Distributed Systems
University of Cambridge
Computer Science Tripos, Part IB
Michaelmas term 2020/21

Course web page: https://www.cst.cam.ac.uk/teaching/2021/ConcDisSys

Lecture videos: https://www.youtube.com/playlist?list=PLeKd45zvjcDFUEv_ohr_HdUFe97RItdiB

Dr. Martin Kleppmann
mk428@cst.cam.ac.uk

1 Introduction

This 8-lecture course on distributed systems forms the second half of *Concurrent and Distributed Systems*. While the first half focussed on concurrency among multiple processes or threads running on the same computer, this second half takes things further by examining systems consisting of multiple communicating computers.

Concurrency on a single computer is also known as *shared-memory concurrency*, since multiple threads running in the same process have access to the same address space. Thus, data can easily be passed from one thread to another: a variable or pointer that is valid for one thread is also valid for another.

This situation changes when we move to distributed systems. We still have concurrency in a distributed system, since different computers can execute programs in parallel. However, we don't typically have shared memory, since each computer in a distributed system runs its own operating system with its own address space, using the memory built into that computer. Different computers can only communicate by sending each other messages over a network.

(Limited forms of distributed shared memory exist in some supercomputers and research systems, and there are technologies like *remote direct memory access* (RDMA) that allow computers to access each others' memory over a network. Also, databases can in some sense be regarded as shared memory, but with a different data model compared to byte-addressable memory. However, broadly speaking, most practical distributed systems are based on message-passing.)

Each of the computers in a distributed system is called a *node*. Here, "computer" is interpreted quite broadly: nodes might be desktop computers, servers in datacenters, mobile devices, internet-connected cars, industrial control systems, sensors, or many other types of device. In this course we don't distinguish them: a node can be any type of communicating computing device.

A distributed system is. . .

- ► *". . . a system in which the failure of a computer you didn't even know existed can render your own computer unusable."* — Leslie Lamport
- ► . . . multiple computers communicating via a network. . .
- ► . . . trying to achieve some task together
- ► Consists of "nodes" (computer, phone, car, robot, . . .)

Start of video section 1.1
(mp4 download)

Slide 1

1.1 About distributed systems

These notes and the lecture recordings should be self-contained, but if you would like to read up on further detail, there are several suggested textbooks:

- Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. ISBN 978-1543057386. Free download from https://www.distributed-systems.net/index.php/books/ds3/ (third edition, 2017).

  This book gives a broad overview over a large range of distributed systems topics, with lots of examples from practical systems.

- Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Second edition, Springer, 2011. ISBN 978-3-642-15259-7.

  Ebook download for Cambridge users: https://link.springer.com/book/10.1007/978-3-642-15260-3 then click Log in → via Shibboleth → type *University of Cambridge* → log in with Raven.

  This book is more advanced, going into depth on several important distributed algorithms, and proving their correctness. Recommended if you want to explore the theory in greater depth than this course covers.

- Martin Kleppmann. *Designing Data-Intensive Applications*, O'Reilly, 2017. ISBN 978-1449373320.

  This book goes more in the direction of databases, but also covers a number of distributed systems topics. It is designed for software engineers in industry working with distributed databases.

- Jean Bacon and Tim Harris. *Operating Systems: Concurrent and Distributed Software Design*. Addison-Wesley, 2003. ISBN 978-0321117892.

  This book provides a link to the *concurrent systems* half of the course, and to operating systems topics.

Where appropriate, these lecture notes also contain references to research papers and other useful background reading (these are given in square brackets, and the details appear at the end of this document). However, only material covered in the lecture notes and videos is examinable.

Recommended reading

▶ van Steen & Tanenbaum.
  "**Distributed Systems**"
  (any ed), free ebook available
▶ Cachin, Guerraoui & Rodrigues.
  "**Introduction to Reliable and Secure Distributed Programming**" (2nd ed), Springer 2011
▶ Kleppmann.
  "**Designing Data-Intensive Applications**",
  O'Reilly 2017
▶ Bacon & Harris.
  "**Operating Systems: Concurrent and Distributed Software Design**", Addison-Wesley 2003

Slide 2

The syllabus, slides, and lecture notes for this course have been substantially revised for 2020/21. This means that new mistakes may have crept in. If you notice anything wrong, or if anything is unclear, please let me know by email (mk428@cst.cam.ac.uk)!

As for other courses, past exam questions are available at https://www.cl.cam.ac.uk/teaching/exams/pastpapers/t-ConcurrentandDistributedSystems.html. Because of syllabus changes, the following past exam questions are no longer applicable: 2018 P5 Q8; 2015 P5 Q8; 2014 P5 Q9 (a); 2013 P5 Q9; 2011 P5 Q8 (b).

These notes also contain exercises, which are suggested material for discussion in supervisions. Solution notes for supervisors are available from the course web page.

This course is related to several other courses in the tripos, as shown on Slide 3.

Relationships with other courses

- **Concurrent Systems** – Part IB
  (every distributed system is also concurrent)
- **Operating Systems** – Part IA
  (inter-process communication, scheduling)
- **Databases** – Part IA
  (many modern databases are distributed)
- **Computer Networking** – Part IB Lent term
  (distributed systems involve network communication)
- **Further Java** – Part IB Michaelmas
  (distributed programming practical exercises)
- **Security** – Part IB Easter term
  (network protocols with encryption & authentication)
- **Cloud Computing** – Part II
  (distributed systems for processing large amounts of data)

There are a number of reasons for creating distributed systems. Some applications are *intrinsically distributed*: if you want to send a message from your phone to your friend's phone, that operation inevitably requires those phones to communicate via some kind of network.

Some distributed systems do things that in principle a single computer could do, but they do it *more reliably*. A single computer can fail and might need to be rebooted from time to time, but if you are using multiple nodes, then one node can continue serving users while another node is rebooting. Thus, a distributed system has the potential to be more reliable than a single computer, at least if it is well-designed (somewhat contradicting Lamport's quote on Slide 1)!

Another reason for distribution is for better *performance*: if a service has users all over the world, and they all have to access a single node, then either the users in the UK or the users in New Zealand are going to find it slow (or both). By placing nodes in multiple locations around the world, we can get around the slowness of the speed of light by routing each user to a nearby node.

Finally, some large-scale data processing or computing tasks are simply *too big* to perform on a single computer, or would be intolerably slow. For example, the Large Hadron Collider at CERN is supported by a worldwide computing infrastructure with 1 million CPU cores for data analysis, and 1 exabyte (10^{18} bytes) of storage! See https://wlcg-public.web.cern.ch/.

Why make a system distributed?

- **It's inherently distributed:**
  e.g. sending a message from your mobile phone to your friend's phone
- **For better reliability:**
  even if one node fails, the system as a whole keeps functioning
- **For better performance:**
  get data from a nearby node rather than one halfway round the world
- **To solve bigger problems:**
  e.g. huge amounts of data, can't fit on one machine

However, there are also downsides to distributed systems, because things can go wrong, and the system needs to deal with such faults. The network may fail, leaving the nodes unable to communicate.
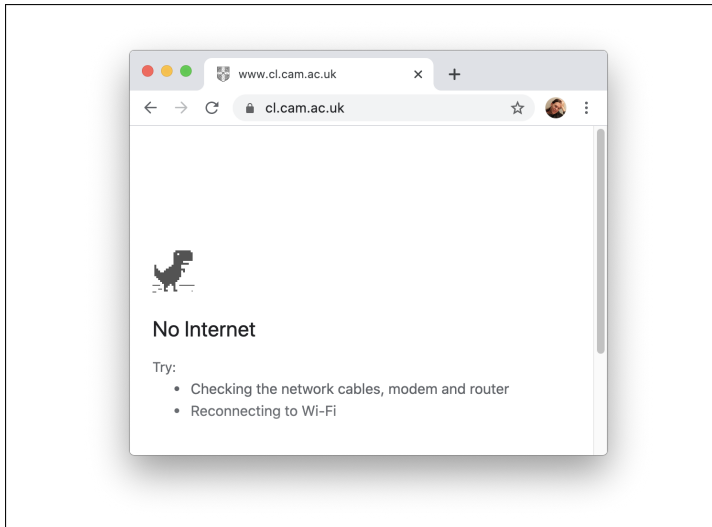
Slide 5

Another thing that can go wrong is that a node may crash, or run much slower than usual, or misbehave in some other way (perhaps due to a software bug or a hardware failure). If we want one node to take over when another node crashes, we need to detect that a crash has happened; as we shall see, even that is not straightforward. Network failures and node failures can happen at any moment, without warning.

In a single computer, if one component fails (e.g. one of the RAM modules develops a fault), we normally don't expect the computer to continue working nevertheless: it will probably just crash. Software does not need to be written in a way that explicitly deals with faulty RAM. However, in a distributed system we often *do* want to tolerate some parts of the system being broken, and for the rest to continue working. For example, if one node has crashed (a *partial failure*), the remaining nodes may still be able to continue providing the service.

If one component of a system stops working, we call that a *fault*, and many distributed systems strive to provide *fault tolerance*: that is, the system as a whole continues functioning despite the fault. Dealing with faults is what makes distributed computing fundamentally different, and often harder, compared to programming a single computer. Some distributed system engineers believe that if you can solve a problem on a single computer, it is basically easy! Though, in fairness to our colleagues in other areas of computer science, this is probably not true.
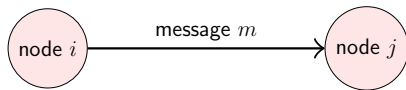

Slide 6

1.2 Distributed systems and computer networking

When studying distributed systems, we usually work with a high-level abstraction of the hardware.

Slide 7

In this course, we just assume that there is some way for one node to send a message to another node. We don't particularly care how that message is physically represented or encoded – the network protocols, informally known as the *bytes on the wire* – because the basic principle of sending and receiving messages remains the same, even as particular networking technologies come and go. The "wire" may actually be radio waves, lasers, a USB thumb drive in someone's pocket, or even hard drives in a van.

Slide 8

Indeed, if you want to send a very large message (think tens of terabytes), it would be slow to send that data over the Internet, and it is in fact faster to write that data to a bunch of hard drives, load them into a van, and to drive them to their destination. But from a distributed systems point of view, the method of delivering the message is not important: we only see an abstract communication channel with a certain *latency* (delay from the time a message is sent until it is received) and *bandwidth* (the volume of data that can be transferred per unit time).

The Computer Networking course in Lent term focusses on the network protocols that enable messages to get to their destination. The study of distributed systems builds upon that facility, and instead focusses on how several nodes should coordinate in order to achieve some shared task. The design of distributed algorithms is about deciding what messages to send, and how to process the messages when they are received.

The web is an example of a distributed system that you use every day.
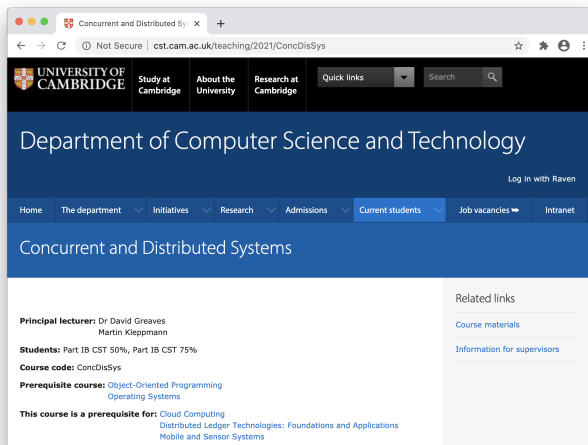
Slide 10

Client-server example: the web

Time flows from top to bottom.

client		server www.cst.cam.ac.uk

GET /teaching/2021/ConcDisSys

<!DOCTYPE html><html>...
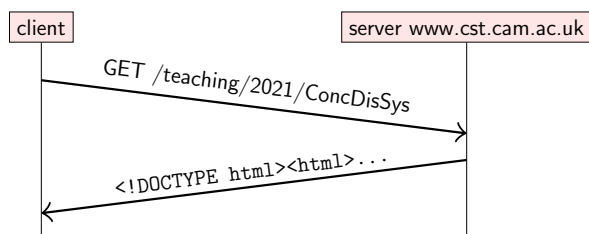
Slide 11

In the web there are two main types of nodes: *servers* host websites, and *clients* (web browsers) display them. When you load a web page, your web browser sends a *HTTP request* message to the appropriate server. On receiving that request, the web server sends a *response* message containing the

6

page contents to the client that requested it. These messages are normally invisible, but we can capture and visualise the network traffic with a tool such as Charles (https://www.charlesproxy.com/), shown on Slide 12. The lecture video includes a demo of this software in action.
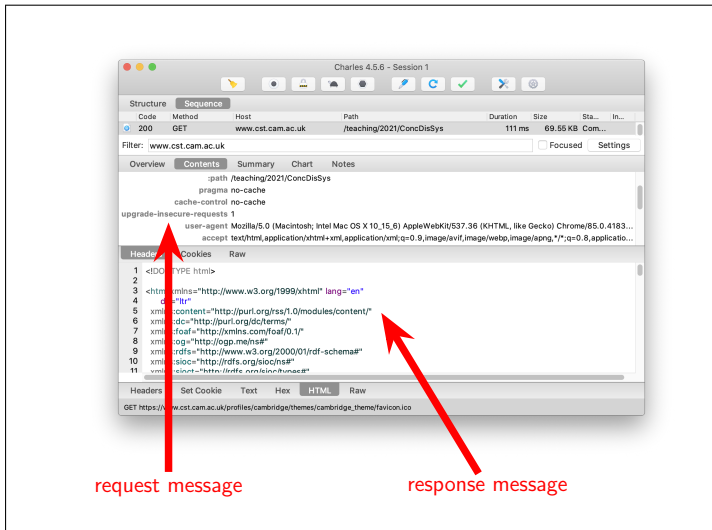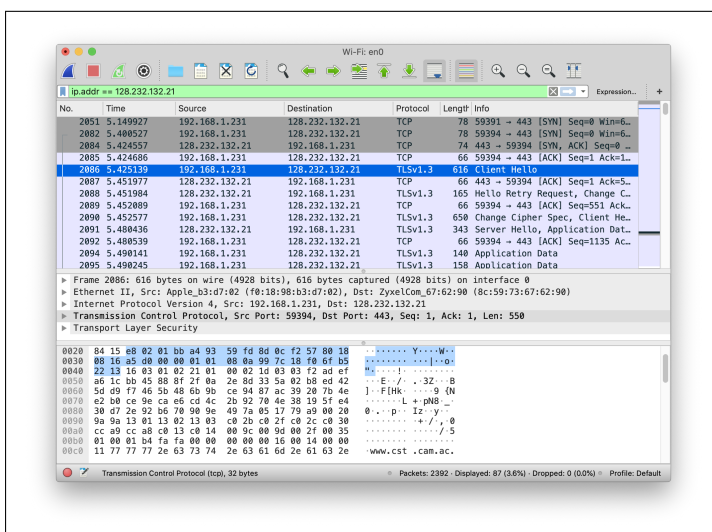

Slide 12

In a URL, the part between the `//` and the following `/` is the hostname of the server to which the client is going to send the request (e.g. `www.cst.cam.ac.uk`), and the rest (e.g. `/teaching/2021/ConcDisSys`) is the path that the client asks for in its request message. Besides the path, the request also contains some extra information, such as the HTTP method (e.g. `GET` to load a page, or `POST` to submit a form), the version of the client software (the *user-agent*), and a list of file formats that the client understands (the *accept header*). The response message contains the file that was requested, and an indicator of its file format (the *content-type*); in the case of a web page, this might be a HTML document, an image, a video, a PDF document, or any other type of file.

Since the requests and responses can be larger than we can fit in a single network packet, the HTTP protocol runs on top of TCP, which breaks down a large chunk of data into a stream of small network packets (see Slide 13), and puts them back together again at the recipient. HTTP also allows multiple requests and multiple responses to be sent over a single TCP connection. However, when looking at this protocol from a distributed systems point of view, this detail is not important: we treat the request as one message and the response as another message, regardless of the number of physical network packets involved in transmitting them. This keeps things independent of the underlying networking technology.

Slide 13

1.3 Example: Remote Procedure Calls (RPC)

Another example of an everyday distributed system is when you buy something online using a credit/debit card. When you enter your card number in some online shop, that shop will send a payment request over the Internet to a service that specialises in processing card payments.
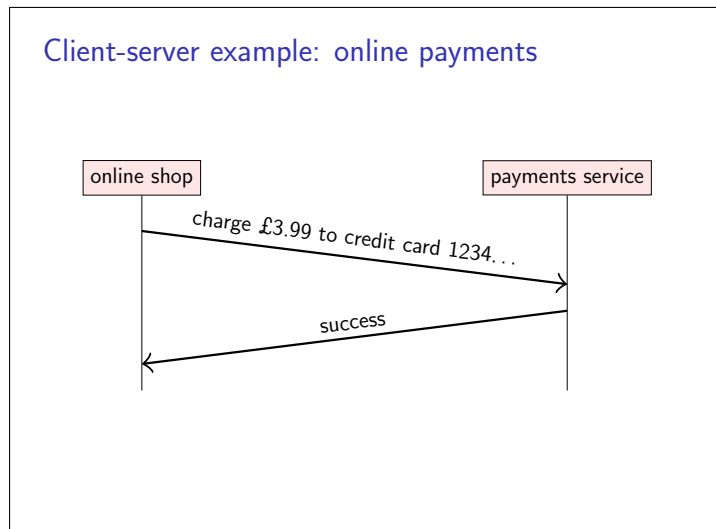
Slide 14

The payments service in turn communicates with a card network such as Visa or MasterCard, which communicates with the bank that issued your card in order to take the payment.

For the programmers who are implementing the online shop, the code for processing the payment may look something like the code on Slide 15.

Slide 15

Calling the `processPayment` function looks like calling any other function, but in fact, what is happening behind the scenes is that the shop is sending a request to the payments service, waiting for a response, and then returning the response it received. The actual implementation of `processPayment` – the logic that communicates with the card network and the banks – does not exist in the code of the shop: it is part of the payments service, which is another program running on another node belonging to a different company.

This type of interaction, where code on one node appears to call a function on another node, is called a *Remote Procedure Call* (RPC). In Java, it is called *Remote Method Invocation* (RMI). The software that implements RPC is called an *RPC framework* or *middleware*. (Not all middleware is based on RPC; there is also middleware that uses different communication models.)

When an application wishes to call a function on another node, the RPC framework provides a *stub* in its place. The stub has the same type signature as the real function, but instead of executing the real function, it encodes the function arguments in a message and sends that message to the remote

node, asking for that function to be called. The process of encoding the function arguments is known as *marshalling*. In the example on Slide 16, a JSON encoding is used for marshalling, but various other formats are also used in practice.
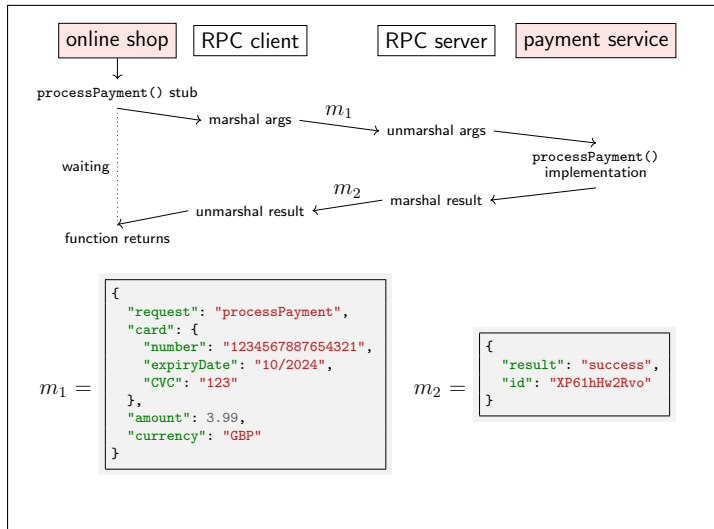
Slide 16

The sending of the message from the RPC client to the RPC server may happen over HTTP (in which case this is also called a *web service*), or one of a range of different network protocols may be used. On the server side, the RPC framework unmarshals (decodes) the message and calls the desired function with the provided arguments. When the function returns, the same happens in reverse: the function's return value is marshalled, sent as a message back to the client, unmarshalled by the client, and returned by the stub. Thus, to the caller of the stub, it looks as if the function had executed locally.

Slide 17

Exercise 1. *Networks and nodes might fail. What are the implications for code that calls another node through RPC? How is RPC different from a local function call? Is location transparency achievable?*

Over the decades many variants of RPC have been developed, with the goal of making it easier to program distributed systems. This includes object-oriented middleware such as CORBA in the 1990s. However, the underlying distributed systems challenges have remained the same [Waldo et al., 1994].

Today, the most common form of RPC is implemented using JSON data sent over HTTP. A popular set of design principles for such HTTP-based APIs is known as *representational state transfer* or *REST* [Fielding, 2000], and APIs that adhere to these principles are called *RESTful*. These principles include:

- communication is stateless (each request is self-contained and independent from other requests),

- resources (objects that can be inspected and manipulated) are represented by URLs, and

- the state of a resource is updated by making a HTTP request with a standard method type, such as POST or PUT, to the appropriate URL.

The popularity of REST is due to the fact that JavaScript code running in a web browser can easily make this type of HTTP request, as shown in Slide 19. In modern websites it is very common to use JavaScript to make HTTP requests to a server without reloading the whole page. This technique is sometimes known as *Ajax*.

RPC/REST in JavaScript

```javascript
let args = {amount: 3.99, currency: 'GBP', /*...*/};
let request = {
  method:  'POST',
  body:    JSON.stringify(args),
  headers: {'Content-Type': 'application/json'}
};

fetch('https://example.com/payments', request)
  .then((response) => {
    if (response.ok) success(response.json());
    else failure(response.status); // server error
  })
  .catch((error) => {
    failure(error); // network error
  });
```

Slide 19

The code on Slide 19 takes the arguments `args`, marshals them to JSON using `JSON.stringify()`, and then sends them to the URL `https://example.com/payments` using a HTTP POST request. There are three possible outcomes: either the server returns a status code indicating success (in which case we unmarshal the response using `response.json()`), or the server returns a status code indicating an error, or the request fails because no response was received from the server (most likely due to a network interruption). The code calls either the `success()` or the `failure()` function in each of these cases.

Even though RESTful APIs and HTTP-based RPC originated on the web (where the client is JavaScript running in a web browser), they are now also commonly used with other types of client (e.g. mobile apps), or for server-to-server communication.

Slide 20

Such server-to-server RPC is especially common in large enterprises, whose software systems are too large and complex to run in a single process on a single machine. To manage this complexity, the system is broken down into multiple services, which are developed and administered by different teams and which may even be implemented in different programming languages. RPC frameworks facilitate the communication between these services.

When different programming languages are used, the RPC framework needs to convert datatypes such that the caller's arguments are understood by the code being called, and likewise for the function's return value. A typical solution is to use an *Interface Definition Language* (IDL) to provide language-independent type signatures of the functions that are being made available over RPC. From the IDL, software developers can then automatically generate marshalling/unmarshalling code and RPC stubs for the respective programming languages of each service and its clients. Slide 21 shows an example of the IDL used by gRPC, called *Protocol Buffers*. The details of the language are not important for this course.

gRPC IDL example

```
message PaymentRequest {
  message Card {
    required string cardNumber  = 1;
    optional int32  expiryMonth = 2;
    optional int32  expiryYear  = 3;
    optional int32  CVC         = 4;
  }
  enum Currency { GBP = 1; USD = 2; }

  required Card     card     = 1;
  required int64    amount   = 2;
  required Currency currency = 3;
}

message PaymentStatus {
  required bool   success      = 1;
  optional string errorMessage = 2;
}

service PaymentService {
  rpc ProcessPayment(PaymentRequest) returns (PaymentStatus) {}
}
```

Slide 21

2 Models of distributed systems

A *system model* captures our assumptions about how nodes and the network behave. It is an abstract description of their properties, which can be implemented by various technologies in practice. To illustrate common system models, we will start this section by looking at two classic thought experiments in distributed systems: the *two generals problem* and the *Byzantine generals problem*.

2.1 The two generals problem

In the two generals problem [Gray, 1978], we imagine two generals, each leading an army, who want to capture a city. (Apologies for the militaristic analogy – I didn't come up with it!) The city's defences are strong, and if only one of the two armies attacks, the army will be defeated. However, if both armies attack at the same time, they will successfully capture the city.
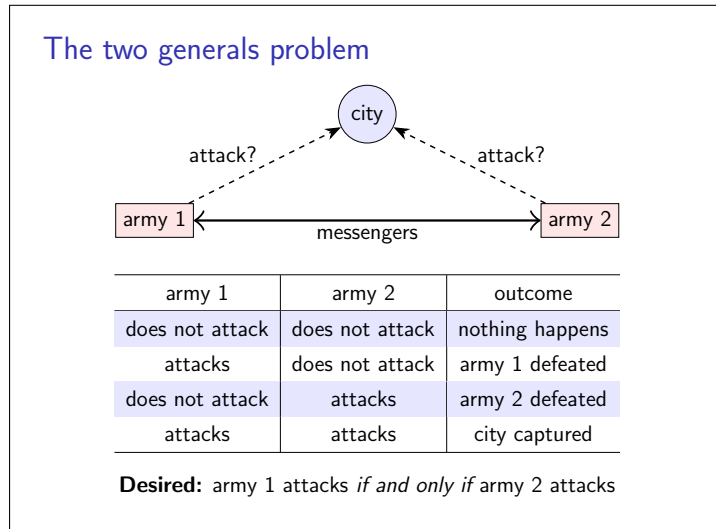
Slide 22

Thus, the two generals need to coordinate their attack plan. This is made difficult by the fact that the two armies are camped some distance apart, and they can only communicate by messenger. The messengers must pass through territory controlled by the city, and so they are sometimes captured. Thus, a message sent by one general may or may not be received by the other general, and the sender does not know whether their message got through, except by receiving an explicit reply from the other party. If a general does not receive any messages, it is impossible to tell whether this is because the other general didn't send any messages, or because all messengers were captured.
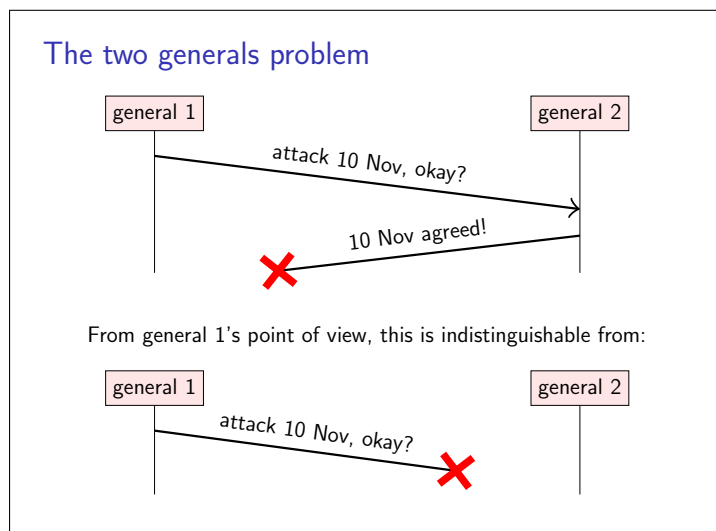
Slide 23

What protocol should the two generals use to agree on a plan? For each general there are two options: either the general promises to go ahead with the attack in any case (even if no response is received), or the general waits for an acknowledgement before committing to attack. In the first case, the general who promises to go ahead risks being alone in the attack. In the second case, the general who awaits acknowledgement shifts the problem to the other general, who must now decide whether to commit to attack (and risk being alone) or wait for an acknowledgement of the acknowledgement.

How should the generals decide?

1. General 1 always attacks, even if no response is received?
   - ▶ Send lots of messengers to increase probability that one will get through
   - ▶ If all are captured, general 2 does not know about the attack, so general 1 loses

2. General 1 only attacks if positive response from general 2 is received?
   - ▶ Now general 1 is safe
   - ▶ But general 2 knows that general 1 will only attack if general 2's response gets through
   - ▶ Now general 2 is in the same situation as general 1 in option 1

No common knowledge: the only way of knowing something is to communicate it

Slide 24

The problem is that no matter how many messages are exchanged, neither general can ever be certain that the other army will also turn up at the same time. A repeated sequence of back-and-forth acknowledgements can build up gradually increasing confidence that the generals are in agreement, but it can be proved that they cannot reach certainty by exchanging any finite number of messages.

This thought experiment demonstrates that in a distributed system, there is no way for one node to have certainty about the state of another node. The only way how a node can know something is by having that knowledge communicated in a message. On a philosophical note, this is perhaps similar to communication between humans: we have no telepathy, so the only way for someone else to know what you are thinking is by communicating it (through speech, writing, body language, etc).

As a practical example of the two generals problem, Slide 25 adapts the model from Slide 22 to the application of paying for goods in an online shop. The shop and the credit card payment processing service communicate per RPC, and some of these messages may be lost. Nevertheless, the shop wants to ensure that it dispatches the goods only if they are paid for, and it only charges the customer card if the goods are dispatched.

The two generals problem applied
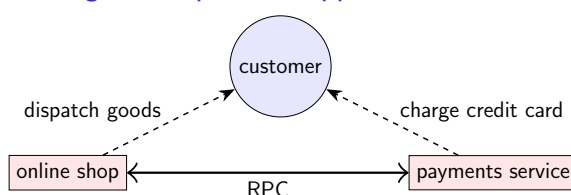
online shop	payments service	outcome
does not dispatch	does not charge	nothing happens
dispatches	does not charge	shop loses money
does not dispatch	charges	customer complaint
dispatches	charges	everyone happy

Desired: online shop dispatches *if and only if* payment made

Slide 25

In practice, the online shopping example does not exactly match the two generals problem: in this scenario, it is safe for the payments service to always go ahead with a payment, because if the shop ends up not being able to dispatch the goods, it can refund the payment. The fact that a payment is something that can be undone (unlike an army being defeated) makes the problem solvable. If the communication between shop and payment service is interrupted, the shop can wait until the connection is restored, and then query the payments service to find out the status of any transactions whose outcome was unknown.

