Paxos Made Moderately Complex

Robbert van Renesse

Department of Computer Science, Cornell University

Abstract

For anybody who has ever tried to implement it, Paxos is by no means a simple protocol, even though it is based on relatively simple invariants. This paper provides pseudo-code for the full Paxos (or Multi-Paxos) protocol without shying away from discussing various implementation details. The initial description avoids optimizations that complicate comprehension. Next we discuss liveness, and list various optimizations that make the protocol practical.

1 Introduction

Paxos [14] is a protocol for state machine replication in an asynchronous environment that admits crash failures. It is useful to consider the terms in this sentence carefully:

- A *state machine* consists of a collection of states, a collection of transitions between states, and a current state. A transition to a new current state happens in response to an issued operation and produces an output. Transitions from the current state to the same state are allowed, and are used to model read-only operations. In a *deterministic state machine*, for any state and operation, the transition enabled by the operation is unique and the output is a function only of the state and the operation.

- In an *asynchronous environment*, there are no bounds on timing. Clocks run arbitrarily fast, network communication takes arbitrarily long, and state machines take arbitrarily long to transition in response to an operation. The term "asynchronous" as used here should not be confused with nonblocking operations on objects that are often called asynchronous as well.

- A state machine has experienced a *crash failure* if it will make no more transitions and thus its current state is fixed indefinitely. No other failures of a state machine, such as experiencing unspecified transitions, are allowed. In a "fail-stop environment" [22], crash failures can be reliably detected—not so in an asynchronous environment.

- *State Machine Replication* (SMR) [13, 23] is a technique to mask failures, and crash failures in particular. A collection of replicas of a deterministic state machine are created. The replicas are then provided with the same sequence of operations, so they end up in the same state and produce the same sequence of outputs. It is assumed that at least one replica never crashes, but we do not know a priori which replica this is.

Deterministic state machines are used to model server processes, such as a file server, a DNS server, and so on. A client process, using a library "stub routine," can send a *command* to such a server over a network and await an output. A command is a triple $\langle \kappa, cid, operation \rangle$, where κ[1] is the identifier of the client that issued the command and *cid* a client-local unique command identifier (such as a sequence number). That is, there cannot be two commands that have the same client identifier and command identifier but have different operations. However, a client can issue the same operation more than once by using

[1] As in [14], we will use Greek letters to identify processes.

different command identifiers. The command identifier must be included in the response from the server to a command so the client can match responses with commands.

In SMR replication, the stub routine is replaced with another to provide the illusion of a single remote server that is highly available. The stub routine sends the command to all replicas (at least one of which is assumed to not crash) and returns only the first response to the command.

The difficulty comes with multiple clients, as concurrent commands may arrive in different orders at different replicas, and thus the replicas may end up taking different transitions, producing different outputs as a result and possibly ending up in different current states. A protocol like Paxos ensures that this cannot happen: all replicas receive commands in the same order and thus the replicated state machine behaves logically identical to a single remote state machine that never crashes [10].

While processes may crash, we assume that messaging between processes is reliable (but not necessarily FIFO):

- a message sent by a non-faulty process to a non-faulty destination process is eventually received (at least once) by the destination process;
- if a message is received by a process, it was sent by some (possibly faulty) process. That is, messages are not garbled and do not appear out of the blue.

This paper gives an *operational description* of the *multi-decree Paxos protocol*, sometimes called *multi-Paxos*. Single-decree Paxos is significantly easier to understand, and is the topic of such papers as [16, 15]. But the multi-decree Paxos protocol is the one that is used (or some variant thereof) within industrial-strength systems like Chubby [5] and ZooKeeper [11]. The paper does not repeat any correctness proofs for Paxos, but it does stress invariants for two reasons. First, the invariants make it possible to understand the operational description, as each operation can be checked against the invariants. For any operation and any invariant you can check the following: if the invariant holds before the operation, then the invariant still holds after the operation. Second, they make it

```
process $Replica(leaders, initial\_state)$
  var $state := initial\_state, slot\_num := 1;$
  var $proposals := \emptyset, decisions := \emptyset;$

  function $propose(c)$
    if $\nexists s : \langle s, c\rangle \in decisions$ then
      $s' := \min\{s \mid s \in \mathbb{N}^+ \wedge$
          $\nexists c' : \langle s, c'\rangle \in proposals \cup decisions\};$
      $proposals := proposals \cup \{\langle s', c\rangle\};$
      $\forall \lambda \in leaders : send(\lambda, \langle \mathbf{propose}, s', c\rangle);$
    end if
  end function

  function $perform(\langle \kappa, cid, op\rangle)$
    if $\exists s : s < slot\_num \wedge$
          $\langle s, \langle \kappa, cid, op\rangle\rangle \in decisions$ then
      $slot\_num := slot\_num + 1;$
    else
      $\langle next, result\rangle := op(state);$
      atomic
        $state := next;$
        $slot\_num := slot\_num + 1;$
      end atomic
      $send(\kappa, \langle \mathbf{response}, cid, result\rangle);$
    end if
  end function

  for ever
    switch $receive()$
      case $\langle \mathbf{request}, c\rangle$ :
        $propose(c);$
      case $\langle \mathbf{decision}, s, c\rangle$ :
        $decisions := decisions \cup \{\langle s, c\rangle\};$
        while $\exists c' : \langle slot\_num, c'\rangle \in decisions$ do
          if $\exists c'' : \langle slot\_num, c''\rangle \in proposals \wedge$
                $c'' \neq c'$ then
            $propose(c'');$
          end if
          $perform(c');$
        end while;
    end switch
  end for
end process
```

Figure 1: Pseudo code for a replica.

reasonably clear why Paxos is correct without having to go into correctness proofs or elaborate and complicated examples.

In Section 2 we start with describing a version of Paxos that maintains more state than strictly necessary, and will not worry too much about liveness of the protocol. Section 3 considers liveness. In Section 4 we describe various optimizations and design decisions that make Paxos a practical solution for state machine replication. The paper is accompanied by a Java implementation, and Section 5 suggests various exercises based on this code.

2 How and Why Paxos Works

Replicas and Slots

In order to tolerate f crashes, Paxos needs at least $f+1$ replicas to maintain copies of the application state. When a client κ wants to execute a command $\langle \kappa, \mathit{cid}, \mathit{op} \rangle$, its stub routine broadcasts a $\langle \textbf{request}, \langle \kappa, \mathit{cid}, \mathit{op} \rangle \rangle$ message to all replicas and waits for a $\langle \textbf{response}, \mathit{cid}, \mathit{result} \rangle$ message from one of the replicas.

The replicas can be thought of as having a sequence of *slots* that need to be filled with commands. Each slot is indexed by a *slot number*. A replica, on receipt of a $\langle \textbf{request}, c \rangle$ message, proposes command c for its lowest unused slot. We call the pair (s, c) a *proposal* for slot s. In the face of concurrently operating clients, different replicas may end up proposing different commands for the same slot. In order to avoid inconsistency, a replica awaits a decision for a slot before actually updating its state and computing a response to send back to the client.

Replicas are not necessarily identical at any time. However, replicas apply operations to the application state in the same order. Figure 1 shows pseudo-code for a replica. Each replica ρ maintains four variables:

- $\rho.\mathit{state}$, the replica's copy of the application state, which we will treat as opaque. All replicas start with the same initial application state;
- $\rho.\mathit{slot\_num}$, the replica's current slot number (equivalent to the version number of the state, and initially 1). It contains the index of the next slot for which it needs to learn a decision before it can update its copy of the application state;
- $\rho.\mathit{proposals}$, a set of proposals that the replica has made in the past (initially empty); and
- $\rho.\mathit{decisions}$, another set of proposals that are known to have been decided (also initially empty).

Before giving an operational description of replicas, we present some important invariants that hold over the collected variables of replicas:

R1: There are no two different commands decided for the same slot: $\forall s, \rho_1, \rho_2, c_1, c_2 : \langle s, c_1 \rangle \in \rho_1.\mathit{decisions} \wedge \langle s, c_2 \rangle \in \rho_2.\mathit{decisions} \Rightarrow c_1 = c_2$

R2: All commands up to *slot_num* are in the set of decisions: $\forall \rho, s : 1 \leq s < \rho.\mathit{slot\_num} \Rightarrow$
$(\exists c : \langle s, c \rangle \in \rho.\mathit{decisions})$

R3: For all replicas ρ, $\rho.\mathit{state}$ is the result of applying the commands in $\langle s, c_s \rangle \in \rho.\mathit{decisions}$ for all s up to *slot_num* to *initial_state*, in order of slot number;

R4: For each ρ, the variable $\rho.\mathit{slot\_num}$ cannot decrease over time.

From Invariants R1-3, it is clear that all replicas apply operations to the application state in the same order, and thus replicas with the same slot number have the same *state*. Invariant R4 captures that a replica cannot go back in time.

