Vertical Paxos and Primary-Backup Replication

Leslie Lamport, Dahlia Malkhi, Lidong Zhou
Microsoft Research

9 February 2009

Abstract

We introduce a class of Paxos algorithms called Vertical Paxos, in which reconfiguration can occur in the middle of reaching agreement on an individual state-machine command. Vertical Paxos algorithms use an auxiliary configuration master that facilitates agreement on reconfiguration. A special case of these algorithms leads to traditional primary-backup protocols. We show how primary-backup systems in current use can be viewed, and shown to be correct, as instances of Vertical Paxos algorithms.

Contents

1 Introduction

Large-scale distributed storage systems built over failure-prone commodity components are increasingly popular. Failures are the norm in those systems, and replication is often the solution to data reliability. We might expect these systems to adopt well-studied consensus algorithms and the replicated state-machine approach that they enable. While some consensus algorithms, such as Paxos [5], have started to find their way into those systems, their uses are limited mostly to the maintenance of the global configuration information in the system, not for the actual data replication. A clear gap remains between the well-known consensus algorithms and the practical replication protocols in real systems.

The gap is not accidental; the abstract models for defining the classic consensus algorithms do not fully capture the requirements from those distributed systems. The classic consensus problem is defined on a single fixed set of n processes as a *replica group* with each process as a *replica*, where at most f of the n processes can fail. In practice, a distributed system consists of a large number of overlapping replica groups, each responsible for maintaining a subset of the system's data. When replicas fail, the system must replace the failed replicas with new ones through *reconfiguration*, before more replica failures lead to permanent data loss. For practical replication protocols, the global resilience, the system throughput, and the cost of reconfiguration tend to be more important than the fault-tolerance of a single replica group or the number of message rounds.

The gap between consensus algorithms and practical replication protocols is not fundamental. Here, we bridge it by viewing Paxos not as a particular algorithm, but as a family of algorithms based on a particular way of achieving consensus. We focus on *primary-backup replication*, a class of replication protocols that has been widely used in practical distributed systems. We develop two new algorithms, in a family of Paxos algorithms called Vertical Paxos, that capture the essence of primary-backup replication. (The only previous algorithm in this family that we know of is an unpublished version of Cheap Paxos [6].) A primary-backup replication protocol becomes a simple instance of our second algorithm, which we call Vertical Paxos II.

Vertical Paxos not only provides a solid theoretical foundation for existing primary-backup replication protocols, but also offers a new way to look at primary-backup replication, leading to further improvements that address issues arising in practical systems. For example, when new processes are to replace failed replicas to restore the desirable level of fault tolerance, *state transfer* to the new processes is necessary before the reconfiguration can take place. In a distributed storage system, state transfer usually involves copying data across machines—an often costly operation. The system therefore faces the difficult decision of either allowing the replica group to continue with a reduced level of resilience or disrupting the service during state transfer. Our first algorithm, Vertical Paxos I addresses this issue by allowing a replica group to operate with the restored resilience, while enabling state transfer concurrently.

Vertical Paxos is also significant from a theoretical perspective. Its value goes beyond the special case of primary-backup replication, offering a different way of reconfiguring state machines from the one described in the original Paxos paper. In Vertical Paxos, reconfiguration relies on an external master, itself implemented as a replicated state machine. This corresponds to an attractive system architecture in which the external master manages the global system state, participates in the reconfiguration of any replica group in the system, and helps achieve the global optimal resilience.

2 Overview of Vertical Paxos

Paxos implements a state machine by executing a sequence of logically separate instances of the Paxos consensus algorithm, instance i choosing the i^{th} state machine command. The consensus algorithm is executed by four conceptually separate but not necessarily disjoint sets of processes: *clients*, *leaders*, *acceptors*, and *learners*. Leaders propose commands that have been requested by clients, acceptors choose a single proposed command, and learners learn what command has been chosen.

Vertical Paxos is very much like traditional Paxos, except for two key differences that stem from experiences with real systems that implement state-machine replication.

Read/Write quorums: Practical protocols use primary-backup structure for better system-wide resilience. Vertical Paxos achieves this structure by distinguishing between *read* and *write* quorums. It has been observed before that Paxos can be generalized by utilizing these two kinds of quorums [7], but this offered little practical benefit in earlier versions of Paxos.