2.2   The Byzantine generals problem

The Byzantine generals problem [Lamport et al., 1982] has a similar setting to the two generals problem. Again we have armies wanting to capture a city, though in this case there can be three or more. Again generals communicate by messengers, although this time we assume that if a message is sent, it is always delivered correctly.
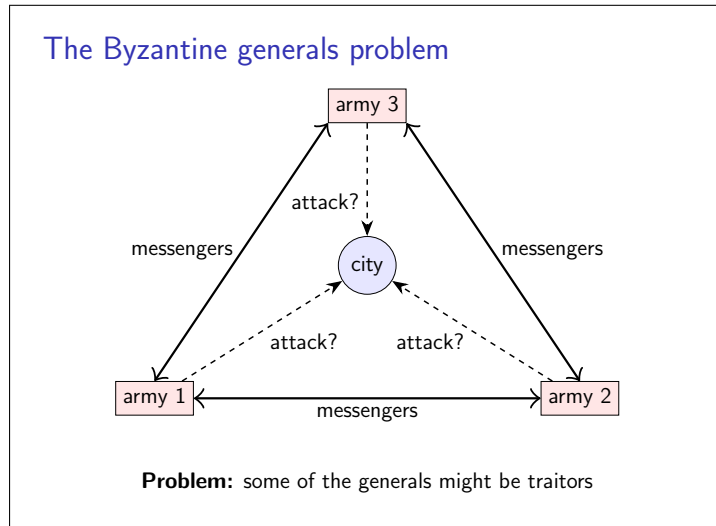
Slide 26

The challenge in the Byzantine setting is that some generals might be "traitors": that is, they might try to deliberately and maliciously mislead and confuse the other generals. One example of such malicious behaviour is shown on Slide 27: here, general 3 receives two contradictory messages from generals 1 and 2. General 1 tells general 3 to attack, whereas general 2 claims that general 1 ordered a retreat. It is impossible for general 3 to determine whether general 2 is lying (the first case), or whether general 2 is honest while general 1 is issuing contradictory orders (the second case).
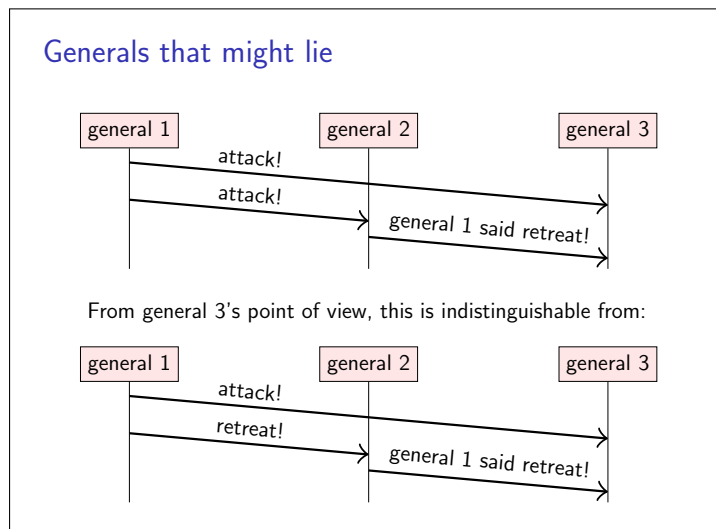
Slide 27

The honest generals don't know who the malicious generals are, but the malicious generals may collude and secretly coordinate their actions. We might even assume that all of the malicious generals are controlled by an evil adversary. The Byzantine generals problem is then to ensure that all honest generals agree on the same plan (e.g. whether to attack or to retreat). It is impossible to specify what the malicious generals are going to do, so the best we can manage is to get the honest generals to agree.

This is difficult: in fact, it can be proved that some variants of the problem can be solved only if strictly fewer than one third of the generals are malicious. That is, in a system with $3f + 1$ generals, no more than f may be malicious. For example, a system with 4 generals can tolerate $f = 1$ malicious general, and a system with 7 generals can tolerate $f = 2$.

The Byzantine generals problem

► Up to f generals might behave maliciously

► Honest generals don't know who the malicious ones are

► The malicious generals may collude

► Nevertheless, honest generals must agree on plan

► Theorem: need $3f + 1$ generals in total to tolerate f malicious generals (i.e. $< \frac{1}{3}$ may be malicious)

► Cryptography (digital signatures) helps – but problem remains hard

Slide 28

The problem is made somewhat easier if generals use cryptography (*digital signatures*) to prove who said what: for example, this would allow general 2 to prove to general 3 what general 1's order was, and thus demonstrate general 2's honesty. We will not go into details of digital signatures in this course, as they are covered in the Security course (Part IB Easter term). However, even with signatures, the Byzantine generals problem remains challenging.

Is the Byzantine generals problem of practical relevance? Real distributed systems do often involve complex trust relationships. For example, a customer needs to trust an online shop to actually deliver the goods they ordered, although they can dispute the payment via their bank if the goods never arrive or if they get charged too much. But if an online shop somehow allowed customers to order goods without paying for them, this weakness would no doubt be exploited by fraudsters, so the shop must assume that customers are potentially malicious. On the other hand, for RPC between services belonging to the shop, running in the same datacenter, one service can probably trust the other services run by the same company. The payments service doesn't fully trust the shop, since someone might set up a fraudulent shop or use stolen credit card numbers, but the shop probably does trust the payments service. And so on. And in the end, we want the customer, the online shop, and the payments service to agree on any order that is placed. The Byzantine generals problem is a simplification of such complex trust relationships, but it is a good starting point for studying systems in which some participants might behave maliciously.
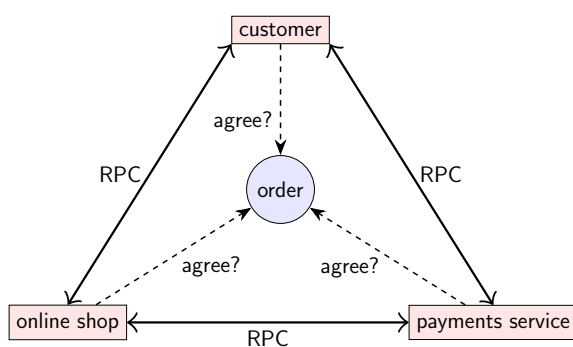
Slide 29

Before we move on, a brief digression about the origin of the word "Byzantine". The term comes from the Byzantine empire, named after its capital city Byzantium or Constantinople, which is now Istanbul in Turkey. There is no historical evidence that the generals of the Byzantine empire were any more prone to intrigue and conspiracy than those elsewhere. Rather, the word *Byzantine* had been used in the sense of "excessively complicated, bureaucratic, devious" long before Leslie Lamport adopted the word to describe the Byzantine generals problem; the exact etymology is unclear.

15

The Byzantine empire (650 CE)

Byzantium/Constantinople/Istanbul

Source: https://commons.wikimedia.org/wiki/File:Byzantiumby650AD.svg

"Byzantine" has long been used for "excessively complicated, bureaucratic, devious" (e.g. *"the Byzantine tax law"*)

Slide 30

2.3 Describing nodes and network behaviour

When designing a distributed algorithm, a *system model* is how we specify our assumptions about what faults may occur.

System models

We have seen two thought experiments:
- ▶ Two generals problem: a model of networks
- ▶ Byzantine generals problem: a model of node behaviour

In real systems, both nodes and networks may be faulty!

Capture assumptions in a **system model** consisting of:
- ▶ Network behaviour (e.g. message loss)
- ▶ Node behaviour (e.g. crashes)
- ▶ Timing behaviour (e.g. latency)

Choice of models for each of these parts.

Slide 31

Networks are unreliable
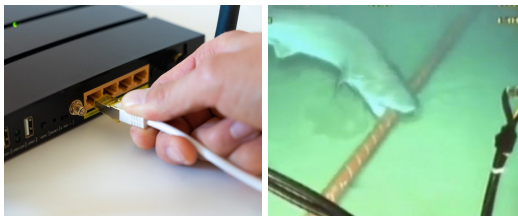
In the sea, sharks bite fibre optic cables
https://slate.com/technology/2014/08/
shark-attacks-threaten-google-s-undersea-internet-cables-video.html

On land, cows step on the cables
https://twitter.com/uhoelzle/status/1263333283107991558

Slide 32

Let's start with the network. No network is perfectly reliable: even in carefully engineered systems with redundant network links, things can go wrong [Bailis and Kingsbury, 2014]. Someone might accidentally unplug the wrong network cable. Sharks and cows have both been shown to cause damage and interruption to long-distance networks (see links on Slide 32). Or a network may be temporarily overloaded, perhaps by accident or perhaps due to a denial-of-service attack. Any of these can cause messages to be lost.

In a system model, we take a more abstract view, which saves us from the details of worrying about sharks and cows. Most distributed algorithms assume that the network provides bidirectional message-passing between a pair of nodes, also known as *point-to-point* or *unicast* communication. Real networks do sometimes allow *broadcast* or *multicast* communication (sending a packet to many recipients at the same time, which is used e.g. for discovering a printer on a local network), but broadly speaking, assuming unicast-only is a good model of the Internet today. Later, in Lecture 4, we will show how to implement broadcast on top of unicast communication.

We can then choose how reliable we want to assume these links to be. Most algorithms assume one of the three choices listed on Slide 33.

System model: network behaviour

Assume bidirectional **point-to-point** communication between two nodes, with one of:
- ▶ **Reliable** (perfect) links:
  A message is received if and only if it is sent.
  Messages may be reordered.
- ▶ **Fair-loss** links:
  Messages may be lost, duplicated, or reordered.
  If you keep retrying, a message eventually gets through.
- ▶ **Arbitrary** links (active adversary):
  A malicious adversary may interfere with messages (eavesdrop, modify, drop, spoof, replay).

Network partition: some links dropping/delaying all messages for extended period of time

retry + dedup

TLS

Slide 33

Interestingly, it is possible to convert some types of link into others. For example, if we have a fair-loss link, we can turn it into a reliable link by continually retransmitting lost messages until they are finally received, and by filtering out duplicated messages on the recipient side. The fair-loss assumption means that any *network partition* (network interruption) will last only for a finite period of time, but not forever, so we can guarantee that every message will eventually be received.

Of course, any messages sent during a network partition will only be received after the interruption is repaired, which may take a long time, but the definitions on Slide 33 do not say anything about network delay or latency. We will get to that topic on Slide 35.

The TCP protocol, which we discussed briefly in Section 1.2, performs this kind of retry and deduplication at the network packet level. However, TCP is usually configured with a timeout, so it will give up and stop retrying after a certain time, typically on the order of one minute. To overcome network partitions that last for longer than this duration, a separate retry and deduplication mechanism needs to be implemented in addition to that provided by TCP.

An arbitrary link is an accurate model for communication over the Internet: whenever your communication is routed through a network (be it a coffee shop wifi or an Internet backbone network), the operator of that network can potentially interfere with and manipulate your network packets in arbitrary ways. Someone who manipulates network traffic is also known as an *active adversary*. Fortunately, it is *almost* possible to turn an arbitrary link into a fair-loss link using cryptographic techniques. The *Transport Layer Security* (TLS) protocol, which provides the "s" for "secure" in `https://`, prevents an active adversary from eavesdropping, modifying, spoofing, or replaying traffic (more on this in the Security course, Part IB Easter term).

The only thing that TLS cannot prevent is the adversary dropping (blocking) communication. Thus, an arbitrary link can be converted into a fair-loss link only if we assume that the adversary does not block communication forever. In some networks, it might be possible to route around the interrupted network link, but this is not always the case.

Thus, the assumption of a reliable network link is perhaps not as unrealistic as it may seem at first glance: generally it is possible for all sent messages to be received, as long as we are willing to wait for a potentially arbitrary period of time for retries during a network partition. However, we also have to consider the possibility that the sender of a message may crash while attempting to retransmit a message, which may cause that message to be permanently lost. This brings us to the topic of node crashes.

System model: node behaviour

Each node executes a specified algorithm, assuming one of the following:

- ▶ **Crash-stop** (fail-stop):
  A node is faulty if it crashes (at any moment).
  After crashing, it stops executing forever.
- ▶ **Crash-recovery** (fail-recovery):
  A node may crash at any moment, losing its in-memory state. It may resume executing sometime later.
- ▶ **Byzantine** (fail-arbitrary):
  A node is faulty if it deviates from the algorithm.
  Faulty nodes may do anything, including crashing or malicious behaviour.

A node that is not faulty is called **"correct"**

Slide 34

In the *crash-stop* model, we assume that after a node crashes, it never recovers. This is a reasonable model for an unrecoverable hardware fault, or for the situation where a person drops their phone in the toilet, after which it is permanently out of order. With a software crash, the crash-stop model might seem unrealistic, because we can just restart the node, after which it will recover. Nevertheless, some algorithms assume a crash-stop model, since that makes the algorithm simpler. In this case, a node that crashes and recovers would have to re-join the system as a new node. Alternatively, the *crash-recovery* model explicitly allows nodes to restart and resume processing after a crash.

Finally, the *Byzantine* model is the most general model of node behaviour: as in the Byzantine generals problem, a faulty node may not only crash, but also deviate from the specified algorithm in arbitrary ways, including exhibiting malicious behaviour. A bug in the implementation of a node could also be classed as a Byzantine fault. However, if all of the nodes are running the same software, they will all have the same bug, and so any algorithm that is predicated on less than one third of nodes being Byzantine-faulty will not be able to tolerate such a bug. In principle, we could try using several different implementations of the same algorithm, but this is rarely a practical option.

In the case of the network, it was possible to convert one model to another using generic protocols. This is not the case with the different models of node behaviour. For instance, an algorithm designed for a crash-recovery system model may look very different from a Byzantine algorithm.

System model: synchrony (timing) assumptions

Assume one of the following for network and nodes:

- ▶ **Synchronous**:
  Message latency no greater than a known upper bound.
  Nodes execute algorithm at a known speed.
- ▶ **Partially synchronous**:
  The system is asynchronous for some finite (but unknown) periods of time, synchronous otherwise.
- ▶ **Asynchronous**:
  Messages can be delayed arbitrarily.
  Nodes can pause execution arbitrarily.
  No timing guarantees at all.

Note: other parts of computer science use the terms "synchronous" and "asynchronous" differently.

Slide 35

The third part of a system model is the synchrony assumption, which is about timing. The three choices we can make here are *synchronous*, *asynchronous*, or *partially synchronous* [Dwork et al., 1988]. (Note that the definitions of these terms differ somewhat across different parts of computer science; our definitions here are standard in the field of distributed computing.)

A synchronous system is what we would love to have: a message sent over the network never takes longer than some known maximum latency, and nodes always execute their algorithm at a predictable speed. Many problems in distributed computing are much easier if you assume a synchronous system. And it is tempting to assume synchrony, because networks and nodes are well-behaved *most of the time*, and so this assumption is often true.

Unfortunately, *most of the time* is not the same as *always*, and algorithms designed for a synchronous model often fail catastrophically if the assumptions of bounded latency and bounded execution speed are violated, even just for a short while, and even if this happens rarely. And in practical systems, there are many reasons why network latency or execution speed may sometimes vary wildly, see Slide 36.

The other extreme is an asynchronous model, in which we make no timing assumptions at all: we allow messages to be delayed arbitrarily in the network, and we allow arbitrary differences in nodes' processing speeds (for example, we allow one node to pause execution while other nodes continue running normally). Algorithms that are designed for an asynchronous model are typically very robust, because they are unaffected by any temporary network interruptions or spikes in latency.

Unfortunately, some problems in distributed computing are impossible to solve in an asynchronous model, and therefore we have the *partially synchronous* model as a compromise. In this model, we assume that our system is synchronous and well-behaved most of the time, but occasionally it may flip into asynchronous mode in which all timing guarantees are off, and this can happen unpredictably. The partially synchronous model is good for many practical systems, but using it correctly requires care.

Violations of synchrony in practice

Networks usually have quite predictable latency, which can occasionally increase:

- ▶ Message loss requiring retry
- ▶ Congestion/contention causing queueing
- ▶ Network/route reconfiguration

Nodes usually execute code at a predictable speed, with occasional pauses:

- ▶ Operating system scheduling issues, e.g. priority inversion
- ▶ Stop-the-world garbage collection pauses
- ▶ Page faults, swap, thrashing

Real-time operating systems (RTOS) provide scheduling guarantees, but most distributed systems do not use RTOS

Slide 36

There are many reasons why a system may violate synchrony assumptions. We have already talked about latency increasing without bound if messages are lost and retransmitted, especially if we have to wait for a network partition to be repaired before the messages can get through. Another reason for latency increases in a network is congestion resulting in queueing of packets in switch buffers. Network reconfiguration can also cause large delays: even within a single datacenter, there have been documented cases of packets being delayed for more than a minute [Imbriaco, 2012].

We might expect that the speed at which nodes execute their algorithms is constant: after all, an instruction generally takes a fixed number of CPU clock cycles, and the clock speed doesn't vary much. However, even on a single node, there are many reasons why a running program may unexpectedly get paused for significant amounts of time. Scheduling in the operating system can preempt a running thread and leave it paused while other programs run, especially on a machine under heavy load. A real problem in memory-managed languages such as Java is that when the garbage collector runs, it needs to pause all running threads from time to time (this is known as a *stop-the-world* garbage collection pause). On large heaps, such pauses can be as long as several minutes [Thompson, 2013]! Page faults are another reason why a thread may get suspended, especially when there is not much free memory left.

As you know from the concurrent systems half of this course, threads can and will get preempted even at the most inconvenient moments, anywhere in a program. In a distributed system, this is particularly

problematic, because for one node, time appears to "stand still" while it is paused, and during this time all other nodes continue executing their algorithms normally. Other nodes may even notice that the paused node is not responding, and assume that it has crashed. After a while, the paused node resumes processing, without even realising that it was paused for a significant period of time.

Combined with the many reasons for variable network latency, this means that in practical systems, it is very rarely safe to assume a synchronous system model. Most distributed algorithms need to be designed for the asynchronous or partially synchronous model.

System models summary

For each of the three parts, pick one:

- **Network:**
  reliable, fair-loss, or arbitrary

- **Nodes:**
  crash-stop, crash-recovery, or Byzantine

- **Timing:**
  synchronous, partially synchronous, or asynchronous

This is the basis for any distributed algorithm.
If your assumptions are wrong, all bets are off!

Slide 37

2.4 Fault tolerance and high availability

As highlighted on Slide 4, one reason for building distributed systems is to achieve higher reliability than is possible with a single computer. We will now explore this idea further in the light of the system models we have discussed.

From a business point of view, what usually matters most is the *availability* of a service, such as a website. For example, an online shop wants to be able to sell products at any time of day or night: any outage of the website means a lost opportunity to make money. For other services, there may even be contractual agreements with customers requiring the service to be available. If a service is unavailable, this can also damage the reputation of the service provider.

The availability of a service is typically measured in terms of its ability to respond correctly to requests within a certain time. The definition of whether a service is "available" or "unavailable" can be somewhat arbitrary: for example, if it takes 5 seconds to load a page, do we still consider that website to be available? What if it takes 30 seconds? An hour?

Start of video section 2.4
(mp4 download)

Availability

Online shop wants to sell stuff 24/7!
Service unavailability = downtime = losing money

Availability = uptime = fraction of time that a service is functioning correctly

- "Two nines" = 99% up = down 3.7 days/year
- "Three nines" = 99.9% up = down 8.8 hours/year
- "Four nines" = 99.99% up = down 53 minutes/year
- "Five nines" = 99.999% up = down 5.3 minutes/year

Service-Level Objective (SLO):
e.g. "99.9% of requests in a day get a response in 200 ms"

Service-Level Agreement (SLA):
contract specifying some SLO, penalties for violation

Slide 38

Typically, the availability expectations of a service are formalised as a *service-level objective* (SLO), which typically specifies the percentage of requests that need to return a correct response within a specified timeout, as measured by a certain client over a certain period of time. A *service-level agreement* (SLA) is a contract that specifies some SLO, as well as the consequences if the SLO is not met (for example, the service provider may need to offer a refund to its customers).

Faults (such as node crashes or network interruptions) are a common cause of unavailability. In order to increase availability, we can reduce the frequency of faults, or we can design systems to continue working despite some of its components being faulty; the latter approach is called *fault tolerance*. Reducing the frequency of faults is possible through buying higher-quality hardware and introducing redundancy, but this approach can never reduce the probability of faults to zero. Instead, fault tolerance is the approach taken by many distributed systems.

Achieving high availability: fault tolerance

Failure: system as a whole isn't working

Fault: some part of the system isn't working
- ▶ Node fault: crash (crash-stop/crash-recovery), deviating from algorithm (Byzantine)
- ▶ Network fault: dropping or significantly delaying messages

Fault tolerance:
system as a whole continues working, despite faults
(some maximum number of faults assumed)

Single point of failure (SPOF):
node/network link whose fault leads to failure

Slide 39

If all nodes crash and don't recover, then no algorithm will be able to get any work done, so it does not make sense to tolerate arbitrary numbers of faults. Rather, an algorithm is designed to tolerate some specified number of faults: for example, some distributed algorithms are able to make progress provided that fewer than half of the nodes have crashed.

In fault-tolerant systems we want to avoid *single points of failure* (SPOF), i.e. components that would cause an outage if they were to become faulty. For example, the Internet is designed to have no SPOF: there is no one server or router whose destruction would bring down the entire Internet (although the loss of some components, such as key intercontinental fibre links, does cause noticeable disruption).

The first step towards tolerating faults is to detect faults, which is often done with a *failure detector*. ("Fault detector" would be more logical, but "failure detector" is the conventional term.) A failure detector usually detects crash faults. Byzantine faults are not always detectable, although in some cases Byzantine behaviour does leave evidence that can be used to identify and exclude malicious nodes.

Failure detectors

Failure detector:
algorithm that detects whether another node is faulty

Perfect failure detector:
labels a node as faulty if and only if it has crashed

Typical implementation for crash-stop/crash-recovery:
send message, await response, label node as crashed if no reply within some timeout

Problem:
cannot tell the difference between crashed node, temporarily unresponsive node, lost message, and delayed message

Slide 40

In most cases, a failure detector works by periodically sending messages to other nodes, and labelling a node as crashed if no response is received within the expected time. Ideally, we would like a timeout to occur if and only if the node really has crashed (this is called a *perfect failure detector*). However, the two generals problem tells us that this is not a totally accurate way of detecting a crash, because the absence of a response could also be due to message loss or delay.

A perfect timeout-based failure detector exists only in a synchronous crash-stop system with reliable links; in a partially synchronous system, a perfect failure detector does not exist. Moreover, in an asynchronous system, no timeout-based failure exists, since timeouts are meaningless in the asynchronous model. However, there is a useful failure detector that exists in partially synchronous systems: the *eventually perfect failure detector* [Chandra and Toueg, 1996].

<div style="border:1px solid black; padding:1em;">

Failure detection in partially synchronous systems

Perfect timeout-based failure detector exists only in a synchronous crash-stop system with reliable links.

Eventually perfect failure detector:
- ► May *temporarily* label a node as crashed, even though it is correct
- ► May *temporarily* label a node as correct, even though it has crashed
- ► But *eventually*, labels a node as crashed if and only if it has crashed

Reflects fact that detection is not instantaneous, and we may have spurious timeouts

</div>

Slide 41

We will see later how to use such a failure detector to design fault-tolerance mechanisms and to automatically recover from node crashes. Using such algorithms it is possible to build systems that are highly available. Tolerating crashes also makes day-to-day operations easier: for example, if a service can tolerate one out of three nodes being unavailable, then a software upgrade can be rolled out by installing it and restarting one node at a time, while the remaining two nodes continue running the service. Being able to roll out software upgrades in this way, without clients noticing any interruption, is important for many organisations that are continually working on their software.

For safety-critical applications, such as air-traffic control systems, it is undoubtedly important to invest in good fault-tolerance mechanisms. However, it is not the case that higher availability is always better. Reaching extremely high availability requires a highly focussed engineering effort, and often conservative design choices. For example, the old-fashioned fixed-line telephone network is designed for "five nines" availability, but the downside of this focus on availability is that it has been very slow to evolve. Most Internet services do not even reach four nines because of diminishing returns: beyond some point, the additional cost of achieving higher availability exceeds the cost of occasional downtime, so it is economically rational to accept a certain amount of downtime.

Exercise 2. *Reliable network links allow messages to be reordered. Give pseudocode for an algorithm that strengthens the properties of a reliable point-to-point link such that messages are received in the order they were sent (this is called a FIFO link), assuming an asynchronous crash-stop system model.*

Exercise 3. *How do we need to change the algorithm from Exercise 2 if we assume a crash-recovery model instead of a crash-stop model?*

3 Time, clocks, and ordering of events

Let's start with a riddle, which will be resolved later in this lecture.

A detective story

In the night from 30 June to 1 July 2012 (UK time), many online services and systems around the world crashed simultaneously.

Servers locked up and stopped responding.

Some airlines could not process any reservations or check-ins for several hours.

What happened?

Slide 42

In this lecture we will look at the concept of time in distributed systems. We have already seen that our assumptions about timing form a key part of the system model that distributed algorithms rely on. For example, timeout-based failure detectors need to measure time to determine when a timeout has elapsed. Operating systems rely extensively on timers and time measurements in order to schedule tasks, keep track of CPU usage, and many other purposes. Applications often want to record the time and date at which events occurred: for example, when debugging an error in a distributed system, timestamps are helpful for debugging, since they allow us to reconstruct which things happened around the same time on different nodes. All of these require more or less accurate measurements of time.

Clocks and time in distributed systems

Distributed systems often need to measure time, e.g.:
- ▶ Schedulers, timeouts, failure detectors, retry timers
- ▶ Performance measurements, statistics, profiling
- ▶ Log files & databases: record when an event occurred
- ▶ Data with time-limited validity (e.g. cache entries)
- ▶ Determining order of events across several nodes

We distinguish two types of clock:
- ▶ **physical clocks**: count number of seconds elapsed
- ▶ **logical clocks**: count events, e.g. messages sent

NB. Clock in digital electronics (oscillator)
$\neq$ clock in distributed systems (source of **timestamps**)

Slide 43

3.1 Physical clocks

We will start by discussing *physical clocks*, which are the types of clocks you are familiar with from everyday usage. Physical clocks include analogue/mechanical clocks based on pendulums or similar mechanisms, and digital clocks based e.g. on a vibrating quartz crystal. Quartz clocks are found in most wristwatches, in every computer and mobile phone, in microwave ovens that display the time, and many other everyday objects.

Quartz clocks

- ► Quartz crystal laser-trimmed to mechanically resonate at a specific frequency
- ► Piezoelectric effect: mechanical force ⇔ electric field
- ► Oscillator circuit produces signal at resonant frequency
- ► Count number of cycles to measure elapsed time

Slide 44

Quartz clocks are cheap, but they are not totally accurate. Due to manufacturing imperfections, some clocks run slightly faster than others. Moreover, the oscillation frequency varies with the temperature. Typical quartz clocks are tuned to be quite stable around room temperature, but significantly higher or lower temperatures slow down the clock. The rate by which a clock runs fast or slow is called *drift*.

Quartz clock error: drift

- ► One clock runs slightly fast, another slightly slow
- ► Drift measured in **parts per million** (ppm)
- ► 1 ppm = 1 microsecond/second = 86 ms/day = 32 s/year
- ► Most computer clocks correct within ≈ 50 ppm

Temperature significantly affects drift
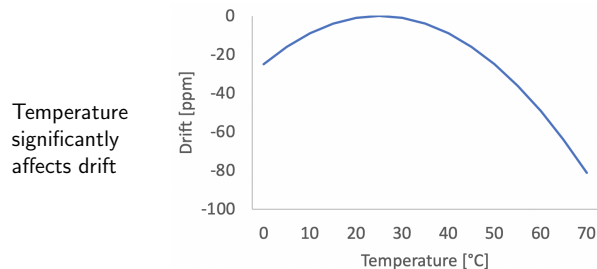
Slide 45

Atomic clocks

- ► Caesium-133 has a resonance ("hyperfine transition") at ≈ 9 GHz
- ► Tune an electronic oscillator to that resonant frequency
- ► 1 second = 9,192,631,770 periods of that signal
- ► Accuracy ≈ 1 in 10^{-14} (1 second in 3 million years)
- ► Price $\approx$ £20,000 (?) (can get cheaper rubidium clocks for $\approx$ £1,000)

https: //www.microsemi.com/product-directory/ cesium-frequency-references/ 4115-5071a-cesium-primary-frequency-standard
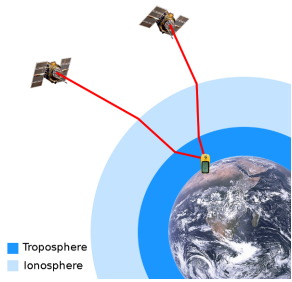
Slide 46

When greater accuracy is required, atomic clocks are used. These clocks are based on quantum-

mechanical properties of certain atoms, such as caesium or rubidium. In fact, the time unit of *one second* in the International System of Units (SI) is defined to be exactly 9,192,631,770 periods of a particular resonant frequency of the caesium-133 atom.

Another high-accuracy method of obtaining the time is to rely on the GPS satellite positioning system, or similar systems such as Galileo or GLONASS. These systems work by having several satellites orbiting the Earth and broadcasting the current time at very high resolution. Receivers measure the time it took the signal from each satellite to reach them, and use this to compute their distance from each satellite, and hence their location. By connecting a GPS receiver to a computer, it is possible to obtain a clock that is accurate to within a fraction of a microsecond, provided that the receiver is able to get a clear signal from the satellites. In a datacenter, there is generally too much electromagnetic interference to get a good signal, so a GPS receiver requires an antenna on the roof of the datacenter building.

We now have a problem: we have two different definitions of time – one based on quantum mechanics, the other based on astronomy – and those two definitions don't match up precisely. One rotation of planet Earth around its own axis does not take exactly $24 \times 60 \times 60 \times 9{,}192{,}631{,}770$ periods of caesium-133's resonant frequency. In fact, the speed of rotation of the planet is not even constant: it fluctuates due to the effects of tides, earthquakes, glacier melting, and some unexplained factors.

The solution is *Coordinated Universal Time* (UTC), which is based on atomic time, but includes corrections to account for variations in the Earth's rotation. In everyday life we use our local *time zone*, which is specified as an offset to UTC.

The UK's local time zone is called *Greenwich Mean Time* (GMT) in winter, and *British Summer Time* (BST) in summer, where GMT is defined to be equal to UTC, and BST is defined to be UTC + 1 hour. Confusingly, the term *Greenwich Mean Time* was originally used to refer to mean solar time on the

Greenwich meridian, i.e. it used to be defined in terms of astronomy, while now it is defined in terms of atomic clocks. Today, the term *UT1* is used to refer to mean solar time at 0° longitude.

