most $(1-\epsilon)^{\ell}$, which goes to zero in the limit.

[3] Literary theorists will recognize this as a three-act structure (preceded by a title card): introduce the main character, make their life difficult, then resolve their problems in time for the final curtain. This is not the only way to organize a talk, but if done right it has the advantage of keeping the audience awake.

audience, you can assume that your listeners know at least everything that we've talked about so far in the class.

3. A description of where this result fits into the literature (e.g., solves an open problem previously proposed in [...], improves on the previous best running time for an algorithm from [...], gives a lower bound or impossibility result for a problem previously proposed by [...], opens up a new area of research for studying [...]), and why it is interesting and/or hard.

4. A description (also possibly high-level) of the main technical mechanism(s) used to get the main result.

You do not have to prepare slides for your presentation if you would prefer to use the blackboard, but you should make sure to practice it in advance to make sure it fits in the allocated time. The instructor will be happy to offer feedback on draft versions if available far enough before the actual presentation date.

Relevant dates:

2016-04-13 Paper selection due.

2016-04-22 Last date to send draft slides or arrange for a practice presentation with the instructor if you want guaranteed feedback.

2016-04-27 Presentations, during the usual class time.

F.5 CS465/CS565 Final Exam, May 10th, 2016

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are four problems on this exam, each worth 20 points, for a total of 80 points. You have approximately three hours to complete this exam.

F.5.1 A slow register (20 points)

Define a **second-to-last register** as having a read operation that always returns the second-to-last value written to it. For example, after `write` (1), `write` (2), `write` (3), a subsequent `read` operation will return 2. If fewer that two `write` operations have occurred, a `read` will return $\perp$.

What is the consensus number of this object?

Solution

The consensus number of this object is 2.

For two processes, have each process i write its input to a standard atomic register $r[i]$, and then write its ID to a shared second-to-last-value register s. We will have whichever process writes to s first win. After writing, process i can detect which process wrote first by reading s once, because it either sees $\perp$ (meaning the other process has not written yet) or it sees the identity of the process that wrote first. In either case it can return the winning process's input.

For three processes, the usual argument gets us to a configuration C where all three processes are about to execute operations x, y, and z on the same object, where each operation moves from a bivalent to a univalent state. Because we know that this object can't be a standard atomic register, it must be a second-to-last register. We can also argue that all of x, y, and z are writes, because if one of them is not, the processes that don't perform it can't tell if it happened or not.

Suppose that Cx is 0-valent and Cy is 1-valent. Then $Cxyz$ is 0-valent and Cyz is 1-valent. But these configurations are indistinguishable to any process but x. It follows that the second-to-last register can't solve consensus for three processes.

F.5.2 Two leaders (20 points)

Assume that you are working in an asynchronous message-passing system organized as a connected graph, where all processes run the same code except that each process starts with an ID and the knowledge of the IDs of all of its neighbors. Suppose that all of these IDs are unique, except that the smallest ID (whatever it is) might appear on either one or two processes.

Is it possible in all cases to detect which of these situations hold? Either give an algorithm that allows all processes to eventually correctly return whether there are one or two minimum-id processes in an arbitrary connected graph, or show that no such algorithm is possible.

Solution

Here is an algorithm.

If there are two processes p and q with the same ID that are adjacent to each other, they can detect this in the initial configuration, and transmit this fact to all the other processes by flooding.

If these processes p and q are not adjacent, we will need some other mechanism to detect them. Define the extended ID of a process as its own ID followed by a list of the IDs of its neighbors in some fixed order. Order the extended IDs lexicographically, so that a process with a smaller ID also has a smaller extended ID.

Suppose now that p and q are not adjacent and have the same extended ID. Then they share the same neighbors, and each of these neighbors will see that p and q have duplicate IDs. So we can do an initial round of messages where each process transmits its extended ID to its neighbors, and if p and q observe that their ID is a duplicate, they can again notify all the processes to return that there are two leaders by flooding.

The remaining case is that p and q have distinct extended IDs, or that only one minimum-process ID exists. In either case we can run any standard broadcast-based leader-election algorithm, using the extended IDs, which will leave us with a tree rooted at whichever process has the minimum extended ID. This process can then perform convergecast to detect if there is another process with the same ID, and perform broadcast to inform all processes of this fact.

F.5.3 A splitter using one-bit registers (20 points)

Algorithm F.6 implements a splitter-like object using one-bit registers. It assumes that each process has a unique ID ID consisting of $k = \lceil \lg n \rceil$ bits $\mathsf{ID}_{k-1}\mathsf{ID}_{k-2}\ldots\mathsf{ID}_0$. We would like this object to have the properties that (a) if exactly one process executes the algorithm, then it wins; and (b) in any execution, at most one process wins.

Show that the algorithm has these properties, or give an example of an execution where it does not.

Solution

The implementation is correct.

If one process runs alone, it sets $A[i][\mathsf{ID}_i]$ for each i, sees 0 in door, then sees 0 in each location $A[i][\neg\mathsf{ID}_i]$ and wins. So we have property (a).

Now suppose that some process with ID p wins in an execution that may involve other processes. Then p writes $A[i][p_i]$ for all i before observing 0 in door, which means that it sets all these bits before any process writes 1 to door. If some other process q also wins, then there is at least one position i where $p_i = \neg q_i$, and q reads $A[i][p_i]$ after writing 1 to door. But then q sees 1 in this location and loses, a contradiction.

```
   shared data:
1  one-bit atomic registers A[i][j] for i = 0 ... ⌈lg n⌉ − 1 and j ∈ {0, 1}, all
   initially 0
2  one-bit atomic register door, initially 0
3  procedure splitter(ID)
4      for i ← 0 to k − 1 do
5          A[i][ID_i] ← 1
6      if door = 1 then
7          return lose
8      door ← 1
9      for i ← 0 to k − 1 do
10         if A[i][¬ID_i] = 1 then
11             return lose
12     return win
```

Algorithm F.6: Splitter using one-bit registers

F.5.4 Symmetric self-stabilizing consensus (20 points)

Suppose we have a synchronous system consisting of processes organized in a connected graph. The state of each process is a single bit, and each process can directly observe the number of neighbors that it has and how many of them have 0 bits and how many have 1 bits. At each round, a process counts the number of neighbors k_0 with zeros, the number k_1 with ones, and its own bit b, and chooses a new bit for the next round $f(b, k_0, k_1)$ according to some rule f that is the same for all processes. The goal of the processes is to reach consensus, where all processes have the same bit forever, starting from an arbitrary initial configuration. An example of a rule that has this property is for f to output 1 if $b = 1$ or $k_1 > 0$.

However, this rule is not symmetric with respect to bit values: if we replace all ones by zeros and vice versa, we get different behavior.

Prove or disprove: There exists a rule f that is symmetric, by which we mean that $f(b, k_0, k_1) = \neg f(\neg b, k_1, k_0)$ always, such that applying this rule starting from an arbitrary configuration in an arbitrary graph eventually converges to all processes having the same bit forever.

Solution

Disproof by counterexample: Fix some f, and consider a graph with two processes p_0 and p_1 connected by an edge. Let p_0 start with 0 and p_1 start with 1. Then p_0's next state is $f(0,0,1) = \neg f(1,1,0) \neq f(1,1,0)$, which is p_1's next state. So either p_0 still has 0 and p_1 still has 1, in which case we never make progress; or they swap their bits, in which case we can apply the same analysis with p_0 and p_1 reversed to show that they continue to swap back and forth forever. In either case the system does not converge.

Appendix G

Sample assignments from Spring 2014

G.1 Assignment 1: due Wednesday, 2014-01-29, at 5:00pm

Bureaucratic part

Send me email! My address is james.aspnes@gmail.com.

In your message, include:

1. Your name.
2. Your status: whether you are an undergraduate, grad student, auditor, etc.
3. Anything else you'd like to say.

(You will not be graded on the bureaucratic part, but you should do it anyway.)

G.1.1 Counting evil processes

A connected bidirectional asynchronous network of n processes with identities has diameter D and may contain zero or more evil processes. Fortunately, the evil processes, if they exist, are not Byzantine, fully conform to RFC 3514 [Bel03], and will correctly execute any code we provide for them.

Suppose that all processes wake up at time 0 and start whatever protocol we have given them. Suppose that each process initially knows whether it is

evil, and knows the identities of all of its neighbors. However, the processes do not know the number of processes n or the diameter of the network D.

Give a protocol that allows every process to correctly return the number of evil processes no later than time D. Your protocol should only return a value once for each process (no converging to the correct answer after an initial wrong guess).

Solution

There are a lot of ways to do this. Since the problem doesn't ask about message complexity, we'll do it in a way that optimizes for algorithmic simplicity.

At time 0, each process initiates a separate copy of the flooding algorithm (Algorithm 3.1). The message $\langle p, N(p), e\rangle$ it distributes consists of its own identity, the identities of all of its neighbors, and whether or not it is evil.

In addition to the data for the flooding protocol, each process tracks a set I of all processes it has seen that initiated a protocol and a set N of all processes that have been mentioned as neighbors. The initial values of these sets for process p are $\{p\}$ and $N(p)$, the neighbors of p.

Upon receiving a message $\langle q, N(q), e\rangle$, a process adds q to I and $N(q)$ to N. As soon as $I = N$, the process returns a count of all processes for which $e =$ **true**.

Termination by D: Follows from the same analysis as flooding. Any process at distance d from p has $p \in I$ by time d, so I is complete by time D.

Correct answer: Observe that $N = \bigcup_{i \in I} N(i)$ always. Suppose that there is some process q that is not in I. Since the graph is connected, there is a path from p to q. Let r be the last node in this path in I, and let s be the following node. Then $s \in N \setminus I$ and $N \neq I$. By contraposition, if $I = N$ then I contains all nodes in the network, and so the count returned at this time is correct.

G.1.2 Avoiding expensive processes

Suppose that you have a bidirectional but not necessarily complete asynchronous message-passing network represented by a graph $G = (V, E)$ where each node in V represents a process and each edge in E connects two processes that can send messages to each other. Suppose further that each process is assigned a weight 1 or 2. Starting at some initiator process, we'd like to construct a shortest-path tree, where each process points to one of

its neighbors as its parent, and following the parent pointers always gives a path of minimum total weight to the initiator.[1]

Give a protocol that solves this problem with reasonable time, message, and bit complexity, and show that it works.

Solution

There's an ambiguity in the definition of total weight: does it include the weight of the initiator and/or the initial node in the path? But since these values are the same for all paths to the initiator from a given process, they don't affect which is lightest.

If we don't care about bit complexity, there is a trivial solution: Use an existing BFS algorithm followed by convergecast to gather the entire structure of the network at the initiator, run your favorite single-source shortest-path algorithm there, then broadcast the results. This has time complexity $O(D)$ and message complexity $O(DE)$ if we use the BFS algorithm from §4.3. But the last couple of messages in the convergecast are going to be pretty big.

A solution by reduction: Suppose that we construct a new graph G' where each weight-2 node u in G is replaced by a clique of nodes $u_1, u_2, \ldots u_k$, with each node in the clique attached to a different neighbor of u. We then run any breadth-first search protocol of our choosing on G', where each weight-2 node simulates all members of the corresponding clique. Because any path that passes through a clique picks up an extra edge, each path in the breadth-first search tree has a length exactly equal to the sum of the weights of the nodes other than its endpoints.

A complication is that if I am simulating k nodes, between them they may have more than one parent pointer. So we define $u.\mathsf{parent}$ to be $u_i.\mathsf{parent}$ where u_i is a node at minimum distance from the initiator in G'. We also re-route any incoming pointers to $u_j \neq u_i$ to point to u_i instead. Because u_i was chosen to have minimum distance, this never increases the length of any path, and the resulting modified tree is a still a shortest-path tree.

Adding nodes blows up $|E'|$, but we don't need to actually send messages between different nodes u_i represented by the same process. So if we use the §4.3 algorithm again, we only send up to D messages per real edge, giving $O(D)$ time and $O(DE)$ messages.

If we don't like reductions, we could also tweak one of our existing algorithms. Gallager's layered BFS (§4.2) is easily modified by changing the

[1] Clarification added 2014-01-26: The actual number of hops is not relevant for the construction of the shortest-path tree. By shortest path, we mean path of minimum total weight.

depth bound for each round to a total-weight bound. The synchronizer-based BFS can also be modified to work, but the details are messy.

G.2 Assignment 2: due Wednesday, 2014-02-12, at 5:00pm

G.2.1 Synchronous agreement with weak failures

Suppose that we modify the problem of synchronous agreement with crash failures from Chapter 9 so that instead of crashing a process forever, the adversary may jam some or all of its outgoing messages for a single round. The adversary has limited batteries on its jamming equipment, and can only cause f such one-round faults. There is no restriction on when these one-round jamming faults occur: the adversary might jam f processes for one round, one process for f rounds, or anything in between, so long as the sum over all rounds of the number of processes jammed in each round is at most f. For the purposes of agreement and validity, assume that a process is non-faulty if it is never jammed.[2]

As a function of f and n, how many rounds does it take to reach agreement in the worst case in this model, under the usual assumptions that processes are deterministic and the algorithm must satisfy agreement, termination, and validity? Give the best upper and lower bounds that you can.

Solution

The par solution for this is an $\Omega(\sqrt{f})$ lower bound and $O(f)$ upper bound. I don't know if it is easy to do better than this.

For the lower bound, observe that the adversary can simulate an ordinary crash failure by jamming a process in every round starting in the round it crashes in. This means that in an r-round protocol, we can simulate k crash failures with kr jamming faults. From the Dolev-Strong lower bound [DS83] (see also Chapter 9), we know that there is no r-round protocol with $k = r$ crash failures faults, so there is no r-round protocol with r^2 jamming faults. This gives a lower bound of $\lfloor\sqrt{f}\rfloor + 1$ on the number of rounds needed to solve synchronous agreement with f jamming faults.[3]

[2] Clarifications added 2014-02-10: We assume that processes don't know that they are being jammed or which messages are lost (unless the recipient manages to tell them that a message was not delivered). As in the original model, we assume a complete network and that all processes have known identities.

[3] Since Dolev-Strong only needs to crash one process per round, we don't really need

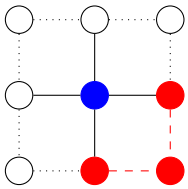


Figure G.1: Connected Byzantine nodes take over half a cut

For the upper bound, have every process broadcast its input every round. After $f+1$ rounds, there is at least one round in which no process is jammed, so every process learns all the inputs and can take, say, the majority value.

G.2.2 Byzantine agreement with contiguous faults

Suppose that we restrict the adversary in Byzantine agreement to corrupting a connected subgraph of the network; the idea is that the faulty nodes need to coordinate, but can't relay messages through the non-faulty nodes to do so.

Assume the usual model for Byzantine agreement with a network in the form of an $m \times m$ torus. This means that each node has a position (x, y) in $\{0, \ldots, m-1\} \times \{0, \ldots, m-1\}$, and its neighbors are the four nodes $(x+1 \bmod m, y)$, $(x-1 \bmod m, y)$, $(x, y+1 \bmod m)$, and $(x, y-1 \bmod m)$.

For sufficiently large m,[4] what is the largest number of faults f; that this system can tolerate and still solve Byzantine agreement?

Solution

The relevant bound here is the requirement that the network have enough connectivity that the adversary can't take over half of a vertex cut (see §10.1.3). This is complicated slightly by the requirement that the faulty nodes be contiguous.

The smallest vertex cut in a sufficiently large torus consists of the four neighbors of a single node; however, these nodes are not connected. But we can add a third node to connect two of them (see Figure G.1).

By adapting the usual lower bound we can use this construction to show that $f = 3$ faults are enough to prevent agreement when $m \geq 3$. The question

the full r jamming faults for processes that crash late. This could be used to improve the constant for this argument.

[4] Problem modified 2014-02-03. In the original version, it asked to compute f for all m, but there are some nasty special cases when m is small.

then is whether $f = 2$ faults is enough.

By a case analysis, we can show that any two nodes in a sufficiently large torus are either adjacent themselves or can be connected by three paths, where no two paths have adjacent vertices. Assume without loss of generality that one of the nodes is at position $(0, 0)$. Then any other node is covered by one of the following cases:

1. Nodes adjacent to $(0, 0)$. These can communicate directly.

2. Nodes at $(0, i)$ or $(i, 0)$. These cases are symmetric, so we'll describe the solution for $(0, i)$. Run one path directly north: $(0, 1), (0, 2), \ldots, (0, i - 1)$. Similarly, run a second path south: $(0, -1), (0, -2), \ldots (0, i + 1)$. For the third path, take two steps east and then run north and back west: $(1, 0), (2, 0), (2, 1), (2, 2), \ldots, (2, i), (1, i)$. These paths are all non-adjacent as long as $m \geq 4$.

3. Nodes at $(\pm 1, i)$ or $(i, \pm 1)$, where i is not -1, 0, or 1. Suppose the node is at $(1, i)$. Run one path east then north through $(1, 0), (1, 1), \ldots, (1, i - 1)$. The other two paths run south and west, with a sideways jog in the middle as needed. This works for m sufficiently large to make room for the sideways jogs.

4. Nodes at $(\pm 1, \pm 1)$ or (i, j) where neither of i or j is -1, 0, or 1. Now we can run one path north then east, one east then north, one south then west, and one west then south, creating four paths in a figure-eight pattern centered on $(0, 0)$.