Auxiliary configuration master: Although it is possible to let the state machine reconfigure itself, in practical settings, a separate configuration master often makes reconfigurations decisions. Vertical Paxos makes use of an auxiliary master to allow the set of acceptors to change within each individual consensus instance. The master determines the set of acceptors and the leader for every configuration. The use of a configuration master allows a more efficient implementation of the individual state machines. In particular, a master allows a state-machine implementation to tolerate k failures using only $k + 1$ processors instead of the $2k + 1$ processors required without it [2].

The configuration master need be called upon by the processors executing an individual state machine only for reconfiguration—that is, for changing the set of processors that are executing the state machine. Since reconfiguration is infrequent—usually in response to a processor failure—executing the master requires little processing power. It is therefore practical to implement a very reliable master by using a sufficient number of different processors that spend most of their time doing other things.

Vertical Paxos integrates these two key ingredients by changing the configuration of acceptors within individual consensus instances. A Paxos consensus algorithm performs a sequence of numbered ballots, each with a unique leader. (In normal operation, only a single ballot is performed in each consensus instance until that ballot's leader fails and a new leader is chosen.) Think of the ballots as arranged in a two-dimensional array, each vertical column consisting of all the ballots within a single instance arranged according to their number. In standard "horizontal" Paxos algorithms, configurations can change only as we move horizontally; they are unchanged when we move vertically (within a single instance). 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 instance.

When a new ballot and its leader is chosen in Paxos, the leader must communicate with acceptors from lower-numbered ballots. Since different ballot numbers have different configurations, a leader in Vertical Paxos must communicate with acceptors from past configurations. The two variants of Vertical Paxos address this differently.

- Algorithm Vertical Paxos I uses the following procedure to eliminate the dependence on acceptors from lower-numbered ballots. When a configuration changes in Vertical Paxos, the new configuration becomes active right away. The previous configuration remains active only for storing old information, the new one also accepts new commands. When the state of the previous configuration has been transferred to the new configuration, the new leader informs the master that this has happened. The master will then tell all future leaders that they need not access that old configuration.

- Suppose a new ballot $b + 1$ is begun, but its leader fails before the state transfer from the ballot b configuration is complete. A new ballot $b + 2$ then begins, but its leader could also fail before any state transfer occurs. This could continue happening until ballot $b + 42$ begins, and its leader must communicate with acceptors from ballots b through $b + 42$. Algorithm Vertical Paxos II avoids the dependence on so many configurations by having a new configuration initially inactive. The new leader notifies the master when the state transfer from the previous configuration is complete, and the master then activates the new configuration. The leader communicates only with acceptors from the new configuration and the previous active configuration. However, a number of new ballots could be started, but remain forever inactive because their leaders failed, before a new configuration becomes active and starts accepting new commands. Vertical Paxos II is especially useful for primary-backup replication.

3 The Primary Backup Case

In a Vertical Paxos consensus algorithm, a ballot leader must access a write quorum of its own ballot and read quorums of one or more lower-numbered ballots. Because a reconfiguration is usually performed in response to a failure, processes participating in lower-numbered ballots are more likely to have failed. We therefore want to keep read quorums small. Read and write quorums must intersect, so there is a tradeoff: making read quorums smaller requires making write quorums larger.

We obtain what is probably the most interesting case of Vertical Paxos by letting read quorums be as small as possible—namely, making any single acceptor a read quorum, so the only write quorum is the set of all acceptors. These are the quorums that allow k-fault tolerance with only $k+1$ acceptors. Suppose that in Vertical Paxos II we also always make the leader one of the acceptors and, upon reconfiguration, always choose the new leader from among the current acceptors. The new leader by itself is a read quorum for the previous ballot. Hence, it can perform the state transfer all by itself, with no messages. (It will still have to interact with the master and may have to exchange messages with acceptors in the new configuration.) If we call the leader the *primary* and all other acceptors *backups*, then we have a traditional primary-backup system.