Returning to Figure 1, a replica runs in an infinite loop, receiving messages. Replicas receive two kinds of messages: requests from clients, and decisions. When it receives a request for command c from a client, the replica invokes *propose*(c). This function checks if there has been a decision for c already. If so, the replica has already sent a response and the request can be ignored. If not, the replica determines the lowest unused slot number s', and adds proposal $\langle s', c \rangle$ to its set of proposals. It then sends a $\langle \textbf{propose}, s', c \rangle$ message to all *leaders*. Leaders are described below.

Decisions may arrive out-of-order and multiple times. For each **decision** message, the replica adds the decision to the set of decisions. Then, in a loop, it considers which decisions are ready for execution before trying to receive more messages. If there is a decision c' corresponding to the current *slot_num*, the replica first checks to see if it has proposed a different command c''. If so, it re-proposes c'', which will be assigned a new slot number. Next, it invokes *perform*(c').

The function *perform*() is invoked with the same sequence of commands at all replicas. First, it checks to see if has already performed the command. Different replicas may end up proposing the same command for different slots, and thus the same command may be decided multiple times. The corresponding operation is evaluated only if the command is new. In either case, *perform*() increments *slot_num*.

Note that both *proposals* and *decisions* are "append-only" in that there is no code that removes entries from these sets. Doing so makes it easier to formulate invariants and reason about the correctness of the code. In Section 4.2 we will discuss correctness-preserving ways of removing entries that are no longer useful.

It is clear that the code enforces Invariant R4. The variables *state* and *slot_num* are updated atomically in order to ensure that Invariant R3 holds, although in practice it is not actually necessary to perform these updates atomically, as the intermediate state is not externally visible. Since *slot_num* is only advanced if the corresponding decision is in *decisions*, it is clear that Invariant R2 holds.

The real difficulty lies in enforcing Invariant R1. It requires that the set of replicas agree on the order of commands. For each slot, the Paxos protocol *chooses* a command from among a collection of commands proposed by clients. This is called *consensus*, and in Paxos the subprotocol that implements consensus is called the "multi-decree Synod" protocol, or just Synod protocol for short as we do not consider the single-decree protocol in this paper.

The Synod Protocol, Ballots, and Acceptors

In the Synod protocol, there is an infinite collection of *ballots*. Ballots are not created; they just are. As we shall see later, ballots are the key to liveness in Paxos. Each ballot has a unique *leader*, a deterministic state machine in its own right. A leader can be working on arbitrarily many ballots, although it will be predominantly working on one at a time, even as multiple slots are being decided. In order to tolerate f failures, there must be at least $f + 1$ leaders, each in charge of a possibly infinite number of ballots. A leader process has a unique identifier called the *leader identifier*. A ballot has a unique identifier as well, called its *ballot number*. Ballot numbers are totally ordered, that is, for any two different ballot numbers, one is before or after the other. Do not confuse ballot numbers and slot numbers; they are orthogonal concepts. One ballot can be used to decide multiple slots, and one slot may be considered by multiple ballots.

In this description, we will have ballot numbers be lexicographically ordered pairs of an integer and its leader identifier (consequently, leader identifiers need to be totally ordered as well). This way, given a ballot number, it is trivial to see who the leader of the ballot is.[2] We will use one special ballot number $\perp$ that is ordered before any normal ballot number, but does not correspond to any ballot.

Besides replicas and leaders, there is a fixed collection of *acceptors*, deterministic state machines as well (although *not* replicas of one another, because they get different sequences of input). Think of acceptors as servers, and leaders as their clients. As we shall see, acceptors maintain the fault tolerant memory of Paxos, preventing conflicting decisions from being made. We will assume that at most a strict minority of acceptors can crash. Thus, in order to tolerate f crash failures, Paxos needs at least $2f + 1$ acceptors.

An acceptor is quite simple, as it is passive and only sends messages in response to requests. Its state consists of two variables. Let a *pvalue* be a triple consisting of a ballot number, a slot number, and a command. If α is the identifier of an acceptor, then the acceptor's state is described by

- α.*ballot_num*: a ballot number, initially $\perp$;
- α.*accepted*: a set of pvalues, initially empty.

Under the direction of request messages sent by leaders, the state of an acceptor can change. Let

[2] In the original Paxos protocol, the leader of a ballot is elected and some ballots may end up without a leader.

$p = \langle b, s, c\rangle$ be a pvalue consisting of a ballot number b, a slot number s, and a command c. When an acceptor α adds p to $\alpha.accepted$, we say that α *accepts* p. (An acceptor may accept the same pvalue multiple times.) When α sets its ballot number to b for the first time, we say that α *adopts* b.

We start by presenting some important invariants that hold over the collected variables of acceptors. Knowing these invariants are an invaluable help to understanding the Synod protocol:

A1: an acceptor can only adopt strictly increasing ballot numbers;

A2: an acceptor α can only add $\langle b, s, c\rangle$ to $\alpha.accepted$ (*i.p.*, accept $\langle b, s, c\rangle$) if $b = \alpha.ballot\_num$;

A3: acceptor α cannot remove pvalues from $\alpha.accepted$ (we will modify this impractical restriction later);

A4: Suppose α and α' are acceptors, with $\langle b, s, c\rangle \in \alpha.accepted$ and $\langle b, s, c'\rangle \in \alpha'.accepted$. Then $c = c'$. Informally, given a particular ballot number and slot number, there can be at most one proposed command under consideration by the set of acceptors.

A5: Suppose that for each α among a majority of acceptors, $\langle b, s, c\rangle \in \alpha.accepted$. If $b' > b$ and $\langle b', s, c'\rangle \in \alpha'.accepted$, then $c = c'$. We will consider this crucial invariant in more detail later.

It is important to realize that an invariant like A5 works "both ways." In one direction, if a majority of acceptors have accepted a particular pvalue $\langle b, s, c\rangle$, then any pvalue for a later ballot will contain the same command c. In the other direction, suppose some acceptor accepts $\langle b', s, c'\rangle$ before there is a majority of acceptors that have accepted some other pvalue on an earlier ballot. That does not rule out that at some later time there will be a pvalue $\langle b, s, c\rangle$ on a ballot b, $b < b'$, that is accepted by a majority of acceptors. The invariant requires that $c = c'$ in that case as well.

Figure 2 shows pseudo-code for an acceptor. It runs in an infinite loop, receiving two kinds of request messages from leaders (note the use of pattern matching):

```
process $Acceptor()$
  var $ballot\_num := \bot$, $accepted := \emptyset$;

  for ever
    switch $receive()$
      case $\langle \mathbf{p1a}, \lambda, b\rangle$ :
        if $b > ballot\_num$ then
          $ballot\_num := b$;
        end if;
        $send(\lambda, \langle \mathbf{p1b}, self(), ballot\_num, accepted\rangle)$;
      end case
      case $\langle \mathbf{p2a}, \lambda, \langle b, s, c\rangle\rangle$ :
        if $b \geq ballot\_num$ then
          $ballot\_num := b$;
          $accepted := accepted \cup \{\langle b, s, c\rangle\}$;
        end if
        $send(\lambda, \langle \mathbf{p2b}, self(), ballot\_num\rangle)$;
      end case
    end switch
  end for
end process
```

Figure 2: Pseudo code for an acceptor.

- $\langle \mathbf{p1a}, \lambda, b\rangle$: Upon receiving a "phase 1a" request message from a leader with identifier λ, for a ballot number b, an acceptor makes the following transition. First, the acceptor adopts b if and only if it exceeds its current ballot number. Then it returns to λ a "phase 1b" response message containing its current ballot number and all pvalues accepted thus far by the acceptor.

- $\langle \mathbf{p2a}, \lambda, \langle b, s, c\rangle\rangle$: Upon receiving a "phase 2a" request message from leader λ with pvalue $\langle b, s, c\rangle$, an acceptor makes the following transition. If b exceeds the current ballot number, then the acceptor first adopts b. If the current ballot number equals b, then the acceptor accepts $\langle b, s, c\rangle$. The acceptor returns to λ a "phase 2b" response message containing its current ballot number.

It is easy to see that the code enforces Invariants A1, A2, and A3. For checking the remaining two invariants, which involve multiple acceptors, we have to study what a leader does first.

Leaders and Commanders

According to Invariant A4, there can be at most one proposed command per ballot number and slot number. The leader of a ballot is responsible for selecting a command for each slot, in such a way that selected commands cannot conflict with decisions on other ballots (Invariant A5). A leader may work on multiple slots at the same time. When, in ballot b, its leader tries to get a command c for slot number s chosen, it spawns a local *commander* thread for $\langle b, s, c\rangle$. While we present it here as a separate process, the commander is really just a thread running within the leader. As we shall see, the following invariants hold in the Synod protocol:

C1: For any b and s, at most one commander is spawned;

C2: Suppose that for each α among a majority of acceptors $\langle b, s, c\rangle \in \alpha.accepted$. If $b' > b$ and a commander is spawned for $\langle b', s, c'\rangle$, then $c = c'$.