The difference between UTC and TAI is that UTC includes *leap seconds*, which are added as needed to keep UTC roughly in sync with the rotation of the Earth.

Slide 49

Due to leap seconds, it is not true that an hour always as 3600 seconds, and a day always has 86,400 seconds. In the UTC timescale, a day can be 86,399 seconds, 86,400 seconds, or 86,401 seconds long due to a leap second. This complicates software that needs to work with dates and times.

In computing, a *timestamp* is a representation of a particular point in time. Two representations of timestamps are commonly used: Unix time and ISO 8601. For Unix time, zero corresponds to the arbitrarily chosen date of 1 January 1970, known as the *epoch*. There are minor variations: for example, Java's `System.currentTimeMillis()` is like Unix time, but uses milliseconds rather than seconds.

Slide 50

To be correct, software that works with timestamps needs to know about leap seconds. For example, if you want to calculate how many seconds elapsed between two timestamps, you need to know how many leap seconds were inserted between those two dates. For dates that are more than about six months into the future, this is impossible to know, because the Earth's rotation has not happened yet!

The most common approach in software is to simply ignore leap seconds, pretend that they don't exist, and hope that the problem somehow goes away. This approach is taken by Unix timestamps, and by the POSIX standard. For software that only needs coarse-grained timings (e.g. rounded to the nearest day), this is fine, since the difference of a few seconds is not significant.

However, operating systems and distributed systems often *do* rely on high-resolution timestamps for accurate measurements of time, where a difference of one second is very noticeable. In such settings, ignoring leap seconds can be dangerous. For example, say you have a Java program that twice calls

`System.currentTimeMillis()`, 500 ms apart, within a positive leap second (i.e. while the clock is saying 23:59:60). What is the difference between those two timestamps going to be? It can't be 500, since the `currentTimeMillis()` clock does not account for leap seconds. Does the clock stop, so the difference between the two timestamps is zero? Or could the difference even be negative, so the clock runs backwards for a brief moment? The documentation is silent about this question. (The best solution is probably to use a *monotonic clock* instead, which we introduce on Slide 58.)

Poor handling of the leap second on 30 June 2012 is what caused the simultaneous failures of many services on that day (Slide 42). Due to a bug in the Linux kernel, the leap second had a high probability of triggering a livelock condition when running a multithreaded process [Allen, 2013, Minar, 2012]. Even a reboot did not fix the problem instead, but setting the system clock reset the bad state in the kernel.

How most software deals with leap seconds

By ignoring them!

However, OS and DistSys often need timings with sub-second accuracy.

30 June 2012: bug in Linux kernel caused livelock on leap second, causing many Internet services to go down

Pragmatic solution: "**smear**" (spread out) the leap second over the course of a day
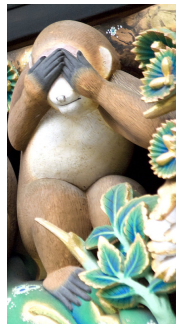
https://www.flickr.com/photos/ru_boff/37915499055/

Slide 51

Today, some software handles leap seconds explicitly, while other programs continue to ignore them. A pragmatic solution that is widely used today is that when a positive leap second occurs, rather than inserting it between 23:59:59 and 00:00:00, the extra second is spread out over several hours before and after that time by deliberately slowing down the clocks during that time (or speeding up in the case of a negative leap second). This approach is called *smearing* the leap second, and it is not without problems. However, it is a pragmatic alternative to making all software aware of and robust to leap seconds, which may well be infeasible.

Exercise 4. *Describe some problems that may arise from leap second smearing.*

3.2 Clock synchronisation and monotonic clocks

Clock synchronisation

Computers track physical time/UTC with a quartz clock (with battery, continues running when power is off)

Due to **clock drift**, clock error gradually increases

Clock skew: difference between two clocks at a point in time

Solution: Periodically get the current time from a server that has a more accurate time source (atomic clock or GPS receiver)

Protocols: Network Time Protocol (**NTP**), Precision Time Protocol (**PTP**)

Start of video section 3.2
(mp4 download)

Slide 52

Atomic clocks are too expensive and bulky to build into every computer and phone, so quartz clocks are used. As discussed on Slide 45, these clocks drift, and need adjustment from time to time. The most common approach is to use the *Network Time Protocol* (NTP). All mainstream operating systems have NTP clients built in; for example, Slide 53 shows the NTP settings dialog in macOS.
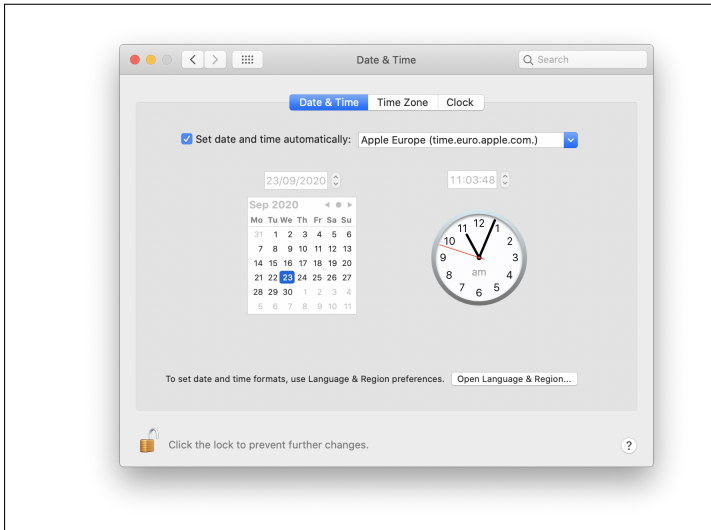

Slide 53

Slide 54

Time synchronisation over a network is made difficult by unpredictable latency. As discussed on Slide 36, both network latency and nodes' processing speed can vary considerably. To reduce the effects of random variations, NTP takes several samples of time measurements and applies statistical filters to eliminate outliers.

Slide 55 shows how NTP estimates the clock skew between the client and the server. When the client sends a request message, it includes the current timestamp t_1 according to the client's clock. When the server receives the request, and before processing it, the server records the current timestamp t_2 according to the server's clock. When the server sends its response, it echoes the value t_1 from the request, and also includes the server's receipt timestamp t_2 and the server's response timestamp t_3 in the reply. Finally, when the client receives the response, it records the current timestamp t_4 according to the client's clock.

We can determine the time that the messages spent travelling across the network by calculating the round-trip time from the client's point of view $(t_4 - t_1)$ and subtracting the processing time on the server $(t_3 - t_2)$. We then estimate the one-way network latency as being half of the total network delay. Thus, by the time the response reaches the client, we can estimate that the server's clock will have moved on to t_3 plus the one-way network latency. We then subtract the client's current time t_4 from the estimated server time to obtain the estimated skew between the two clocks.

This estimation depends on the assumption that the network latency is approximately the same in both directions. This assumption is probably true if latency is dominated by geographic distance between

client and server. However, if queueing time in the network is a significant factor in the latency (e.g. if one node's network link is heavily loaded while the other node's link has plenty of spare capacity), then there could be a large difference between request and response latency. Unfortunately, most networks do not give nodes any indication of the actual latency that a particular packet has experienced.

Estimating time over a network

NTP client — NTP server

request: t_1

response: (t_1, t_2, t_3)

Round-trip network delay: $\delta = (t_4 - t_1) - (t_3 - t_2)$

Estimated server time when client receives response: $t_3 + \dfrac{\delta}{2}$

Estimated clock skew: $\theta = t_3 + \dfrac{\delta}{2} - t_4 = \dfrac{t_2 - t_1 + t_3 - t_4}{2}$
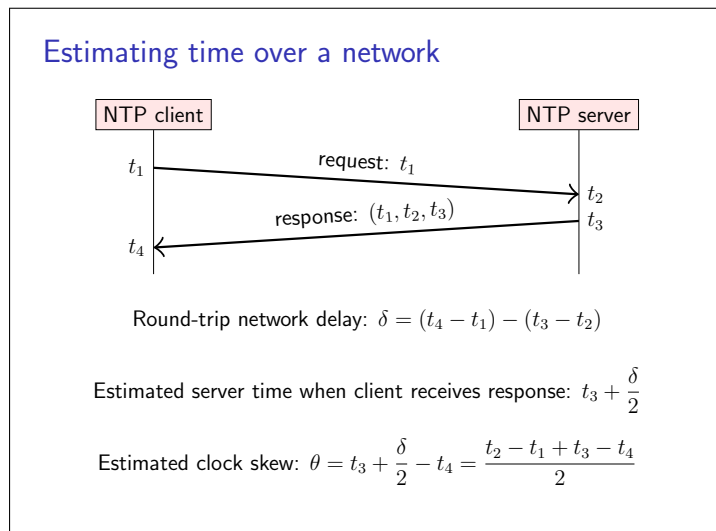
Slide 55

Exercise 5. *What is the maximum possible error in the NTP client's estimate of skew with regard to one particular server, assuming that both nodes correctly follow the protocol?*
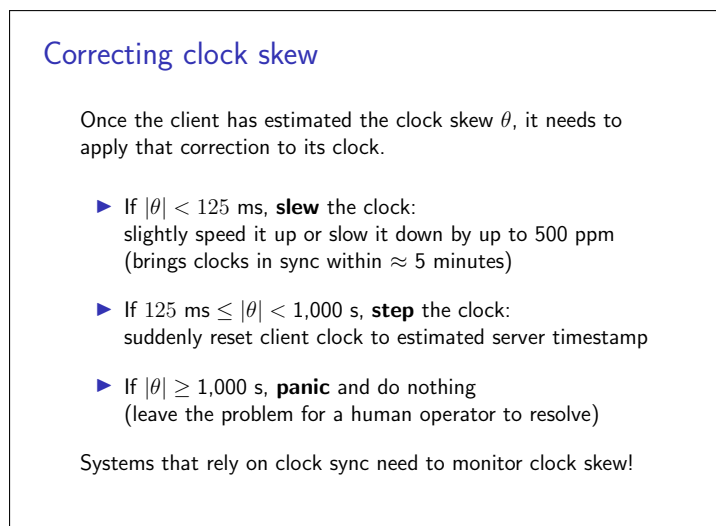
Once NTP has estimated the clock skew between client and server, the next step is to adjust the client's clock to bring it in line with the server. The method used for this depends on the amount of skew. The client corrects small differences gently by adjusting the clock speed to run slightly faster or slower as needed, which gradually reduces the skew over the course of a few minutes. This process is called *slewing* the clock.

Slide 57 shows an example of slewing, in which the client's clock frequency converges to the same rate as the server, keeping the two in sync to within a few milliseconds. Of course, the exact accuracy achieved in a particular system depends on the timing properties of the network between client and server.

However, if the skew is larger, slewing would take too long, so the NTP client instead forcibly sets its clock to the estimated correct time based on the server timestamp. This is called *stepping* the clock. Any applications that are watching the clock on the client will see time suddenly jump forwards or backwards.

And finally, if the skew is very large (by default, more than about 15 minutes), the NTP client may decide that something must be wrong, and refuse to adjust the clock, leaving the problem for a user or operator to correct. For this reason, any system that depends on clock synchronisation needs to be carefully monitored for clock skew: just because a node is running NTP, that does not guarantee that its clock will be correct, since it could get stuck in a panic state in which it refuses to adjust the clock.

Correcting clock skew

Once the client has estimated the clock skew θ, it needs to apply that correction to its clock.

► If $|\theta| < 125$ ms, **slew** the clock:
slightly speed it up or slow it down by up to 500 ppm
(brings clocks in sync within ≈ 5 minutes)

► If 125 ms $\leq |\theta| < 1{,}000$ s, **step** the clock:
suddenly reset client clock to estimated server timestamp

► If $|\theta| \geq 1{,}000$ s, **panic** and do nothing
(leave the problem for a human operator to resolve)

Systems that rely on clock sync need to monitor clock skew!

Slide 56

Initial run of NTP 3.5f on HP L2000-44/2

http://www.ntp.org/ntpfaq/NTP-s-algo.htm

Slide 57

The fact that clocks may be stepped by NTP, i.e. suddenly moved forwards or backwards, has an important implication for any software that needs to measure elapsed time. Slide 58 shows an example in Java, in which we want to measure the running time of a function `doSomething()`. Java has two core functions for getting the current timestamp from the operating system's local clock: `currentTimeMillis()` and `nanoTime()`. Besides the different resolution (milliseconds versus nanoseconds), the key difference between the two is how they behave in the face of clock adjustments from NTP or other sources.

`currentTimeMillis()` is a *time-of-day* clock (also known as *real-time* clock) that returns the time elapsed since a fixed reference point (in this case, the Unix epoch of 1 January 1970). When the NTP client steps the local clock, a time-of-day clock may jump. Thus, if you use such a clock to measure elapsed time, the resulting difference between end timestamp and start timestamp may be much greater than the actual elapsed time (if the clock was stepped forwards), or it may even be negative (if the clock was stepped backwards). This type of clock is therefore not suitable for measuring elapsed time.

On the other hand, `nanoTime()` is a *monotonic* clock, which is not affected by NTP stepping: it still counts seconds elapsed, but it always moves forward. Only the rate at which it moves forward may be adjusted by NTP slewing. This makes a monotonic clock much more robust for measuring elapsed time. The downside is that a timestamp from a monotonic clock is meaningless by itself: it measures the time since some arbitrary reference point, such as the time since this computer was started up. When using a monotonic clock, only the difference between two timestamps from the same node is meaningful. It does not make sense to compare monotonic clock timestamps across different nodes.

Monotonic and time-of-day clocks

```java
// BAD:
long startTime = System.currentTimeMillis();
doSomething();
long endTime = System.currentTimeMillis();
long elapsedMillis = endTime - startTime;
// elapsedMillis may be negative!

              NTP client steps the clock during this

// GOOD:
long startTime = System.nanoTime();
doSomething();
long endTime = System.nanoTime();
long elapsedNanos = endTime - startTime;
// elapsedNanos is always >= 0
```

Slide 58

Most operating systems and programming languages provide both a time-of-day clock and a monotonic clock, since both are useful for different purposes.
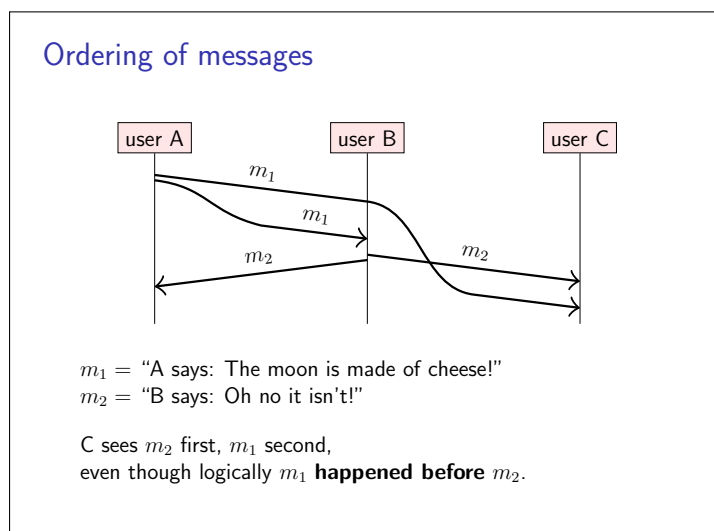
3.3 Causality and happens-before

We will now move on to the problem of ordering events in a distributed system, which is closely related to the concept of time. Consider the scenario on Slide 60, in which user A makes a statement m_1 and sends it as a message to the other two users, B and C. On receiving m_1, user B reacts by sending a reply m_2 to the other two users, A and C. However, even if we assume the network links are reliable, they allow reordering (Slide 33), so C might receive m_2 before m_1 if m_1 is slightly delayed in the network.

From C's point of view, the result is confusing: C first sees the reply, and then the message it is replying to. It almost looks as though B was able to see into the future and anticipate A's statement before A even said it. In real life this sort of reordering of spoken words does not happen, and so we intuitively don't expect it to happen in computer systems either.

As a more technical example, consider m_1 to be an instruction that creates an object in a database, and m_2 to be an instruction that updates this same object. If a node processes m_2 before m_1 it would first attempt to update a nonexistent object, and then create an object which would not subsequently be updated. The database instructions only make sense if m_1 is processed before m_2.

Ordering of messages

$m_1 =$ "A says: The moon is made of cheese!"
$m_2 =$ "B says: Oh no it isn't!"

C sees m_2 first, m_1 second,
even though logically m_1 **happened before** m_2.

Start of video section 3.3
(mp4 download)

Slide 60

How can C determine the correct order in which it should put the messages? A monotonic clock won't work since its timestamps are not comparable across nodes. A first attempt might be to get a timestamp from a time-of-day clock whenever a user wants to send a message, and to attach that timestamp to the message. In this scenario, we might reasonably expect m_2 to have a later timestamp than m_1, since m_2 is a response to m_1 and so m_2 must have happened after m_1.

Unfortunately, in a partially synchronous system model, this does not work reliably. The clock synchronisation performed by NTP and similar protocols always leaves some residual uncertainty about

the exact skew between two clocks, especially if the network latency in the two directions is asymmetric. We therefore cannot rule out the following scenario: A sends m_1 with timestamp t_1 according to A's clock. When B receives m_1, the timestamp according to B's clock is t_2, where $t_2 < t_1$, because A's clock is slightly ahead of B's clock. Thus, if we order messages based on their timestamps from time-of-day clocks, we might again end up with the wrong order.

Physical timestamps inconsistent with causality

$m_1 = (t_1, \text{"A says: The moon is made of cheese!"})$
$m_2 = (t_2, \text{"B says: Oh no it isn't!"})$

Problem: even with synced clocks, $t_2 < t_1$ is possible. Timestamp order is inconsistent with expected order!
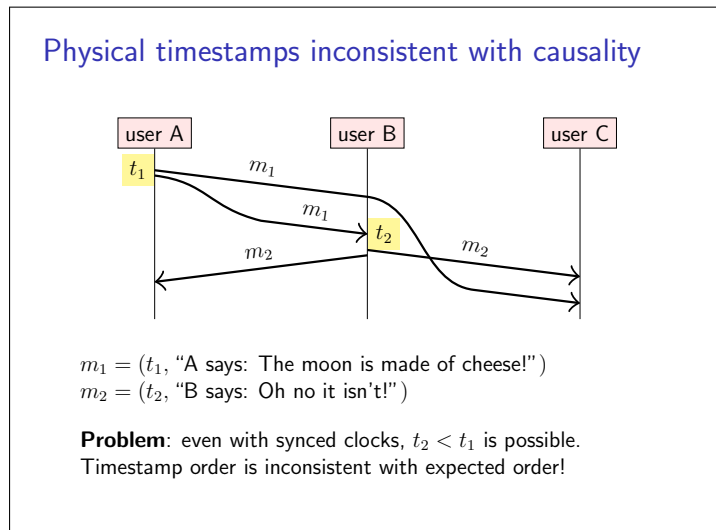
Slide 61

To formalise what we mean with the "correct" order in this type of scenario, we use the *happens-before relation* as defined on Slide 62. This definition assumes that each node has only a single thread of execution, so for any two execution steps of a node, it is clear which one happened first. More formally, we assume that there is a *strict total order* on the events that occur at the same node. A multithreaded process can be modelled by using a separate node to represent each thread.

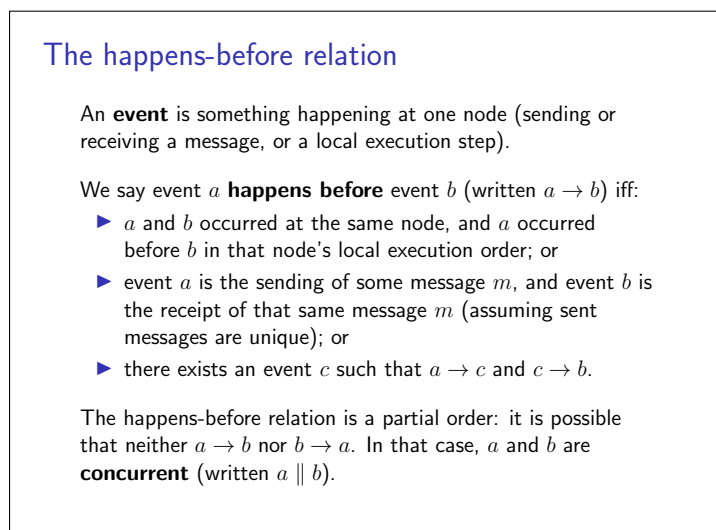We then extend this order across nodes by defining that a message is sent before that same message is received (in other words, we rule out time travel: it is not possible to receive a message that has not yet been sent). For convenience, we assume that every sent message is unique, so when a message is received, we always know unambiguously where and when that message was sent. In practice, duplicate messages may exist, but we can make them unique, for example by including the ID of the sender node and a sequence number in each message.

Finally, we take the transitive closure, and the result is the happens-before relation. This is a *partial order*, which means that it is possible that for some events a and b, neither a happened before b, nor b happened before a. In that case, we call a and b *concurrent*. Note that here, "concurrent" does not mean literally "at the same time", but rather that a and b are independent in the sense that there is no sequence of messages leading from one to the other.
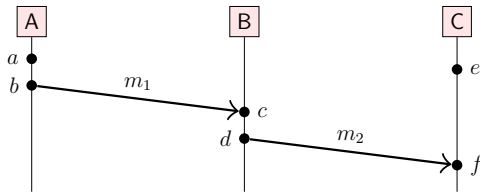
The happens-before relation

An **event** is something happening at one node (sending or receiving a message, or a local execution step).

We say event a **happens before** event b (written $a \to b$) iff:
- ▶ a and b occurred at the same node, and a occurred before b in that node's local execution order; or
- ▶ event a is the sending of some message m, and event b is the receipt of that same message m (assuming sent messages are unique); or
- ▶ there exists an event c such that $a \to c$ and $c \to b$.

The happens-before relation is a partial order: it is possible that neither $a \to b$ nor $b \to a$. In that case, a and b are **concurrent** (written $a \parallel b$).

Slide 62

32

Exercise 6. *A relation R is a* strict partial order *if it is irreflexive* ($\nexists a.\ (a, a) \in R$) *and transitive* ($\forall a, b, c.\ (a, b) \in R \land (b, c) \in R \implies (a, c) \in R$). *(These two conditions also imply that it R asymmetric, i.e. that* $\forall a, b.\ (a, b) \in R \implies (b, a) \notin R$.*) Prove that the happens-before relation is a strict partial order. You may assume that any two nodes are a nonzero distance apart, as well as the physical principle that information cannot travel faster than the speed of light.*

Exercise 7. *Show that for any two events a and b, exactly one of the three following statements must be true: either* $a \to b$, *or* $b \to a$, *or* $a \parallel b$.

The happens-before relation is a way of reasoning about *causality* in distributed systems. Causality considers whether information could have flowed from one event to another, and thus whether one event may have influenced another. In the example of Slide 60, m_2 ("Oh no it isn't!") is a reply to m_1 ("The moon is made of cheese!"), and so m_1 influenced m_2. Whether one event truly "caused" another is a philosophical question that we don't need to answer now; what matters for our purposes is that the sender of m_2 had already received m_1 at the time of sending m_2.
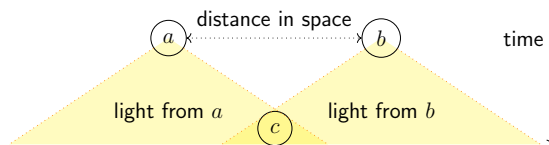
The notion of causality is borrowed from physics, where it is generally believed that it is not possible for information to travel faster than the speed of light. Thus, if you have two events a and b that occur sufficiently far apart in space, but close together in time, then it is impossible for a signal sent from a to arrive at b's location before event b, and vice versa. Therefore, a and b must be *causally unrelated*.

An event c that is sufficiently close in space to a, and sufficiently long after a in time, will be within a's *light cone*: that is, it is possible for a signal from a to reach c, and therefore a might influence c. In distributed systems, we usually work with messages on a network rather than beams of light, but the principle is very similar.

4 Broadcast protocols and logical time

In this lecture we will examine *broadcast protocols* (also known as *multicast protocols*), that is, algorithms for delivering one message to multiple recipients. These are a useful building blocks for higher-level distributed algorithms, as we will see in Lecture 5.

Several different broadcast protocols are used in practice, and their main difference is the *order* in which they deliver messages. As we saw in the last lecture, the concept of ordering is closely related to clocks and time. Hence we will start this lecture by examining more closely how clocks can help us keep track of ordering within a distributed system.

4.1 Logical time

Physical clocks, which we discussed in Section 3.1, measure the number of seconds elapsed. However, recall that on Slide 61, we saw that timestamps from physical clocks can be inconsistent with causality, even if those clocks are synchronised using something like NTP. That is, if $\mathsf{send}(m)$ is the event of sending message m, and if the happens-before relation indicates that $\mathsf{send}(m_1) \to \mathsf{send}(m_2)$, then it could nevertheless be the case that the the physical timestamp of $\mathsf{send}(m_1)$ (according to the clock of m_1's sender) is less than the physical timestamp of $\mathsf{send}(m_2)$ (according to the clock of m_2's sender).

In contrast, *logical clocks* focus on correctly capturing the order of events in a distributed system.

Logical vs. physical clocks

    ▶ Physical clock: count number of **seconds elapsed**
    ▶ Logical clock: count number of **events occurred**

Physical timestamps: useful for many things, but may be **inconsistent with causality**.

Logical clocks: designed to **capture causal dependencies**.

$$(e_1 \to e_2) \implies (T(e_1) < T(e_2))$$

We will look at two types of logical clocks:
    ▶ Lamport clocks
    ▶ Vector clocks

Start of video section 4.1
(mp4 download)

Slide 65

The first type of logical clock we will examine is the Lamport clock, introduced by Lamport [1978] in one of the seminal papers of distributed computing.

Lamport clocks algorithm

on initialisation **do**
    $t := 0$ ▷ each node has its own local variable t
end on

on any event occurring at the local node **do**
    $t := t + 1$
end on

on request to send message m **do**
    $t := t + 1$; send (t, m) via the underlying network link
end on

on receiving (t', m) via the underlying network link **do**
    $t := \max(t, t') + 1$
    deliver m to the application
**end on**

Slide 66

Slide 67

A Lamport timestamp is essentially an integer that counts the number of events that have occurred. As such, it has no direct relationship to physical time. On each node, time increases because the integer is incremented on every event. The algorithm assumes a crash-stop model (or a crash-recovery model if the timestamp is maintained in stable storage, i.e. on disk).

When a message is sent over the network, the sender attaches its current Lamport timestamp to that message. In the example on Slide 68, $t = 2$ is attached to m_1 and $t = 4$ is attached to m_2. When the recipient receives a message, it moves its local Lamport clock forward to the timestamp in the message plus one; if the recipient's clock is already ahead of the timestamp in the message, it is only incremented.

Lamport timestamps have the property that if a happened before b, then b always has a greater timestamp than a; in other words, the timestamps are consistent with causality. However, the converse is not true: in general, if b has a greater timestamp than a, we know that $b \nrightarrow a$, but we do not know whether it is the case that $a \to b$ or that $a \parallel b$.

It is also possible for two different events to have the same timestamp. In the example on Slide 68, the third event on node A and the first event on node B both have a timestamp of 3. If we need a unique timestamp for every event, each timestamp can be extended with the name or identifier of the node on which that event occurred. Within the scope of a single node, each event is assigned a unique timestamp; thus, assuming each node has a unique name, the combination of timestamp and node name is globally unique (across all nodes).

Lamport clocks example
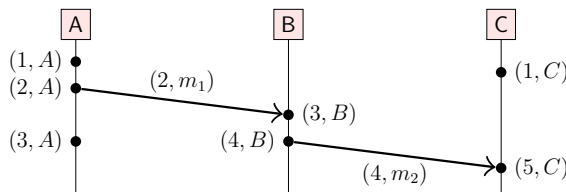
Let $N(e)$ be the node at which event e occurred.
Then the pair $(L(e), N(e))$ **uniquely identifies** event e.

Define a **total order** $\prec$ using Lamport timestamps:

$$(a \prec b) \Longleftrightarrow (L(a) < L(b) \lor (L(a) = L(b) \land N(a) < N(b)))$$

This order is **causal**: $(a \to b) \Longrightarrow (a \prec b)$

Slide 68

Recall that the happens-before relation is a partial order (Slide 62). Using Lamport timestamps we can extend this partial order into a *total order*. We use the lexicographic order over (timestamp, node name) pairs: that is, we first compare the timestamps, and if they are the same, we break ties by comparing the node names.

This relation $\prec$ puts all events into a linear order: for any two events $a \neq b$ we have either $a \prec b$ or

35

$b \prec a$. It is a causal order: that is, whenever $a \to b$ we have $a \prec b$. In other words, $\prec$ is a *linear extension* of the partial order $\to$. However, if $a \parallel b$ we could have either $a \prec b$ or $b \prec a$, so the order of the two events is determined arbitrarily by the algorithm.

Exercise 8. *Given the sequence of messages in the following execution, show the Lamport timestamps at each send or receive event.*
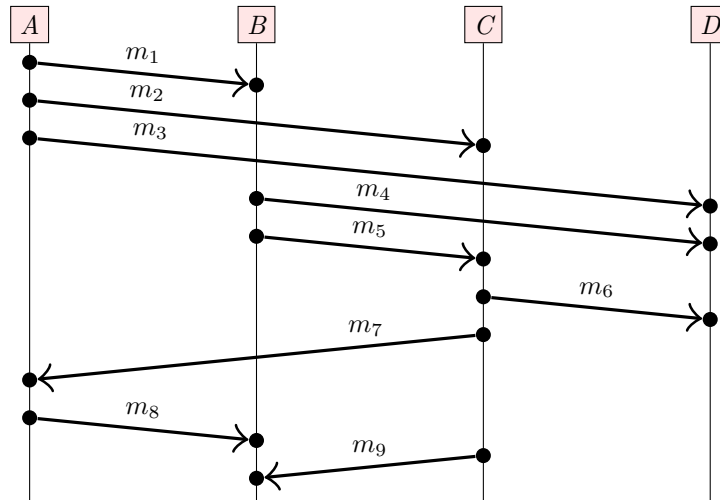
Exercise 9. *Prove that the total order $\prec$ using Lamport timestamps is a causal order.*

Given the Lamport timestamps of two events, it is in general not possible to tell whether those events are concurrent or whether one happened before the other. If we do want to detect when events are concurrent, we need a different type of logical time: a *vector clock*.