G.3 Assignment 3: due Wednesday, 2014-02-26, at 5:00pm

G.3.1 Among the elect

The adversary has decided to be polite and notify each non-faulty processes when he gives up crashing it. Specifically, we have the usual asynchronous message-passing system with up to f faulty processes, but every non-faulty process is eventually told that it is non-faulty. (Faulty processes are told nothing.)

For what values of f can you solve consensus in this model?

Solution

We can tolerate $f < n/2$, but no more.

If $f < n/2$, the following algorithm works: Run Paxos, where each process i waits to learn that it is non-faulty, then acts as a proposer for proposal number i. The highest-numbered non-faulty process then carries out a proposal round that succeeds because no higher proposal is ever issued, and both the proposer (which is non-faulty) and a majority of accepters participate.

If $f \geq n/2$, partition the processes into two groups of size $\lfloor n/2 \rfloor$, with any leftover process crashing immediately. Make all of the processes in both groups non-faulty, and tell each of them this at the start of the protocol. Now do the usual partitioning argument: Run group 0 with inputs 0 with no messages delivered from group 1 until all processes decide 0 (we can do this because the processes can't distinguish this execution from one in which the group 1 processes are in fact faulty). Run group 1 similarly until all processes decide 1. We have then violated agreement, assuming we didn't previously violate termination of validity.

G.3.2 Failure detectors on the cheap

Suppose we do not have the budget to equip all of our machines with failure detectors. Instead, we order an eventually strong failure detector for k machines, and the remaining $n - k$ machines get fake failure detectors that never suspect anybody. The choice of which machines get the real failure detectors and which get the fake ones is under the control of the adversary.

This means that every faulty process is eventually permanently suspected by every non-faulty process with a real failure detector, and there is at least one non-faulty process that is eventually permanently not suspected by anybody. Let's call the resulting failure detector $\Diamond S_k$.

Let f be the number of actual failures. Under what conditions on f, k, and n can you still solve consensus in the usual deterministic asynchronous message-passing model using $\Diamond S_k$?

Solution

First observe that $\Diamond S$ can simulate $\Diamond S_k$ for any k by having $n - k$ processes ignore the output of their failure detectors. So we need $f < n/2$ by the usual lower bound on $\Diamond S$.

If $f \geq k$, we are also in trouble. The $f > k$ case is easy: If there exists a consensus protocol for $f > k$, then we can transform it into a consensus protocol for $n - k$ processes and $f - k$ failures, with no failure detectors at all, by pretending that there are an extra k processes with real failure detectors

that crash immediately. The FLP impossibility result rules this out.

If $f = k$, we have to be a little more careful. By immediately crashing $f - 1$ processes with real failure detectors, we can reduce to the $f = k = 1$ case. Now the adversary runs the FLP strategy. If no processes crash, then all $n - k + 1$ surviving process report no failures; if it becomes necessary to crash a process, this becomes the one remaining process with the real failure detector. In either case the adversary successfully prevents consensus.

So let $f < k$. Then we have weak completeness, because every faulty process is eventually permanently suspected by at least $k - f > 0$ processes. We also have weak accuracy, because it is still the case that some process is eventually permanently never suspected by anybody. By boosting weak completeness to strong completeness as described in §13.2.3, we can turn out failure detector into $\Diamond S$, meaning we can solve consensus precisely when $f < \min(k, n/2)$.

G.4 Assignment 4: due Wednesday, 2014-03-26, at 5:00pm

G.4.1 A global synchronizer with a global clock

Consider an asynchronous message-passing system with n processes in a bidirectional ring with no failures. Suppose that the processes are equipped with a global clock, which causes a local event to occur simultaneously at each process every c time units, where as usual 1 is the maximum message delay. We would like to use this global clock to build a global synchronizer. Provided c is at least 1, a trivial approach is to have every process advance to the next round whenever the clock pulse hits. This gives one synchronous round every c time units.

Suppose that c is greater than 1 but still $o(n)$. Is it possible to build a global synchronizer in this model that runs more than a constant ratio faster than this trivial global synchronizer in the worst case?

Solution

No. We can adapt the lower bound on the session problem from §7.4.2 to apply in this model.

Consider an execution of an algorithm for the session problem in which each message is delivered exactly one time unit after it is sent. Divide it as in the previous proof into a prefix β containing special actions and a suffix δ containing no special actions. Divide β further into segments

$\beta_1, \beta_2, \beta_3, \ldots, \beta_k$, where each segment ends with a clock pulse. Following the standard argument, because each segment has duration less than the diameter of the network, there is no causal connection between any special actions done by processes at opposite ends of the network that are in the same segment β_i. So we can causally shuffle each β_i to get a new segment β_i' where all special actions of process p_0 (say) occur before all special actions of process $p_{n/2}$. This gives at most one session per segment, or at most one session for every c time units.

Since a globally synchronous system can do one session per round, this means that our global synchronizer can only produce one session per c time units as well.

G.4.2 A message-passing counter

A **counter** is a shared object that support operations `inc` and `read`, where `read` returns the number of previous `inc` operations.

Algorithm G.1 purports to implement a counter in an asynchronous message-passing system subject to $f < n/2$ crash failures. In the algorithm, each process i maintains a vector c_i of contributions to the counter from all the processes, as well as a nonce r_i used to distinguish responses to different read operations from each other. All of these values are initially zero.

Show that the implemented counter is linearizable, or give an example of an execution where it isn't.

Solution

This algorithm is basically implementing an array of ABD registers [ABND95], but it omits the second phase on a `read` where any information the reader learns is propagated to a majority. So we expect it to fail the same way ABD would without this second round, by having two `read` operations return values that are out of order with respect to their observable ordering.

Here is one execution that produces this bad outcome:

1. Process p_1 starts an `inc` by updating $c_1[1]$ to 1.

2. Process p_2 carries out a `read` operation in which it receives responses from p_1 and p_2, and returns 1.

3. After p_2 finishes, process p_3 carries out a `read` operation in which it receives responses from p_2 and p_3, and returns 0.

```
1  procedure inc
2      c_i[i] ← c_i[i] + 1
3      Send c_i[i] to all processes.
4      Wait to receive ack(c_i[i]) from a majority of processes.
5  upon receiving c from j do
6      c_i[j] ← max(c_i[j], c)
7      Send ack(c) to j.
8  procedure read
9      r_i ← r_i + 1
10     Send read(r_i) to all processes.
11     Wait to receive respond(r_i, c_j) from a majority of processes j.
12     return ∑_k max_j c_j[k]
13 upon receiving read(r) from j do
14     Send respond(r, c_i) to j
```

Algorithm G.1: Counter algorithm for Problem G.4.2.

If we want to be particularly perverse, we can exploit the fact that p_2 doesn't record what it learns in its first `read` to have p_2 do the second `read` that returns 0 instead of p_3. This shows that Algorithm G.1 isn't even sequentially consistent.

The patch, if we want to fix this, is to include the missing second phase from ABD in the `read` operation: after receiving values from a majority, I set $c_i[k]$ to $\max_j c_j[k]$ and send my updated values to a majority. That the resulting counter is linearizable is left as an exercise.

G.5 Assignment 5: due Wednesday, 2014-04-09, at 5:00pm

G.5.1 A concurrency detector

Consider the following optimistic mutex-like object, which we will call a **concurrency detector**. A concurrency detector supports two operations for each process i, $\texttt{enter}_i$ and $\texttt{exit}_i$. These operations come in pairs: a process enters a critical section by executing $\texttt{enter}_i$, and leaves by executing $\texttt{exit}_i$. The behavior of the object is undefined if a process calls $\texttt{enter}_i$ twice without an intervening $\texttt{exit}_i$, or calls $\texttt{exit}_i$ without first calling $\texttt{enter}_i$.

Unlike mutex, a concurrency detector does not enforce that only one

process is in the critical section at a time; instead, $\texttt{exit}_i$ returns 1 if the interval between it and the previous $\texttt{enter}_i$ overlaps with some interval between a $\texttt{enter}_j$ and corresponding $\texttt{exit}_j$ for some $j \neq i$, and returns 0 if there is no overlap.

Is there a deterministic linearizable wait-free implementation of a concurrency detector from atomic registers? If there is, give an implementation. If there is not, give an impossibility proof.

Solution

It is not possible to implement this object using atomic registers.

Suppose that there were such an implementation. Algorithm G.2 implements two-process consensus using a two atomic registers and a single concurrency detector, initialized to the state following $\texttt{enter}_1$.

```
1  procedure consensus1(v)
2      r1 ← v
3      if exit1() = 1 then
4          return r2
5      else
6          return v
7  procedure consensus2(v)
8      r2 ← v
9      enter2()
10     if exit2() = 1 then
11         return v
12     else
13         return r1
```

Algorithm G.2: Two-process consensus using the object from Problem G.5.1

Termination is immediate from the absence of loops in the code.

To show validity and termination, observe that one of two cases holds:

1. Process 1 executes $\texttt{exit}_1$ before process 2 executes $\texttt{enter}_2$. In this case there is no overlap between the interval formed by the implicit $\texttt{enter}_1$ and $\texttt{exit}_1$ and the interval formed by $\texttt{enter}_2$ and $\texttt{exit}_2$. So the $\texttt{exit}_1$ and $\texttt{exit}_2$ operations both return 0, causing process 1 to return its own value and process 2 to return the contents of r_1. These

will equal process 1's value, because process 2's read follows its call to $\texttt{enter}_2$, which follows $\texttt{exit}_1$ and thus process 1's write to r_1.

2. Process 1 executes $\texttt{exit}_1$ after process 2 executes $\texttt{enter}_2$. Now both `exit` operations return 1, and so process 2 returns its own value while process 1 returns the contents of r_2, which it reads after process 2 writes its value there.

In either case, both processes return the value of the first process to access the concurrency detector, satisfying both agreement and validity. This would give a consensus protocol for two processes implemented from atomic registers, contradicting the impossibility result of Loui and Abu-Amara [LAA87].

G.5.2 Two-writer sticky bits

A **two-writer sticky bit** is a sticky bit that can be read by any process, but that can only be written to by two specific processes.

Suppose that you have an unlimited collection of two-writer sticky bits for each pair of processes, plus as many ordinary atomic registers as you need. What is the maximum number of processes for which you can solve wait-free binary consensus?

Solution

If $n = 2$, then a two-writer sticky bit is equivalent to a sticky bit, so we can solve consensus.

If $n \geq 3$, suppose that we maneuver our processes as usual to a bivalent configuration C with no bivalent successors. Then there are three pending operations x, y, and z, that among them produce both 0-valent and 1-valent configurations. Without loss of generality, suppose that Cx and Cy are both 0-valent and Cz is 1-valent. We now consider what operations these might be.

1. If x and z apply to different objects, then $Cxz = Czx$ must be both 0-valent and 1-valent, a contradiction. Similarly if y and z apply to different objects. This shows that all three operations apply to the same object O.

2. If O is a register, then the usual case analysis of Loui and Abu-Amara [LAA87] gives us a contradiction.

3. If O is a two-writer sticky bit, then we can split cases further based on z:

(a) If z is a read, then either:
   i. At least one of x and y is a read. But then $Cxz = Czx$ or $Cyz = Czy$, and we are in trouble.
   ii. Both x and y are writes. But then Czx (1-valent) is indistinguishable from Cx (0-valent) by the two processes that didn't perform z: more trouble.

(b) If z is a write, then at least one of x or y is a read; suppose it's x. Then Cxz is indistinguishable from Cz by the two processes that didn't perform x.

Since we reach a contradiction in all cases, it must be that when $n \geq 3$, every bivalent configuration has a bivalent successor, which shows that we can't solve consensus in this case. The maximum value of n for which we can solve consensus is 2.

G.6 Assignment 6: due Wednesday, 2014-04-23, at 5:00pm

G.6.1 A rotate register

Suppose that you are asked to implement a concurrent m-bit register that supports in addition to the usual `read` and `write` operations a `RotateLeft` operation that rotates all the bits to the left; this is equivalent to doing a left shift (multiplying the value in the register by two) followed by replacing the lowest-order bit with the previous highest-order bit.

For example, if the register contains 1101, and we do `RotateLeft`, it now contains 1011.

Show that if m is sufficiently large as a function of the number of processes n, $\Theta(n)$ steps per operation in the worst case are necessary and sufficient to implement a linearizable wait-free m-bit shift register from atomic registers.

Solution

The necessary part is easier, although we can't use JTT (Chapter 21) directly because having write operations means that our rotate register is not perturbable. Instead, we argue that if we initialize the register to 1, we get a mod-m counter, where increment is implemented by `RotateLeft` and read is implemented by taking the log of the actual value of the counter. Letting $m \geq 2n$ gives the desired $\Omega(n)$ lower bound, since a mod-$2n$ counter *is* perturbable.

For sufficiency, we'll show how to implement the rotate register using snapshots. This is pretty much a standard application of known techniques [AH90b, AM93], but it's not a bad exercise to write it out.

Pseudocode for one possible solution is given in Algorithm G.3.

The register is implemented using a single snapshot array A. Each entry in the snapshot array holds four values: a timestamp and process ID indicating which write the process's most recent operations apply to, the initial write value corresponding to this timestamp, and the number of rotate operations this process has applied to this value. A write operation generates a new timestamp, sets the written value to its input, and resets the rotate count to 0. A rotate operation updates the timestamp and associated write value to the most recent that the process sees, and adjusts the rotate count as appropriate. A read operation combines all the rotate counts associated with the most recent write to obtain the value of the simulated register.

```
1  procedure write(A, v)
2      s ← snapshot(A)
3      A[id] ← ⟨max_i s[i].timestamp + 1, id, v, 0⟩
4  procedure RotateLeft(A)
5      s ← snapshot(A)
6      Let i maximize ⟨s[i].timestamp, s[i].process⟩
7      if s[i].timestamp = A[id].timestamp and
       s[i].process = A[id].process then
           // Increment my rotation count
8          A[id].rotations ← A[id].rotations + 1
9      else
           // Reset and increment my rotation count
10         A[id] ← ⟨s[i].timestamp, s[i].process, s[i].value, 1⟩
11 procedure read(A)
12     s ← snapshot(A)
13     Let i maximize ⟨s[i].timestamp, s[i].process⟩
14     Let
       r = ∑_{j,s[j].timestamp=s[i].timestamp∧s[j].process=s[i].process} s[j].rotations
15     return s[i].value rotated r times.
```

Algorithm G.3: Implementation of a rotate register

Since each operation requires one snapshot and at most one update, the cost is $O(n)$ using the linear-time snapshot algorithm of Inoue *et al.* [IMCT94].

Linearizability is easily verified by observing that ordering all operations by the maximum timestamp/process tuple that they compute and then by the total number of rotations that they observe produces an ordering consistent with the concurrent execution for which all return values of reads are correct.

G.6.2 A randomized two-process test-and-set

Algorithm G.4 gives pseudocode for a protocol for two processes p_0 and p_1. It uses two shared unbounded single-writer atomic registers r_0 and r_1, both initially 0. Each process also has a local variable s.

```
1  procedure TAS_i()
2      while true do
3          with probability 1/2 do
4              r_i ← r_i + 1
5          else
6              r_i ← r_i
7          s ← r_¬i
8          if s > r_i then
9              return 1
10         else if s < r_i − 1 do
11             return 0
```

Algorithm G.4: Randomized two-process test-and-set for G.6.2

1. Show that any return values of the protocol are consistent with a linearizable, single-use test-and-set.

2. Will this protocol always terminate with probability 1 assuming an oblivious adversary?

3. Will this protocol always terminate with probability 1 assuming an adaptive adversary?

Solution

1. To show that this implements a linearizable test-and-set, we need to show that exactly one process returns 0 and the other 1, and that if one process finishes before the other starts, the first process to go returns 1.

Suppose that p_i finishes before $p_{\neg i}$ starts. Then p_i reads only 0 from $r_{\neg i}$, and cannot observe $r_i < r_{\neg i}$: p_i returns 0 in this case.

We now show that the two processes cannot return the same value. Suppose that both processes terminate. Let i be such that p_i reads $r_{\neg i}$ for the last time before $p_{\neg i}$ reads r_i for the last time. If p_i returns 0, then it observes $r_i \geq r_{\neg i} + 2$ at the time of its read; $p_{\neg i}$ can increment $r_{\neg i}$ at most once before reading r_i again, and so observed $r_{\neg i} < r_i$ and returns 1.

Alternatively, if p_i returns 1, it observed $r_i < r_{\neg i}$. Since it performs no more increments on r_i, p_i also observes $r_i < r_{\neg i}$ in all subsequent reads, and so cannot also return 1.

2. Let's run the protocol with an oblivious adversary, and track the value of $r_0^t - r_1^t$ over time, where r_i^t is the value of r_i after t writes (to either register). Each write to r_0 increases this value by 1/2 on average, with a change of 0 or 1 equally likely, and each write to r_1 decreases it by 1/2 on average.

   To make things look symmetric, let Δ^t be the change caused by the t-th write and write Δ^t as $c^t + X^t$ where $c^t = \pm 1/2$ is a constant determined by whether p_0 or p_1 does the t-th write and $X^t = \pm 1/2$ is a random variable with expectation 0. Observe that the X^t variables are independent of each other and the constants c^t (which depend only on the schedule).

   For the protocol to run forever, at every time t it must hold that $\left|r_0^t - r_1^t\right| \leq 3$; otherwise, even after one or both processes does its next write, we will have $\left|r_0^{t'} - r_1^{t'}\right|$ and the next process to read will terminate. But

$$\begin{aligned}\left|r_0^t - r_1^t\right| &= \left|\sum_{s=1}^{t} \Delta^s\right| \\ &= \left|\sum_{s=1}^{t} (c_s + X_s)\right| \\ &= \left|\sum_{s=1}^{t} c_s + \sum_{s=1}^{t} X_s\right|.\end{aligned}$$

   The left-hand sum is a constant, while the right-hand sum has a binomial distribution. For any fixed constant, the probability that a binomial distribution lands within ± 2 of the constant goes to zero in

the limit as $t \to \infty$, so with probability 1 there is some t for which this event does not occur.