Invariant C1 implies Invariant A4, because by C1 all acceptors that accept a pvalue for a particular ballot and slot number received the pvalue from the same commander. Similarly, Invariant C2 implies Invariant A5.

Figure 3(a) shows the pseudo-code for a commander. A commander sends a $\langle \mathbf{p2a}, \lambda, \langle b, s, c\rangle\rangle$ message to all acceptors, and waits for responses of the form $\langle \mathbf{p2b}, \alpha, b'\rangle$. In each such response $b' \geq b$ will hold (see the code for acceptors). There are two cases:

1. If a commander receives $\langle \mathbf{p2b}, \alpha, b\rangle$ from all acceptors in a majority of acceptors, then the commander learns that command c has been chosen for slot s. In this case the commander notifies the replicas and exits. In order to satisfy Invariant R1, we need to enforce that if a commander learns that c is chosen for slot s, and another commander learns that c' is chosen for the same slot s, then $c = c'$. This is a consequence of Invariant A5: if a majority of acceptors accept $\langle b, s, c\rangle$, then for any later ballot b' and the same slot number s, acceptors can only accept $\langle b', s, c\rangle$. Thus if the commander of $\langle b', s, c'\rangle$ learns that c' has been chosen for s, it is guaranteed that $c = c'$ and no inconsistency occurs, assuming—of course—that Invariant C2 holds.

2. If a commander receives $\langle \mathbf{p2b}, \alpha', b'\rangle$ from some acceptor α', with $b' \neq b$, then it learns that a ballot b' (which must be larger than b as guaranteed by acceptors) is active. This means that ballot b may no longer be able to make progress, as there may no longer exist a majority of acceptors that can accept $\langle b, s, c\rangle$. In this case, the commander notifies its leader about the existence of b', and exits.

Under the assumption that at most a minority of acceptors can crash, messages are delivered reliably, and the commander does not crash, the commander will eventually do one or the other.

Scouts, Passive and Active Modes

The leader must enforce Invariants C1 and C2. Because there is only one leader per ballot, invariant C1 is trivial to enforce by the leader not spawning more than one commander per ballot number and slot number. In order to enforce Invariant C2, the leader runs what is often called a *view change* protocol for some ballot before spawning commanders for that ballot.[3] The leader spawns a *scout* thread to run the view change protocol for some ballot b. A leader starts at most one of these for any ballot b, and only for its own ballots.

Figure 3(b) shows the pseudo-code for a scout. The code is similar to that of a commander, except that it sends and receives phase 1 instead of phase 2 messages. A scout completes successfully when it has collected $\langle \mathbf{p1b}, \alpha, b, r_\alpha\rangle$ messages from all acceptors in a majority (again, guaranteed to complete eventually), and returns an $\langle \mathbf{adopted}, b, \bigcup r_\alpha\rangle$ message to its leader λ. As we will see later, the leader uses $\bigcup r_\alpha$, the union of all pvalues accepted by this majority of acceptors, in order to enforce Invariant C2.

Figure 4 shows the main code of a leader. Leader λ maintains three state variables:

- $\lambda.ballot\_num$: a monotonically increasing ballot number, initially $(0, \lambda)$;

[3] The term "view change" is used in the Viewstamped Replication protocol [20], similar to the Paxos protocol.

```
process $Commander(\lambda, acceptors, replicas, \langle b, s, c\rangle)$
  var $waitfor := acceptors$;

  $\forall\alpha \in acceptors : send(\alpha, \langle \mathbf{p2a}, self(), \langle b, s, c\rangle\rangle)$;
  for ever
    switch $receive()$
      case $\langle \mathbf{p2b}, \alpha, b'\rangle$ :
        if $b' = b$ then
          $waitfor := waitfor - \{\alpha\}$;
          if $|waitfor| < |acceptors|/2$ then
            $\forall\rho \in replicas$ :
              $send(\rho, \langle \mathbf{decision}, s, c\rangle)$;
            $exit()$;
          end if;
        else
          $send(\lambda, \langle \mathbf{preempted}, b'\rangle)$;
          $exit()$;
        end if;
      end case
    end switch
  end for
end process
```

(a)

```
process $Scout(\lambda, acceptors, b)$
  var $waitfor := acceptors, pvalues := \emptyset$;

  $\forall\alpha \in acceptors : send(\alpha, \langle \mathbf{p1a}, self(), b\rangle)$;
  for ever
    switch $receive()$
      case $\langle \mathbf{p1b}, \alpha, b', r\rangle$ :
        if $b' = b$ then
          $pvalues := pvalues \cup r$;
          $waitfor := waitfor - \{\alpha\}$;
          if $|waitfor| < |acceptors|/2$ then
            $send(\lambda, \langle \mathbf{adopted}, b, pvalues\rangle)$;
            $exit()$;
          end if;
        else
          $send(\lambda, \langle \mathbf{preempted}, b'\rangle)$;
          $exit()$;
        end if;
      end case
    end switch
  end for
end process
```

(b)

Figure 3: (a) Pseudo code for a commander. Here λ is the identifier of its leader, *acceptors* the set of acceptor identifiers, *replicas* the set of replicas, and $\langle b, s, c\rangle$ the pvalue the commander is responsible for. (b) Pseudo code for a scout. Here λ is the identifier of its leader, *acceptors* the identifiers of the acceptors, and b the desired ballot number.

- $\lambda.active$: a boolean flag, initially `false`; and
- $\lambda.proposals$: a map of slot numbers to proposed commands in the form of a set of $\langle$*slot number*, *command*$\rangle$ pairs, initially empty. At any time, there is at most one entry per slot number in the set.

The leader starts by spawning a scout for its initial ballot number, and then enters into a loop awaiting messages. There are three types of messages that cause transitions:

- $\langle \mathbf{propose}, s, c\rangle$: A replica proposes command c for slot number s;
- $\langle \mathbf{adopted}, ballot\_num, pvals\rangle$: Sent by a scout, this message signifies that the current ballot number *ballot_num* has been adopted by a majority of acceptors. (If an **adopted** message arrives for an old ballot number, it is ignored.) The set *pvals* contains all pvalues accepted by these acceptors prior to *ballot_num*.
- $\langle \mathbf{preempted}, \langle r', \lambda'\rangle\rangle$: Sent by either a scout or a commander, it means that some acceptor has adopted $\langle r', \lambda'\rangle$. If $\langle r', \lambda'\rangle > ballot\_num$, it may no longer be possible to use ballot *ballot_num* to choose a command.

A leader goes between *passive* and *active* modes. When passive, the leader is waiting for an $\langle \mathbf{adopted}, ballot\_num, pvals\rangle$ message from the last scout that it spawned. When this message arrives,

```
process $Leader(acceptors, replicas)$
    var $ballot\_num = (0, self()), active =$ false, $proposals = \emptyset$;

    $spawn(Scout(self(), acceptors, ballot\_num))$;
    for ever
        switch $receive()$
            case $\langle \textbf{propose}, s, c\rangle$ :
                if $\nexists c' : \langle s, c'\rangle \in proposals$ then
                    $proposals := proposals \cup \{\langle s, c\rangle\}$;
                    if $active$ then
                        $spawn(Commander(self(), acceptors, replicas, \langle ballot\_num, s, c\rangle))$;
                    end if
                end if
            end case
            case $\langle \textbf{adopted}, ballot\_num, pvals\rangle$ :
                $proposals := proposals \lhd pmax(pvals)$;
                $\forall\langle s, c\rangle \in proposals : spawn(Commander(self(), acceptors, replicas, \langle ballot\_num, s, c\rangle))$;
                $active :=$ true;
            end case
            case $\langle \textbf{preempted}, \langle r', \lambda'\rangle\rangle$ :
                if $(r', \lambda') > ballot\_num$ then
                    $active :=$ false;
                    $ballot\_num := (r' + 1, self())$;
                    $spawn(Scout(self(), acceptors, ballot\_num))$;
                end if
            end case
        end switch
    end for
end process
```

Figure 4: Pseudo code skeleton for a leader. Here *acceptors* is the set of acceptor identifiers, and *replicas* the set of replica identifiers.

the leader becomes active and spawns commanders for each of the slots for which it has a proposed command, but must select commands that satisfy Invariant C2. We will now consider how the leader goes about this.

