Paxos Made Moderately Complex

ROBBERT VAN RENESSE and DENIZ ALTINBUKEN, Cornell University

This article explains the full reconfigurable multidecree Paxos (or multi-Paxos) protocol. Paxos is by no means a simple protocol, even though it is based on relatively simple invariants. We provide pseudocode and explain it guided by invariants. We initially avoid optimizations that complicate comprehension. Next we discuss liveness, list various optimizations that make the protocol practical, and present variants of the protocol.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Network operating systems*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*

General Terms: Design, Reliability

Additional Key Words and Phrases: Replicated state machines, consensus, voting

ACM Reference Format:
Robbert van Renesse and Deniz Altinbuken. 2015. Paxos made moderately complex. ACM Comput. Surv. 47, 3, Article 42 (February 2015), 36 pages.
DOI: http://dx.doi.org/10.1145/2673577

1. INTRODUCTION

Paxos [Lamport 1998] 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. Logically, a deterministic state machine handles one operation at a time.

—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; they are often referred to as 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 (so-called Byzantine failures), are

This work was supported in part by NSF grants CNS-1111698 and SA4897-10808PG (TRUST), DARPA grant FA8750-10-2-0238 (CRASH), and AFOSR grant FA9550-11-1-0137.
Authors' address: Robbert van Renesse and Deniz Altinbuken, Computer Science Department, Cornell University, Ithaca, NY 14853.

allowed. In a “fail-stop environment” [Schlichting and Schneider 1983], crash failures can be reliably detected. In an asynchronous environment, one machine cannot tell whether another machine is slow or has crashed.

—*State machine replication* (SMR) [Lamport 1978; Schneider 1990] is a technique to mask failures, particularly crash failures. A collection of *replicas* of a deterministic state machine are created. The replicas are then provided with the same sequence of operations, so they go through the same sequence of state transitions and 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 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* is a client-local unique command identifier (e.g., a sequence number). In other words, 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 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, the stub routine is replaced with another stub routine to provide the illusion of a single remote server that is highly available. The new stub routine sends the command to all replicas (at least one of which is assumed not to 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 take different transitions, produce different outputs as a result, and possibly end up in different current states. A protocol like Paxos ensures that this cannot happen: all replicas process commands in the same order, and thus the replicated state machine behaves logically identical to a single remote state machine that never crashes.

For simplicity, we assume that messaging between correct processes is reliable (but not necessarily FIFO):

—A message sent by a nonfaulty process to a nonfaulty 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. In other words, messages are not garbled and do not appear out of the blue.

Although Paxos can tolerate process crashes from the initial state, each process crash reduces the ability of Paxos to tolerate more crashes. A solution is to *reconfigure* Paxos after a crash to replace the crashed process with a fresh one.

This article gives an *operational description* of a reconfigurable version of the *multidecree Paxos protocol*, sometimes called *multi-Paxos*, and briefly describes variants of the Paxos protocol. Single-decree Paxos is significantly easier to understand and is the topic of papers by Lampson [1996] and Lamport [2001]. But the multidecree Paxos protocol is the one that is used (or some variant thereof) within industrial-strength systems like Google’s Chubby [Burrows 2006]. The article 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, the following can be checked: if the invariant holds before the operation, then the invariant still holds

[1]As in Lamport [1998], we will use Greek letters to identify processes.

Table I. Types of Processes in Paxos

Process Type	Description	Minimum Number
Replica	Maintains application state Receives requests from clients Asks leaders to serialize the requests so all replicas see the same sequence Applies serialized requests to the application state Responds to clients	$f + 1$
Leader	Receives requests from replicas Serializes requests and responds to replicas	$f + 1$
Acceptor	Maintains the fault tolerant memory of Paxos	$2f + 1$

Note: f is the number of failures tolerated.

after the operation. Second, invariants make it reasonably clear why Paxos is correct without having to go into correctness proofs.

In Section 2, we start with describing a reconfigurable 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 SMR. In Section 5, we survey important variants of the Paxos protocol. The article is accompanied by a basic Python implementation, and Section 4.6 suggests various exercises based on this code. Section 6 concludes the article.

2. HOW AND WHY PAXOS WORKS

Paxos employs *leaders* and *acceptors*—specialized processes that coordinate the replicas. Table I summarizes the functions of each type of process and how many of each are needed to tolerate f failures. Each such process is a deterministic state machine in its own right, maintaining state and undergoing state transitions in response to incoming messages. For each type of process, we will first describe what state it maintains and what invariants need to hold for correctness. We then give an operational description and discuss how the invariants are maintained.

2.1. Clients, Replicas, Slots, and Configurations

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 $c = \langle \kappa, cid, op \rangle$, its stub routine broadcasts a $\langle \mathbf{request}, c \rangle$ message to all replicas and waits for a $\langle \mathbf{response}, cid, result \rangle$ 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 that make up the inputs to the state machine. Each slot is indexed by a *slot number*. Replicas receive requests from clients and assign them to specific slots, creating a sequence of commands. A replica, on receipt of a $\langle \mathbf{request}, c \rangle$ 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. To avoid inconsistency, a *consensus protocol* chooses a single command from the proposals. The consensus protocol runs between a set of processes called the *configuration* of the slot. The configuration contains the leaders and the acceptors (but not the replicas). Leaders receive proposed commands from replicas and are responsible for *deciding* a single command for the slot. A replica awaits the decision before actually updating its state and computing a response to send back to the client that issued the request.

Usually, the configuration for consecutive slots is the same. Sometimes, such as when a process in the configuration is suspected of having crashed, it is useful to be able to change the configuration. Paxos supports reconfiguration: a client can propose a special reconfiguration command, which is decided in a slot just like any other command. However, if s is the index of the slot in which a new configuration is decided, it does not take effect until slot $s + \texttt{WINDOW}$. This allows up to WINDOW slots to have proposals pending—the configuration of later slots may change [Lamport 1978, §6.2]. It is always possible to add new replicas—this does not require a reconfiguration of the leaders and acceptors.

State and Invariants

Each replica ρ maintains seven variables:

—ρ*.state*: The replica's copy of the application state, which we will treat as opaque. All replicas start with the same initial application state.
—ρ*.slot_in*: The index of the next slot in which the replica has not yet proposed any command, initially 1.
—ρ*.slot_out*: The index of the next slot for which it needs to learn a decision before it can update its copy of the application state, equivalent to the state's version number (i.e., number of updates) and initially 1.
—ρ*.requests*: An initially empty set of requests that the replica has received and are not yet proposed or decided.
—ρ*.proposals*: An initially empty set of proposals that are currently outstanding.
—ρ*.decisions*: Another set of proposals that are known to have been decided (also initially empty).
—ρ*.leaders*: The set of leaders in the current configuration. The leaders of the initial configuration are passed as an argument to the replica.

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_out* are in the set of decisions:
$\forall \rho, s : 1 \leq s < \rho.\mathit{slot\_out} \Rightarrow \exists c : \langle s, c\rangle \in \rho.\mathit{decisions}$.
R3: For all replicas ρ, ρ*.state* is the result of applying the commands $\langle s, c_s\rangle \in \rho.\mathit{decisions}$ to *initial_state* for all s up to *slot_out*, in order of slot number.
R4: For each ρ, the variable ρ*.slot_out* cannot decrease over time.
R5: A replica proposes commands only for slots for which it knows the configuration:
$\forall \rho : \rho.\mathit{slot\_in} < \rho.\mathit{slot\_out} + \texttt{WINDOW}$.

To understand the significance of such invariants, it is useful to consider what would happen if one of the invariants would not hold. If R1 would not hold, replicas could *diverge*, ending up in different states even if they have applied the same number of commands. In addition, without R1, the same replica could decide multiple different commands for the same slot, because ρ_1 and ρ_2 could be the same process. Thus, the application state of that replica would be ambiguous.

Invariants R2 through R4 ensure that for each replica ρ, the sequence of the first ρ*.slot_out* commands is recorded and fixed. If any of these invariants were invalidated, a replica could change its history and end up with a different application state. The invariants do not imply that the slots have to be decided in order; they only imply that decided commands have to be applied to the application state in order and that there is no way to roll back.

Invariant R5 guarantees that replicas do not propose commands for slots that have an uncertain configuration. Because a configuration for slot s takes effect at slot $s + \texttt{WINDOW}$ and all decisions up to $\mathit{slot\_in} - 1$ are known, configurations up to slot $\rho.\mathit{slot\_in} + \texttt{WINDOW} - 1$ are known.

Operational Description

Figure 1 shows pseudocode for a replica. A replica runs in an infinite loop, receiving messages. Replicas receive two kinds of messages: requests and decisions. When it receives a request for command c from a client, the replica adds the request to set *requests*. Next, the replica invokes the function *propose*().