3. For an adaptive adversary, the following strategy prevents agreement:

   (a) Run p_0 until it is about to increment r_0.

   (b) Run p_1 until it is about to increment r_1.

   (c) Allow both increments to proceed and repeat.

   The effect is that both processes always observe $r_0 = r_1$ whenever they do a read, and so never finish. This works because the adaptive adversary can see the coin-flips done by the processes before they act on them; it would not work with an oblivious adversary or in a model that supported probabilistic writes.

G.7 CS465/CS565 Final Exam, May 2nd, 2014

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are four problems on this exam, each worth 20 points, for a total of 80 points. You have approximately three hours to complete this exam.

G.7.1 Maxima (20 points)

Some deterministic processes organized in an anonymous, synchronous ring are each given an integer input (which may or may not be distinct from other processes' inputs), but otherwise run the same code and do not know the size of the ring. We would like the processes to each compute the maximum input. As usual, each process may only return an output once, and must do so after a finite number of rounds, although it may continue to participate in the protocol (say, by relaying messages) even after it returns an output.

Prove or disprove: It is possible to solve this problem in this model.

Solution

It's not possible.

Consider an execution with $n = 3$ processes, each with input 0. If the protocol is correct, then after some finite number of rounds t, each process returns 0. By symmetry, the processes all have the same states and send the same messages throughout this execution.

Now consider a ring of size $2(t+1)$ where every process has input 0, except for one process p that has input 1. Let q be the process at maximum distance from p. By induction on r, we can show that after r rounds of communication, every process that is more than $r+1$ hops away from p has the same state as all of the processes in the 3-process execution above. So in particular, after t rounds, process q (at distance $t+1$) is in the same state as it would be in the 3-process execution, and thus it returns 0. But—as it learns to its horror, one round too late—the correct maximum is 1.

G.7.2 Historyless objects (20 points)

Recall that a shared-memory object is **historyless** if any operation on the object either (a) always leaves the object in the same state as before the operation, or (b) always leaves the object in a new state that doesn't depend on the state before the operation.

What is the maximum possible consensus number for a historyless object? That is, for what value n is it possible to solve wait-free consensus for n processes using some particular historyless object but not possible to solve wait-free consensus for $n+1$ processes using any historyless object?

Solution

Test-and-sets are (a) historyless, and (b) have consensus number 2, so n is at least 2.

To show that no historyless object can solve wait-free 3-process consensus, consider an execution that starts in a bivalent configuration and runs to a configuration C with two pending operations x and y such that Cx is 0-valent and Cy is 1-valent. By the usual arguments x and y must both be operations on the same object. If either of x and y is a read operation, then (0-valent) Cxy and (1-valent) Cyx are indistinguishable to a third process p_z if run alone, because the object is left in the same state in both configurations; whichever way p_z decides, it will give a contradiction in an execution starting with one of these configurations. If neither of x and y is a read, then x overwrites y, and Cx is indistinguishable from $Cyx$to p_z if p_z runs alone; again we get a contradiction.

G.7.3 Hams (20 points)

Hamazon, LLC, claims to be the world's biggest delivery service for canned hams, with guaranteed delivery of a canned ham to your home anywhere on Earth via suborbital trajectory from secret launch facilities at the North

and South Poles. Unfortunately, these launch facilities may be subject to crash failures due to inclement weather, trademark infringement actions, or military retaliation for misdirected hams.

For this problem, you are to evaluate Hamazon's business model from the perspective of distributed algorithms. Consider a system consisting of a client process and two server processes (corresponding to the North and South Pole facilities) that communicate by means of asynchronous message passing. In addition to the usual message-passing actions, each server also has an irrevocable `launch` action that launches a ham at the client. As with messages, hams are delivered asynchronously: it is impossible for the client to tell if a ham has been launched until it arrives.

A ham protocol is correct provided (a) a client that orders no ham receives no ham; and (b) a client that orders a ham receives exactly one ham. Show that there can be no correct deterministic protocol for this problem if one of the servers can crash.

Solution

Consider an execution in which the client orders ham. Run the northern server together with the client until the server is about to issue a `launch` action (if it never does so, the client receives no ham when the southern server is faulty).

Now run the client together with the southern server. There are two cases:

1. If the southern server ever issues `launch`, execute both this and the northern server's `launch` actions: the client gets two hams.

2. If the southern server never issues `launch`, never run the northern server again: the client gets no hams.

In either case, the one-ham rule is violated, and the protocol is not correct.[5]

[5]It's tempting to try to solve this problem by reduction from a known impossibility result, like Two Generals or FLP. For these specific problems, direct reductions don't appear to work. Two Generals assumes message loss, but in this model, messages are not lost. FLP needs any process to be able to fail, but in this model, the client never fails. Indeed, we can solve consensus in the Hamazon model by just having the client transmit its input to both servers.

G.7.4 Mutexes (20 points)

A swap register s has an operation `swap`(s, v) that returns the argument to the previous call to `swap`, or $\perp$ if it is the first such operation applied to the register. It's easy to build a mutex from a swap register by treating it as a test-and-set: to grab the mutex, I swap in 1, and if I get back $\perp$ I win (and otherwise try again); and to release the mutex, I put back $\perp$.

Unfortunately, this implementation is not starvation-free: some other process acquiring the mutex repeatedly might always snatch the $\perp$ away just before I try to swap it out. Algorithm G.5 uses a swap object s along with an atomic register r to try to fix this.

```
1 procedure mutex()
2     predecessor ← swap(s, myId)
3     while r ≠ predecessor do
4         try again
      // Start of critical section
5     ...
      // End of critical section
6     r ← myId
```

Algorithm G.5: Mutex using a swap object and register

Prove that Algorithm G.5 gives a starvation-free mutex, or give an example of an execution where it fails. You should assume that s and r are both initialized to $\perp$.

Solution

Because processes use the same ID if they try to access the mutex twice, the algorithm doesn't work.

Here's an example of a bad execution:

1. Process 1 swaps 1 into s and gets $\perp$, reads $\perp$ from r, performs its critical section, and writes 1 to r.

2. Process 2 swaps 2 into s and gets 1, reads 1 from r, and enters the critical section.

3. Process 1 swaps 1 into s and gets 2, and spins waiting to see 2 in r.

4. Process 3 swaps 3 into s and gets 1. Because r is still 1, process 3 reads this 1 and enters the critical section. We now have two processes in the critical section, violating mutual exclusion.

I believe this works if each process adopts a new ID every time it calls `mutex`, but the proof is a little tricky.[6]

[6]The simplest proof I can come up with is to apply an invariant that says that (a) the processes that have executed `swap`(s, myId) but have not yet left the while loop have predecessor values that form a linked list, with the last pointer either equal to $\perp$ (if no process has yet entered the critical section) or the last process to enter the critical section; (b) r is $\perp$ if no process has yet left the critical section, or the last process to leave the critical section otherwise; and (c) if there is a process that is in the critical section, its predecessor field points to the last process to leave the critical section. Checking the effects of each operation shows that this invariant is preserved through the execution, and (a) combined with (c) show that we can't have two processes in the critical section at the same time. Additional work is still needed to show starvation-freedom. It's a good thing this algorithm doesn't work as written.

Appendix H

Sample assignments from Fall 2011

H.1 Assignment 1: due Wednesday, 2011-09-28, at 17:00

Bureaucratic part

Send me email! My address is aspnes@cs.yale.edu.

In your message, include:

1. Your name.
2. Your status: whether you are an undergraduate, grad student, auditor, etc.
3. Anything else you'd like to say.

(You will not be graded on the bureaucratic part, but you should do it anyway.)

H.1.1 Anonymous algorithms on a torus

An $n \times m$ **torus** is a two-dimensional version of a ring, where a node at position (i, j) has a neighbor to the north at $(i, j-1)$, the east at $(i+1, j)$, the south at $(i, j+1)$, and the west at $(i-1, j)$. These values wrap around modulo n for the first coordinate and modulo m for the second; so $(0, 0)$ has neighbors $(0, m-1)$, $(1, 0)$, $(0, 1)$, and $(n-1, 0)$.

Suppose that we have a synchronous message-passing network in the form of an $n \times m$ torus, consisting of anonymous, identical processes that do not know n, m, or their own coordinates, but do have a sense of direction (meaning they can tell which of their neighbors is north, east, etc.).

Prove or disprove: Under these conditions, there is a deterministic[1] algorithm that computes whether $n > m$.

Solution

Disproof: Consider two executions, one in an $n \times m$ torus and one in an $m \times n$ torus where $n > m$ and both n and m are at least 2.[2] Using the same argument as in Lemma 5.1.1, show by induction on the round number that, for each round r, all processes in both executions have the same state. It follows that if the processes correctly detect $n > m$ in the $n \times m$ execution, then they incorrectly report $m > n$ in the $m \times n$ execution.

H.1.2 Clustering

Suppose that k of the nodes in an asynchronous message-passing network are designated as cluster heads, and we want to have each node learn the identity of the nearest head. Given the most efficient algorithm you can for this problem, and compute its worst-case time and message complexities.

You may assume that processes have unique identifiers and that all processes know how many neighbors they have.[3]

Solution

The simplest approach would be to run either of the efficient distributed breadth-first search algorithms from Chapter 4 simultaneously starting at all cluster heads, and have each process learn the distance to all cluster heads at once and pick the nearest one. This gives $O(D^2)$ time and $O(k(E + VD))$ messages if we use layering and $O(D)$ time and $O(kDE)$ messages using local synchronization.

We can get rid of the dependence on k in the local-synchronization algorithm by running it almost unmodified, with the only difference being the attachment of a cluster head ID to the exactly messages. The simplest way to show that the resulting algorithm works is to imagine coalescing

[1] Clarification added 2011-09-28.

[2] This last assumption is not strictly necessary, but it avoids having to worry about what it means when a process sends a message to itself.

[3] Clarification added 2011-09-26.

all cluster heads into a single initiator; the clustering algorithm effectively simulates the original algorithm running in this modified graph, and the same proof goes through. The running time is still $O(D)$ and the message complexity $O(DE)$.

H.1.3 Negotiation

Two merchants A and B are colluding to fix the price of some valuable commodity, by sending messages to each other for r rounds in a synchronous message-passing system. To avoid the attention of antitrust regulators, the merchants are transmitting their messages via carrier pigeons, which are unreliable and may become lost. Each merchant has an initial price p_A or p_B, which are integer values satisfying $0 \leq p \leq m$ for some known value m, and their goal is to choose new prices p'_A and p'_B, where $|p'_A - p'_B| \leq 1$. If $p_A = p_B$ and no messages are lost, they want the stronger goal that $p'_A = p'_B = p_A = p_B$.

Prove the best lower bound you can on r, as a function of m, for all protocols that achieve these goals.

Solution

This is a thinly-disguised version of the Two Generals Problem from Chapter 8, with the agreement condition $p'_A = p'_B$ replaced by an **approximate agreement** condition $|p'_A - p'_B| \leq 1$. We can use a proof based on the indistinguishability argument in §8.2 to show that $r \geq m/2$.

Fix r, and suppose that in a failure-free execution both processes send messages in all rounds (we can easily modify an algorithm that does not have this property to have it, without increasing r). We will start with a sequence of executions with $p_A = p_B = 0$. Let X_0 be the execution in which no messages are lost, X_1 the execution in which A's last message is lost, X_2 the execution in which both A and B's last messages are lost, and so on, with X_k for $0 \leq k \leq 2r$ losing k messages split evenly between the two processes, breaking ties in favor of losing messages from A.

When i is even, X_i is indistinguishable from X_{i+1} by A; it follows that p'_A is the same in both executions. Because we no longer have agreement, it may be that $p'_B(X_i)$ and $p'_B(X_{i+1})$ are not the same as p'_A in either execution; but since both are within 1 of p'_A, the difference between them is at most 2. Next, because X_{i+1} to X_{i+2} are indistinguishable to B, we have $p'_B(X_{i+1}) = p'_B(X_{i+2})$, which we can combine with the previous claim to get $|p'_B(X_i) - p'_B(X_{i+2})|$. A simple induction then gives $p'_B(X_{2r}) \leq 2r$, where

X_{2r} is an execution in which all messages are lost.

Now construct executions X_{2r+1} and X_{2r+2} by changing p_A and p_B to m one at a time. Using essentially the same argument as before, we get $|p'_B(X_{2r}) - p'_B(X_{2r+2})| \leq 2$ and thus $p'_B(X_{2r+2}) \leq 2r + 2$.

Repeat the initial $2r$ steps backward to get to an execution X_{4r+2} with $p_A = p_B = m$ and no messages lost. Applying the same reasoning as above shows $m = p'_B(X_{4r+2}) \leq 4r + 2$ or $r \geq \frac{m-2}{4} = \Omega(m)$.

Though it is not needed for the solution, it is not too hard to unwind the lower bound argument to extract an algorithm that matches the lower bound up to a small constant factor. For simplicity, let's assume m is even.

The protocol is to send my input in the first message and then use $m/2-1$ subsequent acknowledgments, stopping immediately if I ever fail to receive a message in some round; the total number of rounds r is exactly $m/2$. If I receive s messages in the first s rounds, I decide on $\min(p_A, p_B)$ if that value lies in $[m/2 - s, m/2 + s]$ and the nearest endpoint otherwise. (Note that if $s = 0$, I don't need to compute $\min(p_A, p_B)$, and if $s > 0$, I can do so because I know both inputs.)

This satisfies the approximate agreement condition because if I see only s messages, you see at most $s+1$, because I stop sending once I miss a message. So either we both decide $\min(p_A, p_B)$ or we choose endpoints $m/2 \pm s_A$ and $m/2 \pm s_B$ that are within 1 of each other. It also satisfies the validity condition $p'_A = p'_B = p_A = p_B$ when both inputs are equal and no messages are lost (and even the stronger requirement that $p'_A = p'_B$ when no messages are lost), because in this case $[m/2 - s, m/2 + s]$ is exactly $[0, m]$ and both processes decide $\min(p_A, p_B)$.

There is still a factor-of-2 gap between the upper and lower bounds. My guess would be that the correct bound is very close to $m/2$ on both sides, and that my lower bound proof is not quite clever enough.

H.2 Assignment 2: due Wednesday, 2011-11-02, at 17:00

H.2.1 Consensus with delivery notifications

The FLP bound (Chapter 11) shows that we can't solve consensus in an asynchronous system with one crash failure. Part of the reason for this is that only the recipient can detect when a message is delivered, so the other processes can't distinguish between a configuration in which a message has or has not been delivered to a faulty process.

Suppose that we augment the system so that senders are notified immediately when their messages are delivered. We can model this by making the delivery of a single message an event that updates the state of both sender and recipient, both of which may send additional messages in response. Let us suppose that this includes attempted deliveries to faulty processes, so that any non-faulty process that sends a message m is eventually notified that m has been delivered (although it might not have any effect on the recipient if the recipient has already crashed).

1. Show that this system can solve consensus with one faulty process when $n = 2$.

2. Show that this system cannot solve consensus with two faulty processes when $n = 3$.

Solution

1. To solve consensus, each process sends its input to the other. Whichever input is delivered first becomes the output value for both processes.

2. To show impossibility with $n = 3$ and two faults, run the usual FLP proof until we get to a configuration C with events e' and e such that Ce is 0-valent and $Ce'e$ is 1-valent (or vice versa). Observe that e and e' may involve two processes each (sender and receiver), for up to four processes total, but only a process that is involved in both e and e' can tell which happened first. There can be at most two such processes. Kill both, and get that $Ce'e$ is indistinguishable from Cee' for the remaining process, giving the usual contradiction.

H.2.2 A circular failure detector

Suppose we equip processes $0 \dots n-1$ in an asynchronous message-passing system with n processes subject to crash failures with a failure detector that is strongly accurate (no non-faulty process is ever suspected) and causes process $i+1 \pmod n$ to eventually permanently suspect process i if process i crashes. Note that this failure detector is not even weakly complete (if both i and $i+1$ crash, no non-faulty process suspects i). Note also that the ring structure of the failure detector doesn't affect the actual network: even though only process $i+1 \pmod n$ may suspect process i, any process can send messages to any other process.

Prove the best upper and lower bounds you can on the largest number of failures f that allows solving consensus in this system.

Solution

There is an easy reduction to FLP that shows $f \leq n/2$ is necessary (when n is even), and a harder reduction that shows $f < 2\sqrt{n} - 1$ is necessary. The easy reduction is based on crashing every other process; now no surviving process can suspect any other survivor, and we are back in an asynchronous message-passing system with no failure detector and 1 remaining failure (if f is at least $n/2 + 1$).

The harder reduction is to crash every $(\sqrt{n})$-th process. This partitions the ring into $\sqrt{n}$ segments of length $\sqrt{n} - 1$ each, where there is no failure detector in any segment that suspects any process in another segment. If an algorithm exists that solves consensus in this situation, then it does so even if (a) all processes in each segment have the same input, (b) if any process in one segment crashes, all $\sqrt{n} - 1$ process in the segment crash, and (c) if any process in a segment takes a step, all take a step, in some fixed order. Under this additional conditions, each segment can be simulated by a single process in an asynchronous system with no failure detectors, and the extra $\sqrt{n} - 1$ failures in $2\sqrt{n} - 1$ correspond to one failure in the simulation. But we can't solve consensus in the simulating system (by FLP), so we can't solve it in the original system either.