When active, the leader knows that a majority of acceptors, say $\mathcal{A}$, have adopted *ballot_num* and thus no longer accept pvalues for ballot numbers less than *ballot_num* (because of Invariants A1 and A2). In addition, it has all pvalues accepted by the acceptors in $\mathcal{A}$ prior to *ballot_num*. There are two cases to consider:

1. If, for some slot s, there is no pvalue in *pvals*, then, prior to *ballot_num*, it is not possible that any pvalue has been chosen or will be chosen for slot s. After all, suppose that some pvalue $\langle b, s, c\rangle$ were chosen, with $b < ballot\_num$. This would require a majority of acceptors $\mathcal{A}'$ to accept $\langle b, s, c\rangle$, but we have responses from a majority $\mathcal{A}$ that have adopted *ballot_num* and have not accepted, nor can accept, pvalues with a ballot number smaller than *ballot_num* (Invariants A1 and A2). Because both $\mathcal{A}$ and $\mathcal{A}'$ are majorities, $\mathcal{A} \cap \mathcal{A}'$ is non-empty—some acceptor in the

intersection must have violated invariant A1, A2, or A3, which we know cannot happen. Because no pvalue has been or will be chosen for slot s prior to *ballot_num*, the leader can propose any command for that slot without causing a conflict on an earlier ballot, thus enforcing Invariant C2.

2. Otherwise, let $\langle b, s, c\rangle$ be the pvalue with the maximum ballot number for slot s. Because of Invariant A4, this pvalue is unique—there cannot be two different commands for the same ballot number and slot number. Also note that $b <$ *ballot_num* (because acceptors only report pvalues they accepted before *ballot_num*). Like the leader of *ballot_num*, the leader of b must have picked c carefully to ensure that Invariant C2 holds, and thus if a pvalue is chosen before or at b, the command it contains must be c. Since all acceptors in $\mathcal{A}$ have adopted *ballot_num*, no pvalues between b and *ballot_num* can be chosen (Invariants A1 and A2). Thus, by using c as a command, λ enforces Invariant C2.

This inductive argument is the crux for the correctness of the Synod protocol. It demonstrates that Invariant C2 holds, which in turn implies Invariant A5, which in turn implies Invariant R1 that ensures that all replicas apply the same operations in the same order.

Back to the code, after the leader receives $\langle$**adopted**, *ballot_num*, *pvals*$\rangle$, it determines for each slot the command corresponding to the maximum ballot number in *pvals* by invoking the function *pmax*. Formally, the function *pmax*(*pvals*) is defined as follows:

$$\begin{aligned} \mathit{pmax}(\mathit{pvals}) \equiv \{\langle s,c\rangle \mid \exists b : \langle b,s,c\rangle \in \mathit{pvals} \wedge \\ \forall b', c' : \langle b', s, c'\rangle \in \mathit{pvals} \Rightarrow b' \le b \ \} \end{aligned}$$

The update operator $\lhd$ applies to two sets of proposals. $x \lhd y$ returns the elements of y as well as the elements of x that are not in y. Formally:

$$\begin{aligned} x \lhd y \quad \equiv \quad & \{\langle s,c\rangle \mid \langle s,c\rangle \in y \ \vee \\ & (\langle s,c\rangle \in x \wedge \nexists c' : \langle s,c'\rangle \in y)\} \end{aligned}$$

Thus the line *proposals* := *proposals*$\lhd$*pmax*(*pvals*); updates the set of proposals, replacing for each slot number the command corresponding to the maximum pvalue in *pvals*, if any. Now the leader can start commanders for each slot while satisfying Invariant C2.

If a new proposal arrives while the leader is active, the leader checks to see if it already has a proposal for the same slot (and has thus spawned a commander for that slot) in its set *proposals*. If not, the new proposal will satisfy Invariant C2, and thus the leader adds the proposal to *proposals* and spawns a commander.

If either a scout or a commander detects that an acceptor has adopted a ballot number b, with $b >$ *ballot_num*, then it sends the leader a `preempted` message. The leader becomes passive and spawns a new scout with a ballot number that is higher than b.

Figure 5 shows an example of a leader λ_2 spawning a scout to become active, and a client sending a request to a replica, which in turns sends a proposal to the leader.

3 When Paxos Works

It would clearly be desirable that, if a client broadcasts a new command to all replicas, that it eventually receives at least one response. This is often referred to as *liveness*. It requires that if one or more commands have been proposed for a particular slot, that some command is eventually decided for that slot. Unfortunately, the Synod protocol as described does not guarantee this, even in the absence of any failure whatsoever.[4]

Consider the following scenario, with two leaders with identifiers λ and λ' such that $\lambda < \lambda'$. Both start at the same time, respectively proposing commands c and c' for slot number 1. Suppose there are three acceptors, α_1, α_2, and α_3. In ballot $\langle 0, \lambda\rangle$, leader λ is successful in getting α_1 and α_2 to adopt the ballot, and α_1 to accept pvalue $\langle\langle 0,\lambda\rangle, 1, c\rangle$.

Now leader λ' gets α_2 and α_3 to adopt ballot $\langle 0, \lambda'\rangle$ (which is after $\langle 0, \lambda\rangle$ because $\lambda < \lambda'$). Note that neither α_2 or α_3 accepted any pvalues, so leader λ' is free to select any proposal. Leader λ' then gets α_3 to accept $\langle\langle 0,\lambda'\rangle, 1, c'\rangle$.

[4]In fact, failures tend to be good for liveness. If all leaders but one fail, Paxos is guaranteed to terminate.

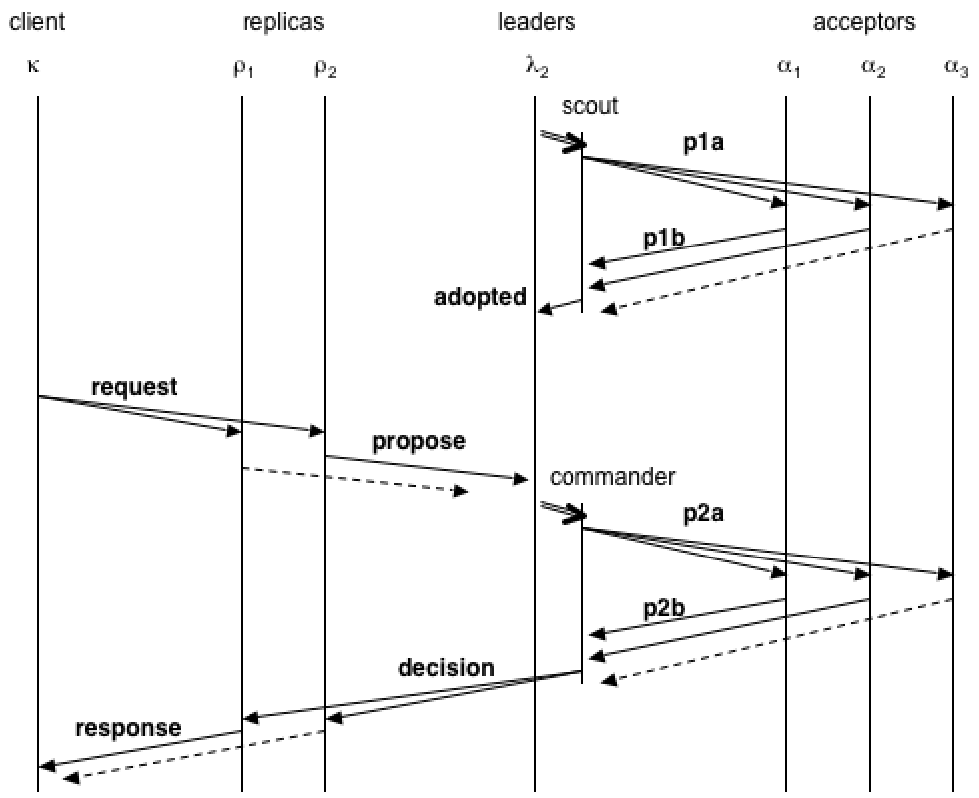

Figure 5: The time diagram shows a client, two replicas, a leader (with a scout and a commander), and three acceptors, with time progressing downward. Arrows represent messages. Dashed arrows are messages that end up being ignored. The leader first runs a scout in order to become active. Later, when a replica proposes a command (in response to a client's request), the leader runs a commander, which notifies the replicas upon learning a decision.

At this point, acceptors α_2 and α_3 are unable to accept $\langle\langle 0, \lambda\rangle, 1, c\rangle$ and thus leader λ is unable to get a majority of acceptors to accept $\langle\langle 0, \lambda\rangle, 1, c\rangle$. Trying again, leader λ gets α_1 and α_2 to adopt $\langle 1, \lambda\rangle$. The maximum pvalue accepted by α_1 and α_2 is $\langle\langle 0, \lambda\rangle, 1, c\rangle$, and thus λ must propose c. Suppose λ gets α_1 to accept $\langle\langle 1, \lambda\rangle, 1, c\rangle$.

Because acceptors α_1 and α_2 adopted $\langle 1, \lambda\rangle$, they are unable to accept $\langle\langle 0, \lambda'\rangle, 1, c'\rangle$. Trying to make progress, leader λ' gets α_2 and α_3 to adopt $\langle 1, \lambda'\rangle$, and gets α_3 to accept $\langle\langle 1, \lambda'\rangle, 1, c'\rangle$.