Function *propose*() tries to transfer requests from the set *requests* to *proposals*. It uses *slot_in* to look for unused slots within the window of slots with known configurations. For each such slot s, it first checks if the configuration for s is different from the prior slot by checking if the decision in slot s − `WINDOW` is a reconfiguration command. If so, the function updates the set of leaders for slot s. Then, the function removes a request r from *requests* and adds proposal (s, r) to the set *proposals*. Finally, it sends a $\langle$**propose**$, s, c\rangle$ message to all leaders in the configuration of slot s (Figure 2).

Decisions may arrive out of order and multiple times. For each **decision** message, the replica adds the decision to the set *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_out*, the replica first checks to see if it has proposed a command c'' for that slot. If so, the replica removes $\langle slot\_out, c''\rangle$ from the set *proposals*. If $c'' \neq c'$, that is, the replica proposed a different command for that slot, the replica returns c'' to set *requests* so c'' is proposed again at a later time. Next, the replica invokes *perform*(c').

The function *perform*() is invoked with the same sequence of commands at all replicas. First, it checks to see if it 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 and it is not a reconfiguration request. If so, *perform*() applies the requested operation to the application state. In either case, the function increments *slot_out*.

Figure 2 presents an example of an execution involving two clients, two replicas, and two leaders (in the same configuration). Both clients send a request to the replicas at approximately the same time, and the replicas learn about the two requests in opposite orders, assigning the commands to different slots. Both replicas send proposals to the leaders, which run a protocol that is not shown—it will be described in detail later. The result is that leaders decide which command gets assigned to which slot and replicas receive decisions for the two slots, upon which both replicas end up executing the commands in the same order and responding to the clients. It may happen that the same command is decided for both slots. The *Replica* code will skip the second decision and repropose the other command.

Maintenance of Invariants

Note that *decisions* is "append-only" in that there is no code that removes entries from this set. 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_out* 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_out* is only advanced if the corresponding decision is in *decisions*, it is clear that Invariant R2 holds. A command is proposed for a slot only if that slot is within the current `WINDOW`, and since replicas execute reconfiguration commands after a `WINDOW` of operations, it is ensured that Invariant R5 holds.

```
process $Replica(leaders, initial\_state)$
  var $state := initial\_state, slot\_in := 1, slot\_out := 1$;
  var $requests := \emptyset, proposals := \emptyset, decisions := \emptyset$;
  function $propose()$
    while $slot\_in < slot\_out + \texttt{WINDOW} \land \exists c : c \in requests$ do
      if $\exists op : \langle slot\_in - \texttt{WINDOW}, \langle \cdot, \cdot, op \rangle\rangle \in decisions \land isreconfig\ (op)$ then
        $leaders := op.leaders$;
      end if
      if $\nexists c' : \langle slot\_in, c' \rangle \in decisions$ then
        $requests := requests \backslash \{c\}$;
        $proposals := proposals \cup \{\langle slot\_in, c \rangle\}$;
        $\forall \lambda \in leaders : send(\lambda, \langle \mathbf{propose}, slot\_in, c \rangle)$;
      end if
      $slot\_in := slot\_in + 1$;
    end while
  end function
  function $perform(\langle \kappa, cid, op \rangle)$
    if $(\exists s : s < slot\_out \land \langle s, \langle \kappa, cid, op \rangle\rangle \in decisions) \lor isreconfig\ (op)$ then
      $slot\_out := slot\_out + 1$;
    else
      $\langle next, result \rangle := op(state)$;
      atomic
        $state := next; slot\_out := slot\_out + 1$;
      end atomic
      $send(\kappa, \langle \mathbf{response}, cid, result \rangle)$;
    end if
  end function
  for ever
    switch $receive()$
      case $\langle \mathbf{request}, c \rangle$ :
        $requests := requests \cup \{c\}$;
      end case
      case $\langle \mathbf{decision}, s, c \rangle$ :
        $decisions := decisions \cup \{\langle s, c \rangle\}$;
        while $\exists c' : \langle slot\_out, c' \rangle \in decisions$ do
          if $\exists c'' : \langle slot\_out, c'' \rangle \in proposals$ then
            $proposals := proposals \backslash \{\langle slot\_out, c'' \rangle\}$;
            if $c'' \neq c'$ then
              $requests := requests \cup \{c''\}$;
            end if
          end if
          $perform(c')$;
        end while
      end case
    end switch
    $propose()$;
  end for
end process
```

Fig. 1. Pseudocode for a replica.

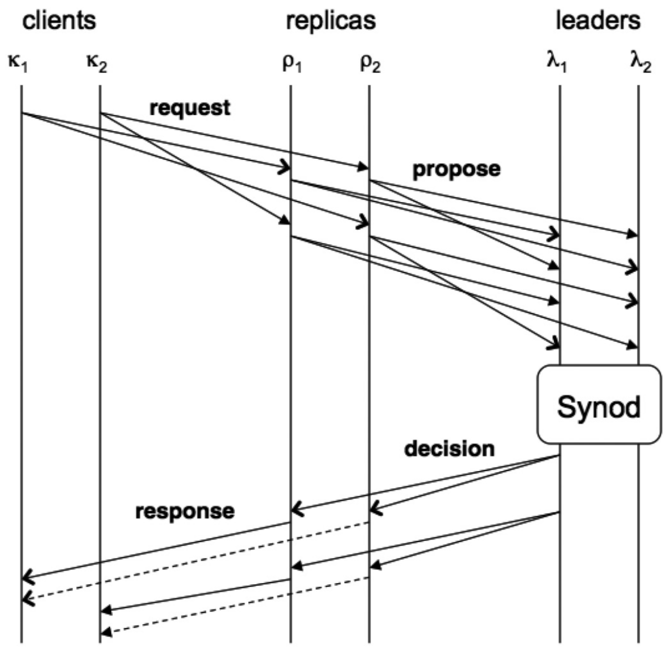

Fig. 2. The time diagram shows two clients, two replicas, and two leaders, with time progressing downward. Arrows represent messages. Open arrows are messages concerning the request from client κ_1. Closed arrows are messages concerning the request from client κ_2. Dashed arrows are messages that end up being ignored. The clients broadcast to all replicas, who receive requests from the two clients in different orders. The replicas forward proposals to the leaders, who decide on the ordering using the Synod protocol (message traffic not shown) and inform the replicas. Finally, the replicas respond to the clients.

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 *multidecree protocol*, or just *Synod protocol* for short, as we do not consider the single-decree protocol in this article.

2.2. The Synod Protocol, Ballots, and Acceptors

Replicas can propose multiple commands for the same slot. The Synod protocol chooses from these a single command to be decided by all nonfaulty replicas. This is not a trivial problem. A process might fail at any time, and because there is no bound on timing for delivering and processing messages, it is impossible for other processes to know for certain that the process has failed.

In the Synod protocol, there is an unbounded collection of *ballots*. Ballots are not created; they just *are*. As we shall see later, ballots are the key to liveness properties in Paxos. Each ballot has a unique *leader*. A leader can be working on arbitrarily many ballots, although it will be predominantly working on one at a time. To tolerate f failures, there must be at least $f + 1$ leaders, each in charge of an unbounded 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

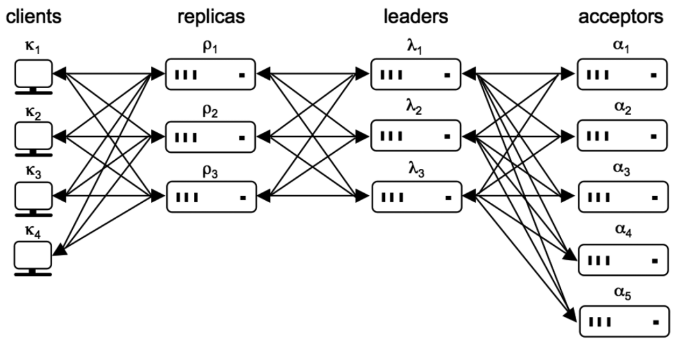

Fig. 3. Communication pattern between types of processes in a setting where $f = 2$.

ballot can be used to decide multiple slots, and one slot may be targeted 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, the protocol employs a collection of *acceptors*. Acceptors are deterministic state machines, although they are *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. Acceptors use a *voting protocol*, allowing a unanimous majority of acceptors to decide without needing input from the remaining acceptors. Thus, to tolerate f crash failures, Paxos needs at least $2f + 1$ acceptors, always leaving at least $f+1$ acceptors to maintain the fault-tolerant memory. Figure 3 illustrates the communication patterns between the various types of processes in a setting where $f = 2$.

State and Invariants

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

—$\alpha.ballot\_num$, a ballot number, initially $\perp$; and
—$\alpha.accepted$, a set of pvalues, initially empty.

Under the direction of messages sent by leaders, the state of an acceptor can change. Let $p = \langle b, s, c\rangle$ 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 is an invaluable help to understanding the Synod protocol:

A1: An acceptor can only adopt strictly increasing ballot numbers.