While Lamport timestamps are just a single integer (possibly with a node name attached), vector timestamps are a list of integers, one for each node in the system. By convention, if we put the n nodes into a vector $\langle N_1, N_2, \ldots, N_n \rangle$, then a vector timestamp is a similar vector $\langle t_1, t_2, \ldots, t_n \rangle$ where t_i is the entry corresponding to node N_i. Concretely, t_i is the number of events known to have occurred at node N_i.

In a vector $T = \langle t_1, t_2, \ldots, t_n \rangle$ we refer to element t_i as $T[i]$, like an index into an array.

Slide 69

Apart from the difference between a scalar and a vector, the vector clock algorithm is very similar to a Lamport clock (compare Slide 66 and Slide 70). A node initialises its vector clock to contain a zero for each node in the system. Whenever an event occurs at node N_i, it increments the ith entry (its own entry) in its vector clock. (In practice, this vector is often implemented as a map from node IDs to integers rather than an array of integers.) When a message is sent over the network, the sender's current vector timestamp is attached to the message. Finally, when a message is received, the recipient merges the vector timestamp in the message with its local timestamp by taking the element-wise maximum of the two vectors, and then the recipient increments its own entry.

Vector clocks algorithm

on initialisation at node N_i **do**
    $T := \langle 0, 0, \ldots, 0 \rangle$ $\triangleright$ local variable at node N_i
end on

on any event occurring at node N_i **do**
    $T[i] := T[i] + 1$
end on

on request to send message m at node N_i **do**
    $T[i] := T[i] + 1$; send (T, m) via network
end on

on receiving (T', m) at node N_i via the network **do**
    $T[j] := \max(T[j], T'[j])$ for every $j \in \{1, \ldots, n\}$
    $T[i] := T[i] + 1$; deliver m to the application
end on

Slide 71 shows an example of this algorithm in action. Note that when C receives message m_2 from B, the vector entry for A is also updated to 2 because this event has an indirect causal dependency on the two events that happened at A. In this way, the vector timestamps mirror the transitivity of the happens-before relation.

Vector clocks example

Assuming the vector of nodes is $N = \langle A, B, C \rangle$:
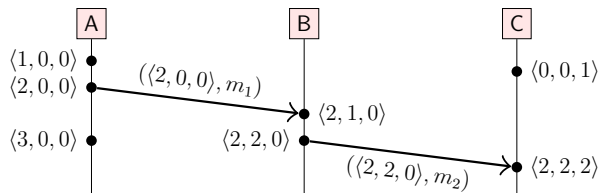
The vector timestamp of an event e represents a set of events, e and its causal dependencies: $\{e\} \cup \{a \mid a \to e\}$

For example, $\langle 2, 2, 0 \rangle$ represents the first two events from A, the first two events from B, and no events from C.

Vector clocks ordering

Define the following order on vector timestamps
(in a system with n nodes):
- $T = T'$ iff $T[i] = T'[i]$ for all $i \in \{1, \ldots, n\}$
- $T \leq T'$ iff $T[i] \leq T'[i]$ for all $i \in \{1, \ldots, n\}$
- $T < T'$ iff $T \leq T'$ and $T \neq T'$
- $T \parallel T'$ iff $T \not\leq T'$ and $T' \not\leq T$

$V(a) \leq V(b)$ iff $(\{a\} \cup \{e \mid e \to a\}) \subseteq (\{b\} \cup \{e \mid e \to b\})$

Properties of this order:
- $(V(a) < V(b)) \iff (a \to b)$
- $(V(a) = V(b)) \iff (a = b)$
- $(V(a) \parallel V(b)) \iff (a \parallel b)$

We then define a partial order over vector timestamps as shown on Slide 72. We say that one vector

is less than or equal to another vector if every element of the first vector is less than or equal to the corresponding element of the second vector. One vector is strictly less than another vector if they are less than or equal, and if they differ in at least one element. However, two vectors are incomparable if one vector has a greater value in one element, and the other has a greater value in a different element. For example, $T = \langle 2, 2, 0 \rangle$ and $T' = \langle 0, 0, 1 \rangle$ are incomparable because $T[1] > T'[1]$ but $T[3] < T'[3]$.

The partial order over vector timestamps corresponds exactly to the partial order defined by the happens-before relation. Thus, the vector clock algorithm provides us with a mechanism for computing the happens-before relation in practice.

Exercise 10. *Given the same sequence of messages as in Exercise 8, show the vector clocks at each send or receive event.*

Exercise 11. *Using the Lamport and vector timestamps calculated in Exercise 8 and 10, state whether or not the following events can be determined to have a happens-before relationship.*

Events		Lamport	Vector
send(m_2)	send(m_3)		
send(m_3)	send(m_5)		
send(m_5)	send(m_9)		

Exercise 12. *We have seen several types of physical clocks (time-of-day clocks with NTP, monotonic clocks) and logical clocks. For each of the following uses of time, explain which type of clock is the most appropriate: process scheduling; I/O; distributed filesystem consistency; cryptographic certificate validity; concurrent database updates.*

This completes our discussion of logical time. We have seen two key algorithms: Lamport clocks and vector clocks, one providing a total order and the other capturing the partial order of happens-before. Various other constructions have been proposed: for example, there are hybrid clocks that combine some of the properties of logical and physical clocks [Kulkarni et al., 2014].

4.2 Delivery order in broadcast protocols

Many networks provide point-to-point (unicast) messaging, in which a message has one specified recipient. We will now look at *broadcast protocols*, which generalise networking such that a message is sent to *all* nodes in some group. The group membership may be fixed, or the system may provide mechanisms for nodes to join and leave the group.

Some local-area networks provide multicast or broadcast at the hardware level (for example, *IP multicast*), but communication over the Internet typically only allows unicast. Moreover, hardware-level multicast is typically provided on a *best-effort* basis, which allows messages to be dropped; making it reliable requires retransmission protocols similar to those discussed here.

The system model assumptions about node behaviour (Slide 34) and synchrony (Slide 35) carry over directly to broadcast groups.

Broadcast protocols

Broadcast (multicast) is **group communication**:
- ▶ One node sends message, all nodes in group deliver it
- ▶ Set of group members may be fixed (static) or dynamic
- ▶ If one node is faulty, remaining group members carry on
- ▶ Note: concept is more general than IP multicast
  (we build upon point-to-point messaging)

Build upon system models from lecture 2:
- ▶ Can be **best-effort** (may drop messages) or
  reliable (non-faulty nodes deliver every message,
  by retransmitting dropped messages)
- ▶ Asynchronous/partially synchronous timing model
  $\implies$ **no upper bound** on message latency

Slide 73

Before we go into the details, we should clarify some terminology. When an application wants to send a message to all nodes in the group, it uses an algorithm to *broadcast* it. To make this happen, the broadcast algorithm then *sends* some messages to other nodes over point-to-point links, and another node *receives* such a message when it arrives over the point-to-point link. Finally, the broadcast algorithm may *deliver* the message to the application. As we shall see shortly, there is sometimes a delay between the time when a message is received and when it is delivered.
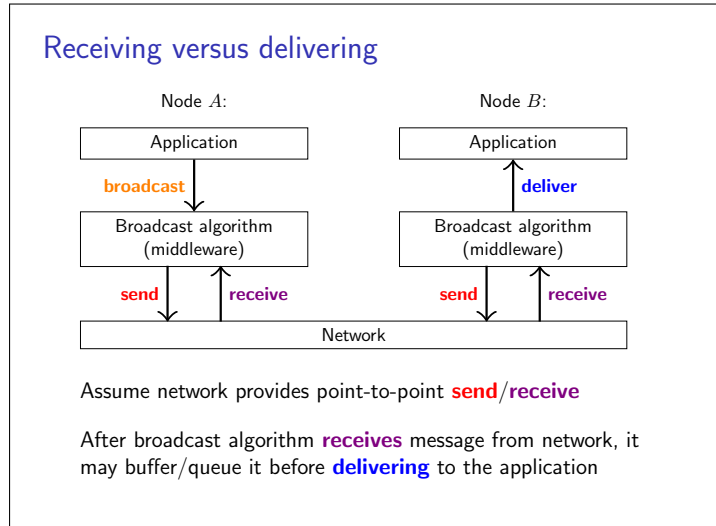
Slide 74

We will examine three different forms of broadcast. All of these are *reliable*: every message is eventually delivered to every non-faulty node, with no timing guarantees. However, they differ in terms of the order in which messages may be delivered at each node. It turns out that this difference in ordering has very fundamental consequences for the algorithms that implement the broadcast.

Forms of reliable broadcast

FIFO broadcast:
If m_1 and m_2 are broadcast by the same node, and broadcast(m_1) → broadcast(m_2), then m_1 must be delivered before m_2

Causal broadcast:
If broadcast(m_1) → broadcast(m_2) then m_1 must be delivered before m_2

Total order broadcast:
If m_1 is delivered before m_2 on one node, then m_1 must be delivered before m_2 on all nodes

FIFO-total order broadcast:
Combination of FIFO broadcast and total order broadcast

Slide 75

The weakest type of broadcast is called *FIFO broadcast*, which is closely related to FIFO links (see Exercise 2). In this model, messages sent by the same node are delivered in the order they were sent. For example, on Slide 76, m_1 must be delivered before m_3, since they were both sent by A. However, m_2 can be delivered at any time before, between, or after m_1 and m_3.

Another detail about these broadcast protocols: we assume that whenever a node broadcasts a message, it also delivers that message to itself (represented as a loopback arrow on Slide 76). This may seem unnecessary at first – after all, a node knows what messages it has itself broadcast! – but we will need this for total order broadcast.

FIFO broadcast

A   B   C

m_1

m_1

m_1

m_2 m_2 m_2

m_3 m_3

m_3
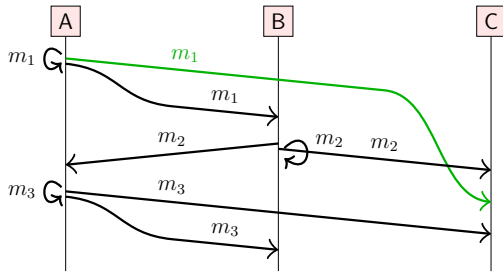
Messages sent by the same node must be delivered in the order they were sent.
Messages sent by different nodes can be delivered in any order.
Valid orders: (m_2, m_1, m_3) or (m_1, m_2, m_3) or (m_1, m_3, m_2)

The example execution on Slide 76 is valid FIFO broadcast, but it violates causality: node C delivers m_2 before m_1, even though B broadcast m_2 after delivering m_1. *Causal broadcast* provides a stricter ordering property than FIFO broadcast. As the name suggests, it ensures that messages are delivered in causal order: that is, if the broadcast of one message happened before the broadcast of another message, then all nodes must deliver those two messages in that order. If two messages are broadcast concurrently, a node may deliver them in either order.

In the example on Slide 76, if node C receives m_2 before m_1, the broadcast algorithm at C will have to *hold back* (delay or buffer) m_2 until after m_1 has been delivered, to ensure the messages are delivered in causal order. In the example on Slide 77, messages m_2 and m_3 are broadcast concurrently. Nodes A and C deliver messages in the order m_1, m_3, m_2, while node B delivers them in the order m_1, m_2, m_3. Either of these delivery orders is acceptable, since they are both consistent with causality.

Causal broadcast

A   B   C

m_1

m_1

m_1

m_3 m_3 m_2 m_2

m_3

Causally related messages must be delivered in causal order.
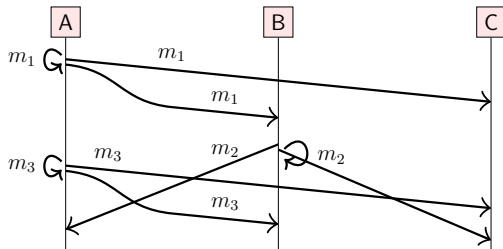Concurrent messages can be delivered in any order.

Here: broadcast(m_1) $\rightarrow$ broadcast(m_2) and
broadcast(m_1) $\rightarrow$ broadcast(m_3)
$\implies$ valid orders are: (m_1, m_2, m_3) or (m_1, m_3, m_2)

The third type of broadcast is *total order broadcast*, sometimes also known as *atomic broadcast*. While FIFO and causal broadcast allow different nodes to deliver messages in different orders, total order broadcast enforces consistency across the nodes, ensuring that all nodes deliver messages in the same order. The precise delivery order is not defined, as long as it is the same on all nodes.

Slide 78 and 79 show two example executions of total order broadcast. On Slide 78, all three nodes deliver the messages in the order m_1, m_2, m_3, while on Slide 79, all three nodes deliver the messages in the order of m_1, m_3, m_2. Either of these executions is valid, as long as the nodes agree.

As with causal broadcast, nodes may need to hold back messages, waiting for other messages that need to be delivered first. For example, node C could receive messages m_2 and m_3 in either order. If the algorithm has determined that m_3 should be delivered before m_2, but if node C receives m_2 first, then C will need to hold back m_2 until after m_3 has been received.

Another important detail can be seen on these diagrams: in the case of FIFO and causal broadcast,

when a node broadcasts a message, it can immediately deliver that message to itself, without having to wait for communication with any other node. This is no longer true in total order broadcast: for example, on Slide 78, m_2 needs to be delivered before m_3, so node A's delivery of m_3 to itself must wait until after A has received m_2 from B. Likewise, on Slide 79, node B's delivery of m_2 to itself must wait for m_3.

Total order broadcast (1)
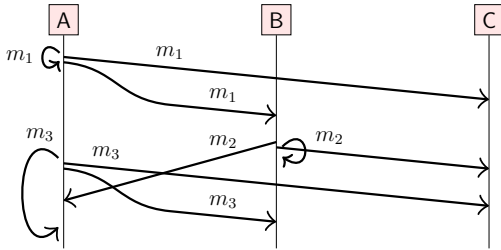
All nodes must deliver messages in **the same** order
(here: m_1, m_2, m_3)

This includes a node's deliveries to itself!

Slide 78

Total order broadcast (2)
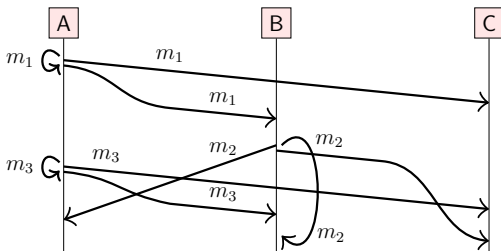
All nodes must deliver messages in **the same** order
(here: m_1, m_3, m_2)

This includes a node's deliveries to itself!

Slide 79

Relationships between broadcast models

FIFO-total order broadcast

Causal broadcast

Total order broadcast

FIFO broadcast

Reliable broadcast

Best-effort broadcast

$\longrightarrow$ = stronger than

Slide 80

41
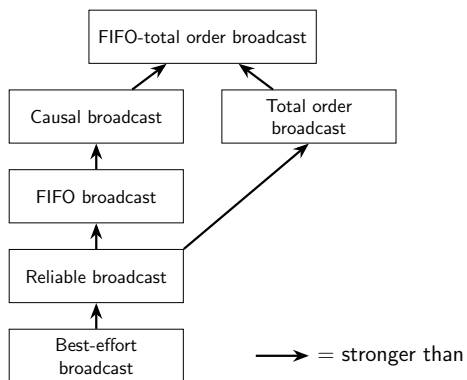
Finally, *FIFO-total order broadcast* is like total order broadcast, but with the additional FIFO requirement that any messages broadcast by the same node are delivered in the order they were sent. The examples on Slide 78 and 79 are in fact valid FIFO-total order broadcast executions, since m_1 is delivered before m_3 in both.

Exercise 13. *Prove that causal broadcast also satisfies the requirements of FIFO broadcast, and that FIFO-total order broadcast also satisfies the requirements of causal broadcast.*

4.3   Broadcast algorithms

We will now move on to algorithms for implementing broadcast. Roughly speaking, this involves two steps: first, ensuring that every message is received by every node; and second, delivering those messages in the right order. We will first look at disseminating the messages reliably.

The first algorithm we might try is: when a node wants to broadcast a message, it individually sends that message to every other node, using reliable links as discussed on Slide 33 (i.e. retransmitting dropped messages). However, it could happen that a message is dropped, and the sender crashes before retransmitting it. In this situation, one of the nodes will never receive that message.

Slide 81

To improve the reliability, we can enlist the help of the other nodes. For example, we could say that the first time a node receives a particular message, it forwards it to every other node (this is called *eager reliable broadcast*). This algorithm ensures that even if some nodes crash, all of the remaining (non-faulty) nodes will receive every message. However, this algorithm is fairly inefficient: in the absence of faults, every message is sent $O(n^2)$ times in a group of n nodes, as each node will receive every message $n-1$ times. This means it uses a large amount of redundant network traffic.
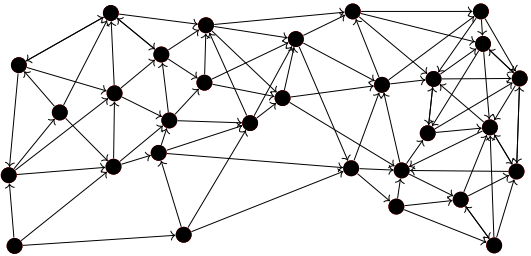
Slide 82

Many variants of this algorithm have been developed, optimising along various dimensions such as the fault tolerance, the time until all nodes receive a message, and the network bandwidth used. One particularly common family of broadcast algorithms are *gossip protocols* (also known as *epidemic protocols*). In these protocols, a node that wishes to broadcast a message sends it to a small fixed number of nodes that are chosen randomly. On receiving a message for the first time, a node forwards it to a fixed number of randomly chosen nodes. This resembles the way gossip, rumours, or an infectious disease may spread through a population.

Gossip protocols do not strictly guarantee that all nodes will receive a message: it is possible that in the random selection of nodes, some node is always omitted. However, if the parameters of the algorithm are chosen appropriately, the probability of a message not being delivered can be very small. Gossip protocols are appealing because, with the right parameters, they are very resilient to message loss and node crashes while also remaining efficient.

Slide 83

Now that we have reliable broadcast (using eager reliable broadcast or a gossip protocol), we can build FIFO, causal, or total order broadcast on top of it. Let's start with FIFO broadcast.

Slide 84

Each FIFO broadcast message sent by node N_i is tagged with the sending node number i and a sequence number that is 0 for the first message sent by N_i, 1 for the second message, and so on. The local state at each node consists of the sequence number *sendSeq* (counting the number of messages broadcast by this node), *delivered* (a vector with one entry per node, counting the number of messages from each sender that this node has delivered), and *buffer* (a buffer for holding back messages until they are ready to be delivered). The algorithm checks for messages from any sender that match the expected next sequence number, and then increments that number, ensuring that messages from each particular sender are delivered in order of increasing sequence number.

The causal broadcast algorithm is somewhat similar to FIFO broadcast; instead of attaching a sequence number to every message that is broadcast, we attach a vector of integers. This algorithm is sometimes called a *vector clock* algorithm, even though it is quite different from the algorithm on Slide 70. In the vector clock algorithm from Slide 70 the vector elements count the number of *events* that have occurred at each node, while in the causal broadcast algorithm, the vector elements count the number of *messages* from each sender that have been delivered.

The local state at each node consists of *sendSeq*, *delivered*, and *buffer*, which have the same meaning as in the FIFO broadcast algorithm. When a node wants to broadcast a message, we attach the sending node number i and *deps*, a vector indicating the *causal dependencies* of that message. We construct *deps* by taking a copy of *delivered*, the vector that counts how many messages from each sender have been delivered at this node. This indicates that all messages that have been delivered locally prior to this broadcast must appear before the broadcast message in the causal order. We then update the sending node's own element of this vector to equal *sendSeq*, which ensures that each message broadcast by this node has a causal dependency on the previous message broadcast by the same node.

When receiving a message, the algorithm first adds it to the buffer like in FIFO broadcast, and then searches the buffer for any messages that are ready to be delivered. The comparison $deps \leq delivered$ uses the $\leq$ operator on vectors defined on Slide 72. This comparison is true if this node has already delivered all of the messages that must precede this message in the causal order. Any messages that are causally ready are then delivered to the application and removed from the buffer, and the appropriate element of the *delivered* vector is incremented.

Finally, total order broadcast (and FIFO-total order broadcast) are trickier. Two simple approaches are outlined on Slide 86, one based on a designated leader node, and a leaderless algorithm using Lamport timestamps. However, neither of these approaches is fault tolerant: in both cases, the crash of a single

node can stop all other nodes from being able to deliver messages. In the single-leader approach, the leader is a single point of failure. We will return to the problem of fault-tolerant total order broadcast in Lecture 6.

Exercise 14. *Give pseudocode for an algorithm that implements FIFO-total order broadcast using Lamport clocks. You may assume that each node has a unique ID, and that the set of all node IDs is known. Further assume that the underlying network provides reliable FIFO broadcast. [2020 Paper 5 Question 8]*

5 Replication

We will now turn to the problem of *replication*, which means to maintain a copy of the same data on multiple nodes, each of which is called a *replica*. Replication is a standard feature of many distributed databases, filesystems, and other storage systems. It is one of the main mechanisms we have for achieving *fault tolerance*: if one replica becomes faulty, we can continue accessing the copies of the data on other replicas.

Replication

- ▶ Keeping a copy of the same data on multiple nodes
- ▶ Databases, filesystems, caches, . . .
- ▶ A node that has a copy of the data is called a **replica**
- ▶ If some replicas are faulty, others are still accessible
- ▶ Spread load across many replicas
- ▶ Easy if the data doesn't change: just copy it
- ▶ We will focus on data changes

Compare to **RAID** (Redundant Array of Independent Disks): replication within a single computer

- ▶ RAID has single controller; in distributed system, each node acts independently
- ▶ Replicas can be distributed around the world, near users

Slide 87

5.1 Manipulating remote state

If the data doesn't change, replication is easy, since it just requires making a one-time copy of the data. Therefore, the main problem in replication is managing changes to the data. Before we get into the details of replication, let's look at how data changes happen in a distributed system.

Retrying state updates

User A: The moon is not actually made of cheese!

⚫ Like   12,300 people like this.

client

increment post.likes

ack → 12,301

✗

increment post.likes

ack → 12,302

✗
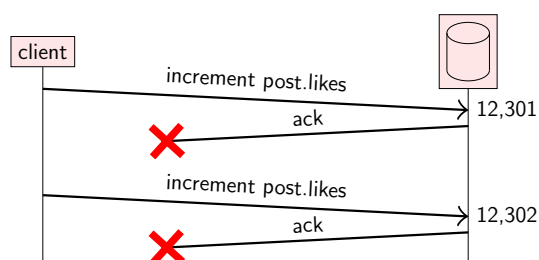
Deduplicating requests requires that the database tracks which requests it has already seen (in stable storage)

Slide 88

Let's consider as example the act of "liking" a status update on a social network. When you click the "like" button, the fact that you have liked it, and the number of people who have liked it, need to be stored somewhere so that they can be displayed to you and to other users. This usually happens in a database on the social network's servers. We can consider the data stored in a database to be its *state*.

A request to update the database may be lost in the network, or an acknowledgement that an update has been performed might be lost. As usual, we can improve reliability by retrying the request. However, if we are not careful, the retry could lead to the request being processed multiple times, leading to an incorrect state in the database.
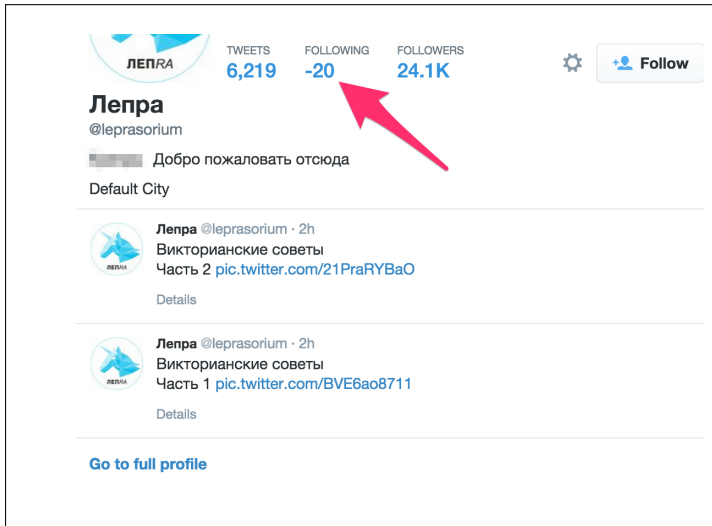
Slide 89

Lest you think this is a purely hypothetical problem, consider Slide 89, a genuine (I promise!) screenshot of Twitter that I made in 2014, which shows the profile of a user who is apparently following a negative number of people. I don't have insight into Twitter's internals to know exactly what happened here, but my guess is that this person used to follow several people, then unfollowed them, and due to a network problem during the unfollowing process, the decrement of the follow counter was retried, resulting in more decrements than the user was originally following.

If following a negative number of people seems like a trivial problem, then instead of decrementing a follow counter, consider the act of deducting £1,000 from your bank account balance. The database operation is essentially the same, but performing this operation too many times has the potential to make you rather unhappy.

One way of preventing an update from taking effect multiple times is to deduplicate requests. However, in a crash-recovery system model, this requires storing requests (or some metadata about requests, such as a vector clock) in stable storage, so that duplicates can be accurately detected even after a crash.

An alternative to recording requests for deduplication is to make requests *idempotent*.

Slide 90

Incrementing a counter is not idempotent, but adding an element to a set is. Therefore, if a counter is required (as in the number of likes), it might be better to actually maintain the set of elements in the database, and to derive the counter value from the set by computing its cardinality.

An idempotent update can safely be retried, because performing it several times has the same effect as performing it once. Idempotence allows an update to have *exactly-once* semantics: that is, the update may actually be applied multiple times, but the effect is the same as if it had been applied exactly once. Idempotence is a very useful property in practical systems, and it is often found in the context of RPC (Section 1.3), where retries are often unavoidable.

However, idempotence has a limitation that becomes apparent when there are multiple updates in progress. On Slide 91, client 1 adds a user ID to the set of likes for a post, but the acknowledgement is lost. Client 2 reads the set of likes from the database (including the user ID added by client 1), and then makes a request to remove the user ID again. Meanwhile, client 1 retries its request, unaware of the update made by client 2. The retry therefore has the effect of adding the user ID to the set again. This is unexpected since client 2 observed client 1's change, so the removal happened causally after the addition of the set element, and therefore we might expect that in the final state, the user ID should *not* be present in the set. In this case, the fact that adding an element to a set is idempotent is not sufficient to make the retry safe.

Adding and then removing again

$f(likes) = likes \cup \{userID\}$
$g(likes) = likes \setminus \{userID\}$
Idempotent? $f(f(x)) = f(x)$ but $f(g(f(x)) \neq g(f(x))$
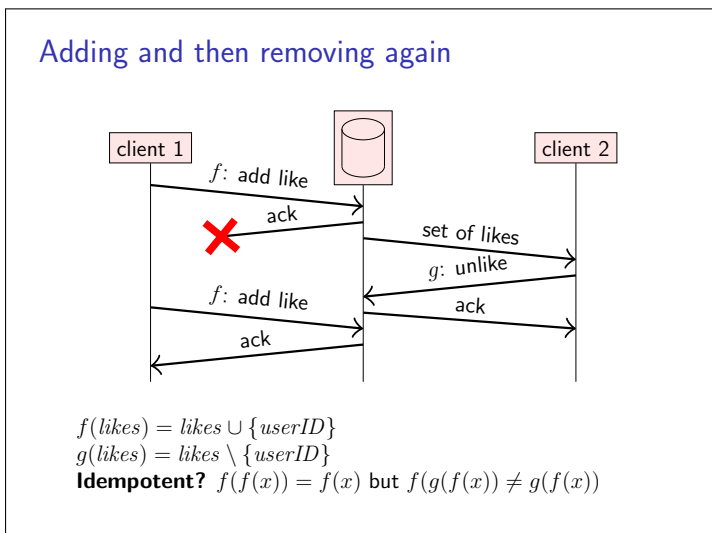
Slide 91

A similar problem occurs on Slide 92, in which we have two replicas. In the first scenario a client first adds x to both replicas of the database, then tries to remove x again from both. However, the remove request to replica B is lost, and the client crashes before it is able to retry. In the second scenario a client tries to add x to both replicas, but the request to replica A is lost, and again the client crashes.

Another problem with adding and removing

Final state ($x \notin A$, $x \in B$) is the same as in this case:
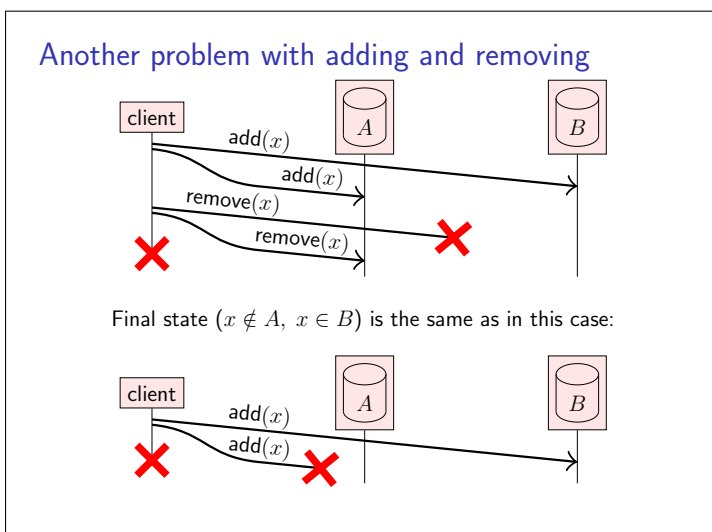
Slide 92

In both scenarios the outcome is the same: x is present on replica B, and absent from replica A.

47

Yet the intended effect is different: in the first scenario, the client wanted x to be removed from both replicas, whereas in the second scenario, the client wanted x to be present on both replicas. When the two replicas reconcile their inconsistent states, we want them to both end up in the state that the client intended. However, this is not possible if the replicas cannot distinguish between these two scenarios.