The optimizations that are possible with traditional primary-backup systems all seem to have corresponding optimizations in Vertical Paxos. One example is the way efficient local reads can be done from the primary while maintaining linearizability. The primary obtains a lease that gives it permission to reply directly to reads from its local state. A new primary cannot be chosen until the lease expires. In a similar way, a Vertical Paxos leader can use a lease to reply immediately to commands that do not change the state. A new leader cannot be chosen until the previous leader's lease has expired.

Boxwood employs this local reading protocol for Replicated Logical Devices [8]. Boxwood also uses another optimization that applies as well to Vertical Paxos. A Boxwood configuration consists of a primary and a single backup. When either of them fails, the other takes over and continues in a solo configuration. It may start processing new client requests immediately upon the reconfiguration decree by the configuration master. This applies to Vertical Paxos because the solo process is both the leader and a write quorum of the new configuration, as well as a read quorum of the previous (two-acceptor) configuration.

Most primary-backup protocols maintain a single active configuration, as captured by Vertical Paxos II. While having a single active configuration might seem like a simple and natural choice, the master activates a new configuration only after the state transfer from the previous configuration is complete. In practical systems, the state transfer tends to involve copying a large amount of data and is therefore costly. By allowing multiple active configurations, Vertical Paxos I decouples state transfer from reconfiguration: a new configuration can be activated to accept new requests, while the state is transferred from the old configuration. This simple variation of the existing primary-backup protocols is practically significant.

4 Vertical Paxos

Paxos is an instance of the state-machine approach, in which a sequence of commands is chosen by executing a separate instance of a consensus algorithm to choose the i^{th} command, for each i. What makes Paxos efficient is that part of the consensus algorithm is performed simultaneously for all the different instances. However, correctness depends only on correctness of the consensus algorithm. We therefore concentrate on the consensus algorithm.

4.1 Paxos Preliminaries

A consensus algorithm must solve the following problem. There is a set of *client* processes, each of which can propose a value from some set *PValues* of proposable values. The algorithm must choose a unique value that was proposed by some client.

A Paxos consensus algorithm uses a sequence of ballots, with numbers belonging to an infinite set *Ballots* sequentially ordered by $<$ and having minimal element 0. In Vertical Paxos I, *Ballots* is the set *Nat* of natural numbers. In Vertical Paxos II, the numbers of ballots that are started but never activated are not considered to be in *Ballots*, so *Ballots* is a subset of *Nat*. For any non-zero b in *Ballots*, let $Prev(b)$ be the next lower element of *Ballots*, and let $Prev(0)$ be a value less than 0 that is not a ballot number.

For each ballot number b, there are two sets of processes: the sets $RQuorums(b)$ and $WQuorums(b)$ of *read* and *write* *b-quorums*. They satisfy the property that every write b-quorum has a non-empty intersection with every read b-quorum and every write b-quorum. An *acceptor* is a process that is in a read or write b-quorum, for some ballot number b.

In the original (horizontal) Paxos consensus algorithm, *Ballots* is the set of natural numbers, the sets of read and write b-quorums are the same, and the sets of b-acceptors and b-quorums are independent of b. In Vertical Paxos, the master computes the read and write b-quorums in reaction to changes in the system; and in Vertical Paxos II it dynamically adds elements to *Ballots*. The master may wait to choose if b is in *Ballots* and what the b-quorums are until it needs to know, but once made its choice is fixed. For simplicity, we assume that *Ballots* and the read and

write quorums are constant, fixed in advance by an oracle that predicts what the master will do. (Formally, the oracle is a prophecy variable [1].)

Each acceptor a maintains a value $vote[a][b]$ for each ballot number b. Its value is initially equal to a non-proposable value *None* and can be set by the acceptor to a proposable value v in an action that we call *a voting for v in ballot b*. An acceptor cannot change its vote. In an implementation, acceptor a does not need to remember the value $vote[a][b]$ for all ballot numbers b, but it is easiest to explain the algorithm in terms of the complete array $vote[a]$.

A proposable value v is *chosen in* ballot b iff all acceptors in a write b-quorum have voted for v in ballot b; and v is *chosen* iff it is chosen in some ballot. We now explain how Paxos ensures that at most one proposable value is chosen.