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

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

Fig. 4. Pseudocode for an acceptor.

A2: An acceptor α can only 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 that 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 $c = c'$.

It is important to realize that Invariant A5 works in two ways. In one way, if all acceptors in a majority have accepted a particular pvalue $\langle b, s, c \rangle$, then any pvalue for a later ballot will contain the same command c for slot s. In the other way, suppose that some acceptor accepts $\langle b', s, c' \rangle$. At a later time, if any majority of acceptors accepts pvalue $\langle b, s, c \rangle$ on an earlier ballot $b < b'$, then $c = c'$.

Operational Description

Figure 4 shows pseudocode for an acceptor. It runs in an infinite loop, receiving two kinds of request messages from leaders (note the use of pattern matching):

—$\langle \mathbf{p1a}, \lambda, b \rangle$: 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 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.

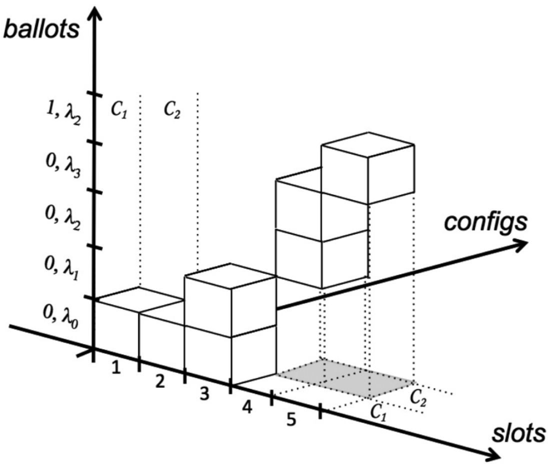

Fig. 5. In the Synod protocol, slot numbers and ballot numbers are orthogonal concepts. One ballot can be used to decide on multiple slots, like in slot 1 and slot 2. A slot may be considered by multiple ballots, such as in slot 3. A configuration can span multiple ballots and multiple slots, but they each belong to a single configuration.

Maintenance of Invariants

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, which is described in the following sections.

An instance of the Synod protocol uses a fixed configuration $\mathcal{C}$ consisting of at least $f+1$ leaders and $2f+1$ acceptors. For simplicity, assume that configurations have no processes in common. Instances follow each other, creating a reconfigurable protocol. Figure 5 shows the relation between ballots, slots, and configurations. A leader can use a single ballot to decide multiple slots, as in the case for slots 1 and 2. Multiple leaders might use multiple ballots during a single slot, as shown in slot 3. A configuration can have multiple ballots and can span multiple slots, but each slot and each ballot has only one configuration associated with it.

2.3. 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. For each such slot, the leader selects a command and spawns a new process that we call a *commander*. Although we present it as a separate process, the commander is really just a thread running within the leader. The commander runs what is known as phase 2 of the Synod protocol.

As we shall see, the following invariants hold in the Synod protocol:

C1: For any ballot b and slot s, at most one command c is selected and at most one commander for $\langle b, s, c\rangle$ is spawned.
C2: Suppose that for each α among a majority of acceptors, $\langle b, s, c\rangle \in \alpha.\mathit{accepted}$. 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 6(a) shows the pseudocode 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 in Figure 4). 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 all replicas and exits. 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 assumptions that at most a minority of acceptors can crash, that messages are delivered reliably, and that the commander does not crash, the commander will eventually do one or the other.

2.4. 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. To enforce Invariant C2, the leader runs what is often called *phase 1* of the Synod protocol or 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 6(b) shows the pseudocode 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 to enforce Invariant C2.

Figure 7 shows the main code of a leader. Leader λ maintains three state variables:

—$\lambda.\mathit{ballot\_num}$, a monotonically increasing ballot number, initially $(0, \lambda)$;
—$\lambda.\mathit{active}$, a Boolean flag, initially `false`; and
—$\lambda.\mathit{proposals}$, a map of slot numbers to proposed commands in the form of a set of $\langle \mathit{slot\ number}, \mathit{command}\rangle$ 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.

[3]The term *view change* is used in the Viewstamped Replication protocol [Oki and Liskov 1988], which is similar to the Paxos protocol [Van Renesse et al. 2014].

(a)

```
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
```

(b)

```
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
```

Fig. 6. (a) Pseudocode for a commander. Here, λ is the identifier of its leader, *acceptors* is the set of acceptor identifiers, *replicas* is the set of replica identifiers, and $\langle b, s, c\rangle$ is the pvalue for which the commander is responsible. (b) Pseudocode for a scout. Here, λ is the identifier of its leader, *acceptors* is the identifiers of the acceptors, and b is the desired ballot number.