This ping-pong scenario can be continued indefinitely, with no ballot ever succeeding in choosing a pvalue. This is true even if $c = c'$, that is, the leaders propose the same command. The well-known "FLP impossibility result" [8] demonstrates that in an asynchronous environment that admits crash failures, no consensus protocol can guarantee termination, and the Synod protocol is no exception. The argument does not apply directly if transitions have non-deterministic actions—for example changing state in a randomized manner. However, it can be demonstrated that such protocols cannot guarantee a decision either.

If we could somehow guarantee that some leader would be able to work long enough to get a majority of acceptors to adopt a high ballot and also accept a pvalue, then Paxos would be guaranteed to choose a proposed command. A possible approach could be as follows: when a leader λ discovers (through a `preempted` message) that there is a higher ballot with leader λ' active, rather than starting a new scout with an even higher ballot number, it starts monitoring the leader of b by pinging it on a regular basis. As long as λ' responds timely to pings, leader λ waits patiently. Only if λ' stops responding will λ select a higher ballot number and start a scout.

This concept is called *failure detection*, and theoreticians have been interested in the weakest properties failure detection should have in order to support a consensus algorithm that is guaranteed to terminate [7]. In a purely asynchronous environment it is impossible to determine through pinging or any other method whether a particular leader has crashed or is simply slow. However, under fairly weak assumptions about timing, we can design a version of Paxos that is guaranteed to choose a proposal. In particular, we will assume that both the following are bounded:

- the clock drift of a process, that is, the rate of its clock is within some factor of the rate of real-time;
- the time between when a non-faulty process initiates sending a message, and the message having been received and handled by a non-faulty destination process.

We do not need to assume that we know what those bounds are—only that such bounds exist. From a practical point of view, this seems entirely reasonable. Modern clocks progress certainly within a factor of 2 of real-time. A message between two non-faulty processes is likely delivered and processed within a year, say.

These assumptions can be exploited as follows: we use a scheme similar to the one described above, based on pinging and timeouts, but the value of the timeout interval depends on the ballot number: the higher the competing ballot number, the longer a leader waits before trying to preempt it with a higher ballot number. Eventually the timeout at each of the leaders becomes so high that some correct leader will always be able to get its proposals chosen.

For good performance, one would like the timeout period to be long enough so that a leader can be successful, but short enough so that the ballots of a faulty leader are preempted as quickly as possible. This can be achieved with a TCP-like AIMD (Additive Increase, Multiplicative Decrease) approach for choosing timeouts. The leader associates an initial timeout with each ballot. If a ballot gets preempted, the next ballot uses a timeout that is multiplied by some small factor larger than one. With each chosen proposal, this initial timeout is decreased linearly. Eventually the timeout will become too short, and the ballot replaced with another even if its leader is non-faulty, but this does not affect correctness.

(As an aside: some people call this process *leader election* or *weak leader election*. This is, however, confusing, as each ballot has a fixed leader that is not elected.)

For further improved liveness, crashes should be avoided. The Paxos protocol can tolerate a minority of its acceptors failing, and all but one of its replicas and leaders failing. If more than that fail, consistency is still guaranteed, but liveness will be violated. For this reason, one may want to keep the state of acceptors replicas, and leaders on disk. A process that suffers from a power failure but can recover from disk is not theoretically considered crashed—it is simply slow for a while. Only a process that suffers a permanent disk failure would be considered crashed.

For completeness, we note that for liveness we also rely on reliable communication: messages between correct processes are eventually delivered, typically implemented by periodic retransmission until an ack is received. In particular, a correct client retransmits its request to replicas until it has received a response. Because there are at least $f + 1$ replicas, at least one of those replicas will not fail and will assign a slot to the request and send a proposal to the $f + 1$ or more leaders. Thus at least one correct leader will try to get a command decided for that slot. Should a competing command get decided, the replica will reassign the request to a new slot and so on. While this may lead to starvation, in the absence of new requests any outstanding request will eventually get decided in at least one slot.

4 Paxos Made Pragmatic

We have described a relatively simple version of the Paxos protocol with the intention to make it understandable, but the described protocol is not practical. The state of the various components, as well as the contents of `p1b` messages, grows much too quickly. This section presents various optimizations and design decisions.

4.1 State Reduction

First note that although a leader obtains a set of all accepted pvalues from a majority of acceptors, it only needs to know, for each slot, if this set is empty or not, and if not, what the maximum pvalue is. Thus, a large step toward practicality is that acceptors only maintain the most recently accepted pvalue for each slot ($\perp$ if no pvalue has been accepted) and return only these pvalues in a `p1b` message to the scout. This gives the leader all information needed to enforce Invariant C2.

While this optimization affects none of the presented invariants, it leads to an worrisome effect. We know that when a majority of acceptors have accepted the same pvalue $\langle b, s, c\rangle$, then proposal c is chosen for slot s. Consider now the following scenario. Suppose (as in the example of Section 3) there are two leaders λ and λ' such that $\lambda < \lambda'$, and there are three acceptors α_1, α_2 and α_3. Acceptors α_1 and α_2 accept $\langle\langle 0, \lambda\rangle, 1, c\rangle$, and thus proposal c is chosen for slot 1 by ballot $\langle 0, \lambda\rangle$. However, leader λ crashes before learning this. Now leader λ' gets acceptors α_2 and α_3 to adopt ballot $\langle 0, \lambda'\rangle$. After determining the maximum pvalue among the responses, leader λ' has to select proposal c. Now say that acceptor α_2 accepts $\langle\langle 0, \lambda'\rangle, 1, c\rangle$.

At this point there is no majority of acceptors that have the same most recently accepted pvalue, and in fact no proof that ballot $\langle 0, \lambda\rangle$ even chose proposal c as that part of the history has been overwritten. This appears to be a problem. However, any ballot b after $\langle 0, \lambda\rangle$ can only select $\langle b, s, c\rangle$ (by Invariant C2) and thus there can be no inconsistency.[5]

Another large amount of overhead can be avoided if leaders keep track of which slots have already been decided. Before turning active, a leader can include on the `p1a` request the first slot for which it does not know the decision. Acceptors do not need to respond with pvalues for smaller slot numbers. Upon turning active, a leader does not need to start commanders for slots for which it knows the decision, and also does not need to maintain proposals for such slots. Leaders can learn which slots have been decided from their co-located commanders, or alternatively from replicas in case leaders and replicas are co-located (see Section 4.3).

Also note that the set *requests* maintained by a replica only needs to contain those requests for slot numbers higher than *slot_num*.

4.2 Garbage Collection

When *all* replicas have learned that some slot has been decided, then it is no longer necessary for an acceptor to maintain the corresponding pvalues in its *accepted* set. To enable garbage collection, replicas could respond to leaders when they have performed a command, and upon a leader learning that all replicas have done so it could notify the acceptors to release the corresponding state.

The state of an acceptor would have to be extended with a new variable that contains a slot number: all pvalues lower than that slot number have been garbage collected. This slot number must be included in `p1b` messages so that another leader does not mistakenly conclude that the acceptors have not accepted any pvalues for those slots.

This garbage collection technique does not work if one of the replicas is faulty or exceedingly slow, leaving the acceptors no other option but to maintain state for all slots, just in case the replica recovers and needs to learn the decisions it missed. A solution is to use $2f + 1$ or more replicas instead of just $f + 1$. Acceptor state is garbage collected when more than f replicas have performed a command. If, because of garbage collection, a replica is not able to learn the decision for a slot that it needs, then it can always

[5]It might be tempting to conclude that Paxos has decided a proposal c as soon as a majority of acceptors have most recently accepted a pvalue containing c, possibly on different ballots. However, such a conclusion would be wrong.

obtain a snapshot of the state from another replica that has learned the operation and continue on.

Another solution is to make the set of replicas dynamic, by having replicas themselves keep track of which $f + 1$ replicas are currently active. A special command is used to change the set of replicas in case one or more are suspected of having failed. Once all replicas in a configuration of replicas have learned the decision for a slot, the corresponding acceptor state can be garbage collected (as can be the state maintained by replicas in old configurations).[6]

4.3 Co-location

In practice, leaders are typically co-located with replicas. That is, each machine that runs a replica also runs a leader. This leads to some useful optimizations. A client sends its proposals to replicas. If co-located, the replica can send a proposal for a particular slot to its local leader, say λ, rather than broadcasting the request to all leaders. If λ is passive, monitoring another leader λ', it forwards the proposal to λ'. If λ is active, it will start a commander.

An alternative not often considered is to have clients and leaders be co-located instead of replicas and leaders. Thus, each client runs a local leader. By doing so, one obtains a protocol that is much like Quorum Replication [24, 2]. While traditional quorum replication protocols can only support read and write operations, this Paxos version of quorum replication could support arbitrary (deterministic) operations. However, this approach would place a lot of trust in clients for both integrity and liveness and is therefore not popular.