To solve this problem, we can do two things. First, we attach a logical timestamp to every update operation, and store that timestamp in the database as part of the data written by the update. Second, when asked to remove a record from the database, we don't actually remove it, but rather write a special type of update (called a *tombstone*) marking it as deleted. On Slide 93, records containing false are tombstones.
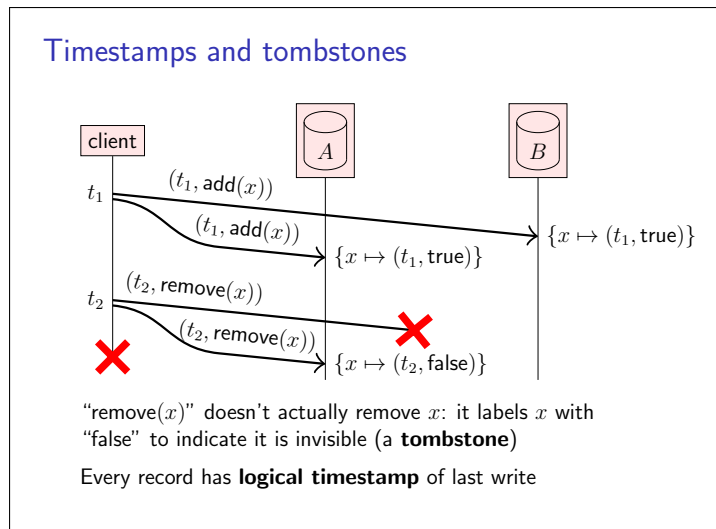
Slide 93

In many replicated systems, replicas run a protocol to detect and reconcile any differences (this is called *anti-entropy*), so that the replicas eventually hold consistent copies of the same data. Thanks to tombstones, the anti-entropy process can tell the difference between a record that has been deleted and a record that has not yet been created. And thanks to timestamps, we can tell which version of a record is older and which is newer. The anti-entropy process then keeps the newer and discards the older record.

This approach also helps address the problem on Slide 91: a retried request has the same timestamp as the original request, so a retry will not overwrite a value written by a causally later request with a greater timestamp.
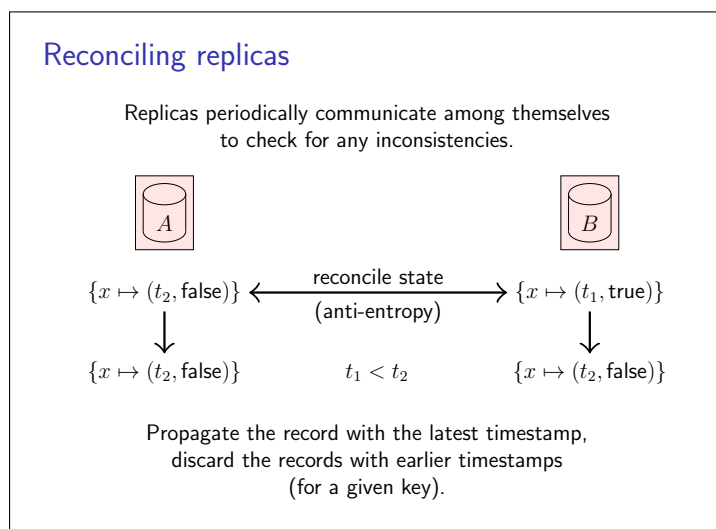
Slide 94

The technique of attaching a timestamp to every update is also useful for handling concurrent updates. On Slide 95, client 1 wants to set key x to the value v_1 (with timestamp t_1), while concurrently client 2 wants to set the same key x to the value v_2 (with timestamp t_2). Replica A receives v_2 first and v_1 second, while replica B receives the updates in the opposite order. To ensure that both replicas end up in the same state, we rely not on the order in which they receive requests, but the order of their timestamps.
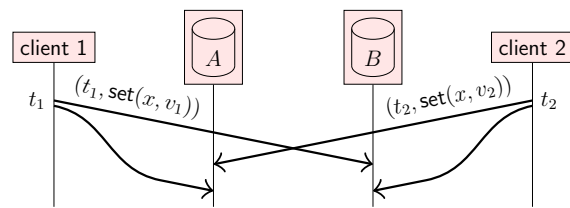
The details of this approach depend on the type of timestamps used. If we use Lamport clocks (with the total order defined on Slide 68), two concurrent updates will be ordered arbitrarily, depending on how the timestamps happen to get assigned. In this case, we get what is known as *last writer wins* (LWW) semantics: the update with the greatest timestamp takes effect, and any concurrent updates with lower timestamps to the same key are discarded. This approach is simple to work with, but it does imply data loss when multiple updates are performed concurrently. Whether or not this is a problem depends on the application: in some systems, discarding concurrent updates is fine.

When discarding concurrent updates is not acceptable, we need to use a type of timestamp that allows us to detect when updates happen concurrently, such as vector clocks. With such partially ordered timestamps, we can tell when a new value should overwrite an old value (when the old update happened before the new update), and when several updates are concurrent, we can keep all of the concurrently written values. These concurrently written values are called *conflicts*, or sometimes *siblings*. The application can later merge conflicts back into a single value, as discussed in Lecture 8.

A downside of vector clocks is that they can become expensive: every client needs an entry in the vector, and in systems with large number of clients (or where clients assume a new identity every time they are restarted), these vectors can become large, potentially taking up more space than the data itself. Further types of logical clocks, such as *dotted version vectors* [Preguiça et al., 2010], have been developed to optimise this type of system.

Exercise 15. *Apache Cassandra, a widely-used distributed database, uses a replication approach similar to the one described here. However, it uses physical timestamps instead of logical timestamps, as discussed here: https://www.datastax.com/blog/2013/09/why-cassandra-doesnt-need-vector-clocks. Write a critique of this blog post. What do you think of its arguments and why? What facts are missing from it? What recommendation would you make to someone considering using Cassandra?*

5.2 Quorums

As discussed at the start of this lecture, replication is useful since it allows us to improve the reliability of a system: when one replica is unavailable, the remaining replicas can continue processing requests. Unavailability could be due to a faulty node (e.g. a crash or a hardware failure), or due to a network partition (inability to reach a node over the network), or planned maintenance (e.g. rebooting a node to install software updates).

However, the details of how exactly the replication is performed have a big impact on the reliability of the system. Without fault tolerance, having multiple replicas would make reliability *worse*: the more replicas you have, the greater the probability that *any one* of the replicas is faulty at any one time (assuming faults are not perfectly correlated). However, if the system continues working despite some faulty replicas, then reliability improves: the probability that *all* replicas are faulty at the same time is much lower than the probability of one replica being faulty.

Probability of faults

A replica may be **unavailable** due to network partition or node fault (e.g. crash, hardware problem).

Assume each replica has probability p of being faulty or unavailable at any one time, and that faults are independent. (Not actually true! But okay approximation for now.)

Probability of **all** n replicas being faulty: p^n
Probability of ≥ 1 out of n replicas being faulty: $1 - (1-p)^n$

Example with $p = 0.01$:

replicas n	$P(\geq 1$ faulty$)$	$P(\geq \frac{n+1}{2}$ faulty$)$	$P($all n faulty$)$
1	0.01	0.01	0.01
3	0.03	$3 \cdot 10^{-4}$	10^{-6}
5	0.049	$1 \cdot 10^{-5}$	10^{-10}
100	0.63	$6 \cdot 10^{-74}$	10^{-200}

Slide 96

We will now explore how to achieve fault tolerance in replication. To start, consider the example on Slide 97. Assume we have two replicas, A and B, which initially both associate the key x with a value v_0 (and timestamp t_0). A client attempts to update the value of x to v_1 (with timestamp t_1). It succeeds in updating B, but the update to A fails as A is temporarily unavailable. Subsequently, the client attempts to read back the value it has written; the read succeeds at A but fails at B. As a result, the read does not return the value v_1 previously written by the same client, but rather the initial value v_0.

Read-after-write consistency
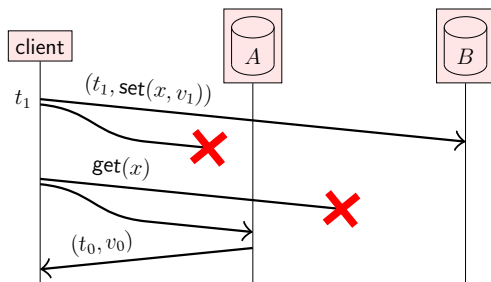
Writing to one replica, reading from another: client does not read back the value it has written

Require writing to/reading from both replicas $\implies$ cannot write/read if one replica is unavailable

Slide 97

This scenario is problematic, since from the client's point of view it looks as if the value it has written has been lost. Imagine you post an update on a social network, then refresh the page, and don't see the update you have just posted. As this behaviour is confusing for users, many systems require *read-after-write consistency* (also known as *read-your-writes consistency*), in which we ensure that after a client writes a value, the same client will be able to read back the same value from the database.

Strictly speaking, with read-after-write consistency, after writing a client may not read the value it wrote because concurrently another client may have overwritten the value. Therefore we say that read-after-write consistency requires reading either the last value written, or a later value.

On Slide 97 we could guarantee read-after-write consistency by ensuring we always write to both replicas and/or read from both replicas. However, this would mean that reads and/or writes are no longer fault-tolerant: if one replica is unavailable, a write or read that requires responses from both replicas would not be able to complete.

We can solve this conundrum by using three replicas, as shown on Slide 98. We send every read and write request to all three replicas, but we consider the request successful as long as we receive ≥ 2 responses. In the example, the write succeeds on replicas B and C, while the read succeeds on replicas A and B. With a "2 out of 3" policy for both reads and writes, it is guaranteed that at least one of the responses to a read is from a replica that saw the most recent write (in the example, this is replica B).

50
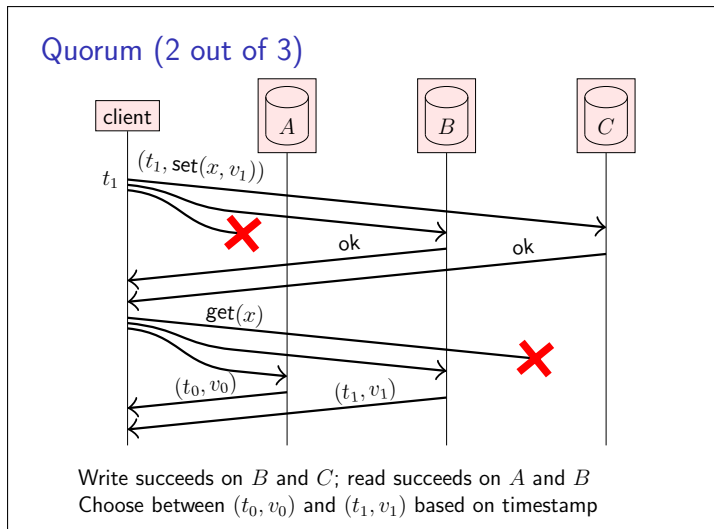
Slide 98

Different replicas may return different responses to the same read request: on Slide 98, the read at A returns the initial value (t_0, v_0), while the read at B returns the value (t_1, v_1) previously written by this client. Using the timestamps, the client can tell which response is the more recent one, and return v_1 to the application.

In this example, the set of replicas $\{B, C\}$ that responded to the write request is a *write quorum*, and the set $\{A, B\}$ that responded to the read is a *read quorum*. In general, a *quorum* is a minimum set of nodes that must respond to some request for it to be successful. (The term comes from politics, where a quorum refers to the minimum number of votes required to make a valid decision, e.g. in a parliament or committee.) In order to ensure read-after-write consistency, the quorum for the write and the quorum for the read must have a non-empty intersection: in other words, the read quorum must contain at least one node that has acknowledged the write.

A common choice of quorum in distributed systems is a *majority quorum*, which is any subset of nodes that comprises strictly more than half of the nodes. In a system with three nodes $\{A, B, C\}$, the majority quorums are $\{A, B\}$, $\{A, C\}$, and $\{B, C\}$. In general, in a system with an odd number of nodes n, any subset of size $\frac{n+1}{2}$ is a majority quorum (2 out of 3, or 3 out of 5, ...). With an even number of nodes n, this needs to be rounded up to $\lceil \frac{n+1}{2} \rceil = \frac{n+2}{2}$. For example, 3 out of 4 form a majority quorum. Majority quorums have the property that any two quorums always have at least one element in common. However, other quorum constructions besides majorities are also possible.
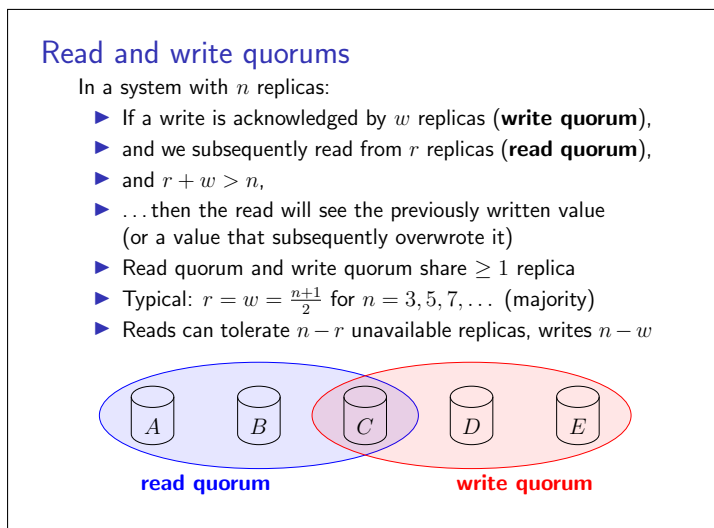
Slide 99

A system that requires w acknowledgements for writes (i.e. a write quorum of size w) can continue processing updates as long as no more than $n - w$ replicas are unavailable, and a system that requires r responses for reads can continue reading as long as no more than $n - r$ replicas are unavailable. With majority quorums, this means that a system of three replicas can tolerate one replica being unavailable, a system of five replicas can tolerate two being unavailable, and so on.

51

In this quorum approach to replication, some updates may be missing from some replicas at any given moment: for example, on Slide 98, the (t_1, v_1) update is missing from replica A, since that write request was dropped. To bring replicas back in sync with each other, one approach is to rely on an anti-entropy process, as discussed on Slide 94.
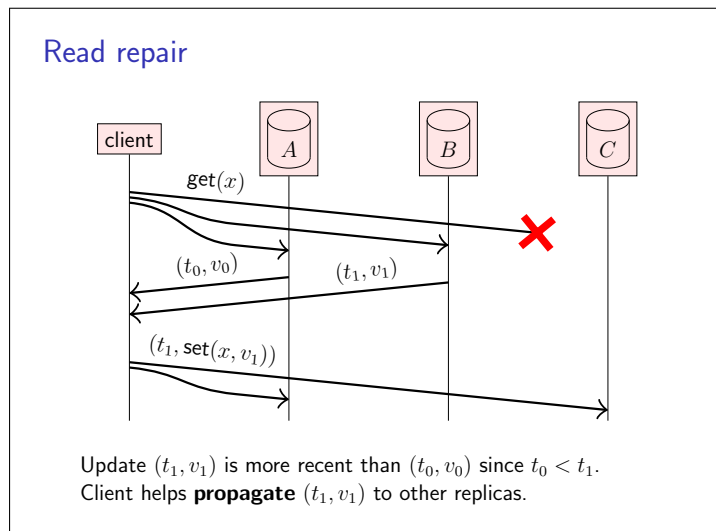
Slide 100

Another option is to get clients to help with the process of disseminating updates. For example, on Slide 100, the client reads (t_1, v_1) from B, but it receives an older value (t_0, v_0) from A, and no response from C. Since the client now knows that the update (t_1, v_1) needs to be propagated to A, it can send that update to A (using the original timestamp t_1, since this is not a new update, only a retry of a previous update). The client may also send the update to C, even though it does not know whether C needs it (if it turns out that C already has this update, only a small amount of network bandwidth is wasted). This process is called *read repair*. The client can perform read repair on any read request it makes, regardless of whether it was the client that originally performed the update in question.

Databases that use this model of replication are often called *Dynamo-style*, after Amazon's Dynamo database [DeCandia et al., 2007], which popularised it. However, the approach actually predates Dynamo [Attiya et al., 1995].

5.3   Replication using broadcast

The quorum approach of Section 5.2 essentially uses best-effort broadcast: a client broadcasts every read or write request to all of the replicas, but the protocol is unreliable (requests might be lost) and provides no ordering guarantees.

An alternative approach to replication is to use the broadcast protocols from Lecture 4. Let's first consider FIFO-total order broadcast, the strongest form of broadcast we have seen.

Start of video section 5.3
(mp4 download)

Slide 101

52

Using FIFO-total order broadcast it is easy to build a replicated system: we broadcast every update request to the replicas, which update their state based on each message as it is delivered. This is called *state machine replication* (SMR), because a replica acts as a state machine whose inputs are message deliveries. We only require that the update logic is *deterministic*: any two replicas that are in the same state, and are given the same input, must end up in the same next state. Even errors must be deterministic: if an update succeeds on one replica but fails on another, they would become inconsistent.

An excellent feature of SMR is that the logic for moving from one state to the next can be arbitrarily complex, as long as it is deterministic. For example, an entire database transaction with arbitrary business logic can be executed, and this logic can depend both on the broadcast message and the current state of the database. Some distributed database perform replication in this way, with each replica independently executing the same deterministic transaction code (this is known as *active replication*). This principle also underpins blockchains, cryptocurrencies, and distributed ledgers: the "chain of blocks" in a blockchain is nothing other than the sequence of messages delivered by a total order broadcast protocol (more on this in Lecture 6), and each replica deterministically executes the transactions described in those blocks to determine the state of the ledger (e.g. who owns which money). A "smart contract" is just a deterministic program that a replica executes when a particular message is delivered.

Slide 102

The downsides of state machine replication are the limitations of total order broadcast. As discussed in Section 4.2, when a node wants to broadcast a message through a total order broadcast, it cannot immediately deliver that message to itself. For this reason, when using state machine replication, a replica that wants to update its state cannot do so immediately, but it has to go through the broadcast process, coordinate with other nodes, and wait for the update to be delivered back to itself. The fault tolerance of state machine replication depends on the fault tolerance of the underlying total order broadcast, which we will discuss in Lecture 6. Nevertheless, replication based on total order broadcast is widely used.
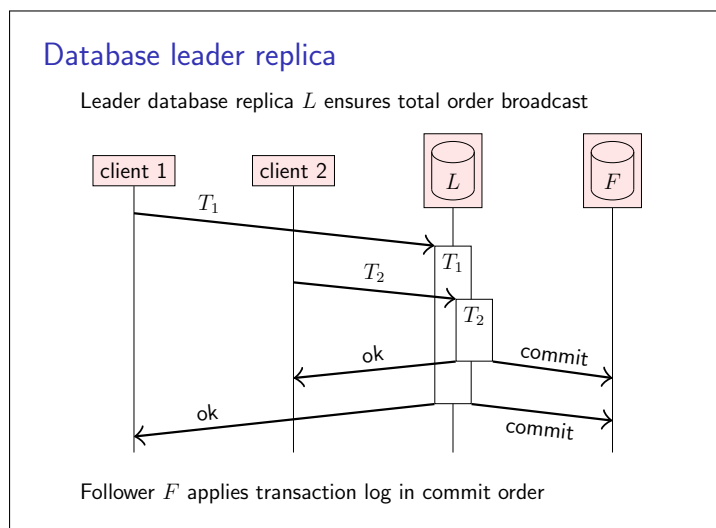
Slide 103

53

Recall from Slide 86 that one way of implementing total order broadcast is to designate one node as the *leader*, and to route all broadcast messages through it in order to impose a delivery order. This principle is also widely used for database replication: many database systems designate one replica as *leader*, *primary*, or *master*. Any transactions that wish to modify the database must be executed on the leader replica. As shown on Slide 103, the leader may execute multiple transactions concurrently; however, it *commits* those transactions in a total order. When a transaction commits, the leader replica broadcasts the data changes from that transaction to all the follower replicas, and the followers apply those changes in commit order. This approach is known as *passive replication*, and we can see that it is equivalent to total order broadcast of transaction commit records.

So much on using total order broadcast for replication. What about the other broadcast models from Lecture 4 – can we use them for replication too? The answer is yes, as shown on Slide 104; however, more care is required to ensure that replicas remain consistent. It is not sufficient to merely ensure that the state update is deterministic.

For example, we can use causal broadcast, which ensures the same delivery order across replicas when one update happened before another, but which may deliver concurrent updates in any order. If we want to ensure that replicas end up in the same state, no matter in which order concurrent updates are delivered, we need to make those updates *commutative*: that is, we have to ensure that the final result is the same, no matter in which order those updates are applied. This can be done, and we will see some techniques for commutativity in Lecture 8.

Replication using causal (and weaker) broadcast

State machine replication uses (FIFO-)total order broadcast. Can we use weaker forms of broadcast too?

If replica state updates are **commutative**, replicas can process updates in different orders and still end up in the same state.

Updates f and g are commutative if $f(g(x)) = g(f(x))$

broadcast	assumptions about state update function
total order	deterministic (SMR)
causal	deterministic, concurrent updates commute
reliable	deterministic, all updates commute
best-effort	deterministic, commutative, idempotent, tolerates message loss

Slide 104

---

6 Consensus

In this lecture we return to the problem of total order broadcast. We saw in Section 5.3 that total order broadcast is very useful for enabling state machine replication. As discussed on Slide 86, one way of implementing total order broadcast is by designating one node as the *leader*, and routing all messages via it. The leader then just needs to distribute the messages via FIFO broadcast, and this is sufficient to ensure that all nodes deliver the same sequence of messages in the same order.

However, the big problem with this approach is that the leader is a single point of failure: if it becomes unavailable, the whole system grinds to a halt. One way of overcoming this is through manual intervention: a human operator can be notified if the leader becomes unavailable, and this person then reconfigures all of the nodes to use a different node as their leader. This process is called *failover*, and it is in fact used in many database systems.

Failover works fine in situations where the leader unavailability is planned in advance, for example when the leader needs to be rebooted to install software updates. However, for sudden and unexpected leader outages (e.g. a crash, hardware failure, or network problem), failover suffers from the fact that humans are limited in how fast they can perform this procedure. Even in the best case, it will take several minutes for an operator to respond, during which the system is not able to process any updates.

This raises the question: can we automatically transfer the leadership from one node to another in the case that the old leader becomes unavailable? The answer is yes, and this is exactly what consensus algorithms do.

Fault-tolerant total order broadcast

Total order broadcast is very useful for state machine replication.

Can implement total order broadcast by sending all messages via a single **leader**.

Problem: what if leader crashes/becomes unavailable?

- ▶ **Manual failover**: a human operator chooses a new leader, and reconfigures each node to use new leader

  Used in many databases! Fine for planned maintenance.

  Unplanned outage? Humans are slow, may take a long time until system recovers. . .

- ▶ Can we **automatically choose a new leader**?

Slide 105

6.1 Introduction to consensus

The consensus problem is traditionally formulated as follows: several nodes want to come to *agreement* about a value. One or more nodes may *propose* a value, and then the consensus algorithm will *decide* on one of those values. The algorithm guarantees that the decided value is one of the proposed values, that all nodes decide on the same value (with the exception of faulty nodes, which may not decide anything), and that the decision is final (a node will not change its mind once it has decided a value).

It has been formally shown that consensus and total order broadcast are equivalent to each other – that is, an algorithm for one can be turned into an algorithm for the other, and vice versa [Chandra and Toueg, 1996]:

- To turn total order broadcast into consensus, a node that wants to propose a value broadcasts it, and the first message delivered by total order broadcast is taken to be the decided value.

- To turn consensus into total order broadcast, we use a separate instance of the consensus protocol to decide on the first, second, third, . . . message to be delivered. A node that wants to broadcast a message proposes it for one of these rounds of consensus. The consensus algorithm then ensures that all nodes agree on the sequence of messages to be delivered.

Consensus and total order broadcast

- ▶ Traditional formulation of consensus: several nodes want to come to **agreement** about a single **value**
- ▶ In context of total order broadcast: this value is the **next message to deliver**
- ▶ Once one node **decides** on a certain message order, all nodes will decide the same order
- ▶ Consensus and total order broadcast are formally equivalent

Common consensus algorithms:

- ▶ **Paxos**: single-value consensus
  Multi-Paxos: generalisation to total order broadcast
- ▶ **Raft**, **Viewstamped Replication**, **Zab**:
  FIFO-total order broadcast by default

Slide 106

The two best-known consensus algorithms are *Paxos* [Lamport, 1998] and *Raft* [Ongaro and Ouster-hout, 2014]. In its original formulation, Paxos provides only consensus on a single value, and the *Multi-Paxos* algorithm is a generalisation of Paxos that provides FIFO-total order broadcast. On the other hand, Raft is designed to provide FIFO-total order broadcast "out of the box".

Slide 107

The design of a consensus algorithm depends crucially on the system model, as discussed in Section 2.3. Paxos and Raft assume a system model with fair-loss links (Slide 33), crash-recovery behaviour of nodes (Slide 34), and partial synchrony (Slide 35).

The assumptions on network and node behaviour can be weakened to Byzantine, and such algorithms are used in blockchains. However, Byzantine fault-tolerant consensus algorithms are significantly more complicated and less efficient than non-Byzantine ones. We will focus on fair-loss, crash-recovery algorithms for now, which are useful in many practical settings (such as datacenters with trusted private networks). The L47 unit in Part III goes into detail of Byzantine consensus.

On the other hand, the assumption of partial synchrony cannot be weakened to asynchrony. The reason is that consensus requires a failure detector (Slide 40), which in turn requires a local clock to trigger timeouts [Chandra and Toueg, 1996]. If we did not have any clocks, then a deterministic consensus algorithm might never terminate. Indeed, it has been proved that no deterministic, asynchronous algorithm can solve the consensus problem with guaranteed termination. This fact is known as the *FLP result*, one of the most important theorems of distributed computing, named after its three authors Fischer, Lynch, and Paterson [Fischer et al., 1985].

It is possible to get around the FLP result by using a nondeterministic (randomised) algorithm. However, most practical systems instead avoid non-termination by using clocks for timeouts. Recall, however, that in a partially synchronous system, we cannot assume bounded network latency or bounded execution speed of nodes. For this reason, consensus algorithms need to guarantee their *safety properties* (namely, that each node decides on the same messages in the same order) regardless of the timing in the system, even if messages are arbitrarily delayed. Only the *liveness* (namely, that a message is eventually delivered) depends on clocks and timing.

Slide 108

At the core of most consensus algorithms is a process for electing a new leader when the existing leader

56

becomes unavailable for whatever reason. The details differ between the algorithms; in this lecture we will concentrate on the approach taken by Raft, but many of the lessons from Raft are equally relevant to other consensus algorithms. Howard and Mortier [2020] give a detailed comparison of Raft and Multi-Paxos (however, Paxos/Multi-Paxos are not examinable).

A leader election is initiated when the other nodes suspect the current leader to have failed, typically because they haven't received any message from the leader for some time. One of the other nodes becomes a *candidate* and asks the other nodes to vote on whether they accept the candidate as their new leader. If a *quorum* (Section 5.2) of nodes vote in favour of the candidate, it becomes the new leader. If a majority quorum is used, this vote can succeed as long as a majority of nodes (2 out of 3, or 3 out of 5, etc.) are working and able to communicate.
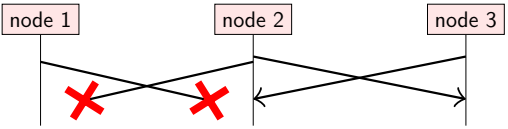
If there were multiple leaders, they could make inconsistent decisions that lead to violations of the safety properties of total order broadcast (a situation known as *split brain*). Therefore, the key thing we want of a leader election is that there should only be one leader at any one time. In Raft, the concept of "at any one time" is captured by having a *term* number, which is just an integer that is incremented every time a leader election is started. If a leader is elected, the voting algorithm guarantees that that it is the only leader within that particular term. Different terms may have different leaders.

Slide 109

However, recall from Slide 41 that in a partially synchronous system, a timeout-based failure detector may be inaccurate: it may suspect a node has having crashed when in fact the node is functioning fine, for example due to a spike in network latency. For example, on Slide 109, node 1 is the leader in term t, but the network between it and nodes 2 and 3 is temporarily interrupted. Nodes 2 and 3 may detect node 1 as having failed, and elect a new leader in term $t + 1$, even though node 1 is still functioning correctly. Moreover, node 1 might not even have noticed the network problem, and it doesn't yet know about the new leader either. Thus, we end up with two nodes both believing to be the leader.

Slide 110

For this reason, even after a node has been elected leader, it must act carefully, since at any moment the system might contain be another leader with a later term that it has not yet heard about. It is not safe for a leader to act unilaterally. Instead, every time a leader wants to decide on the next message to deliver, it must again request confirmation from a quorum of nodes. This is illustrated on Slide 110:

1. In the first round-trip, the left node is elected leader thanks to the votes of the other two nodes.

2. In the second round-trip, the leader proposes the next message to deliver, and the followers acknowledge that they do not know of any leader with a later term than t.

3. Finally, the leader actually delivers m and broadcasts this fact to the followers, so that they can do the same.

If another leader has been elected, the old leader will find out from at least one of the acknowledgements in the second round-trip, because at least one of the nodes in the second-round quorum must have also voted for the new leader. Therefore, even though multiple leaders may exist at the same time, the old leaders will no longer be able to decide on any further messages to deliver, making the algorithm safe.

6.2   The Raft consensus algorithm

To make these ideas concrete, we will now walk through the full Raft algorithm for fault-tolerant FIFO-total order broadcast. It is by far the most complex algorithm that we look at in this course, with pseudocode spanning nine slides. (Paxos and other consensus algorithms are similarly complex, if not worse.) There is no need to memorise the whole algorithm for the exam, but it is worth studying carefully in order to understand the underlying principles. For a graphical visualisation of the algorithm, see http://thesecretlivesofdata.com/raft/.

In order to understand the algorithm, it is worth keeping in mind the state machine on Slide 111. A node can be in one of three states: *leader*, *candidate*, or *follower*. When a node first starts running, or when it crashes and recovers, it starts up in the follower state and awaits messages from other nodes. If it receives no messages from a leader or candidate for some period of time, the follower suspects that the leader is unavailable, and it may attempt to become leader itself. The timeout for detecting leader failure is randomised, to reduce the probability of several nodes becoming candidates concurrently and competing to become leader.