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

   $spawn(Scout(self(), acceptors, ballot\_num))$;
   for ever
      switch $receive()$
         case $\langle \mathbf{propose}, s, c\rangle$ :
            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 $\langle \mathbf{adopted}, ballot\_num, pvals\rangle$ :
            $proposals := proposals \lhd pmax(pvals)$;
            $\forall \langle s, c\rangle \in proposals$ :
               $spawn(Commander(self(), acceptors, replicas, \langle ballot\_num, s, c\rangle))$;
            $active :=$ true;
         end case
         case $\langle \mathbf{preempted}, \langle r', \lambda'\rangle\rangle$ :
            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
```

Fig. 7. Pseudocode skeleton for a leader. Here, *acceptors* is the set of acceptor identifiers, and *replicas* is the set of replica identifiers.

—$\langle$**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*.
—$\langle$**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 $\langle$**adopted**, *ballot_num*, *pvals*$\rangle$ message from the last scout that it spawned. When this message arrives, 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*. The leader uses these pvalues to update its own proposals variable. 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 nonempty—some acceptor in the intersection must have violated Invariant A1, A2, or A3, which we assume 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. In addition, note that $b <$ *ballot_num* (because acceptors only report pvalues that they accepted before adopting *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 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:

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

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,

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

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 8 shows an example of a leader λ spawning a scout to become active, and a client κ sending a request to two replicas ρ_1 and ρ_2, which in turn send proposals to λ.

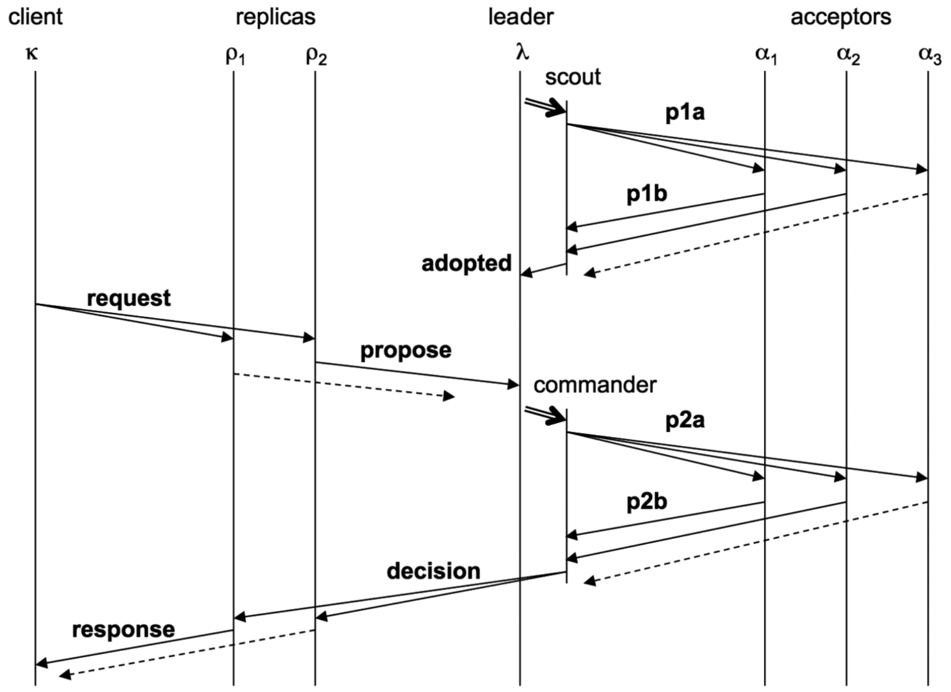

Fig. 8. The time diagram shows a client, two replicas, a leader (with a scout and a commander), and three acceptors, with time progressing downward. Arrows represent messages. Dashed arrows are messages that end up being ignored. The leader first runs a scout to become active. Later, when a replica proposes a command (in response to a client's request), the leader runs a commander, which notifies the replicas upon learning a decision.

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 an example of a *liveness* property. 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 shown in Figure 9, 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 that 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 has a higher ballot number than ballot $\langle 0, \lambda\rangle$ 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 $\langle\langle 0, \lambda'\rangle, 1, c'\rangle$.

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 with a higher ballot, 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 that 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

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

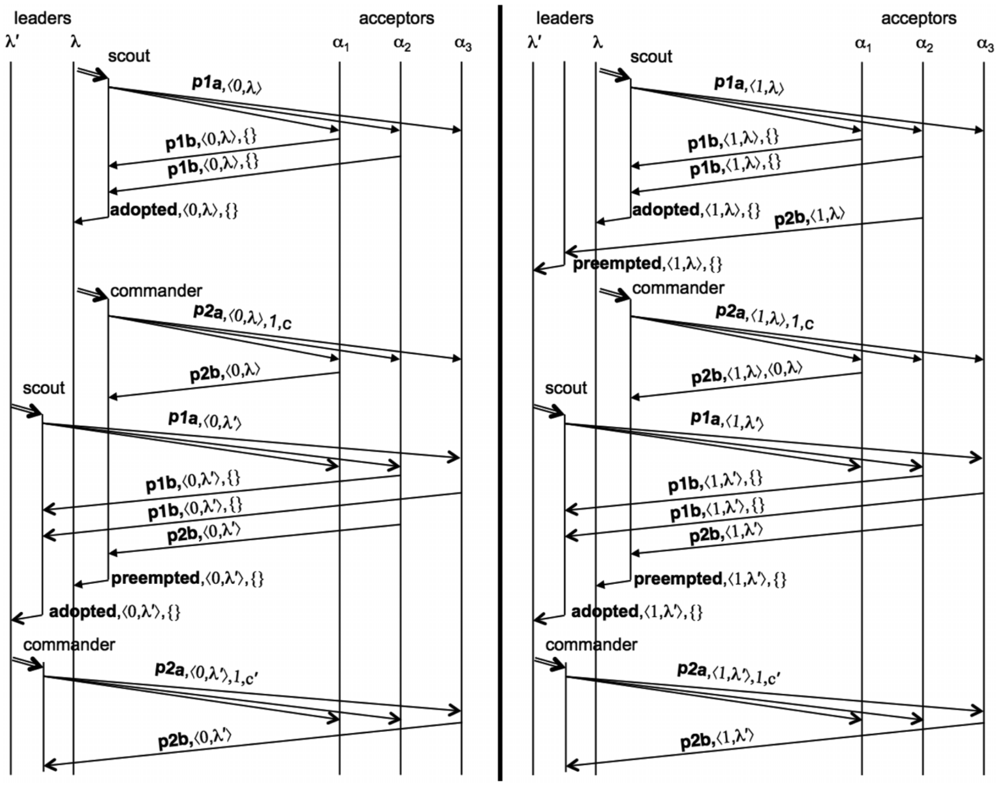

Fig. 9. The time diagram shows two leaders and three acceptors, with time progressing downward; the time diagram on the right follows the one on the left. Arrows represent messages. Closed arrows are messages concerning the proposal from leader λ. Open arrows are messages concerning the proposal from leader λ'. Leaders try to get their proposals accepted by a majority of acceptors, but in each round one leader prevents the other leader from obtaining a majority.

$\langle\langle 0, \lambda'\rangle, 1, c'\rangle$. 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" [Fischer et al. 1985] 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, such as 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, λ starts monitoring λ' by pinging it on a regular basis. As long as λ' responds timely to pings, 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 to support a consensus algorithm that is guaranteed to terminate [Chandra and Toueg 1991]. 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; and
—the time between when a nonfaulty process initiates sending a message and the message has been received and handled by a nonfaulty 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 nonfaulty processes is likely delivered and processed within, say, a year.

These assumptions can be exploited as follows. We use a scheme similar to the one described earlier, 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 quickly. This can be achieved with a TCP-like additive increase, multiplicative decrease (AIMD) 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 1. With each chosen proposal, this initial timeout is decreased linearly. Eventually, the timeout will become too short, and the ballot will be replaced with another even if its leader is nonfaulty.

Liveness can be further improved by keeping state on disk. 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. 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 assumed reliable communication. This assumption can be weakened by using a *fair links* assumption: if a correct process repeatedly sends a message to another correct process, at least one copy is eventually delivered. Reliable transmission of a message can then be implemented by periodic retransmission until an ack is received. To prevent overload on the network, the time intervals between retransmissions can grow until the load imposed on the network is negligible. The fair links assumption can be weakened further, but such a discussion is outside the scope of this article.

As an example of how liveness is achieved, 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 retry. Although 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 improvements.

4.1. State Reduction

First, note that although a leader obtains for each slot a set of all accepted pvalues from a majority of acceptors, it only needs to know 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.

This optimization leads to a 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) that there are two leaders λ and λ' such that $\lambda < \lambda'$ and 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 suppose that acceptor α_2 accepts $\langle\langle 0, \lambda'\rangle, 1, c\rangle$.

At this point, there is no majority of acceptors that *store* 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, the leader of any ballot b after $\langle 0, \lambda\rangle$ can only select $\langle b, 1, c\rangle$. This is by Invariant C2 and because acceptors α_1 and α_2 both accepted $\langle\langle 0, \lambda\rangle, 1, c\rangle$ and together form a majority.

4.2. Garbage Collection

In our description of Paxos, much information is kept about slots that have already been decided. Some of this is unavoidable. For example, because commands may be decided in multiple slots, replicas each maintain a set of all decisions to filter out such duplicates. In practice, it is often sufficient if such information is only kept for a certain amount of time, making the probability of duplicate execution negligible.

But some state, and indeed some work that results from having that state, is unnecessary. The leader maintains state for each slot in which it has a proposal. In addition, when a leader becomes active, it spawns a commander for each such slot. Similarly, even if the state reduction of Section 4.1 is implemented, each acceptor maintains state for each slot. However, once at least $f + 1$ replicas have learned about the decision of some slot, it is no longer necessary for leaders and acceptors to maintain this state—replicas can learn decisions, and the application state that results from those decisions, from one another.

Thus, much state, and work, can be saved if each replica periodically updates leaders and acceptors about its *slot_out* variable. Once a leader or acceptor learns that at least $f + 1$ replicas have received all decisions up to some slot s, all information about lower numbered slots can be garbage collected. However, we must prevent other leaders from mistakenly concluding that the acceptors have not accepted any pvalues for the garbage-collected slots. To achieve this, the state of an acceptor can 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 leaders can skip the lower numbered slots.

Note that if there are fewer than $2f + 1$ replicas, the crash of f replicas would leave fewer than $f + 1$ replicas to send periodic updates and no garbage collection could be done. If having $2f + 1$ replicas is too expensive, the set of replicas could be made dynamic so that suspicious replicas could be removed and fresh replicas could be added. To do so, we add the set of replicas to configurations. Now, $f + 1$ replicas per configuration suffices. Once all ($f + 1$ or more) replicas in a configuration have learned the decision for a slot, the corresponding leader and acceptor state for that slot can be garbage collected. If there are not enough responsive replicas, a new configuration can be created whose replicas can learn the state from responsive replicas in the old configuration. When done, the unresponsive replicas can be cleaned up.

4.3. Keeping State on Disk

There are two reasons for keeping state on disk. One is that the amount of state is so large that it does not fit in memory. Another is to survive power outages. However, keeping state on disk comes with its own set of problems. Disk access is slow relative to memory access, and there are issues with updating data structures on disks atomically. A pragmatic solution to these problems is *Write-Ahead Logging* [Mohan et al. 1992] in which updates are logged sequentially to disk, and, separately, memory caches all or part of the data structure that is also stored on disk. Periodically, the log is truncated after *checkpointing* the current state of the data structure (i.e., the dirty entries in the cache) to disk.

4.4. Colocation

In practice, leaders are typically colocated with replicas. In other words, each machine that runs a replica also runs a leader. This allows some optimizations. A client sends its proposals to replicas. If colocated, 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 colocated 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 [Thomas 1979; Attiya et al. 1995]. Whereas 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 colocated with acceptors. As discussed in Section 4.2, one may desire as many replicas as acceptors in any case. When leaders are colocated with acceptors, one has to be careful that separate ballot number variables are used.

4.5. Read-Only Commands

The protocol that we presented does not treat read-only commands any differently from update commands, leading to more overhead than necessary since read-only commands do not change any state. One would be naïve to think that a client wanting to do a read-only command could simply query one of the replicas—doing so could easily violate consistency, as the selected replica may not have learned all updates yet and thus have stale application state. Therefore, a read-only command should be serialized after the latest completed update command that precedes it. There are, however, various optimizations possible.

First, it is not necessary that a new slot is allocated for a read-only command; in particular, it is not necessary that a majority of acceptors accept a pvalue. It is necessary to ensure that the read-only command is assigned an appropriate slot s that defines the state at which the read-only command is executed. Slot number s has to be at least as high as the maximum among the *slot_out* variables of the replicas, because the read-only command should follow the latest update.

To obtain s, a replica ρ can ask each acceptor for the highest slot number for which it has accepted a pvalue (returning 0 if it has not yet accepted any value). Replica ρ awaits responses from a majority of acceptors and sets s to the maximum among the responses. It is easy to see that s will be at least as large as the maximum *slot_out* among the replicas. Next, replica ρ proposes no-ops for all slots between $\rho.\mathit{slot\_in}$ and s, if any, and waits until $\rho.\mathit{slot\_out} > s$. Finally, ρ evaluates the read-only command and responds to the client.

It is interesting to note that leaders do not need to be involved in read-only commands—they only are involved in serializing the update commands. Indeed, different replicas may end up computing a different value for s, and consequently a client could receive different results for its read-only command. Any of these responses is a correct response, serialized after the latest update that completed before the client issued its command, so the client can just use the first response it receives.

This optimization still requires a round-trip latency to the acceptors. We can avoid this round-trip by using so-called *leases* [Gray and Cheriton 1989; Lamport 1998]. Leases require an additional assumption on timing. Various options for leasing are possible—the one that we describe here is based on the work described in Van Renesse et al. [2011], which assumes that there is a known bound on clock drift but does not assume that clocks are synchronized. For simplicity, we will assume that there is no clock drift whatsoever. The basic idea is that at most one replica can have a lease at a time, and that this replica handles all client requests, both read-only and update commands. A lease is valid for a particular period of time δ.

To obtain a lease, a replica ρ notes its current clock c and, as before, sends a request to each acceptor. The acceptor grants a lease for a period of time δ (on its local clock) to ρ if it has no outstanding lease to another replica and, as before, returns the highest slot number for which it accepted a pvalue. If the replica receives responses from a majority of acceptors, it has a lease until time $c+\delta$ on ρ's clock. Knowing that it has the lease, a replica can directly respond to read-only commands assuming $\rho.\mathit{slot\_out} > s$, where s is the maximum among the responses it received. Update commands still have to follow the usual protocol, involving a leader and a majority of acceptors.

4.6. Exercises

This article is accompanied by a Python package (see the Appendix) that contains a Python implementation for each of the pseudocode listings presented in this work. We list a set of suggested 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 $\{\lambda_1, \ldots, \lambda_n\}$, then the ballot numbers for leader λ_i could be $i, i+n, i+2n, \ldots$. Ballot number $\perp$ could be represented as 0. Modify the Python 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, and inquiry.
(4) In the Python implementation, all processes run as threads within the same Python 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 reconnect until successful.
(5) Implement the failure detection scheme of Section 3 so that most of the time only one leader is active.