On the other side, let's first boost completeness of the failure detector, by having any process that suspects another transmit this submission by reliable broadcast. So now if any non-faulty process i suspects $i + 1$, all the non-faulty processes will suspect $i + 1$. Now with up to t failures, whenever I learn that process i is faulty (through a broadcast message passing on the suspicion of the underlying failure detector, I will suspect processes $i + 1$ through $i + t - f$ as well, where f is the number of failures I have heard about directly. I don't need to suspect process $i + t - f + 1$ (unless there is some intermediate process that has also failed), because the only way that this process will not be suspected eventually is if every process in the range i to $i + t - f$ is faulty, which can't happen given the bound t.

Now if t is small enough that I can't cover the entire ring with these segments, then there is some non-faulty processes that is far enough away from the nearest preceding faulty process that it is never suspected: this gives us an eventually strong failure detector, and we can solve consensus using the standard Chandra-Toueg $\Diamond S$ algorithm from §13.4 or [CT96]. The inequality I am looking for is $f(t - f) < n$, where the left-hand side is maximized by setting $f = t/2$, which gives $t^2/4 < n$ or $t < \sqrt{2n}$. This leaves a gap of about $\sqrt{2}$ between the upper and lower bounds; I don't know which one can be improved.

I am indebted to Hao Pan for suggesting the $\Theta(\sqrt{n})$ upper and lower bounds, which corrected an error in my original draft solution to this problem.

H.2.3 An odd problem

Suppose that each of n processes in a message-passing system with a complete network is attached to a sensor. Each sensor has two states, *active* and *inactive*; initially, all sensors are off. When the sensor changes state, the corresponding process is notified immediately, and can update its state and send messages to other processes in response to this event. It is also guaranteed that if a sensor changes state, it does not change state again for at least two time units. We would like to detect when an odd number of sensors are active, by having at least one process update its state to set off an alarm at a time when this condition holds.

A correct protocol for this problem should satisfy two conditions:

No false positives If a process sets of an alarm, then an odd number of sensors are active.

Termination If at some time an odd number of sensors are active, and from that point on no sensor changes its state, then some process eventually sets off an alarm.

For what values of n is it possible to construct such a protocol?

Solution

It is feasible to solve the problem for $n < 3$.

For $n = 1$, the unique process sets off its alarm as soon as its sensor becomes active.

For $n = 2$, have each process send a message to the other containing its sensor state whenever the sensor state changes. Let s_1 and s_2 be the state of the two process's sensors, with 0 representing inactive and 1 active, and let p_i set off its alarm if it receives a message s such that $s \oplus s_i = 1$. This satisfies termination, because if we reach a configuration with an odd number of active sensors, the last sensor to change causes a message to be sent to the other process that will cause it to set off its alarm. It satisfies no-false-positives, because if p_i sets off its alarm, then $s_{\neg i} = s$ because at most one time unit has elapsed since $p_{\neg i}$ sent s; it follows that $s_{\neg i} \oplus s_i = 1$ and an odd number of sensors are active.

No such protocol is possible for $n \geq 3$. Make p_1's sensor active. Run the protocol until some process p_i is about to enter an alarm state (this occurs

eventually because otherwise we violate termination). Let p_j be one of p_2 or p_3 with $j \neq i$, activate p_j's sensor (we can do this without violating the once-per-time-unit restriction because it has never previously been activated) and then let p_i set off its alarm. We have now violated no-false-positives.

H.3 Assignment 3: due Friday, 2011-12-02, at 17:00

H.3.1 A restricted queue

Suppose you have an atomic queue Q that supports operations `enq` and `deq`, restricted so that:

- `enq`(Q) always pushes the identity of the current process onto the tail of the queue.
- `deq`(Q) tests if the queue is nonempty and its head is equal to the identity of the current process. If so, it pops the head and returns **true**. If not, it does nothing and returns **false**.

The rationale for these restrictions is that this is the minimal version of a queue needed to implement a starvation-free mutex using Algorithm 18.2.

What is the consensus number of this object?

Solution

The restricted queue has consensus number 1.

Suppose we have 2 processes, and consider all pairs of operations on Q that might get us out of a bivalent configuration C. Let x be an operation carried out by p that leads to a b-valent state, and y an operation by q that leads to a $(\neg b)$-valent state. There are three cases:

- Two `deq` operations. If Q is empty, the operations commute. If the head of the Q is p, then y is a no-op and p can't distinguish between Cx and Cyx. Similarly for q if the head is q.
- One `enq` and one `deq` operation. Suppose x is an `enq` and y a `deq`. If Q is empty or the head is not q, then y is a no-op: p can't distinguish Cx from Cyx. If the head is q, then x and y commute. The same holds in reverse if x is a `deq` and y an `enq`.
- Two `enq` operations. This is a little tricky, because Cxy and Cyx are different states. However, if Q is nonempty in C, whichever process

isn't at the head of Q can't distinguish them, because any `deq` operation returns false and never reaches the newly-enqueued values. This leaves the case where Q is empty in C. Run p until it is poised to do $x' = \texttt{deq}(Q)$ (if this never happens, p can't distinguish Cxy from Cyx); then run q until it is poised to do $y' = \texttt{deq}(Q)$ as well (same argument as for p). Now allow both `deq` operations to proceed in whichever order causes them both to succeed. Since the processes can't tell which `deq` happened first, they can't tell which `enq` happened first either. Slightly more formally, if we let α be the sequence of operations leading up to the two `deq` operations, we've just shown $Cxy\alpha x'y'$ is indistinguishable from $Cyx\alpha y'x'$ to both processes.

In all cases, we find that we can't escape bivalence. It follows that Q can't solve 2-process consensus.

H.3.2 Writable fetch-and-increment

Suppose you are given an unlimited supply of atomic registers and fetch-and-increment objects, where the fetch-and-increment objects are all initialized to 0 and supply *only* a fetch-and-increment operation that increments the object and returns the old value. Show how to use these objects to construct a wait-free, linearizable implementation of an augmented fetch-and-increment that also supports a `write` operation that sets the value of the fetch-and-increment and returns nothing.

Solution

We'll use a snapshot object a to control access to an infinite array f of fetch-and-increments, where each time somebody writes to the implemented object, we switch to a new fetch-and-increment. Each cell in a holds $(\mathsf{timestamp}, \mathsf{base})$, where base is the starting value of the simulated fetch-and-increment. We'll also use an extra fetch-and-increment T to hand out timestamps.

Code is in Algorithm H.1.

Since this is all straight-line code, it's trivially wait-free.

Proof of linearizability is by grouping all operations by timestamp, using $s[i].\mathsf{timestamp}$ for `FetchAndIncrement` operations and t for `write` operations, then putting `write` before `FetchAndIncrement`, then ordering `FetchAndIncrement` by return value. Each group will consist of a $\texttt{write}(v)$ for some v followed by zero or more `FetchAndIncrement` operations, which will return increasing values starting at v since they are just returning values

```
1 procedure FetchAndIncrement()
2     s ← snapshot(a)
3     i ← arg max_i(s[i].timestamp)
4     return f[s[i].timestamp] + s[i].base

5 procedure write(v)
6     t ← FetchAndIncrement(T)
7     a[myId] ← (t, v)
```

Algorithm H.1: Resettable fetch-and-increment

from the underlying `FetchAndIncrement` object; the implementation thus meets the specification.

To show consistency with the actual execution order, observe that timestamps only increase over time and that the use of snapshot means that any process that observes or writes a timestamp t does so at a time later than any process that observes or writes any $t' < t$; this shows the group order is consistent. Within each group, the `write` writes $a[\mathsf{myId}]$ before any `FetchAndIncrement` reads it, so again we have consistency between the `write` and any `FetchAndIncrement` operations. The `FetchAndIncrement` operations are linearized in the order in which they access the underlying $f[\dots]$ object, so we win here too.

H.3.3 A box object

Suppose you want to implement an object representing a $w \times h$ box whose width (w) and height (h) can be increased if needed. Initially, the box is 1×1, and the coordinates can be increased by 1 each using `IncWidth` and `IncHeight` operations. There is also a `GetArea` operation that returns the area $w \cdot h$ of the box.

Give an obstruction-free deterministic implementation of this object from atomic registers that optimizes the worst-case individual step complexity of `GetArea`, and show that your implementation is optimal by this measure up to constant factors.

Solution

Let b be the box object. Represent b by a snapshot object a, where $a[i]$ holds a pair $(\Delta w_i, \Delta h_i)$ representing the number of times process i has executed `IncWidth` and `IncHeight`; these operations simply increment the appropriate

value and update the snapshot object. Let `GetArea` take a snapshot and return $(\sum_i \Delta w_i)(\sum_i \Delta h_i)$; the cost of the snapshot is $O(n)$.

To see that this is optimal, observe that we can use `IncWidth` and `GetArea` to represent `inc` and `read` for a standard counter. The Jayanti-Tan-Toueg bound applies to counters, giving a worst-case cost of $\Omega(n)$ for `GetArea`.

H.4 CS465/CS565 Final Exam, December 12th, 2011

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are four problems on this exam, each worth 20 points, for a total of 80 points. You have approximately three hours to complete this exam.

General clarifications added during exam Assume all processes have unique IDs and know n. Assume that the network is complete in the message-passing model.

H.4.1 Lockable registers (20 points)

Most memory-management units provide the ability to control access to specific memory pages, allowing a page to be marked (for example) read-only. Suppose that we model this by a **lockable register** that has the usual register operations `read`(r) and `write`(r, v) plus an additional operation `lock`(r). The behavior of the register is just like a normal atomic register until somebody calls `lock`(r); after this, any call to `write`(r) has no effect.

What is the consensus number of this object?

Solution

The consensus number is ∞; a single lockable register solves consensus for any number of processes. Code is in Algorithm H.2.

```
1 write(r, input)
2 lock(r)
3 return read(r)
```

Algorithm H.2: Consensus using a lockable register

Termination and validity are trivial. Agreement follows from the fact that whatever value is in r when `lock`(r) is first called will never change, and thus will be read and returned by all processes.

H.4.2 Byzantine timestamps (20 points)

Suppose you have an asynchronous message passing system with exactly one Byzantine process.

You would like the non-faulty processes to be able to acquire an increasing sequence of timestamps. A process should be able to execute the timestamp protocol as often as it likes, and it should be guaranteed that when a process is non-faulty, it eventually obtains a timestamp that is larger than any timestamp returned in any execution of the protocol by a non-faulty process that finishes before the current process's execution started.

Note that there is no bound on the size of a timestamp, so having the Byzantine process run up the timestamp values is not a problem, as long as it can't cause the timestamps to go down.

For what values of n is it possible to solve this problem?

Solution

It is possible to solve the problem for all n except $n = 3$. For $n = 1$, there are no non-faulty processes, so the specification is satisfied trivially. For $n = 2$, there is only one non-faulty process: it can just keep its own counter and return an increasing sequence of timestamps without talking to the other process at all.

For $n = 3$, it is not possible. Consider an execution in which messages between non-faulty processes p and q are delayed indefinitely. If the Byzantine process r acts to each of p and q as it would if the other had crashed, this execution is indistinguishable to p and q from an execution in which r is correct and the other is faulty. Since there is no communication between p and q, it is easy to construct and execution in which the specification is violated.

For $n \geq 4$, the protocol given in Algorithm H.3 works.

The idea is similar to the Attiya, Bar-Noy, Dolev distributed shared memory algorithm [ABND95]. A process that needs a timestamp polls $n - 1$ other processes for the maximum values they've seen and adds 1 to it; before returning, it sends the new timestamp to all other processes and waits to receive $n - 1$ acknowledgments. The Byzantine process may choose not to answer, but this is not enough to block completion of the protocol.

```
1  procedure getTimestamp()
2      c_i ← c_i + 1
3      send probe(c_i) to all processes
4      wait to receive response(c_i, v_j) from n − 1 processes
5      v_i ← (max_j v_j) + 1
6      send newTimestamp(c_i, v_i) to all processes
7      wait to receive ack(c_i) from n − 1 processes
8      return v_i

9  upon receiving probe(c_j) from j do
10     send response(c_j, v_i) to j

11 upon receiving newTimestamp(c_j, v_j) from j do
12     v_i ← max(v_i, v_j)
13     send ack(c_j) to j
```

Algorithm H.3: Timestamps with $n \geq 3$ and one Byzantine process

To show the timestamps are increasing, observe that after the completion of any call by i to `getTimestamp`, at least $n-2$ non-faulty processes j have a value $v_j \geq v_i$. Any call to `getTimestamp` that starts later sees at least $n-3 > 0$ of these values, and so computes a max that is at least as big as v_i and then adds 1 to it, giving a larger value.

H.4.3 Failure detectors and k-set agreement (20 points)

Recall that in the k-set agreement problem we want each of n processes to choose a decision value, with the property that the set of decision values has at most k distinct elements. It is known that k-set agreement cannot be solved deterministically in an asynchronous message-passing or shared-memory system with k or more crash failures.

Suppose that you are working in an asynchronous message-passing system with an eventually strong ($\Diamond S$) failure detector. Is it possible to solve k-set agreement deterministically with f crash failures, when $k \leq f < n/2$?

Solution

Yes. With $f < n/2$ and $\Diamond S$, we can solve consensus using Chandra-Toueg [CT96]. Since this gives a unique decision value, it solves k-set

agreement for any $k \geq 1$.

H.4.4 A set data structure (20 points)

Consider a data structure that represents a set S, with an operation $\texttt{add}(S, x)$ that adds x to S by setting $S \leftarrow S \cup \{x\}$), and an operation $\texttt{size}(S)$ that returns the number of distinct[4] elements $|S|$ of S. There are no restrictions on the types or sizes of elements that can be added to the set.

Show that any deterministic wait-free implementation of this object from atomic registers has individual step complexity $\Omega(n)$ for some operation in the worst case.

Solution

Algorithm H.4 implements a counter from a set object, where the counter read consists of a single call to $\texttt{size}(S)$. The idea is that each increment is implemented by inserting a new element into S, so $|S|$ is always equal to the number of increments.

```
1 procedure inc(S)
2     nonce ← nonce + 1
3     add(S, ⟨myId, nonce⟩).

4 procedure read(S)
5     return size(S)
```

Algorithm H.4: Counter from set object

Since the Jayanti-Tan-Toueg lower bound [JTT00] gives a lower bound of $\Omega(n)$ on the worst-case cost of a counter read, there exists an execution in which $\texttt{size}(S)$ takes $\Omega(n)$ steps.

(We could also apply JTT directly by showing that the set object is perturbable; this follows because adding an element not added by anybody else is always visible to the reader.)

[4] Clarification added during exam.

Appendix I

Additional sample final exams

This appendix contains final exams from previous times the course was offered, and is intended to give a rough guide to the typical format and content of a final exam. Note that the topics covered in past years were not necessarily the same as those covered this year.

I.1 CS425/CS525 Final Exam, December 15th, 2005

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are three problems on this exam, each worth 20 points, for a total of 60 points. You have approximately three hours to complete this exam.

I.1.1 Consensus by attrition (20 points)

Suppose you are given a **bounded fetch-and-subtract** register that holds a non-negative integer value and supports an operation fetch-and-subtract(k) for each $k > 0$ that (a) sets the value of the register to the previous value minus k, or zero if this result would be negative, and (b) returns the previous value of the register.

Determine the consensus number of bounded fetch-and-subtract under the assumptions that you can use arbitrarily many such objects, that you can supplement them with arbitrarily many multiwriter/multireader read/write registers, that you can initialize all registers of both types to initial values of

your choosing, and that the design of the consensus protocol can depend on the number of processes N.

Solution

The consensus number is 2.

To implement 2-process wait-free consensus, use a single fetch-and-subtract register initialized to 1 plus two auxiliary read/write registers to hold the input values of the processes. Each process writes its input to its own register, then performs a fetch-and-subtract(1) on the fetch-and-subtract register. Whichever process gets 1 from the fetch-and-subtract returns its own input; the other process (which gets 0) returns the winning process's input (which it can read from the winning process's read/write register.)

To show that the consensus number is at most 2, observe that any two fetch-and-subtract operations commute: starting from state x, after fetch-and-subtract(k_1) and fetch-and-subtract(k_2) the value in the fetch-and-subtract register is $\max(0, x - k_1 - k_2)$ regardless of the order of the operations.

I.1.2 Long-distance agreement (20 points)

Consider an asynchronous message-passing model consisting of N processes $p_1 \dots p_N$ arranged in a line, so that each process i can send messages only to processes $i - 1$ and $i + 1$ (if they exist). Assume that there are no failures, that local computation takes zero time, and that every message is delivered at most 1 time unit after it is sent no matter how many messages are sent on the same edge.

Now suppose that we wish to solve agreement in this model, where the agreement protocol is triggered by a local *input* event at one or more processes and it terminates when every process executes a local *decide* event. As with all agreement problems, we want Agreement (all processes decide the same value), Termination (all processes eventually decide), and Validity (the common decision value previously appeared in some input). We also want no false starts: the first action of any process should either be an *input* action or the receipt of a message.

Define the time cost of a protocol for this problem as the worst-case time between the first *input* event and the last *decide* event. Give the best upper and lower bounds you can on this time as function of N. Your upper and lower bounds should be *exact*: using no asymptotic notation or hidden constant factors. Ideally, they should also be equal.