When a node suspects the leader to have failed, it transitions to the candidate state, increments the term number, and starts a leader election in that term. During this election, if the node hears from another candidate or leader with a higher term, it moves back into the follower state. But if the election succeeds and it receives votes from a quorum of nodes, the candidate transitions to leader state. If not enough votes are received within some period of time, the election times out, and the candidate restarts the election with a higher term.

Once a node is in the leader state, it remains leader until it is shut down or crashes, or until it receives a message from a leader or candidate with a term higher than its own. Such a higher term could occur if a network partition made the leader and another node unable to communicate for long enough that the other node started an election for a new leader. On hearing about a higher term, the former leader steps down to become a follower.

Start of video section 6.2
(mp4 download)

Node state transitions in Raft

Slide 111

[Slide 112](#) shows the pseudocode for starting up, and for starting an election. The variables defined in the initialisation block constitute the state of a node. Four of the variables (*currentTerm*, *votedFor*, *log*, and *commitLength*) need to be maintained in stable storage (e.g. on disk), since their values must not be lost in the case of a crash. The other variables can be in volatile memory, and the crash-recovery function resets their values. Each node has a unique ID, and we assume there is a global constant, *nodes*, containing the set of IDs of all nodes in the system. This version of the algorithm does not deal with reconfiguration (adding or removing nodes in the system).

The variable *log* contains an array of entries, each of which has the properties msg and term. The msg property of each array entry contains a message that we want to deliver through total order broadcast, and the term property contains the term number in which it was broadcast. The log uses zero-based indexing, so $log[0]$ is the first log entry and $log[log.\text{length}-1]$ is the last. The log grows by appending new entries to the end, and Raft replicates this log across nodes. When a log entry (and all of its predecessors) have been replicated to a quorum of nodes, it is *committed*. At the moment when we commit a log entry, we also deliver its msg to the application. Before a log entry is committed, it may yet change, but Raft guarantees that once a log entry is committed, it is final, and all nodes will commit the same sequence of log entries. Therefore, delivering messages from committed log entries in their log order gives us FIFO-total order broadcast.

When a node suspects a leader failure, it starts a leader election as follows: it increments *currentTerm*, it sets its own role to candidate, and it votes for itself by setting *votedFor* and *votesReceived* to its own node ID. It then sends a VoteRequest message to each other node, asking it to vote on whether this candidate should be the new leader. The message contains the *nodeId* of the candidate, its *currentTerm* (after incrementing), the number of entries in its log, and the term property of its last log entry.

Slide 112

Slide 113

59

Slide 113 shows what happens when a node receives a VoteRequest message from a candidate. A node will only vote for a candidate if the candidate's log is at least as up-to-date as the voter's log; this prevents a candidate with an outdated log from becoming leader, which could lead to the loss of committed log entries. The candidate's log is acceptable if the term of its last log entry is higher than the term of the last log entry on the node that received the VoteRequest message. Moreover, the log is also acceptable if the terms are the same and the candidate's log contains at least as many entries as the recipient's log. This logic is reflected in the *logOk* variable.

Moreover, a node will not vote for a candidate if it has already voted for another candidate in the same term or a later term. The variable *termOk* is true if it is okay to vote for the candidate according to this rule. The *votedFor* variable keeps track of any previous vote by the current node in *currentTerm*.

If both *logOk* and *termOk*, then the node updates its current term to the candidate's term, records its vote in *votedFor*, and sends a VoteResponse message containing true (indicating success) to the candidate. Otherwise, the node sends a VoteResponse message containing false (indicating a refusal to vote for the candidate). Besides the flag for success or failure, the response message contains the *nodeId* of the node sending the vote, and the term of the vote.

```
Raft (3/9): collecting votes

  on receiving (VoteResponse, voterId, term, granted) at nodeId do
      if currentRole = candidate ∧ term = currentTerm ∧ granted then
          votesReceived := votesReceived ∪ {voterId}
          if |votesReceived| ≥ ⌈(|nodes| + 1)/2⌉ then
              currentRole := leader; currentLeader := nodeId
              cancel election timer
              for each follower ∈ nodes \ {nodeId} do
                  sentLength[follower] := log.length
                  ackedLength[follower] := 0
                  REPLICATELOG(nodeId, follower)
              end for
          end if
      else if term > currentTerm then
          currentTerm := term
          currentRole := follower
          votedFor := null
          cancel election timer
      end if
  end on
```

Slide 114

Back on the candidate, Slide 114 shows the code for processing the VoteResponse messages. We ignore any responses relating to earlier terms (which could arrive late due to network delays). If the term in the response is higher than the candidate's term, the candidate cancels the election and transitions back into the follower state. But if the term is correct and the success flag *granted* is set to true, the candidate adds the node ID of the voter to the set of votes received.

If the set of votes constitutes a quorum, the candidate transitions into the leader state. As its first action as a leader, it updates the *sentLength* and *ackedLength* variables (explained below), and then calls the REPLICATELOG function (defined on Slide 116) for each follower. This has the effect of sending a message to each follower, informing them about the new leader.

On the leader, *sentLength* and *ackedLength* are variables that map each node ID to an integer (non-leader nodes do not use these variables). For each follower F, *sentLength*$[F]$ tracks how many log entries, counting from the beginning of the log, have been sent to F, and *ackedLength*$[F]$ tracks how many log entries have been acknowledged as received by F. On becoming a leader, a node initialises *sentLength*$[F]$ to *log*.length (i.e. it assumes that the follower has already been sent the whole log), and initialises *ackedLength*$[F]$ to 0 (i.e. nothing has been acknowledged yet). These assumptions might be wrong: for example, the follower might be missing some of the log entries that are present on the leader. In this case, *sentLength*$[F]$ will be corrected through a process that we discuss on Slide 119.

Slide 115 shows how a new entry is added to the log when the application wishes to broadcast a message through total order broadcast. A leader can simply go ahead and append a new entry to its log, while any other node needs to ask the leader to do this on its behalf, via a FIFO link (to ensure FIFO-total order broadcast). The leader then updates its own entry in *ackedLength* to *log*.length, indicating that it has acknowledged its own addition to the log, and calls REPLICATELOG for each other node.

Moreover, a leader also periodically calls REPLICATELOG for each other node, even if there is no new message to broadcast. This serves multiple purposes: it lets the followers know that the leader is still alive; it serves as retransmission of any messages from leader to follower that might have been lost; and

it updates the follower about which messages can be committed, as explained below.

Raft (4/9): broadcasting messages

```
on request to broadcast msg at node nodeId do
    if currentRole = leader then
        append the record (msg : msg, term : currentTerm) to log
        ackedLength[nodeId] := log.length
        for each follower ∈ nodes \ {nodeId} do
            REPLICATELOG(nodeId, follower)
        end for
    else
        forward the request to currentLeader via a FIFO link
    end if
end on

periodically at node nodeId do
    if currentRole = leader then
        for each follower ∈ nodes \ {nodeId} do
            REPLICATELOG(nodeId, follower)
        end for
    end if
end do
```

Slide 115

Raft (5/9): replicating from leader to followers

Called on the leader whenever there is a new message in the log, and also periodically. If there are no new messages, *entries* is the empty list. LogRequest messages with $entries = \langle\rangle$ serve as heartbeats, letting followers know that the leader is still alive.

```
function REPLICATELOG(leaderId, followerId)
    i := sentLength[followerId]
    entries := ⟨log[i], log[i + 1], ..., log[log.length − 1]⟩
    prevLogTerm := 0
    if i > 0 then
        prevLogTerm := log[i − 1].term
    end if
    send (LogRequest, leaderId, currentTerm, i, prevLogTerm,
          commitLength, entries) to followerId
end function
```

Slide 116

The REPLICATELOG function is shown on Slide 116. Its purpose is to send any new log entries from the leader to the follower node with ID *followerId*. It first sets the variable *entries* to the suffix of the log starting with index *sentLength*[*followerId*], if it exists. That is, if *sentLength*[*followerId*] is the number of log entries already sent to *followerId*, then *entries* contains the remaining entries that have not yet been sent. If *sentLength*[*followerId*] = *log*.length, the variable *entries* is set to the empty array.

REPLICATELOG then sends a LogRequest message to *followerId* containing *entries* as well as several other values: the ID of the leader; its current term; the length of the log prefix that precedes *entries*; the term of the last log entry preceding *entries*; and *commitLength*, which is the number of log entries that have been committed, as counted from the start of the log. More on committing log entries shortly.

When a follower receives one of these LogRequest messages from the leader, it processes the message as shown on Slide 117. First, if the message is for a later term than the follower has previously seen, it updates its current term and accepts the sender of the message as leader.

Next, the follower checks if its log is consistent with that of the leader. *logLength* is the number of log entries that precede the new *entries* contained in the LogRequest message. The follower requires that its log is at least as long as *logLength* (i.e. it is not missing any entries), and that the term of the last log entry in the *logLength* prefix of the follower's log is the same as the term of the same log entry on the leader. If both of these checks pass, the *logOk* variable is set to true.

If the LogRequest message is for the expected term and if *logOk*, then the follower accepts the message and calls the APPENDENTRIES function (defined on Slide 118) to add *entries* to its own log. It then replies to the leader with a LogResponse message containing the follower's ID, the current term, an acknowledgement of the number of log entries received, and the value true indicating that the LogRequest

was successful. If the message is from an outdated term or *logOk* is false, the follower replies with a
LogResponse containing false to indicate an error.

Raft (6/9): followers receiving messages

on receiving (LogRequest, *leaderId*, *term*, *logLength*, *logTerm*,
            leaderCommit, *entries*) at node *nodeId* **do**
  if $term > currentTerm$ **then**
    $currentTerm := term;$ $votedFor :=$ null
    $currentRole :=$ follower; $currentLeader := leaderId$
  end if
  if $term = currentTerm \land currentRole =$ candidate **then**
    $currentRole :=$ follower; $currentLeader := leaderId$
  end if
  $logOk := (log.\text{length} \geq logLength) \land$
        $(logLength = 0 \lor logTerm = log[logLength - 1].\text{term})$
  if $term = currentTerm \land logOk$ **then**
    AppendEntries(*logLength*, *leaderCommit*, *entries*)
    $ack := logLength + entries.\text{length}$
    send (LogResponse, *nodeId*, *currentTerm*, *ack*, true) to *leaderId*
  else
    send (LogResponse, *nodeId*, *currentTerm*, 0, false) to *leaderId*
  end if
end on

Slide 117

Raft (7/9): updating followers' logs

function AppendEntries(*logLength*, *leaderCommit*, *entries*)
  if $entries.\text{length} > 0 \land log.\text{length} > logLength$ **then**
    if $log[logLength].\text{term} \neq entries[0].\text{term}$ **then**
      $log := \langle log[0], log[1], \ldots, log[logLength - 1] \rangle$
    end if
  end if
  if $logLength + entries.\text{length} > log.\text{length}$ **then**
    for $i := log.\text{length} - logLength$ **to** $entries.\text{length} - 1$ **do**
      append $entries[i]$ to log
    end for
  end if
  if $leaderCommit > commitLength$ **then**
    for $i := commitLength$ **to** $leaderCommit - 1$ **do**
      deliver $log[i].\text{msg}$ to the application
    end for
    $commitLength := leaderCommit$
  end if
end function

Slide 118

Slide 118 shows the AppendEntries function, which a follower calls to extend its log with entries
received from the leader. *logLength* is the number of log entries (counted from the beginning of the log)
that precede the new *entries*. If the follower's log already contains entries at *log*[*logLength*] and beyond,
it compares the term of that existing entry with the term of the first new entry received from the leader.
If they are inconsistent, the follower discards those existing entries by truncating the log. This happens
if the existing log entries came from a previous leader, which has now been superseded by a new leader.

Next, any new *entries* that are not already present in the follower's log are appended to the log. In
the case of message duplication, this operation is idempotent, since the new entries are understood to
begin at index *logLength* in the log.

Finally, the follower checks whether the integer *leaderCommit* in the LogRequest message is greater
than its local variable *commitLength*. If so, this means that new records are ready to be committed and
delivered to the application. The leader moves its *commitLength* forward and performs the total order
broadcast delivery of the messages in the appropriate log entries.

This completes the algorithm from the followers' point of view. What remains is to switch back to
the leader, and to show how it processes the LogResponse messages from followers (see Slide 119).

Raft (8/9): leader receiving log acknowledgements

```
on receiving (LogResponse, follower, term, ack, success) at nodeId do
    if term = currentTerm ∧ currentRole = leader then
        if success = true ∧ ack ≥ ackedLength[follower] then
            sentLength[follower] := ack
            ackedLength[follower] := ack
            CommitLogEntries()
        else if sentLength[follower] > 0 then
            sentLength[follower] := sentLength[follower] − 1
            ReplicateLog(nodeId, follower)
        end if
    else if term > currentTerm then
        currentTerm := term
        currentRole := follower
        votedFor := null
    end if
end on
```

A leader receiving a LogResponse message first checks the term in the message: if the sender's term is later than the recipient's term, that means a new leader election has been started, and so this node transitions from leader to follower. Messages with an outdated term are ignored. For messages with the correct term, we check the *success* boolean field to see whether the follower accepted the log entries.

If *success* = true, the leader updates *sentLength* and *ackedLength* to reflect the number of log entries acknowledged by the follower, and then calls the CommitLogEntries function (Slide 120). If *success* = false, we know that the follower did not accept the log entries because its *logOk* variable was false. In this case, the leader decrements the *sentLength* value for this follower, and calls ReplicateLog to retry sending the LogRequest message starting with an earlier log entry. This may happen multiple times, but eventually the leader will send the follower an array of entries that cleanly extends the follower's existing log, at which point the follower will accept the LogRequest. (This algorithm could be optimised to require fewer retries, but in this course we will avoid making it more complex than needed.)

Raft (9/9): leader committing log entries

Any log entries that have been acknowledged by a quorum of nodes are ready to be committed by the leader. When a log entry is committed, its message is delivered to the application.

```
define acks(length) = |{n ∈ nodes | ackedLength[n] ≥ length}|

function CommitLogEntries
    minAcks := ⌈(|nodes| + 1)/2⌉
    ready := {len ∈ {1, ..., log.length} | acks(len) ≥ minAcks}
    if ready ≠ {} ∧ max(ready) > commitLength ∧
            log[max(ready) − 1].term = currentTerm then
        for i := commitLength to max(ready) − 1 do
            deliver log[i].msg to the application
        end for
        commitLength := max(ready)
    end if
end function
```

Finally, Slide 120 shows how the leader determines which log entries to commit. We define the function acks(*length*) to take an integer, a number of log entries counted from the start of the log. This function returns the number of nodes that have acknowledged the receipt of *length* log entries or more.

CommitLogEntries uses this function to determine how many log entries have been acknowledged by a majority quorum of nodes or more. The variable *ready* contains the set of log prefix lengths that are ready to commit, and if *ready* is nonempty, max(*ready*) is the maximum log prefix length that we can commit. If this exceeds the current value of *commitLength*, this means there are new log entries that are now ready to commit because they have been acknowledged by sufficiently many nodes. The message in each of these log entries is now delivered to the application on the leader, and the *commitLength* variable is updated. On the next LogRequest message that the leader sends to followers, the new value of *commitLength* will be included, causing the followers to commit and deliver the same log entries.

Exercise 16. *Three nodes are executing the Raft algorithm. At one point in time, each node has the log shown below:*

log at node A: $\boxed{\begin{array}{c}m_1\\1\end{array}}\boxed{\begin{array}{c}m_2\\1\end{array}}$ *log* at node B: $\boxed{\begin{array}{c}m_1\\1\end{array}}\boxed{\begin{array}{c}m_4\\2\end{array}}\boxed{\begin{array}{c}m_5\\2\end{array}}\boxed{\begin{array}{c}m_6\\2\end{array}}$ *log* at node C: $\boxed{\begin{array}{c}m_1\\1\end{array}}\boxed{\begin{array}{c}m_4\\2\end{array}}\boxed{\begin{array}{c}m_7\\3\end{array}}$ ← msg
← term
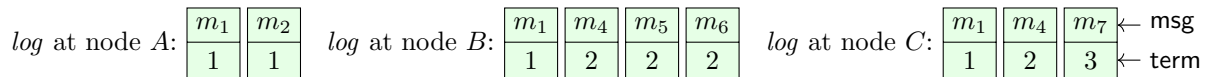
(a) Explain what events may have occurred that caused the nodes to be in this state.
(b) What are the possible values of the commitLength *variable at each node?*
(c) Node A starts a leader election in term 4, while the nodes are in the state above. Is it possible for it to obtain a quorum of votes? What if the election was instead started by one of the other nodes?
(d) Assume that node B is elected leader in term 4, while the nodes are in the state above. Give the sequence of messages exchanged between B and C following this election.

7   Replica consistency

We have seen how to perform replication using read/write quorums, and state machine replication using total order broadcast. In this context we have said that we want replicas to have "consistent copies of the same data", without defining exactly what we mean with *consistent*.

Unfortunately the word "consistency" means different things in different contexts. In the context of transactions, the C in ACID stands for consistency that is a property of a state: that is, we can say that a database is in a consistent or inconsistent state, meaning that the state satisfies or violates certain invariants defined by the application. On the other hand, in the context of replication, we have used "consistency" informally to refer to a relationship between replicas: we want one replica to be consistent with another replica.

Since there is no one true definition of consistency, we speak instead about a variety of *consistency models*. We have seen one particular example of a consistency model, namely read-after-write consistency (Slide 97), which restricts the values that a read operation may return when the same node previously writes to the same data item. We will see more models in this lecture.

<table>
<tr><td>

"Consistency"

A word that means many different things in different contexts!

▶ **ACID**: a transaction transforms the database from one "consistent" state to another

Here, "consistent" = satisfying application-specific invariants

e.g. "every course with students enrolled must have at least one lecturer"

▶ **Read-after-write consistency** (lecture 5)

▶ **Replication**: replica should be "consistent" with other replicas

"consistent" = in the same state? (when exactly?)

"consistent" = read operations return same result?

▶ **Consistency model**: many to choose from

</td><td>

Start of video section 7.1
(mp4 download)

Slide 121

</td></tr>
</table>

7.1   Two-phase commit

Let's start with a consistency problem that arises when executing a *distributed transaction*, i.e. a transaction that reads or writes data on multiple nodes. The data on those nodes may be replicas of the same dataset, or different parts of a larger dataset; a distributed transaction applies in both cases.

Recall from the concurrent systems half of this course that a key property of a transaction is *atomicity*. When a transaction spans multiple nodes, we still want atomicity for the transaction as a whole: that is, either all nodes must commit the transaction and make its updates durable, or all nodes must abort the transaction and discard or roll back its updates.

We thus need *agreement* among the nodes on whether the transaction should abort or commit. You might be wondering: is this agreement the same as *consensus*, which we discussed in Lecture 6? The answer is no: although both are superficially about reaching agreement, the details differ significantly.

Distributed transactions

Recall **atomicity** in the context of ACID transactions:

- ▶ A transaction either **commits** or **aborts**
- ▶ If it commits, its updates are durable
- ▶ If it aborts, it has no visible side-effects
- ▶ ACID consistency (preserving invariants) relies on atomicity

If the transaction updates data on multiple nodes, this implies:

- ▶ Either all nodes must commit, or all must abort
- ▶ If any node crashes, all must abort

Ensuring this is the **atomic commitment** problem.
Looks a bit similar to consensus?

Slide 122

Atomic commit versus consensus

Consensus	Atomic commit
One or more nodes propose a value	Every node votes whether to commit or abort
Any one of the proposed values is decided	Must commit if all nodes vote to commit; must abort if ≥ 1 nodes vote to abort
Crashed nodes can be tolerated, as long as a quorum is working	Must abort if a participating node crashes

Slide 123

The most common algorithm to ensure atomic commitment across multiple nodes is the *two-phase commit* (2PC) protocol [Gray, 1978]. (Not to be confused with *two-phase locking* (2PL), discussed in the first half of this course: 2PL ensures serializable isolation, while 2PC ensures atomic commitment. There is also a *three-phase commit* protocol, but it assumes the unrealistic synchronous system model, so we won't discuss it here.) The communication flow of 2PC is illustrated on Slide 124.

Two-phase commit (2PC)

Slide 124

65
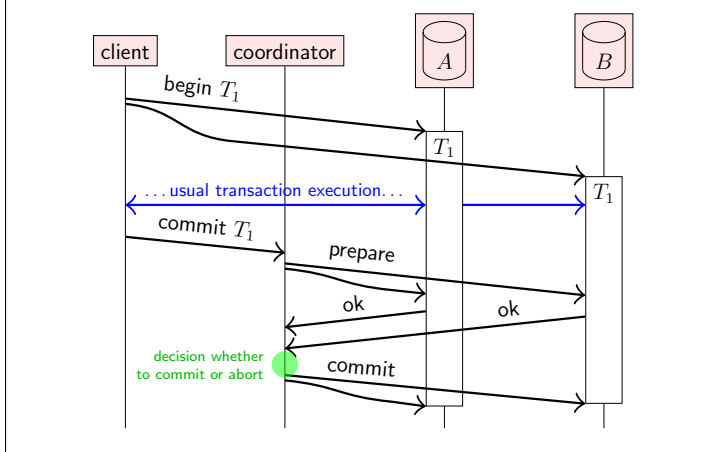
When using two-phase commit, a client first starts a regular single-node transaction on each replica that is participating in the transaction, and performs the usual reads and writes within those transactions. When the client is ready to commit the transaction, it sends a commit request to the *transaction coordinator*, a designated node that manages the 2PC protocol. (In some systems, the coordinator is part of the client.)

The coordinator first sends a *prepare* message to each replica participating in the transaction, and each replica replies with a message indicating whether it is able to commit the transaction (this is the first phase of the protocol). The replicas do not actually commit the transaction yet, but they must ensure that they will definitely be able to commit the transaction in the second phase if instructed by the coordinator. This means, in particular, that the replica must write all of the transaction's updates to disk and check any integrity constraints before replying *ok* to the *prepare* message, while continuing to hold any locks for the transaction.

The coordinator collects the responses, and decides whether or not to actually commit the transaction. If all nodes reply *ok*, the coordinator decides to commit; if any node wants to abort, or if any node fails to reply within some timeout, the coordinator decides to abort. The coordinator then sends its decision to each of the replicas, who all commit or abort as instructed (this is the second phase). If the decision was to commit, each replica is guaranteed to be able to commit its transaction because the previous *prepare* request laid the groundwork. If the decision was to abort, the replica rolls back the transaction.

<div style="border:1px solid #000; padding:1em;">

The coordinator in two-phase commit

What if the coordinator crashes?
- ▶ Coordinator writes its decision to disk
- ▶ When it recovers, read decision from disk and send it to replicas (or abort if no decision was made before crash)
- ▶ **Problem:** if coordinator crashes after prepare, but before broadcasting decision, other nodes do not know how it has decided
- ▶ Replicas participating in transaction cannot commit or abort after responding "ok" to the *prepare* request (otherwise we risk violating atomicity)
- ▶ Algorithm is blocked until coordinator recovers

</div>

Slide 125

The problem with two-phase commit is that the coordinator is a single point of failure. Crashes of the coordinator can be tolerated by having the coordinator write its commit/abort decisions to stable storage, but even so, there may be transactions that have prepared but not yet committed/aborted at the time of the coordinator crash (called *in-doubt transactions*). Any in-doubt transactions must wait until the coordinator recovers to learn their fate; they cannot unilaterally decide to commit or abort, because that decision could end up being inconsistent with the coordinator and other nodes, which might violate atomicity.

Fortunately it is possible to avoid the single point of failure of the coordinator by using a consensus algorithm or total order broadcast protocol. Slide 126 shows a fault-tolerant two-phase commit algorithm based on Paxos Commit [Gray and Lamport, 2006]. The idea is that every node that is participating in the transaction uses total order broadcast to disseminate its vote on whether to commit or abort. Moreover, if node A suspects that node B has failed (because no vote from B was received within some timeout), then A may try to vote to abort on behalf of B. This introduces a race condition: if node B is slow, it might be that node B broadcasts its own vote to commit around the same time that node A suspects B to have failed and votes on B's behalf.

These votes are delivered to each node by total order broadcast, and each recipient independently counts the votes. In doing so, we count only the first vote from any given replica, and ignore any subsequent votes from the same replica. Since total order broadcast guarantees the same delivery order on each node, all nodes will agree on whether the first delivered vote from a given replica was a commit vote or an abort vote, even in the case of a race condition between multiple nodes broadcasting contradictory votes for the same replica.

If a node observes that the first delivered vote from some replica is a vote to abort, then the transaction

can immediately be aborted. Otherwise a node must wait until it has delivered at least one vote from each replica. Once these votes have been delivered, and none of the replicas vote to abort in their first delivered message, then the transaction can be committed. Thanks to total order broadcast, all nodes are guaranteed to make the same decision on whether to abort or to commit, which preserves atomicity.

Fault-tolerant two-phase commit (1/2)

on initialisation for transaction T **do**
  $commitVotes[T] := \{\}$; $replicas[T] := \{\}$; $decided[T] :=$ false
end on

on request to commit transaction T with participating nodes R **do**
  for each $r \in R$ **do** send $(\text{Prepare}, T, R)$ to r
end on

on receiving $(\text{Prepare}, T, R)$ at node $replicaId$ **do**
  $replicas[T] := R$
  $ok =$ "is transaction T able to commit on this replica?"
  total order broadcast $(\text{Vote}, T, replicaId, ok)$ to $replicas[T]$
end on

on a node suspects node $replicaId$ to have crashed **do**
  for each transaction T in which $replicaId$ participated **do**
    total order broadcast $(\text{Vote}, T, replicaId, \text{false})$ to $replicas[T]$
  end for
end on

Slide 126

Fault-tolerant two-phase commit (2/2)

on delivering $(\text{Vote}, T, replicaId, ok)$ by total order broadcast **do**
  if $replicaId \notin commitVotes[T] \;\wedge\; replicaId \in replicas[T] \;\wedge$
        $\neg decided[T]$ **then**
    if $ok =$ true **then**
      $commitVotes[T] := commitVotes[T] \cup \{replicaId\}$
      if $commitVotes[T] = replicas[T]$ **then**
        $decided[T] :=$ true
        commit transaction T at this node
      end if
    else
      $decided[T] :=$ true
      abort transaction T at this node
    end if
  end if
end on

Slide 127

7.2   Linearizability

An atomic commitment protocol is a way of preserving consistency across multiple replicas in the face of faults, by ensuring that all participants of a transaction either commit or abort. However, when there are multiple nodes concurrently reading and modifying some shared data concurrently, ensuring the same commit or abort outcome for all nodes is not sufficient. We also have to reason about the interaction that arises from concurrent activity.

In this section we will introduce one particular consistency model for concurrent system that is called *linearizability*. We will discuss linearizability informally; if you are interested in the details, Herlihy and Wing [1990] give a formal definition. People sometimes say *strong consistency* when referring to linearizability, but the concept of "strong consistency" is rather vague and imprecise. We will stick to the term *linearizability*, which has a precisely defined meaning.

An informal definition of linearizability appears on Slide 128. Over the following slides we will clarify what this means through examples.

Linearizability is a useful concept not only in distributed systems, but also in the context of shared-memory concurrency on a single machine. Interestingly, on a computer with multiple CPU cores (pretty much all servers, laptops and smartphones nowadays), memory access is not linearizable by default! This

is because each CPU core has its own caches, and an update made by one core is not immediately reflected in another core's cache. Thus, even a single computer starts behaving a bit like a replicated system. The L304 unit in Part III goes into detail of multicore memory behaviour.

Don't confuse linearizability with serializability, even though both words seem to mean something like "can be arranged into a sequential order". Serializability means that transactions have the same effect as if they had been executed in *some* serial order, but it does not define what that order should be. Linearizability defines the values that operations must return, depending on the concurrency and relative ordering of those operations. It is possible for a system to provide both serializability and linearizability: the combination of the two is called *strict serializability* or *one-copy serializability*.

Linearizability

Multiple nodes concurrently accessing replicated data.
How do we define "consistency" here?

The strongest option: **linearizability**

- ▶ Informally: every operation takes effect **atomically** sometime after it started and before it finished
- ▶ All operations behave as if executed on a **single copy** of the data (even if there are in fact multiple replicas)
- ▶ Consequence: every operation returns an "up-to-date" value, a.k.a. "strong consistency"
- ▶ Not just in distributed systems, also in shared-memory concurrency (memory on multi-core CPUs is not linearizable by default!)

Note: linearizability $\neq$ serializability!

Slide 128

The main purpose of linearizability is to guarantee that nodes observe the system in an "up-to-date" state; that is, they do not read *stale* (outdated) values. We have previously seen this concept of reading an "up-to-date" value in the context of read-after-write consistency (Slide 97). However, while read-after-write consistency defines only a consistency model for reads and writes made by the same node, linearizability generalises this idea to operations made concurrently by different nodes.

From the point of view of a client, every operation takes some amount of time. We say that an operation *starts* at the moment when it is requested by the application, and it *finishes* when the operation result is returned to the application. Between the start and finish, various network communication steps may happen; for example, if quorums are used, an operation can finish when the client has received responses from a quorum of replicas.

On Slide 129 and the following slides we represent the client's view of a get/set operation as a rectangle covering the period of time from the start to finish of an operation. Inside the rectangle we write the effect of the operation: $\mathsf{set}(x, v)$ means updating the data item x to have the value v, and $\mathsf{get}(x) \to v$ means a read of x that returns the value v.

Read-after-write consistency revisited

Slide 129

68
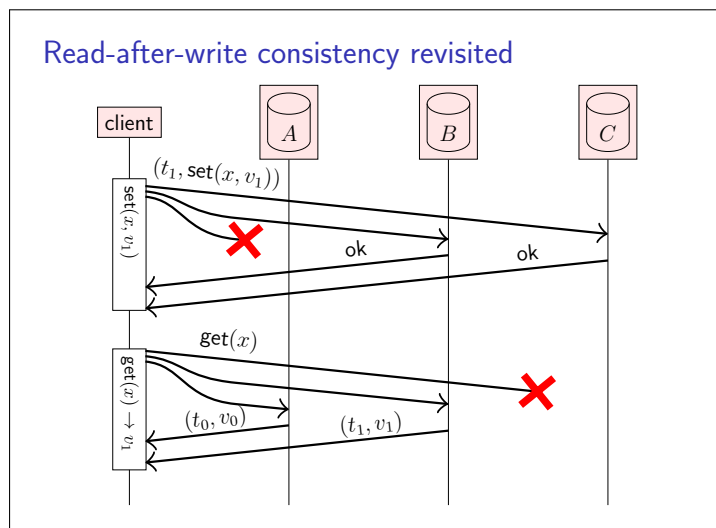
Linearizability is independent of the system implementation and communication protocols: all that matters is the timing of each operation's start and finish, and the outcome of the operation. We can therefore leave out all of the replicas and message-sending arrows, and look at the system's behaviour only from the client's point of view.