(6) Colocate leaders and replicas as suggested in Section 4.4, and garbage collect unnecessary leader state, that is, leaders can forget about proposals for commands that have already been decided. Upon becoming active, leaders do not have to start commanders for such slots either.
(7) 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 partway 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 4.5.

5. PAXOS VARIANTS

There have been many Paxos variants introduced over the years. These variants concentrate on making the basic Paxos algorithm more efficient. In this section, we cover some of these variants chronologically and discuss their significant features.

5.1. Disk Paxos

Every replicated component in multidecree Paxos is a process, actively sending and receiving messages. Disk Paxos [Gafni and Lamport 2003] is a variant of Paxos that replaces acceptor processes with disks that support only `read block` and `write block` operations to store the quorum state. This way, Disk Paxos does not need separate acceptor processes.

In Disk Paxos, each leader owns a block on every disk to which it can write its messages. To run phase 1 of the Synod protocol, a leader executes the following for each disk. The leader writes a **p1a** message in its own block on the disk and reads the blocks of other leaders on the same disk to check if there is a **p1a** message with a higher ballot number. If the leader does not discover a higher ballot number on a majority of disks, its ballot is adopted. If it discovers a **p1a** message with a higher ballot number, it starts over with a higher ballot number. For phase 2, the leader repeats the same process with **p2a** messages to determine if its proposals are accepted.

5.2. Cheap Paxos

Multidecree Paxos requires $2f + 1$ acceptors and $f + 1$ replicas to tolerate f failures. Cheap Paxos [Lamport and Massa 2004] uses only $f + 1$ acceptors combined with f *cheap* additional auxiliary acceptors that are kept idle during normal execution, unless there is a failure. Cheap Paxos is a variation of the reconfigurable multidecree Paxos algorithm that relies on the observation that a leader can send its **p1a** and **p2a** messages to a fixed quorum of acceptors. As long as all acceptors in that quorum are working and can communicate with the leaders, there is no need for acceptors not in the quorum to do anything.

Cheap Paxos runs the multidecree Paxos protocol with a fixed quorum of $f + 1$ acceptors. Upon suspected failure of an acceptor in the fixed quorum, an auxiliary acceptor becomes active so that there are once again $f + 1$ responsive acceptors that can form a quorum. This new quorum completes the execution of any outstanding ballots and reconfigures the system replacing the suspected acceptor with a fresh one, returning the system to normal execution.

5.3. Fast Paxos

In Paxos, each decision takes at least three message delays between when a replica proposes a command and when some replica learns which command has been chosen.

Moreover, if there are multiple leaders working concurrently, it takes four message delays for colliding commands to be assigned to different slots. Fast Paxos [Lamport 2006] is a variant of Paxos that reduces end-to-end message delays experienced by a replica to learn a chosen value. Learning occurs in two message delays when there is no collision, and in three message delays when there is. The price paid is that $3f + 1$ acceptors are necessary instead of $2f + 1$.

In Fast Paxos, the replica sends its proposal directly to the acceptors, bypassing the leader. Fast Paxos distinguishes *fast* and *classic* ballots. In fast ballots, after completing the first phase, the leader sends a **p2a** message without a value to the acceptors. Concurrently, replicas send their proposals directly to acceptors. When an acceptor receives such a **p2a** message, it can choose any replica's proposal as a value to accept. In the absence of collisions, the acceptors end up accepting the same proposals and the end-to-end message delay for a replica is reduced from three message delays to two message delays.

This scheme can fail if replicas send different proposals concurrently and the acceptors accept conflicting pvalues. In that case, a classic ballot—which works the same as in ordinary Paxos—can be used to recover.

5.4. Generalized Paxos

As noted, Fast Paxos does not handle collision efficiently. Generalized Paxos [Lamport 2005] improves upon Fast Paxos by allowing independent commands to be executed in any order. Generalized Paxos generalizes the traditional consensus problem, which chooses a single value, to the problem of choosing monotonically increasing, consistent values. The traditional consensus problem creates a growing sequence of commands by assigning every command to a slot one-by-one. This problem can be abstracted by using *command structures*, or *c-structs* that are formed from a null element, $\perp$, by the operation of appending commands. This way, c-structs are used to abstract command sequences. By abstracting the problem using c-structs, it can be shown that different c-structs can belong to the same equivalence class, that is, if a given set of commands do not have a dependency, multiple c-structs that are equivalent to each other can be constructed using this set of commands in different orders. Consequently, if we use a generalized consensus algorithm that decides on sequences of commands rather than the assignment of a single command to a specific slot, command collisions, as they happen in Fast Paxos, need not be resolved unless there is a direct dependency between the concurrent commands. Different command sequences can be accepted by different acceptors if they are in the same equivalence class, where the ordering of specific commands do not change the end state.

5.5. Stoppable Paxos

Stoppable Paxos [Lamport et al. 2008] implements a *stoppable replicated state machine*. The execution of a stoppable replicated state machine can be stopped with a special command—once decided, no more commands are executed. This variant provides an alternative reconfiguration option: stop the current replicated state machine and start a new one using the final application state of the previous one. This way, the replicated state machine can be thought of as a sequence of stoppable state machines, each running with a fixed configuration.

5.6. Mencius

Mencius [Mao et al. 2008] tries to solve the single leader bottleneck experienced in multidecree Paxos. In Mencius, replicas, acceptors, and leaders are colocated, organized in a logical ring, and preassigned slot numbers to propose their commands. This way,

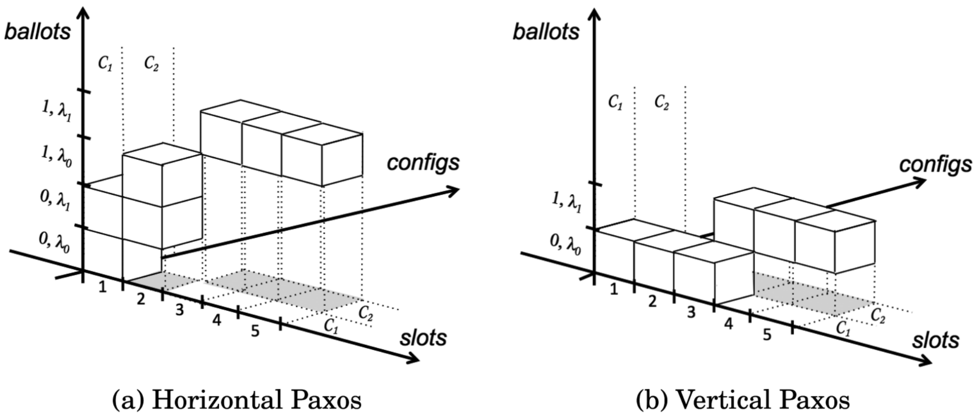

Fig. 10. In Horizontal Paxos, configurations can change only as we move horizontally from one slot to another; they are unchanged when we move vertically, within a single slot. In Vertical Paxos, configurations change when we move vertically but remain the same as we move horizontally from a ballot in one instance to the ballot with the same number in any other slot.

leaders take turns proposing commands, increasing throughput especially when the system is CPU bound.

Because the initial leader for a slot is known, there is no need for phase 1 and the leader only needs to execute phase 2. If it has no command to propose, the leader proposes a no-op command. If the leader is faulty or slow, the other leaders can take over the slot by executing phase 1, but they can only propose a no-op command. Leveraging this knowledge, Mencius allows replicas to learn about a no-op command proposed by a nonfaulty leader in a single message delay.

In normal execution without process failures, the performance of Mencius is similar to multidecree Paxos with a stable leader, without the leader bottleneck. However, idle and slow leaders can diminish overall performance. To limit the number of no-op messages generated by leaders, Mencius uses leases between leaders, where leaders can lease their ballots to other leaders for all slots less than a specified slot number. This can be done voluntarily by an idle leader giving up its ballots or aggressively by a fast leader taking over the ballots of a slow leader.

5.7. Vertical Paxos

Vertical Paxos [Lamport et al. 2009] is a Paxos variant that enables reconfiguration while the replicated state machine is active and deciding on commands. Vertical Paxos uses an auxiliary master to decide on reconfiguration operations, which determines the set of acceptors and the leader for every configuration.

In Vertical Paxos, the leader is set for every configuration, and different ballot numbers have different configurations. Unlike standard "horizontal" Paxos algorithms, every configuration has exactly one ballot number and a leader associated with it, and a slot can have more than one configuration, as shown in Figure 10. In Horizontal Paxos algorithms, configurations can only change when we move horizontally from one slot to another, as in slot 3 in Figure 10(a). In Vertical Paxos, configurations change when we move vertically, as in slot 3 in Figure 10(b), but remain the same as we move horizontally.

After a leader becomes active in a configuration of Vertical Paxos, it has to communicate with the acceptors from lower-numbered ballots to access old state. To limit the number of acceptors with which a leader has to communicate, Vertical Paxos uses two distinct quorums: *read* and *write* quorums. When there is a configuration change, reads happen from the read quorum in the old configuration until this state is transferred to all live acceptors in the new configuration. Once the state transfer is complete, the new leader informs the master and the read and write quorums are unified.

Vertical Paxos can also be viewed as a special case of the Primary-Backup protocol [Budhiraja et al. 1993], where the leader for a given configuration acts as a Primary.

5.8. Egalitarian Paxos

Egalitarian Paxos (EPaxos) [Moraru et al. 2013] is another Paxos variant that tries to solve the single leader bottleneck by allowing all replicas to receive values from clients and letting them propose values with one message delay when there is no dependence between commands. This allows the system to evenly distribute the load to all replicas; in addition, it offers better performance for geo-replicated state machines by enabling clients to send command requests to the replicas closest to them.

Unlike other Paxos variants, EPaxos orders commands dynamically and in a decentralized fashion. EPaxos assumes that replicas, acceptors, and leaders are colocated. In the process of proposing a command for a slot number, a replica attaches ordering constraints to that command, so every replica can use these constraints to independently reach the same ordering locally.

EPaxos runs in two phases: *preaccept* and *accept*. The preaccept phase establishes the ordering constraints for a command c. Upon receiving c from a client, a replica becomes the *command leader* and sends a preaccept message including c, its dependencies dep_c, and a sequence number seq_c to $2f$ replicas. dep_c is the list of all instances that contain commands that interfere with c, and seq_c is a sequence number greater than the sequence number of all interfering commands in dep_c. When a replica receives a preaccept message, it updates its local dep_c and seq_c attributes according to its state and replies to the command leader with this state. The command leader might receive replies from the quorum with dep_c and seq_c matching in all replies—in this case, it can commit c and send a commit message to all other replicas and reply to the client. In the other case, the command leader might receive nonmatching dep_c and seq_c in the replies. Then, the command leader updates its state and goes on to the accept phase.

In the accept phase, the command leader sends an accept message including c, dep_c, and seq_c to at least f other replicas. Upon receiving the accept message, the replicas update their state and send a reply to the command leader including c. When the command leader receives at least f replies, it can send the commit message to all replicas and reply to the client.

Following this protocol, every replica creates a local dependency graph for every command and executes every command recursively in accordance with this dependency graph.

6. CONCLUSION

In this article, we presented a reconfigurable version of Paxos as a collection of specialized processes, each with a simple operational specification that includes invariants. We started with an impractical but relatively easy implementation to simplify comprehension, then showed how various aspects can be improved to render a practical protocol. We also presented variants of the Paxos protocol.

The article is the next in a long line of papers that describe the Paxos protocol, present the experience of implementing it, or make it easier to understand. A partial list follows. The Viewstamped Replication protocol by Oki and Liskov [1988] is a protocol that is similar to multidecree Paxos (similarities and differences are discussed in Van Renesse et al. [2014]). Lamport's seminal "Part-Time Parliament" paper

[Lamport 1998] is the first paper describing Paxos. Lampson presents ways to derive the single-decree Paxos algorithm and pragmatic optimizations [Lampson 1996, 2001]. De Prisco et al. [2000] present an implementation of Paxos in the general timed automaton formalism. In *Paxos Made Simple*, Lamport [2001] gives a simple invariant-based explanation of the protocol. Boichat et al. [2003] give a formal account of various variants of Paxos along with extensive pseudocode. Lorch et al. [2006] present their Paxos SMR implementation SMART that supports high performance through parallelization, dynamic membership changes and migration. Bolosky et al. [2011] describe their high-performance data store that uses SMART. Chandra and Toueg [1991] describe Google's challenges in implementing Paxos. Li et al. [2007] give a novel simplified presentation of Paxos using a write-once register. In *Paxos Made Practical*, [Mazières 2007] gives details of how to build replicated services using Paxos. Kirsch and Amir describe their Paxos implementation experiences in [Kirsch and Amir 2008b], along with detailed pseudocode [Kirsch and Amir 2008a] (intended for developers, not so much for educational purposes). Alvaro et al. [2010] describe implementing Paxos in Overlog, a declarative language. Meling and Jehl [2013] published a tutorial on Paxos [Meling and Jehl 2013]. Ongaro and Ousterhout [2014] present a protocol similar to Paxos and Viewstamped Replication but designed for understandability.

APPENDIX: PYTHON SOURCE CODE LISTING

Python source code corresponding to the pseudocode in this article is available at http://www.paxos.systems. The code can be run using the "python env.py" command. For archival purposes, we also include the code in the following sections.

utils.py

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

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

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).

A reconfiguration command consists of the process identifier of the client submitting the request, a client-local request identifier, and a configuration.

A configuration consists of a list of replicas, a list of acceptors, and a list of leaders.