Solution

Upper bound

Because there are no failures, we can appoint a leader and have it decide. The natural choice is some process near the middle, say $p_{\lfloor (N+1)/2 \rfloor}$. Upon receiving an input, either directly through an *input* event or indirectly from another process, the process sends the input value along the line toward the leader. The leader takes the first input it receives and broadcasts it back out in both directions as the decision value. The worst case is when the protocol is initiated at p_N; then we pay $2(N - \lfloor (N+1)/2 \rfloor)$ time to send all messages out and back, which is N time units when N is even and $N-1$ time units when N is odd.

Lower bound

Proving an almost-matching lower bound of $N-1$ time units is trivial: if p_1 is the only initiator and it starts at time t_0, then by an easy induction argument,in the worst case p_i doesn't learn of any input until time $t_0 + (i-1)$, and in particular p_N doesn't find out until after $N-1$ time units. If p_N nonetheless decides early, its decision value will violate validity in some executions.

But we can actually prove something stronger than this: that N time units are indeed required when N is odd. Consider two slow executions Ξ_0 and Ξ_1, where (a) all messages are delivered after exactly one time unit in each execution; (b) in Ξ_0 only p_1 receives an input and the input is 0; and (c) in Ξ_1 only p_N receives an input and the input is 1. For each of the executions, construct a causal ordering on events in the usual fashion: a send is ordered before a receive, two events of the same process are ordered by time, and other events are partially ordered by the transitive closure of this relation.

Now consider for Ξ_0 the set of all events that precede the *decide(0)* event of p_1 and for Ξ_1 the set of all events that precede the *decide(1)* event of p_N. Consider further the sets of processes S_0 and S_1 at which these events occur; if these two sets of processes do not overlap, then we can construct an execution in which both sets of events occur, violating Agreement.

Because S_0 and S_1 overlap, we must have $|S_0| + |S_1| \geq N+1$, and so at least one of the two sets has size at least $\lceil (N+1)/2 \rceil$, which is $N/2+1$ when N is even. Suppose that it is S_0. Then in order for any event to occur at $p_{N/2+1}$ at all some sequence of messages must travel from the initial input to p_1 to process $p_{N/2+1}$ (taking $N/2$ time units), and the causal ordering implies that an additional sequence of messages travels back from $p_{N/2+1}$ to

p_1 before p_1 decides (taking and additional $N/2$ time units). The total time is thus N.

I.1.3 Mutex appendages (20 points)

An **append** register supports standard read operations plus an append operation that appends its argument to the list of values already in the register. An **append-and-fetch** register is similar to an append register, except that it returns the value in the register after performing the append operation. Suppose that you have an failure-free asynchronous system with anonymous deterministic processes (i.e., deterministic processes that all run exactly the same code). Prove or disprove each of the following statements:

1. It is possible to solve mutual exclusion using only append registers.
2. It is possible to solve mutual exclusion using only append-and-fetch registers.

In either case, the solution should work for arbitrarily many processes—solving mutual exclusion when $N = 1$ is not interesting. You are also not required in either case to guarantee lockout-freedom.

Clarification given during exam

1. If it helps, you may assume that the processes know N. (It probably doesn't help.)

Solution

1. Disproof: With append registers only, it is not possible to solve mutual exclusion. To prove this, construct a failure-free execution in which the processes never break symmetry. In the initial configuration, all processes have the same state and thus execute either the same read operation or the same append operation; in either case we let all N operations occur in some arbitrary order. If the operations are all reads, all processes read the same value and move to the same new state. If the operations are all appends, then no values are returned and again all processes enter the same new state. (It's also the case that the processes can't tell from the register's state which of the identical append operations went first, but we don't actually need to use this fact.)

Since we get a fair failure-free execution where all processes move through the same sequence of states, if any process decides it's in its critical section, all do. We thus can't solve mutual exclusion in this model.

2. Since the processes are anonymous, any solution that depends on them having identifiers isn't going to work. But there is a simple solution that requires only appending single bits to the register.

   Each process trying to enter a critical section repeatedly executes an append-and-fetch operation with argument 0; if the append-and-fetch operation returns either a list consisting only of a single 0 or a list whose second-to-last element is 1, the process enters its critical section. To leave the critical section, the process does append-and-fetch(1).

I.2 CS425/CS525 Final Exam, May 8th, 2008

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are four problems on this exam, each worth 20 points, for a total of 80 points. You have approximately three hours to complete this exam.

I.2.1 Message passing without failures (20 points)

Suppose you have an asynchronous message-passing system with a complete communication graph, unique node identities, and no failures. Show that any deterministic atomic shared-memory object can be simulated in this model, or give an example of a shared-memory object that can't be simulated.

Solution

Pick some leader node to implement the object. To execute an operation, send the operation to the leader node, then have the leader carry out the operation (sequentially) on its copy of the object and send the results back.

I.2.2 A ring buffer (20 points)

Suppose you are given a **ring buffer object** that consists of $k \geq 1$ memory locations $a[0] \dots a[k-1]$ with an atomic *shift-and-fetch* operation that takes an argument v and (a) shifts v into the buffer, so that $a[i] \leftarrow a[i+1]$ for

each i less than $k-1$ and $a[k-1] \leftarrow v$; and (b) returns a snapshot of the new contents of the array (after the shift).

What is the consensus number of this object as a function of k?

Solution

We can clearly solve consensus for at least k processes: each process calls shift-and-fetch on its input, and returns the first non-null value in the buffer.

So now we want to show that we can't solve consensus for $k+1$ processes. Apply the usual FLP-style argument to get to a bivalent configuration C where each of the $k+1$ processes has a pending operation that leads to a univalent configuration. Let e_0 and e_1 be particular operations leading to 0-valent and 1-valent configurations, respectively, and let $e_2 \dots e_k$ be the remaining $k-1$ pending operations.

We need to argue first that no two distinct operations e_i and e_j are operations of different objects. Suppose that Ce_i is 0-valent and Ce_j is 1-valent; then if e_i and e_j are on different objects, Ce_ie_j (still 0-valent) is indistinguishable by all processes from Ce_je_i (still 1-valent), a contradiction. Alternatively, if e_i and e_j are both b-valent, there exists some $(1-b)$-valent e_k such that e_i and e_j both operate on the same object as e_k, by the preceding argument. So all of $e_0 \dots e_k$ are operations on the same object.

By the usual argument we know that this object can't be a register. Let's show it can't be a ring buffer either. Consider the configurations $Ce_0e_1 \dots e_k$ and $Ce_1 \dots e_k$. These are indistinguishable to the process carrying out e_k (because its sees only the inputs to e_1 through e_k in its snapshot). So they must have the same valence, a contradiction.

It follows that the consensus number of a k-element ring buffer is exactly k.

I.2.3 Leader election on a torus (20 points)

An $n \times n$ torus is a graph consisting of n^2 nodes, where each node (i, j), $0 \leq i, j \leq n-1$, is connected to nodes $(i-1, j), (i+1, j), (i, j-1)$, and $(i, j+1)$, where all computation is done mod n.

Suppose you have an asynchronous message-passing system with a communication graph in the form of an $n \times n$ torus. Suppose further that each node has a unique identifier (some large natural number) but doesn't know the value of n. Give an algorithm for leader election in this model with the best message complexity you can come up with.

Solution

First observe that each row and column of the torus is a bidirectional ring, so we can run e.g. Hirschbirg and Sinclair's $O(n \log n)$-message protocol within each of these rings to find the smallest identifier in the ring. We'll use this to construct the following algorithm:

1. Run Hirschbirg-Sinclair in each row to get a local leader for each row; this takes $n \times O(n \log n) = O(n^2 \log n)$ messages. Use an additional n messages per row to distribute the identifier for the row leader to all nodes and initiate the next stage of the protocol.

2. Run Hirschbirg-Sinclair in each column with each node adopting the row leader identifier as its own. This costs another $O(n^2 \log n)$ messages; at the end, every node knows the minimum identifier of all nodes in the torus.

The total message complexity is $O(n^2 \log n)$. (I suspect this is optimal, but I don't have a proof.)

I.2.4 An overlay network (20 points)

A collection of n nodes—in an asynchronous message-passing system with a connected, bidirectional communications graph with $O(1)$ links per node—wish to engage in some strictly legitimate file-sharing. Each node starts with some input pair (k, v), where k is a key and v is a value, and the search problem is to find the value v corresponding to a particular key k.

1. Suppose that we can't do any preparation ahead of time. Give an algorithm for searching with the smallest asymptotic worst-case message complexity you can find as a function of n. You may assume that there are no limits on time complexity, message size, or storage space at each node.

2. Suppose now that some designated leader node can initiate a protocol ahead of time to pre-process the data in the nodes before any query is initiated. Give a pre-processing algorithm (that does not depend on which key is eventually searched for) and associated search algorithm such that the search algorithm minimizes the asymptotic worst-case message complexity. Here you may assume that there are no limits on time complexity, message size, or storage space for either algorithm, and that you don't care about the message complexity of the pre-processing algorithm.

3. Give the best lower bound you can on the total message complexity of the pre-processing and search algorithms in the case above.

Solution

1. Run depth-first search to find the matching key and return the corresponding value back up the tree. Message complexity is $O(|E|) = O(n)$ (since each node has only $O(1)$ links).

2. Basic idea: give each node a copy of all key-value pairs, then searches take zero messages. To give each node a copy of all key-value pairs we could do convergecast followed by broadcast ($O(n)$ message complexity) or just flood each pair $O(n^2)$. Either is fine since we don't care about the message complexity of the pre-processing stage.

3. Suppose the total message complexity of both the pre-processing stage and the search protocol is less than $n - 1$. Then there is some node other than the initiator of the search that sends no messages at any time during the protocol. If this is the node with the matching key-value pair, we don't find it. It follows that any solution to the search problem. requires a total of $\Omega(n)$ messages in the pre-processing and search protocols.

I.3 CS425/CS525 Final Exam, May 10th, 2010

Write your answers in the blue book(s). Justify your answers. Work alone. Do not use any notes or books.

There are four problems on this exam, each worth 20 points, for a total of 80 points. You have approximately three hours to complete this exam.

I.3.1 Anti-consensus (20 points)

A wait-free **anti-consensus** protocol satisfies the conditions:

Wait-free termination Every process decides in a bounded number of its own steps.

Non-triviality There is at least one process that decides different values in different executions.

Disagreement If at least two processes decide, then some processes decide on different values.

Show that there is no deterministic wait-free anti-consensus protocol using only atomic registers for two processes and two possible output values, but there is one for three processes and three possible output values.

Clarification: You should assume processes have distinct identities.

Solution

No protocol for two: turn an anti-consensus protocol with outputs in $\{0, 1\}$ into a consensus protocol by having one of the processes always negate its output.

A protocol for three: Use a splitter.

I.3.2 Odd or even (20 points)

Suppose you have a protocol for a synchronous message-passing ring that is anonymous (all processes run the same code) and uniform (this code is the same for rings of different sizes). Suppose also that the processes are given inputs marking some, but not all, of them as leaders. Give an algorithm for determining if the size of the ring is odd or even, or show that no such algorithm is possible.

Clarification: Assume a bidirectional, oriented ring and a deterministic algorithm.

Solution

Here is an impossibility proof. Suppose there is such an algorithm, and let it correctly decide "odd" on a ring of size $2k + 1$ for some k and some set of leader inputs. Now construct a ring of size $4k + 2$ by pasting two such rings together (assigning the same values to the leader bits in each copy) and run the algorithm on this ring. By the usual symmetry argument, every corresponding process sends the same messages and makes the same decisions in both rings, implying that the processes incorrectly decide the ring of size $4k + 2$ is odd.

I.3.3 Atomic snapshot arrays using message-passing (20 points)

Consider the following variant of Attiya-Bar-Noy-Dolev for obtaining snapshots of an array instead of individual register values, in an asynchronous message-passing system with $t < n/4$ crash failures. The data structure we

are simulating is an array a consisting of an atomic register $a[i]$ for each process i, with the ability to perform atomic snapshots.

Values are written by sending a set of $\langle i, v, t_i \rangle$ values to all processes, where i specifies the segment $a[i]$ of the array to write, v gives a value for this segment, and t_i is an increasing timestamp used to indicate more recent values. We use a set of values because (as in ABD) some values may be obtained indirectly.

To update segment $a[i]$ with value v, process i generates a new timestamp t_i, sends $\{\langle i, v, t_i \rangle\}$ to all processes, and waits for acknowledgments from at least $3n/4$ processes.

Upon receiving a message containing one or more $\langle i, v, t_i \rangle$ triples, a process updates its copy of $a[i]$ for any i with a higher timestamp than previously seen, and responds with an acknowledgment (we'll assume use of nonces so that it's unambiguous which message is being acknowledged).

To perform a snapshot, a process sends SNAPSHOT to all processes, and waits to receive responses from at least $3n/4$ processes, which will consist of the most recent values of each $a[i]$ known by each of these processes together with their timestamps (it's a set of triples as above). The snapshot process then takes the most recent versions of $a[i]$ for each of these responses and updates its own copy, then sends its entire snapshot vector to all processes and waits to receive at least $3n/4$ acknowledgments. When it has received these acknowledgments, it returns its own copy of $a[i]$ for all i.

Prove or disprove: The above procedure implements an atomic snapshot array in an asynchronous message-passing system with $t < n/4$ crash failures.

Solution

Disproof: Let s_1 and s_2 be processes carrying out snapshots and let w_1 and w_2 be processes carrying out writes. Suppose that each w_i initiates a write of 1 to $a[w_i]$, but all of its messages to other processes are delayed after it updates its own copy $a_{w_i}[w_i]$. Now let each s_i receive responses from $3n/4 - 1$ processes not otherwise mentioned plus w_i. Then s_1 will return a vector with $a[w_1] = 1$ and $a[w_2] = 0$ while s_2 will return a vector with $a[w_1] = 0$ and $a[w_2] = 1$, which is inconsistent. The fact that these vectors are also disseminated throughout at least $3n/4$ other processes is a red herring.

I.3.4 Priority queues (20 points)

Let Q be a priority queue whose states are multisets of natural numbers and that has operations `enq`(v) and `deq`(), where `enq`(p) adds a new value v to

the queue, and `deq`() removes and returns the smallest value in the queue, or returns null if the queue is empty. (If there is more than one copy of the smallest value, only one copy is removed.)

What is the consensus number of this object?

Solution

The consensus number is 2. The proof is similar to that for a queue.

To show we can do consensus for $n = 2$, start with a priority queue with a single value in it, and have each process attempt to dequeue this value. If a process gets the value, it decides on its own input; if it gets null, it decides on the other process's input.

To show we can't do consensus for $n = 3$, observe first that starting from any states C of the queue, given any two operations x and y that are both enqueues or both dequeues, the states Cxy and Cyx are identical. This means that a third process can't tell which operation went first, meaning that a pair of enqueues or a pair of dequeues can't get us out of a bivalent configuration in the FLP argument. We can also exclude any split involving two operations on different queues (or other objects) But we still need to consider the case of a dequeue operation d and an enqueue operation e on the same queue Q. This splits into several subcases, depending on the state C of the queue in some bivalent configuration:

1. $C = \{\}$. Then $Ced = Cd = \{\}$, and a third process can't tell which of d or e went first.

2. C is nonempty and $e =$ `enq`(v), where v is greater than or equal to the smallest value in C. Then Cde and Ced are identical, and no third process can tell which of d or e went first.

3. C is nonempty and $e =$ `enq`(v), where v is less than any value in C. Consider the configurations Ced and Cde. Here the process p_d that performs d can tell which operation went first, because it either obtains v or some other value $v' \neq v$. Kill this process. No other process in Ced or Cde can distinguish the two states without dequeuing whichever of v or v' was not dequeued by p_d. So consider two parallel executions $Ced\sigma$ and $Cde\sigma$ where σ consists of an arbitrary sequence of operations ending with a `deq` on Q by some process p (if no process ever attempts to dequeue from Q, then we have already won, since the survivors can't distinguish Ced from Cde). Now the state of all objects is the same after $Ced\sigma$ and $Cde\sigma$, and only p_d and p have different states in these two configurations. So any third process is out of luck.

Appendix J

I/O automata

J.1 Low-level view: I/O automata

An **I/O automaton** [LT87] is an automaton where transitions are labeled by **actions**, which come in three classes: **input actions**, triggered by the outside world; **output actions** triggered by the automaton and visible to the outside world; and **internal actions**, triggered by the automaton but not visible to the outside world. These classes correspond to inputs, outputs, and internal computation steps of the automaton; the latter are provided mostly to give merged input/output actions a place to go when automata are composed together. A **transition relation** $\texttt{trans}(A)$ relates $\texttt{states}(A) \times \texttt{acts}(A) \times \texttt{states}(A)$; if (s, a, s') is in $\texttt{trans}(A)$, it means that A can move from state s to state s' by executing action a.

There is also an equivalence relation $\texttt{task}(A)$ on the output and internal actions, which is used for enforcing fairness conditions—the basic idea is that in a fair execution some action in each equivalence class must be executed eventually (a more accurate definition will be given below).

The I/O automaton model carries with it a lot of specialized jargon. We'll try to avoid it as much as possible. One thing that will be difficult to avoid in reading [Lyn96] is the notion of a **signature**, which is just the tuple $\texttt{sig}(A) = (\texttt{in}(A), \texttt{out}(A), \texttt{int}(A))$ describing the actions of an automaton A.

J.1.1 Enabled actions

An action a is **enabled** in some state s if $\texttt{trans}(A)$ contains at least one transition (s, a, s'). Input actions are *always* enabled—this is a requirement of the model. Output and internal actions—the "locally controlled" actions—are

not subject to this restriction. A state s is **quiescent** if only input actions are enabled in s.