Replicas are also often co-located with acceptors. As shown in Section 4.2, one may need as many replicas as acceptors in any case. When leaders are co-located with acceptors, one has to be careful that they use separate ballot number variables.

4.4 Read-only Commands

The Paxos protocol does not treat read-only commands any different from other commands, and this leads to more overhead than necessary. One would however be naive in thinking that a client that wants to do a read-only command could simply query one of the replicas—doing so would easily violate consistency as the selected replica may have stale application state.

Therefore, read-only commands are typically sent to an active leader just like update commands. One simple optimization is for a leader to send a chosen read-only command to only a single replica instead of to all replicas.[7] After all, the state of none of the replicas needs to change, but one of the replicas has to compute the result and send it to the client. (One has to consider that the selected replica could be faulty and does not send a result to the client. When the leader receives a request that has already been decided, it can broadcast the request to all replicas.)

A read-only command does not actually require that acceptors accept a pvalue at all, but the problem is that the leader cannot know if its ballot is current and no other leader has taken over and gotten new commands decided. To learn this, a leader would have to run a scout and wait for an `adopted` message. While this avoids unnecessary accepts, running a view change for each read-only command is an expensive proposition.

A pragmatic solution involves so-called *leases* [9, 14]. Leases require an additional assumption on timing, which is that there is a *known* bound on clock drift. For simplicity, we will assume that there is no clock drift whatsoever, but the idea is easily generalized if the bound on clock drift is given.

Before a leader sends a `p1a` request to the acceptors, it records the time. The leader includes in the `p1a` request a *lease period*. For example, the lease period could be "10 seconds." An acceptor that adopts the ballot number promises not to adopt another (higher) ballot number until the lease period expires (measured on its local clock from the time the acceptor received the `p1a` request). If a majority of acceptors accept the ballot, the leader can be certain that from the recorded time, until the lease period expires on its own clock, no other leader can preempt its ballot, and thus it is impossible that other leaders introduce update commands.

Knowing that its ballot is current, a leader can send read-only commands to one of the replicas (prefer-

[6]See [3] for a full treatment of reconfiguration in Paxos.

[7]For this to work, a leader needs to be able to recognize read-only commands.

ably a local one), although it has to wait until any outstanding update commands have decided. The leasing technique can be integrated with the adaptive timeout technique described in Section 3.

5 Exercises

This paper is accompanied by a Java package (see Appendix) that contains a Java implementation for each of the pseudo-codes presented in this paper. Below find a list of suggestions for exercises using this code.

1. Implement the state reduction techniques for acceptors and `p1b` messages described in Section 4.1.

2. In the current implementation, ballot numbers are pairs of round numbers and leader process identifiers. If the set of leaders is fixed and ordered, then we can simplify ballot numbers. For example, if the leaders are $\{\lambda_1, ..., \lambda_n\}$, then the ballot numbers for leader λ_i could be $i, i+n, i+2n, ...$ Ballot number $\perp$ could be represented as 0. Modify the Java code accordingly.

3. Implement a simple replicated bank application. The bank service maintains a set of client records, a set of accounts (a client can have zero or more accounts), and operations such as deposit, withdraw, transfer, inquiry.

4. In the Java implementation, all processes run as threads within the same Java machine, and communicate using message queues. Allow processes to run in different machines and have them communicate over TCP connections. Hint: do not consider TCP connections as reliable. If they break, have them periodically try to re-connect until successful.

5. Implement the failure detection scheme of Section 3 so that most of the time only one leader is active.

6. Co-locate leaders and replicas as suggested in Section 4.3, and garbage collect unnecessary leader state, that is, leaders can forget about proposals for commands numbers that have already been decided. Upon becoming active, leaders do not have to start commanders for such slots either.

7. In order to increase fault tolerance, the state of acceptors and leaders can be kept on stable storage (disk). This would allow such processes to recover from crashes. Implement this. Take into consideration that a process may crash part-way during saving its state.

8. Acceptors can garbage collect pvalues for decided commands that have been learned by all replicas. Implement this.

9. Implement the leasing scheme to optimize read-only operations as suggested in Section 4.4.

6 Conclusion

In this paper we presented Paxos as a collection of five kinds of processes, each with a simple operational specification. We started with an impractical but relatively easy implementation to simplify comprehension, and then showed how various aspects can be improved to render a practical protocol.

The paper is the next in a long line of papers that describe the Paxos protocol or the experience of implementing it. A partial list follows. The Viewstamped Replication protocol by Oki and Liskov [20], similar to multi-decree Paxos, was published in 1988. Leslie Lamport's Part-time Parliament paper [14] was first written in 1989, but not published until 1998. Butler Lampson wrote a paper explaining the Part-time Parliament paper using pseudocode in 1996 [16], and in 2001 gives an invariant-based description of various variants of single-decree Paxos [17]. In 2000, De Prisco, Lampson, and Lynch present an implementation of Paxos in the General Timed Automaton formalism [21]. Lamport wrote "Paxos made Simple" in 2001, giving a simple invariant-based explanation of the protocol [15]. In their 2003 presentation, Boichat et al. [4] give a formal account of various variants of Paxos along with extensive pseudo-code. Chandra et al. describe Google's challenges in implementing Paxos in

2007 [6].[8] Also in 2007, Li et al. give a novel simplified presentation of Paxos using a write-once register [18]. In his 2007 report "Paxos made Practical" [19], David Mazières gives details of how to build replicated services using Paxos. Kirch and Amir describe their 2008 Paxos implementation experiences in [12]. Alvaro et al., in 2009 [1], describe implementing Paxos in Overlog, a declarative language.

Acknowledgments

Many people provided invaluable input to this description of Paxos, including Deniz Altinbuken, Lorenzo Alvisi, Mark Bickford, Robert Constable, Danny Dolev, David Guaspari, Chi Ho, Annie Liu, Fred Schneider, Emin Gun Sirer, and Scott Stoller.

References

[1] P. Alvaro, T. Condie, N. Conway, J.M. Hellerstein, and R.C. Sears. I Do Declare: Consensus in a logic language. In *Proceedings of the SOSP Workshop on Networking Meets Databases (NetDB)*, 2009.

[2] H. Attiya, A. Bar Noy, and D. Dolev. Sharing memory robustly in message passing systems. *Journal of the ACM*, 42(1):121–132, 1995.

[3] Ken Birman, Dahlia Malkhi, and Robbert van Renesse. Virtually synchronous methodology for dynamic service replication. Technical Report MSR-TR-2010-151, Microsoft Research, 2010.

[4] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Deconstructing Paxos. *ACM SIGACT News*, 34(1), March 2003.

[5] M. Burrows. The Chubby Lock Service for loosely-coupled distributed systems. In *7th Symposium on Operating System Design and Implementation*, Seattle, WA, November 2006.

[6] T.D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proc. of the 26th ACM Symp. on Principles of Distributed Computing*, pages 398–407, Portland, OR, May 2007. ACM.

[7] T.D. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. In *Proc. of the 11th ACM Symp. on Principles of Distributed Computing*, pages 325–340, Montreal, Quebec, August 1991. ACM SIGOPS-SIGACT.

[8] M.J. Fischer, N.A. Lynch, and M.S. Patterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[9] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. of the Twelfth ACM Symp. on Operating Systems Principles*, pages 202–210, Litchfield Park, AZ, November 1989.

[10] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463 – 492, 1990.

[11] F. Junqueira, P. Hunt, M. Konar, and B. Reed. The ZooKeeper Coordination Service (poster). In *Symposium on Operating Systems Principles (SOSP)*, 2009.

[12] J. Kirsch and Y. Amir. Paxos for system builders. Technical Report CNDS-2008-2, Johns Hopkins University, 2008.

[13] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.

[14] L. Lamport. The part-time parliament. *Trans. on Computer Systems*, 16(2):133–169, 1998.

[15] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, 2001.

[16] B. Lampson. How to build a highly available system using consensus. In O. Babaoglu and K. Marzullo, editors, *Distributed Algorithms*,

[8] There is a subtle bug in their implementation—see if you can spot it. (Courtesy Edward Yang.)

volume 115 of *Lecture Notes on Computer Science*, pages 1–17. Springer-Verlag, 1996.

[17] B.W. Lampson. The ABCD's of Paxos. In *Proc. of the 20th ACM Symp. on Principles of Distributed Computing*, page 13, Newport, RI, 2001. ACM Press.

[18] H.C. Li, A. Clement, A. S. Aiyer, and L. Alvisi. The Paxos register. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 07)*, 2007.

[19] D. Mazières. Paxos made practical. Technical Report on the web at scs.stanford.edu/˜dm/home/papers/paxos.pdf, Stanford University, 2007.