```
from collections import namedtuple

WINDOW = 5

class BallotNumber(namedtuple('BallotNumber',['round','leader_id'])):
  __slots__ = ()
  def __str__(self):
    return "BN(%d,%s)" % (self.round, str(self.leader_id))

class PValue(namedtuple('PValue',['ballot_number',
                                  'slot_number',
                                  'command'])):
  __slots__ = ()
  def __str__(self):
    return "PV(%s,%s,%s)" % (str(self.ballot_number),
                             str(self.slot_number),
                             str(self.command))
```

```
class Command(namedtuple('Command',['client',
                                    'req_id',
                                    'op'])):
  __slots__ = ()
  def __str__(self):
    return "Command(%s,%s,%s)" % (str(self.client),
                                  str(self.req_id),
                                  str(self.op))

class ReconfigCommand(namedtuple('ReconfigCommand',['client',
                                                    'req_id',
                                                    'config'])):
  __slots__ = ()
  def __str__(self):
    return "ReconfigCommand(%s,%s,%s)" % (str(self.client),
                                          str(self.req_id),
                                          str(self.config))

class Config(namedtuple('Config',['replicas',
                                  'acceptors',
                                  'leaders'])):
  __slots__ = ()
  def __str__(self):
    return "%s;%s;%s" % (','.join(self.replicas),
                         ','.join(self.acceptors),
                         ','.join(self.leaders))
```

process.py

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.

```
import multiprocessing
from threading import Thread

class Process(Thread):
  def __init__(self):
    super(Process, self).__init__()
    self.inbox = multiprocessing.Manager().Queue()

  def run(self):
    try:
      self.body()
      self.env.removeProc(self.id)
    except EOFError:
      print "Exiting.."

  def getNextMessage(self):
    return self.inbox.get()

  def sendMessage(self, dst, msg):
    self.env.sendMessage(dst, msg)

  def deliver(self, msg):
    self.inbox.put(msg)
```

message.py

Paxos uses a large variety of message types, collected as follows.