J.1.2 Executions, fairness, and traces

An **execution** of A is a sequence $s_0 a_0 s_1 a_1 \ldots$ where each triple $(s_i, a_i s_{i+1})$ is in $\texttt{trans}(A)$. Executions may be finite or infinite; if finite, they must end in a state.

A **trace** of A is a subsequence of some execution consisting precisely of the external (i.e., input and output) actions, with states and internal actions omitted. If we don't want to get into the guts of a particular I/O automaton—and we usually don't, unless we can't help it because we have to think explicitly about states for some reason—we can describe its externally-visible behavior by just giving its set of traces.

J.1.3 Composition of automata

Composing a set of I/O automata yields a new super-automaton whose state set is the Cartesian product of the state sets of its components and whose action set is the union of the action sets of its components. A transition with a given action a updates the states of all components that have a as an action and has no effect on the states of other components. The classification of actions into the three classes is used to enforce some simple compatibility rules on the component automata; in particular:

1. An internal action of a component is never an action of another component—internal actions are completely invisible.

2. No output action of a component can be an output action of another component.

3. No action is shared by infinitely many components.[1] In practice this means that no action can be an input action of infinitely many components, since the preceding rules mean that any action is an output or internal action of at most one component.

All output actions of the components are also output actions of the composition. An input action of a component is an input of the composition only if some other component doesn't supply it as an output; in this case

[1] Note that infinite (but countable) compositions *are* permitted.

it becomes an output action of the composition. Internal actions remain internal (and largely useless, except for bookkeeping purposes).

The `task` equivalence relation is the union of the `task` relations for the components: this turns out to give a genuine equivalence relation on output and internal actions precisely because the first two compatibility rules hold.

Given an execution or trace X of a composite automaton that includes A, we can construct the corresponding execution or trace $X|A$ of A which just includes the states of A and the actions visible to A (events that don't change the state of A drop out). The definition of composition is chosen so that $X|A$ is in fact an execution/trace of A whenever X is.

J.1.4 Hiding actions

Composing A and B continues to expose the outputs of A even if they line up with inputs of B. While this may sometimes be desirable, often we want to shove such internal communication under the rug. The model lets us do this by redefining the signature of an automaton to make some or all of the output actions into internal actions.

J.1.5 Fairness

I/O automata come with a built-in definition of **fair executions**, where an execution of A is **fair** if, for each equivalence class C of actions in $\texttt{task}(A)$,

1. the execution is finite and no action in C is enabled in the final state, or
2. the execution is infinite and there are infinitely many occurrences of actions in C, or
3. the execution is infinite and there are infinitely many states in which no action in C is enabled.

If we think of C as corresponding to some thread or process, this says that C gets infinitely many chances to do something in an infinite execution, but may not actually do them if it gives ups and stops waiting (the third case). The finite case essentially says that a finite execution isn't fair unless nobody is waiting at the end. The motivation for this particular definition is that it guarantees (a) that any finite execution can be extended to a fair execution and (b) that the restriction $X|A$ of a fair execution or trace X is also fair.

Fairness is useful e.g. for guaranteeing message delivery in a message-passing system: make each message-delivery action its own task class and each message will eventually be delivered; similarly make each message-sending action its own task class and a process will eventually send every message it intends to send. Tweaking the task classes can allow for possibilities of starvation, e.g. if all message-delivery actions are equivalent then a spammer can shut down the system in a "fair" execution where only his (infinitely many) messages are delivered.

J.1.6 Specifying an automaton

The typical approach is to write down preconditions and effects for each action (for input actions, the preconditions are empty). An example would be the spambot in Algorithm J.1.

```
1 input action setMessage(m)
2     effects
3         state ← m
4 output action spam(m)
5     precondition
6         spam = m
7     effects
8         none (keep spamming)
```

Algorithm J.1: Spambot as an I/O automaton

(Plus an initial state, e.g. $\mathsf{state} = \bot$, where $\bot$ is not a possible message, and a task partition, of which we will speak more below when we talk about liveness properties.)

J.2 High-level view: traces

When studying the behavior of a system, traces are what we really care about, and we want to avoid talking about states as much as possible. So what we'll aim to do is to get rid of the states early by computing the set of traces (or fair traces) of each automaton in our system, then compose traces to get traces for the system as a whole. Our typical goal will be to show that the resulting set of traces has some desirable properties, usually of the form (1) nothing bad happens (a **safety property**); (2) something

good eventually happens (a **liveness property**); or (3) the horribly complex composite automaton representing this concrete system acts just like that nice clean automaton representing a specification (a **simulation**).

Very formally, a **trace property** specifies both the signature of the automaton and a set of traces, such that all traces (or perhaps fair traces) of the automata appear in the set. We'll usually forget about the first part.

Tricky detail: It's OK if not all traces in P are generated by A (we want $\texttt{trace}(A) \subseteq P$, but not necessarily $\texttt{trace}(A) = P$). But $\texttt{trace}(A)$ will be pretty big (it includes, for example, all finite sequences of input actions) so hopefully the fact that A has to do something with inputs will tell us something useful.

J.2.1 Example

A property we might demand of the spambot above (or some other abstraction of a message channel) is that it only delivers messages that have previously been given to it. As a trace property this says that in any trace t, if $t_k = \mathsf{spam}(m)$, then $t_j = \mathsf{setMessage}(m)$ for some $j < k$. (As a set, this is just the set of all sequences of external spambot-actions that have this property.) Call this property P.

To prove that the spambot automaton given above satisfies P, we might argue that for any execution $s_0 a_0 s_1 a_1 \ldots$, that $s_i = m$ in the last $\mathsf{setMessage}$ action preceding s_i, or $\perp$ if there is no such action. This is easily proved by induction on i. It then follows that since $\mathsf{spam}(m)$ can only transmit the current state, that if $\mathsf{spam}(m)$ follows $s_i = m$ that it follows some earlier $\mathsf{setMessage}(m)$ as claimed.

However, there are traces that satisfy P that don't correspond to executions of the spambot; for example, consider the trace $\mathsf{setMessage}(0)\mathsf{setMessage}(1)\mathsf{spam}(0)$. This satisfies P (0 was previously given to the automaton $\mathsf{spam}(0)$), but the automaton won't generate it because the 0 was overwritten by the later $\mathsf{setMessage}(1)$ action. Whether this is indicates a problem with our automaton not being nondeterministic enough or our trace property being too weak is a question about what we really want the automaton to do.

J.2.2 Types of trace properties

J.2.2.1 Safety properties

P is a **safety property** if

1. P is nonempty.

2. P is **prefix-closed**, i.e. if xy is in P then x is in P.

3. P is **limit-closed**, i.e. if $x_1, x_1x_2, x_1x_2x_3, \ldots$ are all in P, then so is the infinite sequence obtained by taking their limit.

Because of the last restrictions, it's enough to prove that P holds for all finite traces of A to show that it holds for all traces (and thus for all fair traces), since any trace is a limit of finite traces. Conversely, if there is some trace or fair trace for which P fails, the second restriction says that P fails on any finite prefix of P, so again looking at only finite prefixes is enough. The spambot property mentioned above is a safety property.

Safety properties are typically proved using **invariants**, properties that are shown by induction to hold in all reachable states.

J.2.2.2 Liveness properties

P is a **liveness property** of A if any finite sequence of actions in $\texttt{acts}(A)$ has an extension in P. Note that liveness properties will in general include many sequences of actions that aren't traces of A, since they are extensions of finite sequences that A can't do (e.g. starting the execution with an action not enabled in the initial state). If you want to restrict yourself only to proper executions of A, use a safety property. (It's worth noting that the same property P can't do both: any P that is both a liveness and a safety property includes all sequences of actions because of the closure rules.)

Liveness properties are those that are always eventually satisfiable; asserting one says that the property is eventually satisfied. The typical way to prove a liveness property is with a **progress function**, a function f on states that (a) drops by at least 1 every time something that happens infinitely often happens (like an action from an always-enabled task class) and (b) guarantees P once it reaches 0.

An example would be the following property we might demand of our spambot: any trace with at least one $\mathsf{setMessage}(\ldots)$ action contains infinitely many $\mathsf{spam}(\ldots)$ actions. Whether the spambot automaton will satisfy this property (in fair traces) depends on its task partition. If all $\mathsf{spam}(\ldots)$ actions are in the same equivalence class, then any execution with at least one setMessage will have some $\mathsf{spam}(\ldots)$ action enabled at all times thereafter, so a fair trace containing a setMessage can't be finite (since spam is enabled in the last state) and if infinite contains infinitely many spam messages (since spam messages of some sort are enabled in all but an initial finite prefix). On the other hand, if $\mathsf{spam}(m_1)$ and $\mathsf{spam}(m_2)$ are not equivalent in $\texttt{task}(A)$, then the spambot doesn't satisfy the liveness property: in an execution that alternates

$\mathsf{setMessage}(m_1)\mathsf{setMessage}(m_2)\mathsf{setMessage}(m_1)\mathsf{setMessage}(m_2)\ldots$ there are infinitely many states in which $\mathsf{spam}(m_1)$ is not enabled, so fairness doesn't require doing it even once, and similarly for $\mathsf{spam}(m_2)$.

J.2.2.3 Other properties

Any other property P can be expressed as the intersection of a safety property (the closure of P) and a liveness property (the union of P and the set of all finite sequences that aren't prefixes of traces in P). The intuition is that the safety property prunes out the excess junk we threw into the liveness property to make it a liveness property, since any sequence that isn't a prefix of a trace in P won't go into the safety property. This leaves only the traces in P.

Example: Let $P = \{0^n 1^\infty\}$ be the set of traces where we eventually give up on our pointless 0-action and start doing only 1-actions forever. Then P is the intersection of the safety property $S = \{0^n 1^m\} \cup P$ (the extra junk is from prefix-closure) and the liveness property $L = \{0^n 11^m 0x | xin\{0,1\}^*\} \cup P$. Property S says that once we do a 1 we never do a 0, but allows finite executions of the form 0^n where we never do a 1. Property L says that we eventually do a 1-action, but that we can't stop unless we later do at least one 0-action.

J.2.3 Compositional arguments

The **product** of trace properties $P_1, P_2 \ldots$ is the trace property P where T is in P if and only if $T|\mathtt{sig}(P_i)$ is in P_i for each i. If the $\{A_i\}$ satisfy corresponding propertties $\{P_i\}$ individually, then their composition satisfies the product property. (For safety properties, often we prove something weaker about the A_i, which is that each A_i individually is not the first to violate P—i.e., it can't leave P by executing an internal or output action. In an execution where inputs by themselves can't violate P, P then holds.)

Product properties let us prove trace properties by smashing together properties of the component automata, possibly with some restrictions on the signatures to get rid of unwanted actions. The product operation itself is in a sense a combination of a Cartesian product (pick traces t_i and smash them together) filtered by a consistency rule (the smashed trace must be consistent); it acts much like intersection (and indeed can be made identical to intersection if we treat a trace property with a given signature as a way of describing the set of all T such that $T|\mathtt{sig}(P_i)$ is in P_i).

J.2.3.1 Example

Consider two spambots A_1 and A_2 where we identify the $\mathsf{spam}(m)$ operation of A_1 with the $\mathsf{setMessage}(m)$ operation of A_2; we'll call this combined action $\mathsf{spam}_1(m)$ to distinguish it from the output actions of A_2. We'd like to argue that the composite automaton $A_1 + A_2$ satisfies the safety property (call it P_m) that any occurrence of $\mathsf{spam}(m)$ is preceded by an occurrence of $\mathsf{setMessage}(m)$, where the signature of P_m includes $\mathsf{setMessage}(m)$ and $\mathsf{spam}(m)$ for some specific m but no other operations. (This is an example of where trace property signatures can be useful without being limited to actions of any specific component automaton.)

To do so, we'll prove a stronger property P'_m, which is P_m modified to include the $\mathsf{spam}_1(m)$ action in its signature. Observe that P'_m is the product of the safety properties for A_1 and A_2 restricted to $\mathtt{sig}(P'_m)$, since the later says that any trace that includes $\mathsf{spam}(m)$ has a previous $\mathsf{spam}_1(m)$ and the former says that any trace that includes $\mathsf{spam}_1(m)$ has a previous $\mathsf{setMessage}(m)$. Since these properties hold for the individual A_1 and A_2, their product, and thus the restriction P'_m, holds for $A_1 + A_2$, and so P_m (as a further restriction) holds for $A_1 + A_2$ as well.

Now let's prove the liveness property for $A_1 + A_2$, that at least one occurrence of $\mathsf{setMessage}$ yields infinitely many spam actions. Here we let $L_1 = \{$at least one $\mathsf{setMessage}$ action $\Rightarrow$ infinitely many spam_1 actions$\}$ and $L_2 = \{$at least one spam_1 action $\Rightarrow$ infinitely many spam actions$\}$. The product of these properties is all sequences with (a) no $\mathsf{setMessage}$ actions or (b) infinitely many spam actions, which is what we want. This product holds if the individual properties L_1 and L_2 hold for $A_1 + A_2$, which will be the case if we set $\mathtt{task}(A_1)$ and $\mathtt{task}(A_2)$ correctly.

J.2.4 Simulation arguments

Show that $\mathtt{traces}(A)$ is a subset of $\mathtt{traces}(B)$ (possibly after hiding some actions of A) by showing a **simulation relation** $f : \mathtt{states}(A) \to \mathtt{states}(B)$ between states of A and states of B. Requirements on f are

1. If s is in $\mathtt{start}(A)$, then $f(s)$ includes some element of $\mathtt{start}(B)$.
2. If (s, a, s') is in $\mathtt{trans}(A)$ and s is reachable, then for any reachable u in $f(s)$, there is a sequence of actions x that takes u to some v in $f(s')$ with $\mathtt{trace}(x) = \mathtt{trace}(a)$.

Using these we construct an execution of B matching (in trace) an

execution of A by starting in $f(s_0)$ and applying the second part of the definition to each action in the A execution (including the hidden ones!)

J.2.4.1 Example

A single spambot A can simulate the conjoined spambots $A_1 + A_2$. Proof: Let $f(s) = (s, s)$. Then $f(\bot) = (\bot, \bot)$ is a start state of $A_1 + A_2$. Now consider a transition (s, a, s') of A; the action a is either (a) $\mathsf{setMessage}(m)$, giving $s' = m$; here we let $x = \mathsf{setMessage}(m)\mathsf{spam}_1(m)$ with $\mathtt{trace}(x) = \mathtt{trace}(a)$ since $\mathsf{spam}_1(m)$ is internal and $f(s') = (m, m)$ the result of applying x; or (b) $a = \mathsf{spam}(m)$, which does not change s or $f(s)$; the matching x is $\mathsf{spam}(m)$, which also does not change $f(s)$ and has the same trace.

A different proof could take advantage of f being a relation by defining $f(s) = \{(s, s') | s' \in \mathtt{states}(A_2)\}$. Now we don't care about the state of A_2, and treat a $\mathsf{setMessage}(m)$ action of A as the sequence $\mathsf{setMessage}(m)$ in $A_1 + A_2$ (which updates the first component of the state correctly) and treat a $\mathsf{spam}(m)$ action as $\mathsf{spam}_1(m)\mathsf{spam}(m)$ (which updates the second component—which we don't care about—and has the correct trace.) In some cases an approach of this sort is necessary because we don't know which simulated state we are heading for until we get an action from A.

Note that the converse doesn't work: $A_1 + A_2$ don't simulate A, since there are traces of $A_1 + A_2$ (e.g. $\mathsf{setMessage}(0)\mathsf{spam}_1(0)\mathsf{setMessage}(1)\mathsf{spam}(0)$) that don't restrict to traces of A. See [Lyn96, §8.5.5] for a more complicated example of how one FIFO queue can simulate two FIFO queues and vice versa (a situation called **bisimulation**).

Since we are looking at traces rather than fair traces, this kind of simulation doesn't help much with liveness properties, but sometimes the connection between states plus a liveness proof for B can be used to get a liveness proof for A (essentially we have to argue that A can't do infinitely many action without triggering a B-action in an appropriate task class). Again see [Lyn96, §8.5.5].

Bibliography

[AA11] Dan Alistarh and James Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *Distributed Computing: 25th International Symposium, DISC 2011*, volume 6950 of *Lecture Notes in Computer Science*, pages 97–109. Springer-Verlag, September 2011.

[AAB+11] Yehuda Afek, Noga Alon, Omer Barad, Eran Hornstein, Naama Barkai, and Ziv Bar-Joseph. A biological solution to a fundamental distributed computing problem. *science*, 331(6014):183–185, 2011.

[AABJ+11] Yehuda Afek, Noga Alon, Ziv Bar-Joseph, Alejandro Cornejo, Bernhard Haeupler, and Fabian Kuhn. Beeping a maximal independent set. In *Proceedings of the 25th International Conference on Distributed Computing*, DISC'11, pages 32–50, Berlin, Heidelberg, 2011. Springer-Verlag.

[AACH+11] Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Morteza Zadimoghaddam. Optimal-time adaptive tight renaming, with applications to counting. In *Proceedings of the Thirtieth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 239–248, June 2011.

[AACH12] James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *Journal of the ACM*, 59(1):2:1–2:24, February 2012.

[AACHE15] James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Limited-use snapshots with polylogarithmic step complexity. *Journal of the ACM*, 62(1):3, February 2015.

[AACHE18] James Aspnes, Hagit Attiya, Keren Censor-Hillel, and Faith Ellen. Erratum: Limited-use atomic snapshots with polylogarithmic step complexity. *J. ACM*, 65(6):38:1–38:2, November 2018.