[20] B.M. Oki and B.H. Liskov. Viewstamped replication: A general primary-copy method to support highly-available distributed systems. In *Proc. of the 7th ACM Symp. on Principles of Distributed Computing*, pages 8–17, Toronto, Ontario, August 1988. ACM SIGOPS-SIGACT.

[21] R. De Prisco, B. Lampson, and N. Lynch. Revisiting the Paxos algorithm. *Theoretical Computer Science*, 243(1-2):35–91, July 2000.

[22] R.D. Schlichting and F.B. Schneider. Failstop processors: an approach to designing fault-tolerant computing systems. *Trans. on Computer Systems*, 1(3):222–238, August 1983.

[23] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[24] R.H. Thomas. A solution to the concurrency control problem for multiple copy databases. In *Proc. of COMPCON 78 Spring*, pages 88–93, Washington, D.C., February 1978. IEEE Computer Society.

Appendix: Java Source Code Listing

Java source code corresponding to the pseudo-code in this paper is available at http://www.cs.cornell.edu/home/rvr/Paxos/. After compilation, the code can be run using the command "java Env." For archival purposes we also include the code below.

ProcessId.java

A process identifier is simply a string.

```
public class ProcessId implements Comparable {
  String name;

  public ProcessId(String name){ this.name = name; }

  public boolean equals(Object other){
    return name.equals(((ProcessId) other).name);
  }

  public int compareTo(Object other){
    return name.compareTo(((ProcessId) other).name);
  }

  public String toString(){ return name; }
}
```

Command.java

A command consists of the process identifier of the client submitting the request, a client-local request identifier, and an operation (which can be anything).

```
public class Command {
  ProcessId client;
  int req_id;
  Object op;

  public Command(ProcessId client, int req_id, Object op){
    this.client = client;
    this.req_id = req_id;
    this.op = op;
  }

  public boolean equals(Object o) {
    Command other = (Command) o;
    return client.equals(other.client) && req_id == other.req_id && op.equals(other.op);
  }

  public String toString(){
    return "Command(" + client + ",␣" + req_id + ",␣" + op + ")";
  }
}
```

BallotNumber.java

A ballot number is a lexicographically ordered pair of an integer and the identifier of the ballot's leader.

```
public class BallotNumber implements Comparable {
  int round;
  ProcessId leader_id;

  public BallotNumber(int round, ProcessId leader_id){
    this.round = round;
    this.leader_id = leader_id;
  }

  public boolean equals(Object other){
    return compareTo(other) == 0;
  }

  public int compareTo(Object other){
    BallotNumber bn = (BallotNumber) other;
    if (bn.round != round) {
      return round - bn.round;
    }
    return leader_id.compareTo(bn.leader_id);
  }

  public String toString(){
    return "BN(" + round + ",␣" + leader_id + ")";
  }
}
```

PValue.java

A pvalue consists of a ballot number, a slot number, and a command.

```
public class PValue {
  BallotNumber ballot_number;
  int slot_number;
  Command command;

  public PValue(BallotNumber ballot_number, int slot_number,
                    Command command){
    this.ballot_number = ballot_number;
    this.slot_number = slot_number;
    this.command = command;
  }

  public String toString(){
    return "PV(" + ballot_number + ",␣" + slot_number + ",␣" + command + ")";
  }
}
```

PaxosMessage.java

Paxos uses a large variety of message types. They are collected below.

```
import java.util.*;

public class PaxosMessage {
  ProcessId src;
}

class P1aMessage extends PaxosMessage {
  BallotNumber ballot_number;
  P1aMessage(ProcessId src, BallotNumber ballot_number){
    this.src = src; this.ballot_number = ballot_number;
} }
class P1bMessage extends PaxosMessage {
  BallotNumber ballot_number; Set<PValue> accepted;
  P1bMessage(ProcessId src, BallotNumber ballot_number, Set<PValue> accepted) {
    this.src = src; this.ballot_number = ballot_number; this.accepted = accepted;
} }
class P2aMessage extends PaxosMessage {
  BallotNumber ballot_number; int slot_number; Command command;
  P2aMessage(ProcessId src, BallotNumber ballot_number, int slot_number, Command command){
    this.src = src; this.ballot_number = ballot_number;
    this.slot_number = slot_number; this.command = command;
} }
class P2bMessage extends PaxosMessage {
  BallotNumber ballot_number; int slot_number;
  P2bMessage(ProcessId src, BallotNumber ballot_number, int slot_number){
    this.src = src; this.ballot_number = ballot_number; this.slot_number = slot_number;
} }
class PreemptedMessage extends PaxosMessage {
  BallotNumber ballot_number;
  PreemptedMessage(ProcessId src, BallotNumber ballot_number){
    this.src = src; this.ballot_number = ballot_number;
} }
class AdoptedMessage extends PaxosMessage {
  BallotNumber ballot_number; Set<PValue> accepted;
  AdoptedMessage(ProcessId src, BallotNumber ballot_number, Set<PValue> accepted){
    this.src = src; this.ballot_number = ballot_number; this.accepted = accepted;
} }
class DecisionMessage extends PaxosMessage {
  ProcessId src; int slot_number; Command command;
  public DecisionMessage(ProcessId src, int slot_number, Command command){
    this.src = src; this.slot_number = slot_number; this.command = command;
} }
class RequestMessage extends PaxosMessage {
  Command command;
  public RequestMessage(ProcessId src, Command command){
    this.src = src; this.command = command;
} }
class ProposeMessage extends PaxosMessage {
  int slot_number; Command command;
  public ProposeMessage(ProcessId src, int slot_number, Command command){
    this.src = src; this.slot_number = slot_number; this.command = command;
} }
```

Queue.java

This implementation does not use a real network. Messages are transmitted on queues.

```
import java.util.*;

 public class Queue<T> {
  LinkedList<T> ll = new LinkedList<T>();

  public synchronized void enqueue(T obj){
    ll.add(obj);
    notify();
  }

  public synchronized T bdequeue(){
    while (ll.size() == 0) {
      try { wait(); } catch (InterruptedException e) {}
    }
    return ll.removeFirst();
  }
}
```

Process.java

A process is a thread with a process identifier, a queue of incoming messages, and an "environment" that keeps track of all processes and queues.

```
public abstract class Process extends Thread {
  ProcessId me;
  Queue<PaxosMessage> inbox = new Queue<PaxosMessage>();
  Env env;

  abstract void body();

  public void run(){
    body();
    env.removeProc(me);
  }

  PaxosMessage getNextMessage(){
    return inbox.bdequeue();
  }

  void sendMessage(ProcessId dst, PaxosMessage msg){
    env.sendMessage(dst, msg);
  }

  void deliver(PaxosMessage msg){
    inbox.enqueue(msg);
  }
}
```

Replica.java

This is the Java code for a replica, corresponding to Figure 1. This code does not maintain any application state, however.

```
import java.util.*;

public class Replica extends Process {
  ProcessId[] leaders;
  int slot_num = 1;
  Map<Integer /* slot number */, Command> proposals = new HashMap<Integer, Command>();
  Map<Integer /* slot number */, Command> decisions = new HashMap<Integer, Command>();

  public Replica(Env env, ProcessId me, ProcessId[] leaders){
    this.env = env;
    this.me = me;
    this.leaders = leaders;
    env.addProc(me, this);
  }

  void propose(Command c){
    if (!decisions.containsValue(c)) {
      for (int s = 1;; s++) {
        if (!proposals.containsKey(s) && !decisions.containsKey(s)) {
          proposals.put(s, c);
          for (ProcessId ldr: leaders) {
            sendMessage(ldr, new ProposeMessage(me, s, c));
          }
          break;
        }
      }
    }
  }

  void perform(Command c){
    for (int s = 1; s < slot_num; s++) {
      if (c.equals(decisions.get(s))) {
        slot_num++;
        return;
      }
    }
    System.out.println("" + me + ":␣perform␣" + c);
    slot_num++;
  }
  public void body(){
    System.out.println("Here␣I␣am:␣" + me);
    for (;;) {
      PaxosMessage msg = getNextMessage();

      if (msg instanceof RequestMessage) {
        RequestMessage m = (RequestMessage) msg;
        propose(m.command);
      }

      else if (msg instanceof DecisionMessage) {
        DecisionMessage m = (DecisionMessage) msg;
        decisions.put(m.slot_number, m.command);
        for (;;) {
          Command c = decisions.get(slot_num);
          if (c == null) {
            break;
          }
          Command c2 = proposals.get(slot_num);
          if (c2 != null && !c2.equals(c)) {
            propose(c2);
          }
          perform(c);
        }
      }
      else {
        System.err.println("Replica:␣unknown␣msg␣type");
      }
    }
  }
}
```