```
class Message:
  def __init__(self, src):
    self.src = src

  def __str__(self):
    return str(self.__dict__)

class P1aMessage(Message):
  def __init__(self, src, ballot_number):
    Message.__init__(self, src)
    self.ballot_number = ballot_number

class P1bMessage(Message):
  def __init__(self, src, ballot_number, accepted):
    Message.__init__(self, src)
    self.ballot_number = ballot_number
    self.accepted = accepted

class P2aMessage(Message):
  def __init__(self, src, ballot_number, slot_number, command):
    Message.__init__(self, src)
    self.ballot_number = ballot_number
    self.slot_number = slot_number
    self.command = command

class P2bMessage(Message):
  def __init__(self, src, ballot_number, slot_number):
    Message.__init__(self, src)
    self.ballot_number = ballot_number
    self.slot_number = slot_number

class PreemptedMessage(Message):
  def __init__(self, src, ballot_number):
    Message.__init__(self, src)
    self.ballot_number = ballot_number

class AdoptedMessage(Message):
  def __init__(self, src, ballot_number, accepted):
    Message.__init__(self, src)
    self.ballot_number = ballot_number
    self.accepted = accepted

class DecisionMessage(Message):
  def __init__(self, src, slot_number, command):
    Message.__init__(self, src)
    self.slot_number = slot_number
    self.command = command

class RequestMessage(Message):
  def __init__(self, src, command):
    Message.__init__(self, src)
    self.command = command

class ProposeMessage(Message):
  def __init__(self, src, slot_number, command):
    Message.__init__(self, src)
    self.slot_number = slot_number
    self.command = command
```

replica.py

This is the Python code for a replica, corresponding to Figure 1.

```
from process import Process
from message import ProposeMessage,DecisionMessage,RequestMessage
from utils import *
import time

class Replica(Process):
  def __init__(self, env, id, config):
    Process.__init__(self)
    self.slot_in = self.slot_out = 1
    self.proposals = {}
    self.decisions = {}
    self.requests = []
    self.config = config
    self.id = id
    self.env = env
    self.env.addProc(self)

  def propose(self):
    while len(self.requests) != 0 and self.slot_in < self.slot_out+WINDOW:
      if self.slot_in > WINDOW and self.slot_in-WINDOW in self.decisions:
        if isinstance(self.decisions[self.slot_in-WINDOW],ReconfigCommand):
          r,a,l = self.decisions[self.slot_in-WINDOW].config.split(';')
          self.config = Config(r.split(','),a.split(','),l.split(','))
          print self.id, ": new config:", self.config
      if self.slot_in not in self.decisions:
        cmd = self.requests.pop(0)
        self.proposals[self.slot_in] = cmd
        for ldr in self.config.leaders:
          self.sendMessage(ldr, ProposeMessage(self.id,self.slot_in,cmd))
      self.slot_in +=1

  def perform(self, cmd):
    for s in range(1, self.slot_out):
      if self.decisions[s] == cmd:
        self.slot_out += 1
        return
    if isinstance(cmd, ReconfigCommand):
      self.slot_out += 1
      return
    print self.id, ": perform",self.slot_out, ":", cmd
    self.slot_out += 1

  def body(self):
    print "Here I am: ", self.id
    while True:
      msg = self.getNextMessage()
      if isinstance(msg, RequestMessage):
        self.requests.append(msg.command)
      elif isinstance(msg, DecisionMessage):
        self.decisions[msg.slot_number] = msg.command
        while self.slot_out in self.decisions:
          if self.slot_out in self.proposals:
            if self.proposals[self.slot_out]!=self.decisions[self.slot_out]:
              self.requests.append(self.proposals[self.slot_out])
            del self.proposals[self.slot_out]
          self.perform(self.decisions[self.slot_out])
      else:
        print "Replica: unknown msg type"
      self.propose()
```

acceptor.py

Implementation of the acceptor process, closely corresponding to Figure 4.

```
from utils import PValue
from process import Process
from message import P1aMessage,P1bMessage,P2aMessage,P2bMessage

class Acceptor(Process):
  def __init__(self, env, id):
    Process.__init__(self)
    self.ballot_number = None
    self.accepted = set()
    self.id = id
    self.env = env
    self.env.addProc(self)

  def body(self):
    print "Here I am: ", self.id
    while True:
      msg = self.getNextMessage()
      if isinstance(msg, P1aMessage):
        if msg.ballot_number > self.ballot_number:
          self.ballot_number = msg.ballot_number
        self.sendMessage(msg.src,
                         P1bMessage(self.id,
                                    self.ballot_number,
                                    self.accepted))
      elif isinstance(msg, P2aMessage):
        if msg.ballot_number == self.ballot_number:
          self.accepted.add(PValue(msg.ballot_number,
                                   msg.slot_number,
                                   msg.command))
        self.sendMessage(msg.src,
                         P2bMessage(self.id,
                                    self.ballot_number,
                                    msg.slot_number))
```

commander.py

Implementation of the commander process in Figure 6(a).

```
from message import P2aMessage,P2bMessage,PreemptedMessage,DecisionMessage
from process import Process
from utils import Command

class Commander(Process):
  def __init__(self, env, id, leader, acceptors, replicas,
               ballot_number, slot_number, command):
    Process.__init__(self)
    self.leader = leader
    self.acceptors = acceptors
    self.replicas = replicas
    self.ballot_number = ballot_number
    self.slot_number = slot_number
    self.command = command
    self.id = id
    self.env = env
    self.env.addProc(self)

  def body(self):
    waitfor = set()
    for a in self.acceptors:
      self.sendMessage(a, P2aMessage(self.id, self.ballot_number,
                                     self.slot_number, self.command))
      waitfor.add(a)

    while True:
      msg = self.getNextMessage()
      if isinstance(msg, P2bMessage):
        if self.ballot_number == msg.ballot_number and msg.src in waitfor:
          waitfor.remove(msg.src)
          if len(waitfor) < float(len(self.acceptors))/2:
            for r in self.replicas:
              self.sendMessage(r, DecisionMessage(self.id,
                                                  self.slot_number,
                                                  self.command))
            return
        else:
          self.sendMessage(self.leader, PreemptedMessage(self.id,
                                                         msg.ballot_number))
          return
```

scout.py

Implementation of the scout process in Figure 6(b).

```
from process import Process
from message import P1aMessage,P1bMessage,PreemptedMessage,AdoptedMessage

class Scout(Process):
  def __init__(self, env, id, leader, acceptors, ballot_number):
    Process.__init__(self)
    self.leader = leader
    self.acceptors = acceptors
    self.ballot_number = ballot_number
    self.id = id
    self.env = env
    self.env.addProc(self)

  def body(self):
    waitfor = set()
    for a in self.acceptors:
      self.sendMessage(a, P1aMessage(self.id, self.ballot_number))
      waitfor.add(a)

    pvalues = set()
    while True:
      msg = self.getNextMessage()
      if isinstance(msg, P1bMessage):
        if self.ballot_number == msg.ballot_number and msg.src in waitfor:
          pvalues.update(msg.accepted)
          waitfor.remove(msg.src)
          if len(waitfor) < float(len(self.acceptors))/2:
            self.sendMessage(self.leader,
                             AdoptedMessage(self.id,
                                            self.ballot_number,
                                            pvalues))
            return
        else:
          self.sendMessage(self.leader,
                           PreemptedMessage(self.id,
                                            msg.ballot_number))
          return
      else:
        print "Scout: unexpected msg"
```

leader.py

Implementation of the leader process in Figure 7.