The key thing that linearizability cares about is whether one operation finished before another operation started, regardless of the nodes on which they took place. On Slide 130, the two get operations both start after the set operation has finished, and therefore we expect the get operations to return the value v_1 written by set.

On the other hand, on Slide 131, the get and set operation overlap in time: in this case we don't necessarily know in which order the operations take effect. get may return either the value v_1 written by set, or x's previous value v_0, and either result is acceptable.

Note that "operation A finished before operation B started" is not the same as "A happened before B". The happens-before relation (Section 3.3) is defined in terms of messages sent and received; it is possible to have two operations that do not overlap in time, but are still concurrent according to the happens-before relation, because no communication has occurred between those operations. On the other hand, linearizability is defined in terms of *real time*: that is, a hypothetical global observer who can instantaneously see the state of all nodes (or, a perfectly synchronised clock on each node) determines the start and finish times of each operation. In reality, such a global observer or perfectly synchronised clock does not exist in a system with variable network latency, but we can nevertheless define linearizability in terms of such a hypothetical observer. This has the advantage that if we prove a system to be linearizable, we can be sure that its consistency guarantees hold regardless of whether some communication has taken place or not.

Slide 130

Slide 131

69

Linearizability is not only about the relationship of a `get` operation to a prior `set` operation, but it can also relate one `get` operation to another. Slide 132 shows an example of a system that uses quorum reads and writes, but is nevertheless non-linearizable. Here, client 1 sets x to v_1, and due to a quirk of the network the update to replica A happens quickly, while the updates to replicas B and C are delayed. Client 2 reads from a quorum of $\{A, B\}$, receives responses $\{v_0, v_1\}$, and determines v_1 to be the newer value based on the attached timestamp. After client 2's read has finished, client 3 starts a read from a quorum of $\{B, C\}$, receives v_0 from both replicas, and returns v_0 (since it is not aware of v_1).

Thus, client 3 observes an older value than client 2, even though the real-time order of operations would require client 3's read to return a value that is no older than client 2's result. This behaviour is not allowed in a linearizable system.
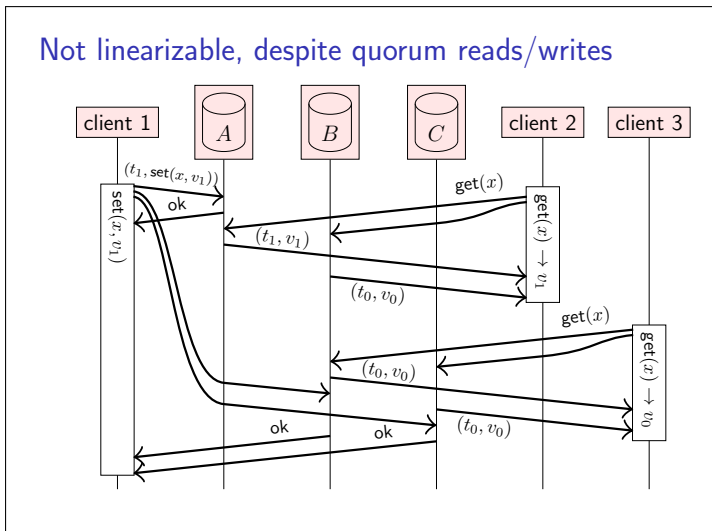
Slide 132
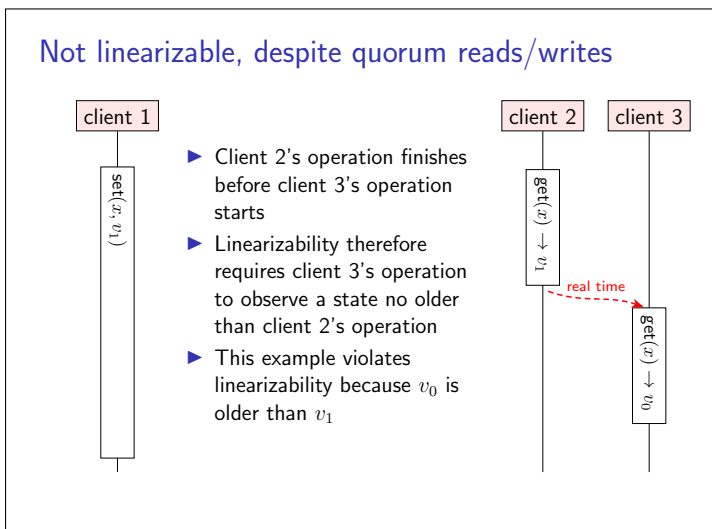
Slide 133

Fortunately, it is possible to make `get` and `set` operations linearizable using quorum reads and writes. `set` operations don't change: as before, they send the update to all replicas, and wait for acknowledgement from a quorum of replicas.

For `get` operations, another step is required, as shown on Slide 134. A client must first send the `get` request to replicas, and wait for responses from a quorum. If some responses include a more recent value than other responses, as indicated by their timestamps, then the client must write back the most recent value to all replicas that did not already respond with the most recent value, like in read repair (Slide 100). The `get` operation finishes only after the client is sure that the most recent value is stored on a quorum of replicas: that is, after a quorum of replicas either responded `ok` to the read repair, or replied with the most recent value in the first place.

This approach is known as the *ABD algorithm*, after its authors Attiya, Bar-Noy, and Dolev [Attiya et al., 1995]. It ensures linearizable reads and writes, because whenever a `get` and `set` operation finishes,

we know that the value read or written is present on a quorum of replicas, and therefore any subsequent quorum read is guaranteed to observe that value (or a later value).
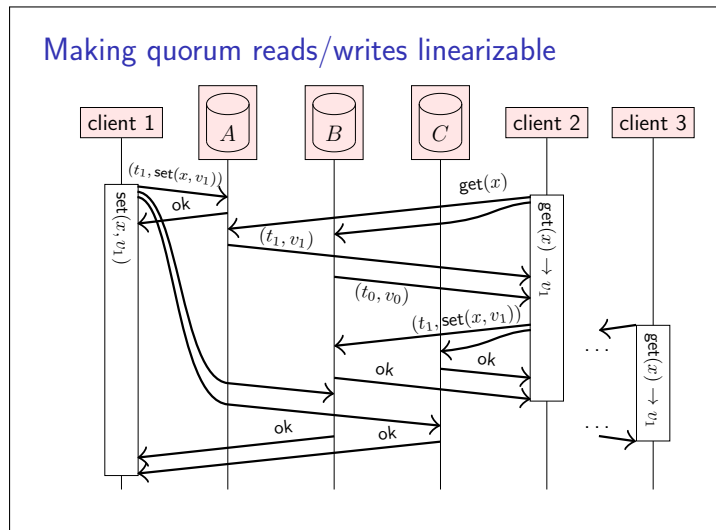
Linearizability for different types of operation

This ensures linearizability of get (quorum read) and set (**blind write** to quorum)

- ▶ When an operation finishes, the value read/written is stored on a quorum of replicas
- ▶ Every subsequent quorum operation will see that value
- ▶ Multiple concurrent writes may overwrite each other

What about an atomic **compare-and-swap** operation?

- ▶ $CAS(x, oldValue, newValue)$ sets x to $newValue$ iff current value of x is $oldValue$
- ▶ Previously discussed in shared memory concurrency
- ▶ Can we implement **linearizable** compare-and-swap in a distributed system?
- ▶ **Yes:** total order broadcast to the rescue again!

The set operation for which the ABD algorithm ensures linearizability is a so-called *blind write* (unconditional write): it simply overwrites the value of a data item, regardless of its previous value. If multiple clients concurrently write to the same item, and if a last-writer-wins conflict resolution policy is used (Slide 95), then one of those writes will end up as the "winner" and the other values will be silently discarded.

In some applications, we want to be more careful and overwrite a value only if it has not been concurrently modified by another node. This can be achieved with an *atomic compare-and-swap* (CAS) operation. A CAS operation for concurrency between threads on a single node was discussed in the first half of this course. This raises the question: how can we implement a linearizable CAS operation in a distributed, replicated system?

Recall that the purpose of linearizability is to make a system behave as if there was only a single copy of the data, and all operations on it happen atomically, even if the system is in fact replicated. This makes CAS a natural operation to want to support in a linearizable context.

The ABD algorithm is not able to implement CAS, because different replicas may see the operations in a different order, and thus reach inconsistent conclusions about whether a particular CAS operation succeeded or not. However, it is possible to implement a linearizable, replicated CAS operation using total order broadcast, as shown on Slide 136. We simply broadcast every operation we want to perform, and actually execute the operation when it is delivered. Like in state machine replication (Slide 101), this algorithm ensures that an operation has the same effect and outcome on every replica.

Slide 136

Exercise 17. *Is the following execution linearizable? If not, where does the violation occur?*
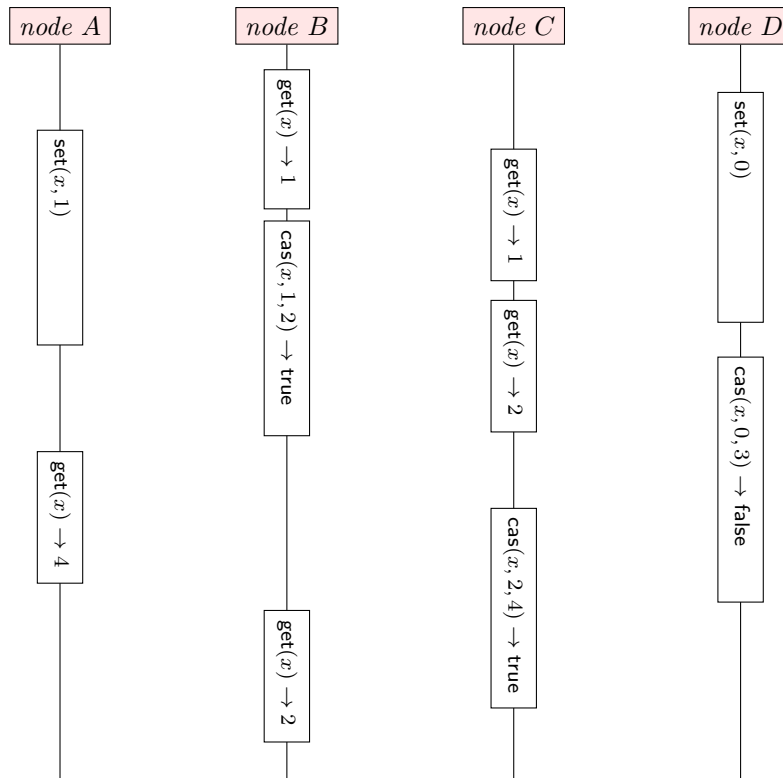
7.3 Eventual consistency

Linearizability is a very convenient consistency model for distributed systems, because it guarantees that a system behaves as if there was only one copy of the data, even if it is in fact replicated. This allows applications to ignore some of the complexities of working with distributed systems. However, this strong guarantee also comes at cost, and therefore linearizability is not suitable for all applications.

Part of the cost is performance: both the ABD algorithm and the linearizable CAS algorithm based on total order broadcast need to send a lot of messages over the network, and require significant amounts of waiting due to network latency. Part is scalability: in algorithms where all updates need to be sequenced through a leader, such as Raft, the leader can become a bottleneck that limits the number of operations that can be processed per second.

Perhaps the biggest problem with linearizability is that every operation requires communication with a quorum of replicas. If a node is temporarily unable to communicate with sufficiently many replicas, it

72

cannot perform any operations. Even though the node may be running, such a communication failure makes it effectively unavailable.

Eventual consistency

Linearizability advantages:
▶ Makes a distributed system behave as if it were non-distributed
▶ Simple for applications to use

Downsides:
▶ **Performance** cost: lots of messages and waiting for responses
▶ **Scalability** limits: leader can be a bottleneck
▶ **Availability** problems: if you can't contact a quorum of nodes, you can't process any operations

Eventual consistency: a weaker model than linearizability. Different trade-off choices.

Slide 137

As an example, consider the calendar app that you can find on most phones, tablets, and computers. We would like the appointments and entries in this app to sync across all of our devices; in other words, we want it to be replicated such that each device is a replica. Moreover, we would like to be able to view, modify, and add calendar events even while a device is offline (e.g. due to poor mobile network coverage). If the calendar app's replication protocol was linearizable, this would not be possible, since an offline device cannot communicate with a quorum of replicas.

Instead, calendar apps allow the user to read and write events in their calendar even while a device is offline, and they sync any updates between devices sometime later, in the background, when an internet connection is available. The video of this lecture includes a demonstration of offline updates to a calendar.
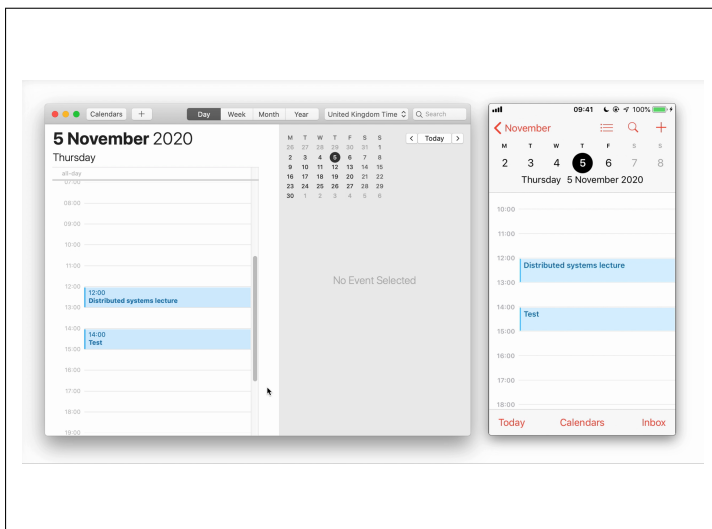
Slide 138

This trade-off is known as the *CAP theorem* (named after *consistency*, *availability*, and *partition tolerance*), which states that if there is a network partition in a system, we must choose between one of the following options [Gilbert and Lynch, 2002]:

1. We can have linearizable consistency, but in this case, some replicas will not be able to respond to requests because they cannot communicate with a quorum. Not being able to respond to requests makes those nodes effectively unavailable.

2. We can allow replicas to respond to requests even if they cannot communicate with other replicas. In this case, they continue to be available, but we cannot guarantee linearizability.

Sometimes the CAP theorem is formulated as a choice of "pick 2 out of 3", but that framing is misleading.

A system can be both linearizable and available as long as there is no network partition, and the choice is forced only in the presence of a partition [Kleppmann, 2015].

This trade-off is illustrated on Slide 139, where node C is unable to communicate with nodes A and B. On A and B's side of the partition, linearizable operations can continue as normal, because A and B constitute a quorum. However, if C wants to read the value of x, it must either wait (potentially indefinitely) until the network partition is repaired, or it must return its local value of x, which does not reflect the value previously written by A on the other side of the partition.
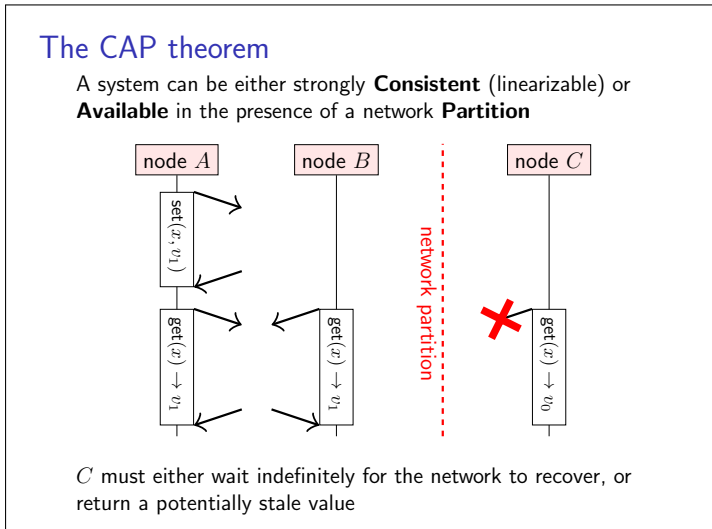
Slide 139

The calendar app chooses option 2: it forgoes linearizability in favour of allowing the user to continue performing operations while a device is offline. Many other systems similarly make this choice for various reasons.

The approach of allowing each replica to process both reads and writes based only on its local state, and without waiting for communication with other replicas, is called *optimistic replication*. A variety of consistency models have been proposed for optimistically replicated systems, with the best-known being *eventual consistency*.

Eventual consistency is defined as: "if no new updates are made to an object, eventually all reads will return the last updated value" [Vogels, 2009]. This is a very weak definition: what if the updates to an object never stop, so the premise of the statement is never true? A slightly stronger consistency model called *strong eventual consistency*, defined on Slide 140, is often more appropriate [Shapiro et al., 2011]. It is based on the idea that as two replicas communicate, they *converge* towards the same state.
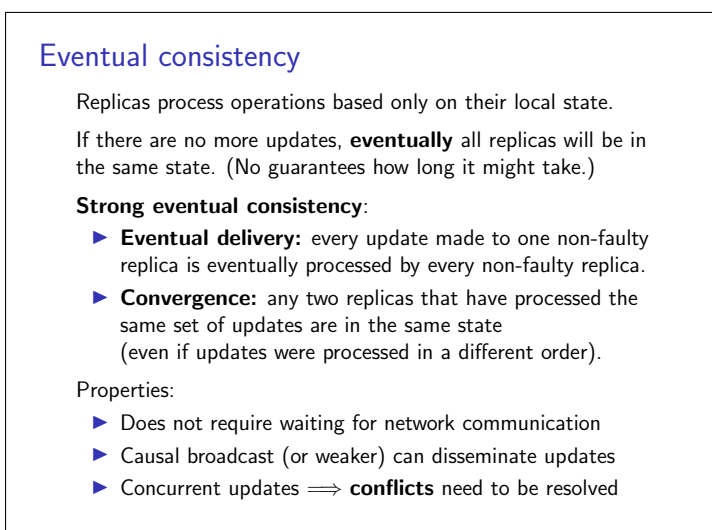
Slide 140

In both eventual consistency and strong eventual consistency, there is the possibility of different nodes concurrently updating the same object, leading to conflicts (as previously discussed on Slide 95). Various algorithms have been developed to resolve those conflicts automatically [Shapiro et al., 2011].

The lecture video shows an example of a conflict in the eventually consistent calendar app: on one device, I update the time of an event, while concurrently on another device, I update the title of the same event. After the two devices synchronise, the update of the time is applied to both devices, while the update of the title is discarded. The state of the two devices therefore converges – at the cost of a small amount of data loss. This is the *last writer wins* approach to conflict resolution that we have seen on Slide 95 (assuming the update to the time is the "last" update in this example). A more refined approach might merge the updates to the time and the title, as shown on Slide 143.

This brings us to the end of our discussion of consistency models. Slide 141 summarises some of the key properties of the models we have seen, in descending order of the minimum strength of assumptions that they must make about the system model.

Atomic commit makes the strongest assumptions, since it must wait for communication with all nodes participating in a transaction (potentially all of the nodes in the system) in order to complete successfully. Consensus, total order broadcast, and linearizable algorithms make weaker assumptions since they only require waiting for communication with a quorum, so they can tolerate some unavailable nodes. The FLP result (Slide 107) showed us that consensus and total order broadcast require partial synchrony. It can be shown that a linearizable CAS operation is equivalent to consensus [Herlihy, 1991], and thus also requires partial synchrony. On the other hand, the ABD algorithm for linearizable get/set is asynchronous, since it does not require any clocks or timeouts. Finally, eventual consistency and strong eventual consistency make the weakest assumptions: operations can be processed without waiting for any communication with other nodes, and without any timing assumptions. Similarly, in causal broadcast and weaker forms of broadcast (FIFO, reliable, etc.), a node broadcasting a message can immediately deliver it to itself without waiting for communication with other nodes, as discussed in Section 4.2; this corresponds to a replica immediately processing its own operations without waiting for communication with other replicas.

This hierarchy has some similarities to the concept of complexity classes of algorithms – for example, sorting generally is $O(n \log n)$ – in the sense that it captures the unavoidable minimum communication and synchrony requirements for a range of common problems in distributed systems.

Summary of minimum system model requirements

Problem	Must wait for communication	Requires synchrony
atomic commit	all participating nodes	partially synchronous
consensus, total order broadcast, linearizable CAS	quorum	partially synchronous
linearizable get/set	quorum	asynchronous
eventual consistency, causal broadcast, FIFO broadcast	local replica only	asynchronous

strength of assumptions

Slide 141

8 Concurrency control in applications

In this last lecture we will look at a couple of examples of distributed systems that need to manage concurrent access to data. In particular, we will include some case studies of practical, real-world systems that need to deal with concurrency, and which build upon the concepts from the rest of this course.

8.1 Collaboration and conflict resolution

Collaboration software is a broad category of software that facilitates several people working together on some task. This includes applications such as Google Docs/Office 365 (multi-user text documents, spreadsheets, presentations, etc.), Overleaf (collaborative LaTeX documents), multi-user graphics software (e.g. Figma), project planning tools (e.g. Trello), note-taking apps (e.g. OneNote, Evernote, Notion), and shared calendars between colleagues or family members (like the calendar sync we saw on Slide 138).

Modern collaboration software allows several people to update a document concurrently, without having to email files back and forth. This makes collaboration another example of replication: each device on which a user has opened a document is a replica, and any updates made to one replica need to be sent over the network to the replicas on other devices.

In principle, it would be possible to use a linearizable replication scheme for collaboration software. However, such software would be slow to use, since every read or write operation would have to contact a quorum of replicas; moreover, it would not work on a device that is offline. Instead, for the sake of better performance and better robustness to network interruptions, most collaboration software uses optimistic replication that provides strong eventual consistency (Slide 140).

<div style="border:1px solid black; padding:10px;">

Collaboration and conflict resolution

Nowadays we use a lot of **collaboration software**:
- ▶ **Examples:** calendar sync (last lecture), Google Docs, . . .
- ▶ Several users/devices working on a shared file/document
- ▶ Each user device has local replica of the data
- ▶ Update local replica anytime (even while offline), sync with others when network available
- ▶ **Challenge:** how to reconcile concurrent updates?

Families of **algorithms**:
- ▶ Conflict-free Replicated Data Types (**CRDTs**)
  - ▶ Operation-based
  - ▶ State-based
- ▶ Operational Transformation (**OT**)

</div>

Start of video section 8.1
(mp4 download)

Slide 142

In this section we will look at some algorithms that are used for this kind of collaboration. As example, consider the calendar sync demo in the lecture recording of Section 7.3. Two nodes initially start with the same calendar entry. On node A, the title is changed from "Lecture" to "Lecture 1", and concurrently on node B the time is changed from 12:00 to 10:00. These two updates happen while the two nodes are temporarily unable to communicate, but eventually connectivity is restored and the two nodes sync their changes. In the outcome shown on Slide 143, the final calendar entry reflects both the change to the title and the change to the time.
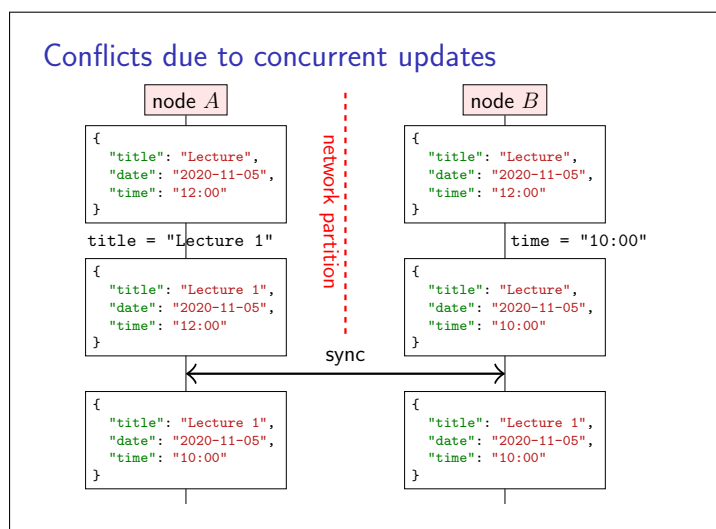
Slide 143

This scenario is an example of *conflict resolution*, which occurs whenever several concurrent writes to the same object need to be integrated into a single final state (see also Slide 95). *Conflict-free replicated data types*, or *CRDTs* for short, are a family of algorithms that perform such conflict resolution [Shapiro et al., 2011]. A CRDT is a replicated object that an application accesses though the object-oriented interface of an abstract datatype, such as a set, list, map, tree, graph, counter, etc.

Slide 144 shows an example of a CRDT that provides a map from keys to values. The application

can invoke two types of operation: reading the value for a given key, and setting the value for a given key (which adds the key if it is not already present).

The local state at each node consists of the set *values* containing $(timestamp, key, value)$ triples. Reading the value for a given key is a purely local operation that only inspects *values* on the current node, and performs no network communication. The algorithm preserves the invariant that *values* contains at most one element for any given key. Therefore, when reading the value for a key, the value is unique if it exists.

Operation-based map CRDT

```
on initialisation do
    values := {}
end on

on request to read value for key k do
    if ∃t, v. (t, k, v) ∈ values then return v else return null
end on

on request to set key k to value v do
    t := newTimestamp()    ▷ globally unique, e.g. Lamport timestamp
    broadcast (set, t, k, v) by reliable broadcast (including to self)
end on

on delivering (set, t, k, v) by reliable broadcast do
    previous := {(t′, k′, v′) ∈ values | k′ = k}
    if previous = {} ∨ ∀(t′, k′, v′) ∈ previous. t′ < t then
        values := (values \ previous) ∪ {(t, k, v)}
    end if
end on
```

Slide 144

To update the value for a given key, we create a globally unique timestamp for the operation – a Lamport timestamp (Slide 66) is a good choice – and then broadcast a message containing the timestamp, key, and value. When that message is delivered, we check if the local copy of *values* already contains an entry with a higher timestamp for the same key; if so, we ignore the message, because the value with the higher timestamp takes precedence. Otherwise we remove the previous value (if any), and add the new $(timestamp, key, value)$ triple to *values*. This means that we resolve concurrent updates to the same key using the last-writer-wins (LWW) approach that we saw on Slide 95.

Operation-based CRDTs

Reliable broadcast may deliver updates in any order:
- ▶ broadcast (set, t_1, "title", "Lecture 1")
- ▶ broadcast (set, t_2, "time", "10:00")

Recall **strong eventual consistency**:
- ▶ **Eventual delivery:** every update made to one non-faulty replica is eventually processed by every non-faulty replica.
- ▶ **Convergence:** any two replicas that have processed the same set of updates are in the same state

CRDT algorithm implements this:
- ▶ Reliable broadcast ensures every operation is eventually delivered to every (non-crashed) replica
- ▶ Applying an operation is **commutative**: order of delivery doesn't matter

Slide 145

This algorithm is an example of an approach that we hinted at on Slide 104, namely, a method for performing replication using reliable broadcast, without requiring totally ordered delivery. It is an *operation-based* CRDT because each broadcast message contains a description of an update operation (as opposed to *state-based* CRDTs that we will see shortly). It allows operations to complete without network connectivity, because the sender of a reliable broadcast can immediately deliver a message to itself, and send it to other nodes sometime later. Moreover, even though messages may be delivered in different orders on different replicas, the algorithm ensures strong eventual consistency because the function that updates a replica's state is commutative.

Exercise 18. *Prove that the operation-based map CRDT algorithm provides strong eventual consistency.*

Exercise 19. *Give pseudocode for a variant of the operation-based map CRDT algorithm that has multi-value register semantics instead of last-writer-wins semantics; that is, when there are several concurrent updates for the same key, the algorithm should preserve all of those updates rather than preserving only the one with the greatest timestamp.*

An alternative CRDT algorithm for the same map datatype is shown on Slide 146. The definition of *values* and the function for reading the value for a key is the same as on Slide 144. However, updates are handled differently: instead of broadcasting each operation, we directly update *values* and then broadcast the whole of *values*. On delivering this message at another replica, we merge together the two replicas' states using a merge function $\sqcup$. This merge function compares the timestamps of entries with the same key, and keeps those with the greater timestamp.

State-based map CRDT

The operator $\sqcup$ merges two states s_1 and s_2 as follows:

$$s_1 \sqcup s_2 = \{(t, k, v) \in (s_1 \cup s_2) \mid \nexists (t', k', v') \in (s_1 \cup s_2).\ k' = k \wedge t' > t\}$$

on initialisation **do**
    $values := \{\}$
end on

on request to read value for key k **do**
    if $\exists t, v.\ (t, k, v) \in values$ **then return** v **else return** null
end on

on request to set key k to value v **do**
    $t := \text{newTimestamp}()$ ▷ globally unique, e.g. Lamport timestamp
    $values := \{(t', k', v') \in values \mid k' \neq k\} \cup \{(t, k, v)\}$
    broadcast $values$ by best-effort broadcast
end on

on delivering V by best-effort broadcast **do**
    $values := values \sqcup V$
end on

Slide 146

This approach of broadcasting the entire replica state and merging it with another replica's state is called a *state-based CRDT*. The downside of the state-based approach is that the broadcast messages are likely to be larger than in the operation-based approach. The advantage of the state-based approach is that it can tolerate lost or duplicated messages: as long as two replicas eventually succeed in exchanging their latest states, they will converge to the same state, even if some earlier messages were lost. Duplicated messages are also fine because the merge operator is idempotent (cf. Slide 90). This is why a state-based CRDT can use unreliable best-effort broadcast, while an operation-based CRDT requires reliable broadcast (and some even require causal broadcast).