Acceptor.java

Implementation of the acceptor process, closely corresponding to Figure 2.

```
import java.util.*;

public class Acceptor extends Process {
  BallotNumber ballot_number = null;
  Set<PValue> accepted = new HashSet<PValue>();

  public Acceptor(Env env, ProcessId me){
    this.env = env;
    this.me = me;
    env.addProc(me, this);
  }

  public void body(){
    System.out.println("Here␣I␣am:␣" + me);
    for (;;) {
      PaxosMessage msg = getNextMessage();

      if (msg instanceof P1aMessage) {
        P1aMessage m = (P1aMessage) msg;

        if (ballot_number == null ||
            ballot_number.compareTo(m.ballot_number) < 0) {
          ballot_number = m.ballot_number;
        }
        sendMessage(m.src, new P1bMessage(me, ballot_number, new HashSet<PValue>(accepted)));
      }
      else if (msg instanceof P2aMessage) {
        P2aMessage m = (P2aMessage) msg;

        if (ballot_number == null ||
            ballot_number.compareTo(m.ballot_number) <= 0) {
          ballot_number = m.ballot_number;
          accepted.add(new PValue(ballot_number, m.slot_number, m.command));
        }
        sendMessage(m.src, new P2bMessage(me, ballot_number, m.slot_number));
      }
    }
  }
}
```

Commander.java

Implementation of the commander process in Figure 3(a).

```
import java.util.*;

public class Commander extends Process {
  ProcessId leader;
  ProcessId[] acceptors, replicas;
  BallotNumber ballot_number;
  int slot_number;
  Command command;

  public Commander(Env env, ProcessId me, ProcessId leader, ProcessId[] acceptors,
      ProcessId[] replicas, BallotNumber ballot_number, int slot_number, Command command){
    this.env = env;
    this.me = me;
    this.acceptors = acceptors;
    this.replicas = replicas;
    this.leader = leader;
    this.ballot_number = ballot_number;
    this.slot_number = slot_number;
    this.command = command;
    env.addProc(me, this);
  }

  public void body(){
    P2aMessage m2 = new P2aMessage(me, ballot_number, slot_number, command);
    Set<ProcessId> waitfor = new HashSet<ProcessId>();
    for (ProcessId a: acceptors) {
      sendMessage(a, m2);
      waitfor.add(a);
    }

    while (2 * waitfor.size() >= acceptors.length) {
      PaxosMessage msg = getNextMessage();

      if (msg instanceof P2bMessage) {
        P2bMessage m = (P2bMessage) msg;

        if (ballot_number.equals(m.ballot_number)) {
          if (waitfor.contains(m.src)) {
            waitfor.remove(m.src);
          }
        }
        else {
          sendMessage(leader, new PreemptedMessage(me, m.ballot_number));
          return;
        }
      }
    }

    for (ProcessId r: replicas) {
      sendMessage(r, new DecisionMessage(me, slot_number, command));
    }
  }
}
```

Scout.java

Implementation of the scout process in Figure 3(b).

```
import java.util.*;

public class Scout extends Process {
  ProcessId leader;
  ProcessId[] acceptors;
  BallotNumber ballot_number;

  public Scout(Env env, ProcessId me, ProcessId leader,
      ProcessId[] acceptors, BallotNumber ballot_number){
    this.env = env;
    this.me = me;
    this.acceptors = acceptors;
    this.leader = leader;
    this.ballot_number = ballot_number;
    env.addProc(me, this);
  }

  public void body(){
    P1aMessage m1 = new P1aMessage(me, ballot_number);
    Set<ProcessId> waitfor = new HashSet<ProcessId>();
    for (ProcessId a: acceptors) {
      sendMessage(a, m1);
      waitfor.add(a);
    }

    Set<PValue> pvalues = new HashSet<PValue>();
    while (2 * waitfor.size() >= acceptors.length) {
      PaxosMessage msg = getNextMessage();

      if (msg instanceof P1bMessage) {
        P1bMessage m = (P1bMessage) msg;

        int cmp = ballot_number.compareTo(m.ballot_number);
        if (cmp != 0) {
          sendMessage(leader, new PreemptedMessage(me, m.ballot_number));
          return;
        }
        if (waitfor.contains(m.src)) {
          waitfor.remove(m.src);
          pvalues.addAll(m.accepted);
        }
      }
      else {
        System.err.println("Scout:␣unexpected␣msg");
      }
    }

    sendMessage(leader, new AdoptedMessage(me, ballot_number, pvalues));
  }
}
```

Leader.java

Implementation of the leader process in Figure 4.

```
import java.util.*;

public class Leader extends Process {
  ProcessId[] acceptors;
  ProcessId[] replicas;
  BallotNumber ballot_number;
  boolean active = false;
  Map<Integer, Command> proposals = new HashMap<Integer, Command>();

  public Leader(Env env, ProcessId me, ProcessId[] acceptors,
                    ProcessId[] replicas){
    this.env = env;
    this.me = me;
    ballot_number = new BallotNumber(0, me);
    this.acceptors = acceptors;
    this.replicas = replicas;
    env.addProc(me, this);
  }

  public void body(){
    System.out.println("Here␣I␣am:␣" + me);

    new Scout(env, new ProcessId("scout:" + me + ":" + ballot_number),
      me, acceptors, ballot_number);
    for (;;) {
      PaxosMessage msg = getNextMessage();

      if (msg instanceof ProposeMessage) {
        ProposeMessage m = (ProposeMessage) msg;
        if (!proposals.containsKey(m.slot_number)) {
          proposals.put(m.slot_number, m.command);
          if (active) {
            new Commander(env,
              new ProcessId("commander:" + me + ":" + ballot_number + ":" + m.slot_number),
              me, acceptors, replicas, ballot_number, m.slot_number, m.command);
          }
        }
      }
      else if (msg instanceof AdoptedMessage) {
        AdoptedMessage m = (AdoptedMessage) msg;

        if (ballot_number.equals(m.ballot_number)) {
          Map<Integer, BallotNumber> max = new HashMap<Integer, BallotNumber>();
          for (PValue pv : m.accepted) {
            BallotNumber bn = max.get(pv.slot_number);
            if (bn == null || bn.compareTo(pv.ballot_number) < 0) {
              max.put(pv.slot_number, pv.ballot_number);
              proposals.put(pv.slot_number, pv.command);
            }
          }

          for (int sn : proposals.keySet()) {
            new Commander(env,
              new ProcessId("commander:" + me + ":" + ballot_number + ":" + sn),
              me, acceptors, replicas, ballot_number, sn, proposals.get(sn));
          }
          active = true;
        }
      }

      else if (msg instanceof PreemptedMessage) {
        PreemptedMessage m = (PreemptedMessage) msg;
        if (ballot_number.compareTo(m.ballot_number) < 0) {
          ballot_number = new BallotNumber(m.ballot_number.round + 1, me);
          new Scout(env, new ProcessId("scout:" + me + ":" + ballot_number),
            me, acceptors, ballot_number);
          active = false;
        }
      }

      else {
        System.err.println("Leader:␣unknown␣msg␣type");
      }
    }
  }
}
```

Env.java

This is the main code in which all processes are created and run. This code also simulates a set of clients submitting requests.

```
import java.util.*;

public class Env {
  Map<ProcessId, Process> procs = new HashMap<ProcessId, Process>();
  public final static int nAcceptors = 3, nReplicas = 2, nLeaders = 2, nRequests = 10;

  synchronized void sendMessage(ProcessId dst, PaxosMessage msg){
    Process p = procs.get(dst);
    if (p != null) {
      p.deliver(msg);
    }
  }

  synchronized void addProc(ProcessId pid, Process proc){
    procs.put(pid, proc);
    proc.start();
  }

  synchronized void removeProc(ProcessId pid){
    procs.remove(pid);
  }

  void run(String[] args){
    ProcessId[] acceptors = new ProcessId[nAcceptors];
    ProcessId[] replicas = new ProcessId[nReplicas];
    ProcessId[] leaders = new ProcessId[nLeaders];

    for (int i = 0; i < nAcceptors; i++) {
      acceptors[i] = new ProcessId("acceptor:" + i);
      Acceptor acc = new Acceptor(this, acceptors[i]);
    }
    for (int i = 0; i < nReplicas; i++) {
      replicas[i] = new ProcessId("replica:" + i);
      Replica repl = new Replica(this, replicas[i], leaders);
    }
    for (int i = 0; i < nLeaders; i++) {
      leaders[i] = new ProcessId("leader:" + i);
      Leader leader = new Leader(this, leaders[i], acceptors, replicas);
    }

    for (int i = 1; i < nRequests; i++) {
      ProcessId pid = new ProcessId("client:" + i);
      for (int r = 0; r < nReplicas; r++) {
        sendMessage(replicas[r],
          new RequestMessage(pid, new Command(pid, 0, "operation␣" + i)));
      }
    }
  }

  public static void main(String[] args){
    new Env().run(args);
  }
}
```