```
from utils import BallotNumber
from process import Process
from commander import Commander
from scout import Scout
from message import ProposeMessage,AdoptedMessage,PreemptedMessage

class Leader(Process):
  def __init__(self, env, id, config):
    Process.__init__(self)
    self.id = id
    self.ballot_number = BallotNumber(0, self.id)
    self.active = False
    self.proposals = {}
    self.config = config
    self.env = env
    self.env.addProc(self)

  def body(self):
    print "Here I am: ", self.id
    Scout(self.env, "scout:%s:%s" % (str(self.id), str(self.ballot_number)),
          self.id, self.config.acceptors, self.ballot_number)
    while True:
      msg = self.getNextMessage()
      if isinstance(msg, ProposeMessage):
        if msg.slot_number not in self.proposals:
          self.proposals[msg.slot_number] = msg.command
          if self.active:
            Commander(self.env,
                      "commander:%s:%s:%s" % (str(self.id),
                                              str(self.ballot_number),
                                              str(msg.slot_number)),
                      self.id, self.config.acceptors, self.config.replicas,
                      self.ballot_number, msg.slot_number, msg.command)
      elif isinstance(msg, AdoptedMessage):
        if self.ballot_number == msg.ballot_number:
          max = {}
          for pv in msg.accepted:
            bn = max[pv.slot_number]
            if bn == None or bn < pv.ballot_number:
              max[pv.slot_number] = pv.ballot_number
              self.proposals[pv.slot_number] = pv.command
          for sn in self.proposals:
            Commander(self.env,
                      "commander:%s:%s:%s" % (str(self.id),
                                              str(self.ballot_number),
                                              str(sn)),
                      self.id, self.config.acceptors, self.config.replicas,
                      self.ballot_number, sn, self.proposals.get(sn))
          self.active = True
      elif isinstance(msg, PreemptedMessage):
        if msg.ballot_number > self.ballot_number:
          self.ballot_number = BallotNumber(msg.ballot_number.round+1,
                                            self.id)
          Scout(self.env, "scout:%s:%s" % (str(self.id),
                                           str(self.ballot_number)),
                self.id, self.config.acceptors, self.ballot_number)
        self.active = False
      else:
        print "Leader: unknown msg type"
```

env.py

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

```
import os, signal, sys, time
from acceptor import Acceptor
from leader import Leader
from message import RequestMessage
from process import Process
from replica import Replica
from utils import *

NACCEPTORS = 3
NREPLICAS = 2
NLEADERS = 2
NREQUESTS = 10
NCONFIGS = 3

class Env:
  def __init__(self):
    self.procs = {}

  def sendMessage(self, dst, msg):
    if dst in self.procs:
      self.procs[dst].deliver(msg)

  def addProc(self, proc):
    self.procs[proc.id] = proc
    proc.start()

  def removeProc(self, pid):
    del self.procs[pid]

  def run(self):
    initialconfig = Config([], [], [])
    c = 0

    for i in range(NREPLICAS):
      pid = "replica: %d" % i
      Replica(self, pid, initialconfig)
      initialconfig.replicas.append(pid)
    for i in range(NACCEPTORS):
      pid = "acceptor: %d.%d" % (c,i)
      Acceptor(self, pid)
      initialconfig.acceptors.append(pid)
    for i in range(NLEADERS):
      pid = "leader: %d.%d" % (c,i)
      Leader(self, pid, initialconfig)
      initialconfig.leaders.append(pid)
    for i in range(NREQUESTS):
      pid = "client: %d.%d" % (c,i)
      for r in initialconfig.replicas:
        cmd = Command(pid,0,"operation %d.%d" % (c,i))
        self.sendMessage(r,RequestMessage(pid,cmd))
        time.sleep(1)

    for c in range(1, NCONFIGS):
      # Create new configuration
      config = Config(initialconfig.replicas, [], [])
      for i in range(NACCEPTORS):
        pid = "acceptor: %d.%d" % (c,i)
        Acceptor(self, pid)
        config.acceptors.append(pid)
      for i in range(NLEADERS):
        pid = "leader: %d.%d" % (c,i)
        Leader(self, pid, config)
        config.leaders.append(pid)
      # Send reconfiguration request
      for r in config.replicas:
        pid = "master: %d.%d" % (c,i)
        cmd = ReconfigCommand(pid,0,str(config))
        self.sendMessage(r, RequestMessage(pid, cmd))
        time.sleep(1)
      for i in range(WINDOW-1):
        pid = "master: %d.%d" % (c,i)
        for r in config.replicas:
          cmd = Command(pid,0,"operation noop")
          self.sendMessage(r, RequestMessage(pid, cmd))
          time.sleep(1)
      for i in range(NREQUESTS):
        pid = "client: %d.%d" % (c,i)
        for r in config.replicas:
          cmd = Command(pid,0,"operation %d.%d"%(c,i))
          self.sendMessage(r, RequestMessage(pid, cmd))
          time.sleep(1)

  def terminate_handler(self, signal, frame):
    self._graceexit()

  def _graceexit(self, exitcode=0):
    sys.stdout.flush()
    sys.stderr.flush()
    os._exit(exitcode)

def main():
  e = Env()
  e.run()
  signal.signal(signal.SIGINT, e.terminate_handler)
  signal.signal(signal.SIGTERM, e.terminate_handler)
  signal.pause()


if __name__=='__main__':
  main()
```

ACKNOWLEDGMENTS

Many people provided invaluable input to this description of Paxos, including Lorenzo Alvisi, Mark Bickford, Robert Constable, Danny Dolev, David Guaspari, Chi Ho, Annie Liu, Stavros Nikolaou, Fred Schneider, Emin Gün Sirer, Scott Stoller, and the anonymous reviewers of *ACM Computing Surveys*.

REFERENCES

Peter Alvaro, Tyson Condie, Neil Conway, Joseph M. Hellerstein, and Russell Sears. 2010. I do declare: Consensus in a logic language. *SIGOPS Operating Systems Review* 43, 4, 25–30.

Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing memory robustly in message-passing systems. *Journal of the ACM* 42, 1, 124–142.

Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. 2003. Deconstructing Paxos. *SIGACT News* 34, 1, 47–67.

William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. 2011. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. 141–154.

Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. 1993. The primary-backup approach. In *Distributed Systems* (2nd ed.), S. Mullender (Ed.). ACM Press/Addison-Wesley, New York, NY, 199–216.

Mike Burrows. 2006. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. 335–350.

Tushar Deepak Chandra and Sam Toueg. 1991. Unreliable failure detectors for asynchronous systems (preliminary version). In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing (PODC'91)*. ACM, New York, NY, 325–340.

Roberto De Prisco, Butler W. Lampson, and Nancy Lynch. 2000. Revisiting the Paxos algorithm. *Theoretical Computer Science* 243, 1–2, 35–91.

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM* 32, 2, 374–382.

Eli Gafni and Leslie Lamport. 2003. Disk Paxos. *Distributed Computing* 16, 1, 1–20.

Cary G. Gray and David R. Cheriton. 1989. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP'89)*. ACM, New York, NY, 202–210.

Jonathan Kirsch and Yair Amir. 2008a. *Paxos for System Builders*. Technical Report CNDS-2008-2. Johns Hopkins University, Baltimore, MD.

Jonathan Kirsch and Yair Amir. 2008b. Paxos for system builders: An overview. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS'08)*. ACM, New York, NY, Article No. 3.

Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Communations of the ACM* 21, 7, 558–565.

Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems* 16, 2, 133–169.

Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News* 32, 4, 51–58.

Leslie Lamport. 2005. *Generalized Consensus and Paxos*. Technical Report MSR-TR-2005-33. Microsoft Research, Mountain View, CA. Available at http://research.microsoft.com/pubs/64631/tr-2005-33.pdf

Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19, 2, 79–103.

Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2008. *Stoppable Paxos*. Technical Report. Microsoft Research, Mountain View, CA. Available at http://research.microsoft.com/apps/pubs/default.aspx?id=101826

Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2009. Vertical Paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC'09)*. ACM, New York, NY, 312–313.

Leslie Lamport and Mike Massa. 2004. Cheap Paxos. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*. IEEE, Los Alamitos, CA, 307–315.

Butler W. Lampson. 1996. How to build a highly available system using consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG'96)*. 1–17.

Butler W. Lampson. 2001. The ABCD's of Paxos. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC'01)*. ACM, New York, NY, 13–14.

Harry C. Li, Allen Clement, Amitanand S. Aiyer, and Lorenzo Alvisi. 2007. The Paxos register. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS'07)*. IEEE, Los Alamitos, CA, 114–126.

Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. 2006. The SMART way to migrate replicated stateful services. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys'06)*. ACM, New York, NY, 103–115.

Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building efficient replicated state machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. 369–384.

David Mazières. 2007. *Paxos Made Practical*. Technical Report. Stanford University, Stanford, CA. Available at http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf

Hein Meling and Leander Jehl. 2013. Tutorial summary: Paxos explained from scratch. In *Principles of Distributed Systems*. Lecture Notes in Computer Science, Vol. 8304. Springer, 1–10.

C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems* 17, 1, 94–162.

Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in egalitarian parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 358–372.

Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC'88)*. ACM, New York, NY, 8–17.

Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'14)*. 305–319.

Richard D. Schlichting and Fred B. Schneider. 1983. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems* 1, 3, 222–238.

Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 22, 4, 299–319.

Robert H. Thomas. 1979. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems* 4, 2, 180–209.

Robbert van Renesse, Nicolas Schiper, and Fred B. Schneider. 2014. Vive la différence: Paxos vs. Viewstamped Replication vs. Zab. *IEEE Transactions on Dependable and Secure Computing* PP, 99, 1.

Robbert van Renesse, Fred B. Schneider, and Johannes Gehrke. 2011. *Nerio: Leader Election and Edict Ordering*. Technical Report. Cornell University, Ithaca, NY. Available at http://hdl.handle.net/1813/23631

Received August 2013; revised June 2014; accepted September 2014