State-based CRDTs

Merge operator $\sqcup$ must satisfy: $\forall s_1, s_2, s_3 \ldots$
- ▶ **Commutative**: $s_1 \sqcup s_2 = s_2 \sqcup s_1$.
- ▶ **Associative**: $(s_1 \sqcup s_2) \sqcup s_3 = s_1 \sqcup (s_2 \sqcup s_3)$.
- ▶ **Idempotent**: $s_1 \sqcup s_1 = s_1$.

State-based versus operation-based:
- ▶ Op-based CRDT typically has smaller messages
- ▶ State-based CRDT can tolerate message loss/duplication

Not necessarily uses broadcast:
- ▶ Can also merge concurrent updates to replicas e.g. in quorum replication, anti-entropy, ...

Slide 147

78

Moreover, state-based CRDTs are not limited to replication systems that use broadcast. Other methods of replication, such as the quorum write algorithms and anti-entropy protocols we saw in Lecture 5, can also use CRDTs for conflict resolution (see Slide 94).

As another example of concurrent updates and the need for conflict resolution, we will consider collaboration software such as Google Docs. When you type in a Google Doc, the keystrokes are immediately applied to the local copy of the document in your web browser, without waiting for them to sync to a server or any other users. This means that when two users are typing concurrently, their documents can temporarily diverge; as network communication takes place, the system needs to ensure that all users converge to the same view of the document. The video of this lecture includes a demo of Google Docs showing this conflict resolution process in action.
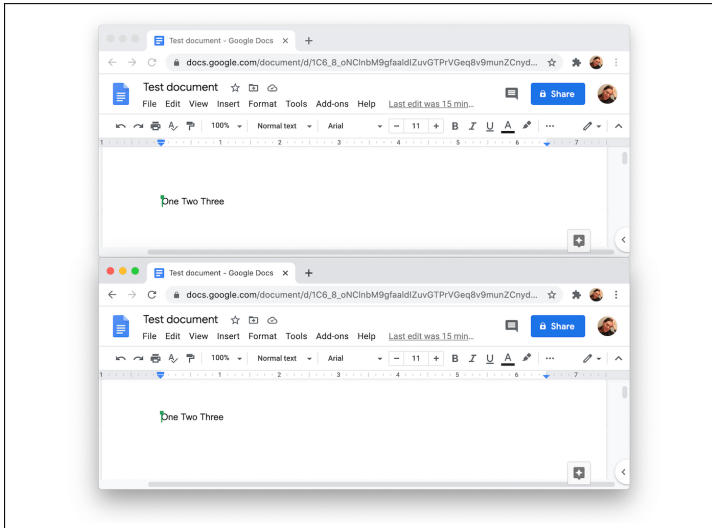
Slide 148

We can think of a collaboratively editable text document as a list of characters, where each user can insert or delete characters at any index in the list. Fonts, formatting, embedded images, tables, and so on add further complexity, so we will just concentrate on plain text for now. When several users may concurrently update a text document, a particular problem arises, which is demonstrated in the example on Slide 149.

In this example, two users A and B both start with the same document, "BC". User A adds the character "A" at the beginning of the document, so that it reads "ABC". Concurrently, user B adds the character "D" at the end of the document, so that it reads "BCD". As A and B merge their edits, we would expect that the final document should read "ABCD".

On Slide 149, the users' replicas communicate by sending each other the operations they have performed. User A sends (insert, 0, "A") to B, and B applies this operation, leading to the desired outcome "ABCD". However, when B sends (insert, 2, "D") to A, and A inserts the character "D" at index 2, the result is "ABDC", not the expected "ABCD".
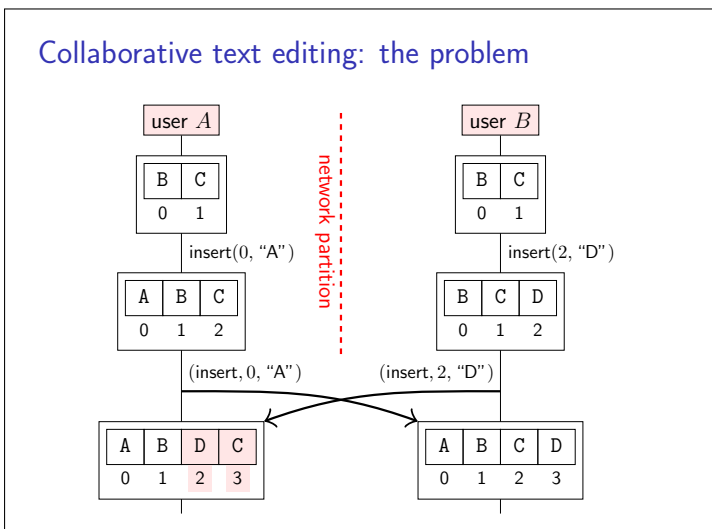
Slide 149

79

The problem is that at the time when B performed the operation insert(2, "D"), index 2 referred to the position after character "C". However, A's concurrent insertion at index 0 had the effect of increasing the indexes of all subsequent characters by 1, so the position after "C" is now index 3, not index 2.

Operational transformation is one approach that is used to solve this problem. There is a family of different algorithms that use this approach and that vary in the details of how they resolve conflicts. But the general principle they have in common is illustrated on Slide 150.

Operational transformation

user A user B

insert(0, "A")   insert(2, "D")

A	B	C
0	1	2

B	C	D
0	1	2

$(\text{insert}, 0, \text{"A"})$ $(\text{insert}, 2, \text{"D"})$

$T((\text{insert}, 2, \text{"D"}), (\text{insert}, 0, \text{"A"})) = (\text{insert}, 3, \text{"D"})$

$T((\text{insert}, 0, \text{"A"}), (\text{insert}, 2, \text{"D"})) = (\text{insert}, 0, \text{"A"})$

A	B	C	D
0	1	2	3

A	B	C	D
0	1	2	3

Slide 150

A node keeps track of the history of operations it has performed. When a node receives another node's operation that is concurrent to one or more of its own operations, it *transforms* the incoming operation relative to its own, concurrent operations.

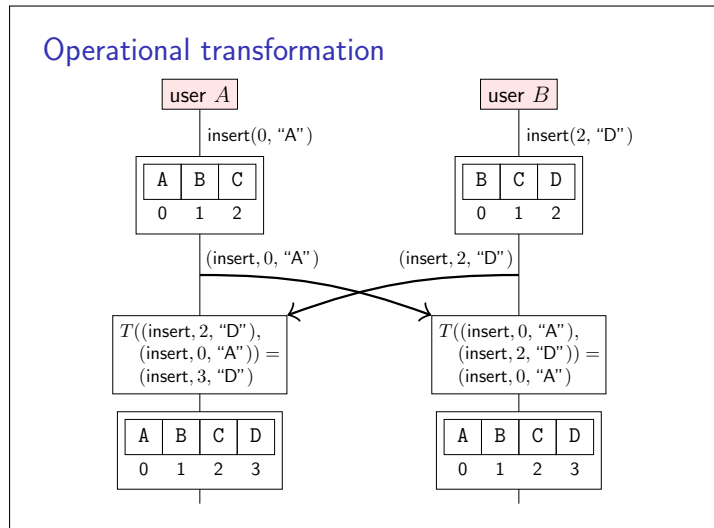The function $T(op_1, op_2)$ takes two operations: op_1 is an incoming operation, and op_2 is a concurrent local operation. T returns a transformed operation op_1' such that applying op_1' to the local state has the effect originally intended by op_1. For example, if $op_1 = (\text{insert}, 2, \text{"D"})$ and $op_2 = (\text{insert}, 0, \text{"A"})$ then the transformed operation is $T(op_1, op_2) = (\text{insert}, 3, \text{"D"})$ because the original insertion op_1 at index 2 now needs to be instead be performed at index 3 due to the concurrent insertion at index 0. On the other hand, $T(op_2, op_1) = op_2$ returns the unmodified op_2 because the insertion at index 0 is not affected by a concurrent insertion later in the document.

The transformation function becomes more complicated when deletions, formatting etc. are taken into account, and we will skip the details. However, this approach is used in practice: for example, the conflict resolution algorithm in Google Docs uses an operational transformation approach based on the Xerox PARC research system Jupiter [Nichols et al., 1995]. A limitation of this approach is that it requires communication between users to use total order broadcast, requiring the use of a designated leader node to sequence the updates, or a consensus algorithm as in Lecture 6.

An alternative to operational transformation, which avoids the need for total order broadcast, is to use a CRDT for text editing. Rather than identifying positions in the text using indexes, and thus necessitating operational transformation, text editing CRDTs work by attaching a unique identifier to each character. These identifiers remain unchanged, even as surrounding characters are inserted or deleted.

Several constructions for these unique identifiers have been proposed, one of which is illustrated on Slide 151. Here, each character is assigned a rational number $i \in \mathbb{Q}$ with $0 < i < 1$, where 0 represents the beginning of the document, 1 is the end, and numbers in between identify the characters in the document in ascending order. We also use the symbol $\vdash$ to represent the beginning of document and $\dashv$ to represent the end; these symbols are part of the algorithm's internal state, not visible to the user.

When we want to insert a new character between two existing adjacent characters with position numbers i and j, we can assign that new character a position number of $\frac{i+j}{2}$, which always lies between i and j. This new position always exists, provided that we use arbitrary-precision arithmetic (floating-point numbers have limited precision, so they would no longer work once the intervals get too small). It is possible for two different nodes to generate characters with the same position number if they concurrently insert at the same position, so we can use the ID of the node that generated a character to break ties for any characters that have the same position number.

Using this approach, conflict resolution becomes easy: an insertion with a particular position number

can simply be broadcast to other replicas, which then add that character to their set of characters, and sort by position number to obtain the current document.

Text editing CRDT
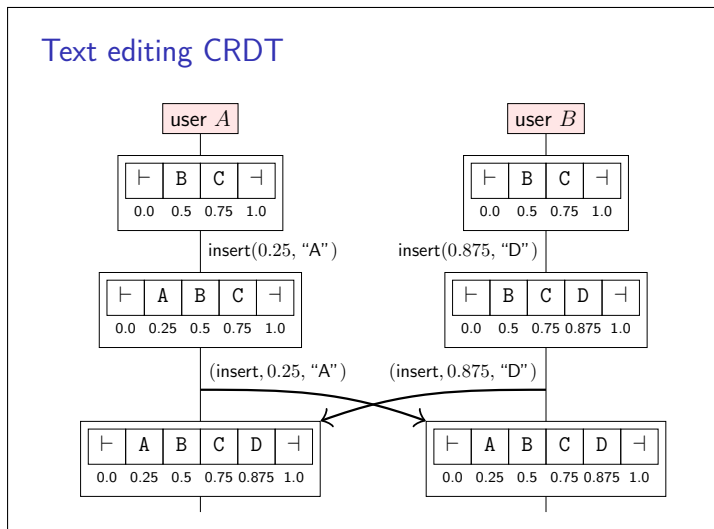
Slide 151

This algorithm is shown on the next two slides. The state of a replica is the set *chars*, which contains ($position, nodeId, character$) triples.

Operation-based text CRDT (1/2)

```
function ELEMENTAT(chars, index)
    min = the unique triple (p, n, v) ∈ chars such that
        ∄(p', n', v') ∈ chars. p' < p ∨ (p' = p ∧ n' < n)}
    if index = 0 then return min
    else return ELEMENTAT(chars \ {min}, index − 1)
end function

on initialisation do
    chars := {(0, null, ⊢), (1, null, ⊣)}
end on

on request to read character at index index do
    let (p, n, v) := ELEMENTAT(chars, index + 1); return v
end on

on request to insert character v at index index at node nodeId do
    let (p₁, n₁, v₁) := ELEMENTAT(chars, index)
    let (p₂, n₂, v₂) := ELEMENTAT(chars, index + 1)
    broadcast (insert, (p₁ + p₂)/2, nodeId, v) by causal broadcast
end on
```

Slide 152

Operation-based text CRDT (2/2)

```
on delivering (insert, p, n, v) by causal broadcast do
    chars := chars ∪ {(p, n, v)}
end on

on request to delete character at index index do
    let (p, n, v) := ELEMENTAT(chars, index + 1)
    broadcast (delete, p, n) by causal broadcast
end on

on delivering (delete, p, n) by causal broadcast do
    chars := {(p', n', v') ∈ chars | ¬(p' = p ∧ n' = n)}
end on
```

▶ Use causal broadcast so that insertion of a character is delivered before its deletion
▶ Insertion and deletion of different characters commute

Slide 153

The function ELEMENTAT iterates over the elements of *chars* in ascending order of position number. It does this by first finding the minimum element, that is, the element for which there does not exist another element with a lower position number. If there are multiple elements with the same position number, the element with the lowest nodeId is chosen. If $index = 0$ we return this minimum element, otherwise we remove the minimum element, decrement the index, and repeat. (This is a rather slow algorithm; a real implementation would make an effort to be more efficient.)

A replica's *chars* is initialised with elements for ⊢ and ⊣. To get the character at a particular index, we use the ELEMENTAT we just defined, adding 1 to the index in order to skip the first element in *chars*, which is always $(0, \mathsf{null}, ⊢)$.

To insert a character at a particular position, we get the position numbers p_1 and p_2 of the immediate predecessor and successor, and then compute the new position number as $(p_1+p_2)/2$. We then disseminate this operation by causal broadcast. On delivery of an insert message we simply add the triple to *chars*.

To delete a character at a particular position we use ELEMENTAT, adding 1 to skip ⊢ as before, to find the position number and nodeId of that character. We then broadcast this information, which uniquely identifies a particular character, by causal broadcast as a delete message. On delivery of a delete message, a replica removes the element in *chars* that matches both the position number and the nodeId in the message, if it exists.

The reason for using causal broadcast (rather than just reliable broadcast) in this algorithm is to ensure that if a character is deleted, all replicas process the insertion of the character before processing the deletion. This restriction is necessary because the operations to insert and delete the same character do not commute. However, insertions and deletions of different characters commute, allowing this algorithm to ensure convergence and strong eventual consistency.

8.2   Google's Spanner

Despite having "strong" in the name, strong eventual consistency is a fairly weak consistency property: for example, when reading a value, there is no guarantee that the operation will return the most up-to-date value, because it may take some time for updates to propagate from one replica to another. In contrast, let's now examine a different system that makes much stronger consistency guarantees. As always, these guarantees come at a cost, but for some applications this is the right choice.

The Spanner database developed by Google [Corbett et al., 2012] is an example of a system that provides the strongest possible consistency guarantees: transactions with serializable isolation and atomic commitment, and linearizable reads and writes. Spanner achieves those properties while remaining very scalable, supporting large data volumes, large transaction throughput, and allowing data to be distributed worldwide. Spanner replicas are designed to be located in datacenters (unlike the collaboration software of the last section, where an end-user device may be a replica).

<div style="border:1px solid">

Google's Spanner

A database system with millions of nodes, petabytes of data, distributed across datacenters worldwide

Consistency properties:
- ▶ **Serializable** transaction isolation
- ▶ **Linearizable** reads and writes
- ▶ Many **shards**, each holding a subset of the data; atomic commit of transactions across shards

Many standard techniques:
- ▶ State machine replication (Paxos) within a shard
- ▶ Two-phase locking for serializability
- ▶ Two-phase commit for cross-shard atomicity

The interesting bit: read-only transactions require **no locks**!

</div>

Slide 154

Many of the techniques used by Spanner are very conventional, and we have seen them earlier in this course: it uses the Paxos consensus algorithm for state machine replication, two-phase locking to ensure serializable isolation between transactions, and two-phase commit to ensure atomic commitment. A lot of engineering effort goes into making these algorithms work well in practice, but at an architectural level,

these well-established choices are quite unsurprising.

However, Spanner is famous for one very unusual aspect of its design, namely its use of atomic clocks. This is the aspect that we will focus on in this section. The reason for this use of clocks is to enable lock-free read-only transactions.

Some read-only transactions need to read a large number of objects in a database; for example, a backup or audit process needs to essentially read the entire database. Performing such transactions with two-phase locking would be extremely disruptive, since the backup may take a long time, and the read lock on the entire database would prevent any clients from writing to the database for the duration of the backup. For this reason, it is very important that large read-only transactions can execute in the background, without requiring any locks and thus without interfering with concurrent read-write transactions.

Spanner avoids locks on read-only transactions by allowing a transaction to read from a *consistent snapshot* of the database: that is, the transaction observes the entire database as it was at a single point in time, even if some parts of the database are subsequently updated by other transactions while the read-only transaction is running. The word "consistent" in the context of a snapshot means that it is consistent with causality: if transaction T_1 happened before transaction T_2, and if the snapshot contains the effects of T_2, then it must also contain the effects of T_1.

Consistent snapshots

A read-only transaction observes a **consistent snapshot**:
If $T_1 \rightarrow T_2$ (e.g. T_2 reads data written by T_1)...

- ▶ Snapshot reflecting writes by T_2 also reflects writes by T_1
- ▶ Snapshot that does not reflect writes by T_1 does not reflect writes by T_2 either
- ▶ In other words, snapshot is **consistent with causality**
- ▶ Even if read-only transaction runs for a long time

Approach: **multi-version concurrency control** (MVCC)

- ▶ Each read-write transaction T_w has commit timestamp t_w
- ▶ Every value is tagged with timestamp t_w of transaction that wrote it (not overwriting previous value)
- ▶ Read-only transaction T_r has snapshot timestamp t_r
- ▶ T_r ignores values with $t_w > t_r$; observes most recent value with $t_w \leq t_r$

Slide 155

Spanner's implementation of snapshots uses *multi-version concurrency control* (MVCC), a particular form of optimistic concurrency control similar to what was discussed in the first half of this course. MVCC is based on assigning a commit timestamp to every transaction; every data object is tagged with the timestamp of the transaction that wrote it. When an object is updated, we don't just overwrite it, but store several old versions (each tagged with a timestamp) in addition to the latest version. A read-only transaction's snapshot is also defined by a timestamp: namely, the transaction reads the most recent version of each object that precedes the snapshot timestamp, and ignores any object versions whose timestamp is greater than that of the snapshot. Many other databases also use MVCC, but what makes Spanner special is the way in which it assigns timestamps to transactions.

In order to ensure that snapshots are consistent with causality, the MVCC algorithm requires that if transaction T_1 happened before transaction T_2, then the commit timestamp of T_1 must be less than that of T_2. However, recall from Slide 61 that timestamps from physical clocks do not necessarily satisfy this property. Thus, our natural response should be to use logical timestamps, such as Lamport timestamps, instead (Section 4.1).

Unfortunately, logical timestamps also have problems. Consider the example on Slide 156, where a user observes the results from transaction T_1, and then takes some action, which is executed in a transaction T_2. This means we have a real-time dependency (Section 7.2) between the transactions: T_1 must have finished before T_2 started, and therefore we expect T_2 to have a greater timestamp than T_1. However, Lamport timestamps cannot necessarily ensure this ordering property: recall that they work by attaching a timestamp to every message that is communicated over the network, and taking the maximum every time such a message is received. However, in the example on Slide 156, there might not be any message sent from replica A, where T_1 is executed, to replica B, where T_2 is executed. Instead, the communication goes via a user, and we cannot expect a human to include a properly formed timestamp

on every action they perform. Without a reliable mechanism for propagating the timestamp on every communication step, logical timestamps cannot provide the ordering guarantee we need.

Slide 156

Another option for generating logical timestamps would be to have a single designated server that assigns timestamps to transactions. However, this approach breaks down in a globally distributed database, as that server would become a single point of failure and a performance bottleneck. Moreover, if transactions executing on a different continent from the timestamp server need to wait for a response, the inevitable round-trip time due to speed-of-light delays would make transactions slow to execute. A less centralised approach to timestamps is required.

This is where Spanner's TrueTime mechanism comes in. TrueTime is a system of physical clocks that does not return a single timestamp, but rather returns an *uncertainty interval*. Even though we cannot ensure perfectly synchronised clocks in practical systems (Section 3.2), we *can* keep track of the errors that may be introduced at various points in the system. For atomic clocks, error bounds are reported by the manufacturer. For GPS receivers, the error depends on the quality of the signals from the satellites currently within range. The error introduced by synchronising clocks over a network depends on the round-trip time (Exercise 5). The error of a quartz clock depends on its drift rate and the time since its last sync with a more accurate clock.

When you ask TrueTime for the current timestamp, it returns an interval $[t_{\text{earliest}}, t_{\text{latest}}]$. The system does not know the true current physical timestamp t_{real}, but it can ensure that $t_{\text{earliest}} \leq t_{\text{real}} \leq t_{\text{latest}}$ with very high probability by taking all of the above sources of error into account.
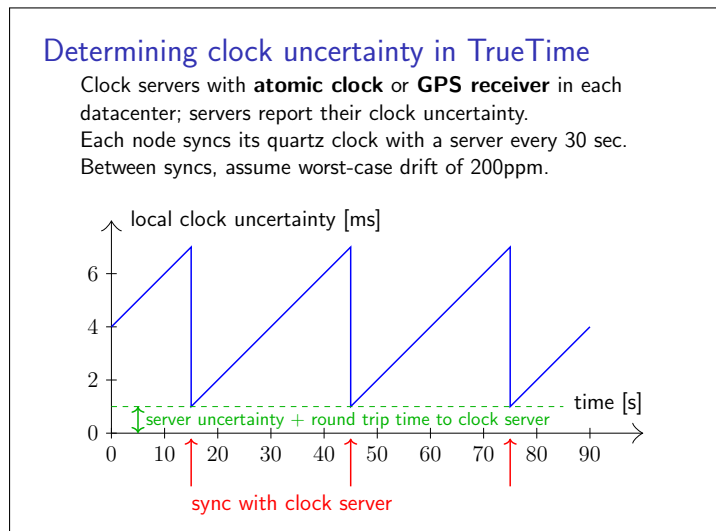

Slide 157

When transaction T_i wants to commit in Spanner, it gets a timestamp interval $[t_{i,\text{earliest}}, t_{i,\text{latest}}]$ from TrueTime, and assigns $t_{i,\text{latest}}$ to be the commit timestamp of T_i. However, before the transaction actually commits and releases its locks, it first pauses and waits for a duration equal to the clock uncertainty period

$\delta_i = t_{i,\text{latest}} - t_{i,\text{earliest}}$. Only after this wait time has elapsed, the transaction is committed and its writes become visible to other transactions.

Even though we don't have perfectly synchronised clocks, and thus a node cannot know the exact physical time of an event, this algorithm ensures that the timestamp of a transaction is less than the true physical time at the moment when the transaction commits. Therefore, if T_2 begins later in real time than T_1, the earliest possible timestamp that could be assigned to T_2 must be greater than T_1's timestamp. Put another way, the waiting ensures that the timestamp intervals of T_1 and T_2 do not overlap, even if the transactions are executed on different nodes with no communication between the two transactions.

Since every transaction has to wait for the uncertainty interval to elapse, the challenge is now to keep that uncertainty interval as small as possible so that transactions remain fast. Google achieves this by installing atomic clocks and GPS receivers in every datacenter, and synchronising every node's quartz clock with a time server in the local datacenter every 30 seconds. In the local datacenter, round-trips are usually below 1 ms, so the clock error introduced by network latency is quite small. If the network latency increases, e.g. due to congestion, TrueTime's uncertainty interval grows accordingly to account for the increased error.

Slide 158

In between the periodic clock synchronisations every 30 seconds, a node's clock is determined only by its local quartz oscillator. The error introduced here depends on the drift rate of the quartz. To be on the safe side, Google assumes a maximum drift rate of 200 ppm, which is significantly higher than the drift observed under normal operating conditions (Slide 45). Moreover, Google monitors the drift of each node and alerts administrators about any outliers.

Is the drift rate of 200 ppm a safe assumption? According to the Spanner paper: "Our machine statistics show that bad CPUs are 6 times more likely than bad clocks. That is, clock issues are extremely infrequent, relative to much more serious hardware problems. As a result, we believe that TrueTime's implementation is as trustworthy as any other piece of software upon which Spanner depends." [Corbett et al., 2012].

If we assume a quartz drift of 200 ppm, and it has been 30 seconds since the last clock sync, this implies a clock uncertainty of 6 ms due to quartz drift (on top of any uncertainty from network latency, GPS receiver and atomic clocks). The result is an uncertainty interval that grows gradually larger with the time since the last clock sync, up to about 7 ms, and which resets to about 1 ms (round-trip time + clock server uncertainty) on every clock sync, as shown on Slide 158.

The average uncertainty interval is therefore approximately 4 ms in normal operating conditions, and this 4 ms is therefore the average time that a transaction must wait before it can commit. This is much faster than transactions could be if they had to wait for an intercontinental network round-trip (which would take on the order of 100 ms or more).

To summarise: through careful accounting of uncertainty, TrueTime provides upper and lower bounds on the current physical time; through high-precision clocks it keeps the uncertainty interval small; by waiting out the uncertainty, interval Spanner ensures that timestamps are consistent with causality; and by using those timestamps for MVCC, Spanner provides serializable transactions without requiring any locks for read-only transactions. This approach keeps transactions fast, without placing any requirement on clients to propagate logical timestamps.

That's all, folks!

Any questions?   Email mk428@cst.cam.ac.uk!

Summary:
- ▶ Distributed systems are everywhere
- ▶ You use them every day: e.g. web apps
- ▶ Key goals: availability, scalability, performance
- ▶ Key problems: concurrency, faults, unbounded latency
- ▶ Key abstractions: replication, broadcast, consensus
- ▶ No one right way, just trade-offs

Slide 159

This brings us to the end of the course on Concurrent and Distributed Systems. We started from a simple premise: when you send a message over the network and you don't get a response, you don't know what happened. Maybe the message got lost, or the response got lost, or either message got delayed, or the remote node crashed, and we cannot distinguish between these types of fault.

Distributed systems are fascinating because we have to work with partial knowledge and uncertain truths. We never have certainty about the state of the system, because by the time we hear about something, that state may already be outdated. In this way it resembles real life more than most of computing! In real life you need to often make decisions with incomplete information.

But distributed systems are also immensely practical: every web site and most apps are distributed systems, and the servers and databases that underlie most websites are in turn further distributed systems. After graduating, many of you will end up working on such systems. Hopefully, the ideas in this course have given you a solid grounding so that you can go and make those systems reliable and understandable.

References

Steven L. Allen. Planes will crash! Things that leap seconds didn't, and did, cause, 2013. URL http://www.hanksville.org/futureofutc/preprints/files/2_AAS%2013-502_Allen.pdf.

Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995. doi:10.1145/200836.200869. URL http://www.cse.huji.ac.il/course/2004/dist/p124-attiya.pdf.

Peter Bailis and Kyle Kingsbury. The network is reliable. *ACM Queue*, 12(7), 2014. doi:10.1145/2639988.2639988. URL https://queue.acm.org/detail.cfm?id=2655736.

Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996. doi:10.1145/226643.226647. URL http://courses.csail.mit.edu/6.852/08/papers/CT96-JACM.pdf.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J.J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI 2012, October 2012. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, December 2007. doi:10.1145/1323293.1294281. URL http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf.

Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988. doi:10.1145/42282.42283. URL http://www.net.t-labs.tu-berlin.de/~petr/ADC-07/papers/DLS88.pdf.

Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. URL https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

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. doi:10.1145/3149.214121. URL https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf.

Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, June 2002. doi:10.1145/564585.564601. URL https://www.comp.nus.edu.sg/~gilbert/pubs/BrewersConjecture-SigAct.pdf.

Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems*, 31(1):133–160, March 2006. doi:10.1145/1132863.1132867. URL http://db.cs.berkeley.edu/cs286/papers/paxoscommit-tods2006.pdf.

Jim N. Gray. Notes on data base operating systems. In R. Bayer, R.M. Graham, and G. Seegmüller, editors, *Operating Systems*, volume 60 of *LNCS*, pages 393–481. Springer, 1978. doi:10.1007/3-540-08755-9_9. URL http://jimgray.azurewebsites.net/papers/dbos.pdf.

Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991. doi:10.1145/114005.102808. URL http://cs.brown.edu/~mph/Herlihy91/p124-herlihy.pdf.

Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990. doi:10.1145/78969.78972. URL http://cs.brown.edu/~mph/HerlihyW90/p463-herlihy.pdf.

Heidi Howard and Richard Mortier. Paxos vs Raft: have we reached consensus on distributed consensus? In *7th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC, April 2020. doi:10.1145/3380787.3393681. URL https://arxiv.org/abs/2004.05074.

Mark Imbriaco. Downtime last Saturday, December 2012. URL https://github.com/blog/1364-downtime-last-saturday.

Martin Kleppmann. A critique of the CAP theorem. *arXiv*, September 2015. URL http://arxiv.org/abs/1509.05393.

Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In *18th International Conference on Principles of Distributed Systems (OPODIS)*, volume 8878 of *LNCS*, pages 17–32. Springer, December 2014. doi:10.1007/978-3-319-14472-6_2. URL https://cse.buffalo.edu/~demirbas/publications/hlc.pdf.

Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563. URL http://research.microsoft.com/en-US/um/people/Lamport/pubs/time-clocks.pdf.

Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998. doi:10.1145/279227.279229. URL http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf.

Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982. doi:10.1145/357172.357176. URL http://research.microsoft.com/en-us/um/people/lamport/pubs/byz.pdf.

Nelson Minar. Leap second crashes half the internet, July 2012. URL http://www.somebits.com/weblog/tech/bad/leap-second-2012.html.

David A Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *8th Annual ACM Symposium on User Interface and Software Technology*, UIST 1995, pages 111–120, November 1995. doi:10.1145/215585.215706. URL http://www.lively-kernel.org/repository/webwerkstatt/projects/Collaboration/paper/Jupiter.pdf.

Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, ATC. USENIX, June 2014. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

Nuno Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. Dotted version vectors: Logical clocks for optimistic replication, November 2010. URL http://arxiv.org/pdf/1011.5808v1.pdf.

Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS, pages 386–400, October 2011. doi:10.1007/978-3-642-24550-3_29. URL https://pages.lip6.fr/Marek.Zawirski/papers/RR-7687.pdf.

Martin Thompson. Java garbage collection distilled, June 2013. URL https://www.infoq.com/articles/Java_Garbage_Collection_Distilled/.

Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, January 2009. doi:10.1145/1435417.1435432. URL http://cacm.acm.org/magazines/2009/1/15666-eventually-consistent/fulltext.

Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report TR-94-29, Sun Microsystems Laboratories, 1994. URL http://m.mirror.facebook.net/kde/devel/smli_tr-94-29.pdf.