[AACV17] Yehuda Afek, James Aspnes, Edo Cohen, and Danny Vainstein. Brief announcement: Object oriented consensus. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 367–369. ACM, 2017.

[AAD+93] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[AAD+06] Dana Angluin, James Aspnes, Zoë Diamadi, Michael J. Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, pages 235–253, March 2006.

[AAE06] Dana Angluin, James Aspnes, and David Eisenstat. Stably computable predicates are semilinear. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 292–299, New York, NY, USA, 2006. ACM Press.

[AAE08a] Dana Angluin, James Aspnes, and David Eisenstat. Fast computation by population protocols with a leader. *Distributed Computing*, 21(3):183–199, September 2008.

[AAE08b] Dana Angluin, James Aspnes, and David Eisenstat. A simple population protocol for fast robust approximate majority. *Distributed Computing*, 21(2):87–102, July 2008.

[AAE+23] Dan Alistarh, James Aspnes, Faith Ellen, Rati Gelashvili, and Leqi Zhu. Why extension-based proofs fail. *SIAM Journal on Computing*, 52(4):913–944, 2023.

[AAG+10] Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In Nancy A. Lynch and Alexander A. Shvartsman, editors, *Distributed Computing, 24th International Symposium, DISC*

2010, Cambridge, MA, USA, September 13-15, 2010. Proceedings, volume 6343 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2010.

[AAGG11] Dan Alistarh, James Aspnes, Seth Gilbert, and Rachid Guerraoui. The complexity of renaming. In *Fifty-Second Annual IEEE Symposium on Foundations of Computer Science*, pages 718–727, October 2011.

[AAGW13] Dan Alistarh, James Aspnes, George Giakkoupis, and Philipp Woelfel. Randomized loose renmaing in $O(\log \log n)$ time. In *2013 ACM Symposium on Principles of Distributed Computing*, pages 200–209, July 2013.

[ABHMT20] Mirza Ahad Baig, Danny Hendler, Alessia Milani, and Corentin Travers. Long-lived snapshots with polylogarithmic amortized step complexity. In *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, page 3140, New York, NY, USA, 2020. Association for Computing Machinery.

[ABND+90] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, and Rüdiger Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, 1990.

[ABND95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[Abr88] Karl Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 291–302, 1988.

[AC08] Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. *Journal of the ACM*, 55(5):20, October 2008.

[AC09] James Aspnes and Keren Censor. Approximate shared-memory counting despite a strong adversary. In *SODA '09: Proceedings of the Nineteenth Annual ACM -SIAM Symposium on Discrete Algorithms*, pages 441–450, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[ACAH16] James Aspnes, Keren Censor-Hillel, Hagit Attiya, and Danny Hendler. Lower bounds for restricted-use objects. *SIAM J. Comput.*, 45(3):734–762, 2016.

[ACH10] Hagit Attiya and Keren Censor-Hillel. Lower bounds for randomized consensus under a weak adversary. *SIAM J. Comput.*, 39(8):3885–3904, 2010.

[ACH13] James Aspnes and Keren Censor-Hillel. Atomic snapshots in $O(log^3 n)$ steps using randomized helping. In Yehuda Afek, editor, *Distributed Computing: 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14–18, 2013. Proceedings*, volume 8205 of *Lecture Notes in Computer Science*, pages 254–268. Springer Berlin Heidelberg, 2013.

[ACHS16] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are lock-free concurrent algorithms practically wait-free? *Journal of the ACM (JACM)*, 63(4):1–20, 2016.

[AE11] James Aspnes and Faith Ellen. Tight bounds for anonymous adopt-commit objects. In *23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 317–324, June 2011.

[AEG16] Yehuda Afek, Faith Ellen, and Eli Gafni. Deterministic objects: Life beyond consensus. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, pages 97–106, New York, NY, USA, 2016. ACM.

[AEH75] E. A. Akkoyunlu, K. Ekanadham, and R. V. Huber. Some constraints and tradeoffs in the design of network communications. *SIGOPS Oper. Syst. Rev.*, 9:67–74, November 1975.

[AF01] Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM Journal on Computing*, 31(2):642–664, 2001.

[AFL83] Eshrat Arjomandi, Michael J. Fischer, and Nancy A. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *J. ACM*, 30(3):449–456, 1983.

[AG91] Yehuda Afek and Eli Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. *SIAM Journal on Computing*, 20(2):376–394, 1991.

[AG95] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. Technical Report 95/7, DEC Western Research Laboratory, 1995.

[AG18] Dan Alistarh and Rati Gelashvili. Recent algorithmic advances in population protocols. *SIGACT News*, 49(3):63–73, October 2018.

[AGGT09] Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. Of choices, failures and asynchrony: The many faces of set agreement. In Yingfei Dong, Ding-Zhu Du, and Oscar H. Ibarra, editors, *ISAAC*, volume 5878 of *Lecture Notes in Computer Science*, pages 943–953. Springer, 2009.

[AGHK06] Hagit Attiya, Rachid Guerraoui, Danny Hendler, and Petr Kouznetsov. Synchronizing without locks is inherently expensive. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 300–307, New York, NY, USA, 2006. ACM.

[AGTV92] Yehuda Afek, Eli Gafni, John Tromp, and Paul M. B. Vitányi. Wait-free test-and-set (extended abstract). In Adrian Segall and Shmuel Zaks, editors, *Distributed Algorithms, 6th International Workshop, WDAG '92, Haifa, Israel, November 2-4, 1992, Proceedings*, volume 647 of *Lecture Notes in Computer Science*, pages 85–94. Springer, 1992.

[AH90a] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, September 1990.

[AH90b] James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Second Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 340–349, July 1990.

[AHM09] Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In Friedhelm Meyer auf der Heide and Michael A. Bender, editors, *SPAA 2009: Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures, Calgary, Alberta, Canada, August 11-13, 2009*, pages 69–78. ACM, 2009.

[AHR95] Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.

[AHS94] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, September 1994.

[AHTW18] James Aspnes, Bernhard Haeupler, Alexander Tong, and Philipp Woelfel. Allocate-on-use space complexity of shared-memory algorithms. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15–19, 2018*, volume 121 of *LIPIcs*, pages 8:1–8:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[AHW08] Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, STOC '08, pages 217–226, New York, NY, USA, 2008. ACM.

[AJK05] James Aspnes, Collin Jackson, and Arvind Krishnamurthy. Exposing computationally-challenged Byzantine impostors. Technical Report YALEU/DCS/TR-1332, Yale University Department of Computer Science, July 2005.

[AKM+93] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. Time optimal self-stabilizing synchronization. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 652–661. ACM, 1993.

[AKM+07] Baruch Awerbuch, Shay Kutten, Yishay Mansour, Boaz Patt-Shamir, and George Varghese. A time-optional self-stabilizing synchronizer using a phase clock. *IEEE Transactions on Dependable and Secure Computing*, 4(3):180–190, July–September 2007.

[AKP+06] Hagit Attiya, Fabian Kuhn, C. Greg Plaxton, Mirjam Wattenhofer, and Roger Wattenhofer. Efficient adaptive collect using randomization. *Distributed Computing*, 18(3):179–188, 2006.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An $o(n \log n)$ sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 1–9, New York, NY, USA, 1983. ACM.

[AM93] James H. Anderson and Mark Moir. Towards a necessary and sufficient condition for wait-free synchronization (extended abstract). In André Schiper, editor, *Distributed Algorithms, 7th International Workshop, WDAG '93, Lausanne, Switzerland, September 27-29, 1993, Proceedings*, volume 725 of *Lecture Notes in Computer Science*, pages 39–53. Springer, 1993.

[AM94] Hagit Attiya and Marios Mavronicolas. Efficiency of semisynchronous versus asynchronous networks. *Mathematical Systems Theory*, 27(6):547–571, November 1994.

[AM99] Yehuda Afek and Michael Merritt. Fast, wait-free $(2k-1)$-renaming. In *PODC*, pages 105–112, 1999.

[AMW11] Yehuda Afek, Adam Morrison, and Guy Wertheim. From bounded to unbounded concurrency objects and back. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 119–128. ACM, 2011.

[And90] Thomas E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, 1990.

[And94] James H. Anderson. Multi-writer composite registers. *Distributed Computing*, 7(4):175–195, 1994.

[Ang80] Dana Angluin. Local and global properties in networks of processors (extended abstract). In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, STOC '80, pages 82–93, New York, NY, USA, 1980. ACM.

[AP06] Noa Agmon and David Peleg. Fault-tolerant gathering algorithms for autnonomous mobile robots. *SIAM Journal on Computing*, 36(1):56–82, 2006.

[AR09] James Aspnes and Eric Ruppert. An introduction to population protocols. In Benoît Garbinato, Hugo Miranda, and

Luís Rodrigues, editors, *Middleware for Network Eccentric and Mobile Applications*, pages 97–120. Springer-Verlag, 2009.

[Asp98] James Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. *Journal of the ACM*, 45(3):415–450, May 1998.

[Asp10] James Aspnes. Slightly smaller splitter networks. Technical Report YALEU/DCS/TR-1438, Yale University Department of Computer Science, November 2010.

[Asp11] James Aspnes. Notes on randomized algorithms. http://www.cs.yale.edu/homes/aspnes/classes/469/notes.pdf, July 2011.

[Asp12a] James Aspnes. Faster randomized consensus with an oblivious adversary. In *2012 ACM Symposium on Principles of Distributed Computing*, pages 1–8, July 2012.

[Asp12b] James Aspnes. A modular approach to shared-memory consensus, with applications to the probabilistic-write model. *Distributed Computing*, 25(2):179–188, May 2012.

[ASW88] Hagit Attiya, Marc Snir, and Manfred K. Warmuth. Computing on an anonymous ring. *J. ACM*, 35:845–875, October 1988.

[ASZ96] D. Atkins, W. Stallings, and P. Zimmerman. PGP Message Exchange Formats. RFC 1991 (Informational), August 1996.

[Att14] Hagit Attiya. Lower bounds and impossibility results for transactional memory computing. *Bulletin of the European Association for Computer Science*, 112:38–52, February 2014.

[Aum97] Yonatan Aumann. Efficient asynchronous consensus with the weak adversary scheduler. In *PODC '97: Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 209–218, New York, NY, USA, 1997. ACM.

[AW99] Yehuda Afek and Eytan Weisberger. The instancy of snapshots and commuting objects. *J. Algorithms*, 30(1):68–105, 1999.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley, second edition, 2004. On-line version: http://dx.doi.org/10.1002/0471478210. (This may not work outside Yale.).

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32:804–823, October 1985.

[AWW93] Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects (extended abstract). In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, pages 159–170, 1993.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference 32*, pages 307–314, 1968.

[BBE+15] P. Berenbrink, A. Brinkmann, R. Elsässer, T. Friedetzky, and L. Nagel. Randomized renaming in shared memory systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 542–549, May 2015.

[BDLP08] Christian Boulinier, Ajoy K. Datta, Lawrence L. Larmore, and Franck Petit. Space efficient and time optimal distributed BFS tree construction. *Information Processing Letters*, 108(5):273–278, November 2008. http://dx.doi.org/10.1016/j.ipl.2008.05.016.

[BDT12] Zohir Bouzid, Shantanu Das, and Sébastien Tixeuil. Wait-free gathering of mobile robots. *CoRR*, abs/1207.0226, 2012.

[Bel58] RE Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[Bel03] S. Bellovin. The Security Flag in the IPv4 Header. RFC 3514 (Informational), April 2003.

[BEW11] Alex Brodsky, Faith Ellen, and Philipp Woelfel. Fully-adaptive algorithms for long-lived renaming. *Distributed Computing*, 24(2):119–134, 2011.

[BG93] Elizabeth Borowsky and Eli Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *STOC*, pages 91–100, 1993.

[BG97] Elizabeth Borowsky and Eli Gafni. A simple algorithmically reasoned characterization of wait-free computations (extended abstract). In *PODC*, pages 189–198, 1997.

[BGA94] Elizabeth Borowsky, Eli Gafni, and Yehuda Afek. Consensus power makes (some) sense! (extended abstract). In *PODC*, pages 363–372, 1994.

[BGLR01] E. Borowsky, E. Gafni, N. Lynch, and S. Rajsbaum. The bg distributed simulation algorithm. *Distrib. Comput.*, 14(3):127–146, October 2001.

[BGM+18] Christian Badertscher, Juan Garay, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. But why does it work? a rational protocol design treatment of bitcoin. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 34–65, Cham, 2018. Springer International Publishing.

[BGP89] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Towards optimal distributed consensus (extended abstract). In *30th Annual Symposium on Foundations of Computer Science, 30 October-1 November 1989, Research Triangle Park, North Carolina, USA*, pages 410–415, 1989.

[BK07] Rida Bazzi and Goran Konjevod. On the establishment of distinct identities in overlay networks. *Distributed Computing*, 19:267–287, 2007. 10.1007/s00446-006-0012-y.

[BL93] James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Inf. Comput.*, 107(2):171–184, 1993.

[BND89] A. Bas-Noy and D. Dolev. Shared-memory vs. message-passing in an asynchronous distributed environment. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, PODC '89, pages 307–318, New York, NY, USA, 1989. ACM.

[BO83] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 27–30, Montreal, Quebec, Canada, August 1983.

[Bor95] Elizabeth Borowsky. *Capturing the Power of Resiliency and Set Consensus in Distributed Systems*. PhD thesis, University of California, Los Angeles, 1995.

[BPSV06] Harry Buhrman, Alessandro Panconesi, Riccardo Silvestri, and Paul Vitányi. On the importance of having an identity or, is consensus really universal? *Distrib. Comput.*, 18:167–176, February 2006.

[BR91] Gabriel Bracha and Ophir Rachman. Randomized consensus in expected $O(n^2 \log n)$ operations. In Sam Toueg, Paul G. Spirakis, and Lefteris M. Kirousis, editors, *Distributed Algorithms, 5th International Workshop*, volume 579 of *Lecture Notes in Computer Science*, pages 143–150, Delphi, Greece, 7–9 October 1991. Springer, 1992.

[Bra87] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[BT15] Quentin Bramas and Sébastien Tixeuil. Wait-free gathering without chirality. In Christian Scheideler, editor, *Structural Information and Communication Complexity: 22nd International Colloquium, SIROCCO 2015, Montserrat, Spain, July 14-16, 2015. Post-Proceedings*, pages 313–327, Cham, 2015. Springer International Publishing.

[Bur80] James E. Burns. A formal model for message passing systems. Technical Report 91, Computer Science Department, Indiana University, September 1980. http://www.cs.indiana.edu/pub/techreports/TR91.pdf.

[BW21] Benyamin Bashari and Philipp Woelfel. An efficient adaptive partial snapshot implementation. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 545–555, 2021.

[CCN12] Luca Cardelli and Attila Csikász-Nagy. The cell cycle switch computes approximate majority. *Scientific Reports*, 2, 2012.

[Cha93] Soma Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Inf. Comput.*, 105(1):132–158, 1993.

[Cha96] Tushar Deepak Chandra. Polylog randomized wait-free consensus. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 166–175, Philadelphia, Pennsylvania, USA, 23–26 May 1996.

[CHKM19] Anne Condon, Monir Hajiaghayi, David Kirkpatrick, and Ján Maňuch. Approximate majority analyses using tri-molecular chemical reaction networks. *Natural Computing*, pages 1–22, 2019.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43:685–722, July 1996.

[CIL94] Benny Chor, Amos Israeli, and Ming Li. Wait-free consensus using asynchronous hardware. *SIAM J. Comput.*, 23(4):701–712, 1994.

[CK10] Alejandro Cornejo and Fabian Kuhn. Deploying wireless networks with beeps. In *Proceedings of the 24th International Conference on Distributed Computing*, DISC'10, pages 148–162, Berlin, Heidelberg, 2010. Springer-Verlag.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[CLM+16] Michael B. Cohen, Yin Tat Lee, Gary L. Miller, Jakub Pachocki, and Aaron Sidford. Geometric median in nearly linear time. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 9–21. ACM, 2016.

[CR79] Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22:281–283, May 1979.

[CR93] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal, editors, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, pages 42–51. ACM, 1993.

[CR08] Armando Castañeda and Sergio Rajsbaum. New combinatorial topology upper and lower bounds for renaming. In Rida A. Bazzi and Boaz Patt-Shamir, editors, *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*, pages 295–304. ACM, 2008.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43:225–267, March 1996.

[CV86] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.

[DFF+23] Carole Delporte, Hugues Fauconnier, Pierre Fraigniaud, Sergio Rajsbaum, and Corentin Travers. The computational power of distributed shared-memory models with bounded-size registers. *CoRR*, abs/2309.13977, 2023.

[DH04] Robert Danek and Vassos Hadzilacos. Local-spin group mutual exclusion algorithms. In Rachid Guerraoui, editor, *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, volume 3274 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2004.

[DHW97] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *J. ACM*, 44(6):779–805, 1997.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[Dij75] Edsger W. Dijkstra. Guarded commands, non-determinacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DLP+86] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33(3):499–516, 1986.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[DN93] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, volume 740 of *Lecture Notes in Computer Science*, pages 139–147. Springer, 1993.

[Dol00] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[Dou02] John R. Douceur. The sybil attack. In Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron, editors, *Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers*, volume 2429 of *Lecture Notes in Computer Science*, pages 251–260. Springer, 2002.

[DS83] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(4):656–666, 1983.

[DS15] David Doty and David Soloveichik. Stable leader election in population protocols requires linear time. In Yoram Moses, editor, *Distributed Computing: 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 602–616, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[EGSZ20] Faith Ellen, Rati Gelashvili, Nir Shavit, and Leqi Zhu. A complexity-based classification for multiprocessor synchronization. *Distributed Computing*, 33(2):125–144, Apr 2020.

[EHS12] Faith Ellen, Danny Hendler, and Nir Shavit. On the inherent sequentiality of concurrent objects. *SIAM Journal on Computing*, 41(3):519–536, 2012.

[Eli75] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[ER+18] Robert Elsässer, Tomasz Radzik, et al. Recent results in population protocols for exact majority and leader election. *Bulletin of EATCS*, 3(126), 2018.

[EW13] Faith Ellen and Philipp Woelfel. An optimal implementation of fetch-and-increment. In Yehuda Afek, editor, *Distributed Computing*, pages 284–298, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[FH07] Keir Fraser and Timothy L. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007.

[FHS98] Faith Ellen Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, 1998.

[FHS05] Faith Ellen Fich, Danny Hendler, and Nir Shavit. Linear lower bounds on real-world implementations of concurrent objects. In *Foundations of Computer Science, Annual IEEE Symposium on*, pages 165–173, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[Fic05] Faith Fich. How hard is it to take a snapshot? In Peter Vojtáš, Mária Bieliková, Bernadette Charron-Bost, and Ondrej Sýkora, editors, *SOFSEM 2005: Theory and Practice of Computer Science*, volume 3381 of *Lecture Notes in Computer Science*, pages 28–37. Springer Berlin / Heidelberg, 2005.

[Fid91] Colin J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.

[FK07] Panagiota Fatourou and Nikolaos D. Kallimanis. Time-optimal, space-efficient single-scanner snapshots & multi-scanner snapshots using CAS. In Indranil Gupta and Roger Wattenhofer, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, pages 33–42. ACM, 2007.

[FL82] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Inf. Process. Lett.*, 14(4):183–186, 1982.

[FL87] Greg N. Frederickson and Nancy A. Lynch. Electing a leader in a synchronous ring. *J. ACM*, 34(1):98–115, 1987.

[FL06] Rui Fan and Nancy A. Lynch. An $\omega(n \log n)$ lower bound on the cost of mutual exclusion. In Eric Ruppert and Dahlia Malkhi, editors, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing, PODC 2006, Denver, CO, USA, July 23-26, 2006*, pages 275–284. ACM, 2006.

[Fle59] Ian Fleming. *Goldfinger*. Jonathan Cape, 1959.

[FLM86] Michael J. Fischer, Nancy A. Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.

[FLMS05] Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-free algorithms can be practically wait-free. In Pierre Fraigniaud, editor, *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, volume 3724 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 2005.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[For56] Lester Randolph Ford. *Network Flow Theory*. Rand Corporation, 1956.

[FR98] Michael J. Fischer and Michael O. Rabin. Super-exponential complexity of presburger arithmetic. In Bob F. Caviness and Jeremy R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 122–135. Springer Vienna, Vienna, 1998.

[FW10] Roland Flury and Roger Wattenhofer. Slotted programming for sensor networks. In Tarek F. Abdelzaher, Thiemo Voigt, and Adam Wolisz, editors, *Proceedings of the 9th International Conference on Information Processing in Sensor Networks, IPSN 2010, April 12-16, 2010, Stockholm, Sweden*, pages 24–34. ACM, 2010.

[Gaf98] Eli Gafni. Round-by-round fault detectors: Unifying synchrony and asynchrony (extended abstract). In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, pages 143–152, 1998.

[Gaf09] Eli Gafni. The extended BG-simulation and the characterization of t-resiliency. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 85–92. ACM, 2009.

[Gal82] Robert G. Gallager. Distributed minimum hop algorithms. Technical Report LIDS-P-1175, M.I.T. Laboratory for Information and Decision Systems, January 1982.

[Gel15] Rati Gelashvili. On the optimal space complexity of consensus for anonymous processes. In Yoram Moses, editor, *Distributed Computing: 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*, pages 452–466, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[GHHW12] Wojciech Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. RMR-efficient implementations of comparison primitives using read and write operations. *Distributed Computing*, 25(2):109–162, May 2012.

[GHHW13] George Giakkoupis, Maryam Helmi, Lisa Higham, and Philipp Woelfel. An $o(\sqrt{n})$ space bound for obstruction-free leader election. In *Proceedings of the 27th International Symposium on Distributed Computing (DISC)*, pages 46–60, October 14–18 2013.

[GHHW15] George Giakkoupis, Maryam Helmi, Lisa Higham, and Philipp Woelfel. Test-and-set in optimal space. In *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing*, STOC '15, pages 615–623, New York, NY, USA, 2015. ACM.

[GHW11] Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing*, STOC '11, pages 373–382, New York, NY, USA, 2011. ACM.

[GK22] Mohsen Ghaffari and Fabian Kuhn. Deterministic distributed vertex coloring: Simpler, faster, and without network decomposition. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1009–1020, 2022.

[GKL15] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310. Springer, 2015.

[GKM+19] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 307–316, New York, NY, USA, 2019. Association for Computing Machinery.

[GM98] Juan A. Garay and Yoram Moses. Fully polynomial byzantine agreement for $n > 3t$ processors in $t + 1$ rounds. *SIAM J. Comput.*, 27(1):247–290, 1998.

[Gol11] Wojciech M. Golab. A complexity separation between the cache-coherent and distributed shared memory models. In Cyril Gavoille and Pierre Fraigniaud, editors, *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 109–118. ACM, 2011.

[Gra78] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, London, UK, 1978.

[GRS90] Ronald L. Graham, Bruce L. Rothschild, and Joel H. Spencer. *Ramsey Theory*. Wiley-Interscience, 2nd edition, 1990.

[GW12a] George Giakkoupis and Philipp Woelfel. On the time and space complexity of randomized test-and-set. In Darek Kowalski and

Alessandro Panconesi, editors, *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 19–28. ACM, 2012.

[GW12b] George Giakkoupis and Philipp Woelfel. A tight RMR lower bound for randomized mutual exclusion. In *Proceedings of the 44th symposium on Theory of Computing*, pages 983–1002. ACM, 2012.

[GW14] G. Giakkoupis and P. Woelfel. Randomized mutual exclusion with constant amortized RMR complexity on the DSM. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 504–513, Oct 2014.

[GW17] George Giakkoupis and Philipp Woelfel. Randomized abortable mutual exclusion with constant amortized RMR complexity on the cc model. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 221–229, New York, NY, USA, 2017. ACM.

[Her91a] Maurice Herlihy. Impossibility results for asynchronous PRAM (extended abstract). In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, SPAA '91, pages 327–336, New York, NY, USA, 1991. ACM.

[Her91b] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.

[HFP02] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In Dahlia Malkhi, editor, *Distributed Computing, 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002 Proceedings*, volume 2508 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 2002.

[HHT20] Vassos Hadzilacos, Xing Hu, and Sam Toueg. On linearizability and the termination of randomized algorithms, 2020.

[HK14] Danny Hendler and Vitaly Khait. Complexity tradeoffs for read and update operations. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 186–195, New York, NY, USA, 2014. ACM.

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA*, pages 522–529. IEEE Computer Society, 2003.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[HS80] Daniel S. Hirschberg and J. B. Sinclair. Decentralized extrema-finding in circular configurations of processors. *Commun. ACM*, 23(11):627–628, 1980.

[HS91] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. In Alfred J. Menezes and Scott A. Vanstone, editors, *Advances in Cryptology-CRYPTO' 90*, pages 437–455, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[HS99] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *J. ACM*, 46(6):858–923, 1999.

[HW90] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[HW11] Danny Hendler and Philipp Woelfel. Randomized mutual exclusion with sub-logarithmic RMR-complexity. *Distributed Computing*, 24(1):3–19, 2011.

[IMCT94] Michiko Inoue, Toshimitsu Masuzawa, Wei Chen, and Nobuki Tokura. Linear-time snapshot using multi-writer multi-reader registers. In Gerard Tel and Paul Vitányi, editors, *Distributed Algorithms*, volume 857 of *Lecture Notes in Computer Science*, pages 130–140. Springer Berlin / Heidelberg, 1994.

[IR09] Damien Imbs and Michel Raynal. Visiting Gafni's reduction land: From the BG simulation to the extended BG simulation.

In *Stabilization, Safety, and Security of Distributed Systems*, pages 369–383. Springer, 2009.

[Jay97] Prasad Jayanti. Robust wait-free hierarchies. *J. ACM*, 44(4):592–614, 1997.

[Jay98] Prasad Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '98, pages 201–210, New York, NY, USA, 1998. ACM.

[Jay02] Prasad Jayanti. f-arrays: implementation and applications. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 270–279, New York, NY, USA, 2002. ACM.

[Jay11] Prasad Jayanti. personal communication, 19 October 2011.

[JT92] Prasad Jayanti and Sam Toueg. Some results on the impossibility, universality, and decidability of consensus. In Adrian Segall and Shmuel Zaks, editors, *Distributed Algorithms: 6th International Workshop, WDAG '92 Haifa, Israel, November 2–4, 1992 Proceedings*, pages 69–84, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[JTT00] Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 30(2):438–456, 2000.

[Kaw00] Jawal Y. Kawash. *Limitations and Capabilities of Weak Memory Consistency Systems*. PhD thesis, University of Calgary, January 2000.

[KP09] Elias Koutsoupias and Christos Papadimitriou. Worst-case equilibria. *Computer Science Review*, 3(2):65–69, 2009.

[LAA87] Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In Franco P. Preparata, editor, *Parallel and Distributed Computing*, volume 4 of *Advances in Computing Research*, pages 163–183. JAI Press, 1987.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[Lam77] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, C-28(9):690–691, Sept 1979.

[Lam83] Leslie Lamport. The weak byzantine generals problem. *J. ACM*, 30(3):668–676, 1983.

[Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[Lam01] Leslie Lamport. Paxos made simple. *SIGACT News*, 32(4):18–25, 2001.

[Lin92] Nathan Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.

[LL77] Gérard Le Lann. Distributed systems—towards a formal approach. In B. Gilchrist, editor, *Information Processing 77*, pages 155–160. North-Holland, 1977.

[LPS23] Jacob Leshno, Rafael Pass, and Elaine Shi. Can open decentralized ledgers be economically secure? Cryptology ePrint Archive, Paper 2023/1516, 2023. https://eprint.iacr.org/2023/1516.

[LS14] Juhana Laurinharju and Jukka Suomela. Brief announcement: Linial's lower bound made easy. In Magnús M. Halldórsson and Shlomi Dolev, editors, *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 377–378. ACM, 2014.

[LT87] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 137–151, New York, NY, USA, 1987. ACM.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[MA95] Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Program.*, 25(1):1–39, 1995.

[Mat93] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *J. Parallel Distrib. Comput.*, 18(4):423–434, 1993.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[Mor95] Shlomo Moran. Using approximate agreement to obtain complete disagreement: the output structure of input-free asynchronous computations. In *Third Israel Symposium on the Theory of Computing and Systems*, pages 251–257, January 1995.

[MPR18] Achour Mostefaoui, Matthieu Perrin, and Michel Raynal. A simple object that spans the whole consensus hierarchy. *Parallel Processing Letters*, 28(02):1850006, 2018.

[MPRJ17] Achour Mostéfaoui, Matoula Petrolia, Michel Raynal, and Claude Jard. Atomic read/write memory in signature-free byzantine asynchronous message-passing systems. *Theory of Computing Systems*, 60(4):677–694, May 2017.

[MR98] Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[MR10] Michael Merideth and Michael Reiter. Selected results from the latest decade of quorum systems research. In Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors, *Replication*, volume 5959 of *Lecture Notes in Computer Science*, pages 185–206. Springer, 2010.

[MRRT08] Achour Mostefaoui, Sergio Rajsbaum, Michel Raynal, and Corentin Travers. The combined power of conditions and information on failures to solve asynchronous set agreement. *SIAM Journal on Computing*, 38(4):1574–1601, 2008.

[MRWW01] Dahlia Malkhi, Michael K. Reiter, Avishai Wool, and Rebecca N. Wright. Probabilistic quorum systems. *Inf. Comput.*, 170(2):184–206, 2001.

[MRZ15] Yves Métivier, John Michael Robson, and Akka Zemmari. On distributed computing with beeps. *CoRR*, abs/1507.02721, 2015.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[NT87] Gil Neiger and Sam Toueg. Substituting for real time and common knowledge in asynchronous distributed systems. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 281–293, New York, NY, USA, 1987. ACM.

[NW98] Moni Naor and Avishai Wool. The load, capacity, and availability of quorum systems. *SIAM J. Comput.*, 27(2):423–447, 1998.

[Oka99] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

[OO14] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In Garth Gibson and Nickolai Zeldovich, editors, *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, pages 305–319. USENIX Association, 2014.

[Pel00] David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, 2000.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Technische Hochschule Darmstadt, 1962.

[Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[Pet82] Gary L. Peterson. An $O(n \log n)$ unidirectional algorithm for the circular extrema problem. *ACM Trans. Program. Lang. Syst.*, 4(4):758–762, 1982.

[PF77] Gary L. Peterson and Michael J. Fischer. Economical solutions for the critical section problem in a distributed system (extended abstract). In John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison, editors, *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 91–97. ACM, 1977.

[Plo89] S. A. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, PODC '89, pages 159–175, New York, NY, USA, 1989. ACM.

[Pos81] J. Postel. Transmission Control Protocol. RFC 793 (INTERNET STANDARD), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[PR01] Alessandro Panconesi and Romeo Rizzi. Some simple distributed algorithms for sparse networks. *Distributed Comput.*, 14(2):97–100, 2001.

[Pre29] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes-Rendus du I Congrès de Mathématiciens des Pays Slaves, Warszawa*, pages 92–101, 1929.

[Pri01] Giusseppe Principe. CORDA: Distributed coordination of a set of autonomous mobile robots. In *Proceedings of the European Research Seminar on Advances in Distributed Systems*, pages 185–190, 2001.

[PS18] Rafael Pass and Elaine Shi. Rethinking large-scale consensus. *IACR Cryptol. ePrint Arch.*, page 302, 2018.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[PVV09] Etienne Perron, Dinkar Vasudevan, and Milan Vojnovic. Using three states for binary consensus on complete graphs. In *IEEE INFOCOM 2009*, pages 2527–2535. IEEE, 2009.

[PW95] David Peleg and Avishai Wool. The availability of quorum systems. *Inf. Comput.*, 123(2):210–223, 1995.

[PW97a] David Peleg and Avishai Wool. The availability of crumbling wall quorum systems. *Discrete Applied Mathematics*, 74(1):69–83, 1997.

[PW97b] David Peleg and Avishai Wool. Crumbling walls: A class of practical and efficient quorum systems. *Distributed Computing*, 10(2):87–97, 1997.

[Rab83] M. O. Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 403–409, Nov 1983.

[RST01] Yaron Riany, Nir Shavit, and Dan Touitou. Towards a practical snapshot algorithm. *Theor. Comput. Sci.*, 269(1-2):163–201, 2001.

[Rup00] Eric Ruppert. Determining consensus numbers. *SIAM J. Comput.*, 30(4):1156–1168, 2000.

[Sch73] Flora Rheta Schreiber. *Sybil*. Regnery, 1973.

[Sch95] Eric Schenk. Faster approximate agreement with multi-writer registers. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*, pages 714–723. IEEE Computer Society, 1995.

[Spe28] E. Sperner. Neuer Beweis für die Invarianz der Dimensionszahl und des Gebietes. *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, 6:265–272, 1928. 10.1007/BF02940617.

[SRS08] Christian Scheideler, Andréa W. Richa, and Paolo Santi. An o(log n) dominating set protocol for wireless ad-hoc networks under the physical interference model. In Xiaohua Jia, Ness B. Shroff, and Peng-Jun Wan, editors, *Proceedings of the 9th ACM Interational Symposium on Mobile Ad Hoc Networking*

and Computing, MobiHoc 2008, Hong Kong, China, May 26-30, 2008, pages 91–100. ACM, 2008.

[SSW91] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus - making resilient algorithms fast in practice. In *Proc. of the 2nd ACM Symposium on Discrete Algorithms (SODA)*, pages 351–362, 1991.

[ST97] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[SY99] Ichiro Suzuki and Masafumi Yamashita. Distributed anonymous mobile robots: formation of geometric patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999.

[SZ00] Michael E. Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM J. Comput.*, 29(5):1449–1483, 2000.

[TV02] John Tromp and Paul M. B. Vitányi. Randomized two-process wait-free test-and-set. *Distributed Computing*, 15(3):127–135, 2002.

[vABHL03] Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. Captcha: Using hard ai problems for security. In Eli Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, pages 294–311, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[VL92] George Varghese and Nancy A. Lynch. A tradeoff between safety and liveness for randomized coordinated attack protocols. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '92, pages 241–250, New York, NY, USA, 1992. ACM.

[Wel87] Jennifer L. Welch. Simulating synchronous processors. *Inf. Comput.*, 74(2):159–170, 1987.

[YA95] Jae-Heon Yang and James H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995.

[YKGF06] Haifeng Yu, Michael Kaminsky, Phillip B. Gibbons, and Abraham Flaxman. Sybilguard: Defending against sybil attacks

via social networks. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '06, page 267–278, New York, NY, USA, 2006. Association for Computing Machinery.

[Yu06] Haifeng Yu. Signed quorum systems. *Distributed Computing*, 18(4):307–323, 2006.

[Zhu16] Leqi Zhu. A tight space bound for consensus. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 345–350. ACM, 2016.

Index