Acceptor a also maintains a ballot number $maxBallot[a]$, initially 0, that never decreases. The acceptor will not vote in a ballot whose number is less than $maxBallot[a]$. We say that a value v is *choosable in* ballot b iff v is chosen in b or could become chosen in b by some acceptors (with $maxBallot[a] \leq b$) voting for v in ballot b. We define a proposable value v to be *safe at* ballot number b iff no proposable value other than v is choosable in any ballot numbered less than b. (Thus, all values are safe at 0.) The fundamental invariant maintained by a Paxos consensus algorithm is that an acceptor votes in any ballot b only for a proposable value safe at b. It can be shown that this implies that two different values cannot be chosen.

To get a value v chosen, we just have to choose a ballot number b and get a write b-quorum Q of acceptors to vote for v in ballot b. We can do this by choosing Q so that $vote[a][b] = None$ and $b \geq MaxBallot[a]$ for every a in Q. However, we must choose v so it is safe at b. We now show how such a v is chosen. First we observe that because read and write b-quorums have non-empty intersection, no value is choosable in a ballot b if there is a read b-quorum Q such that $maxBallot[a] > b$ and $vote[a][b] = None$ for every acceptor a in Q. Vertical Paxos maintains the invariant that different acceptors cannot vote for different values in the same ballot. We can then use algorithm *FindSafe* of Figure 1 to compute a value safe at ballot number b.

Algorithm *FindSafe* is written in the PlusCal algorithm language [4], except that the grain of atomicity is not shown. The algorithm is designed to be as general as possible, rather than to be efficient, so it can be implemented in situations where complete information about the values of variables may not be known. The PlusCal statement **with** $(x \in S)${ body } is executed by waiting until the set S is not empty and executing the body with x a nondeterministically chosen element of S. The statement **await** P waits until P is true. The **either/or** statement nondeterministically chooses to execute one of its two clauses; however, the choice is deferred until it is determined which of the clauses are executable, so deadlock cannot be caused by making the wrong choice.

The result is computed in the variable *safeVal*, which upon termination is left equal either to a proposable value safe at b or the special (non-proposable) value *AllSafe* indicating that all proposable values are safe at b. It is not hard to show that algorithm *FindSafe* is correct if the entire computation is performed atomically, while the variables *vote* and *maxBallot* do not change. Because acceptors vote at most once in any ballot and $maxBallot[a]$ does not decrease, the algorithm remains correct even if executed nonatomically, with the waiting conditions evaluated repeatedly, as long as: (i) the read of $vote[acc][c]$ in the **with** body is atomic, and (ii) for each individual acceptor a and ballot number c, the values of $maxBallot[a]$ and $vote[a][c]$ are both read in a single atomic step when evaluating the **await** condition.

Algorithm *FindSafe* is used in ordinary (horizontal) Paxos. However, it may require knowing the votes of acceptors in c-quorums for every $c \leq b$. This is unacceptable in Vertical Paxos,

$safeVal := AllSafe$;
$c := Prev(b)$;
while ($(safeVal = AllSafe) \ \wedge \ (c \geq 0)$) **{**
either with ($acc \in \{a \in Acceptors : vote[a][c] \neq None\}$) **{**
$safeVal := vote[acc][c]$ **}**
or **await** $\exists\, Q \in RQuorums(c) : \forall\, a \in Q :$
$(maxBallot[a] > c) \wedge (vote[a][c] = None)$;
$c := Prev(c)$ **}**

Figure 1: Algorithm *FindSafe* for computing a value safe at ballot b.

where acceptors that participated in lower-numbered ballots may have been reconfigured out of the system. To solve this problem, we first define ballot b to be *complete* if a value has been chosen in b or all values are safe at b. We can modify algorithm *FindSafe* so it stops at a complete ballot rather than at ballot 0. That is, if ballot number d is complete, we can replace the condition $c \geq 0$ by $c \geq d$ in the **while** test. We call the modified algorithm *VFindSafe*.

4.2 Vertical Paxos I

We describe our algorithms using PlusCal. The PlusCal code describes what actions processes are allowed to perform. It says nothing about when they should or should not perform those actions, which affects only liveness. The liveness property satisfied by a Vertical Paxos algorithm is similar to that of ordinary Paxos algorithms, except with the added complication caused by reconfiguration—a complication that arises in any algorithm employing reconfiguration. A discussion of liveness is beyond the scope of this paper.

In Paxos, the invariant that two different acceptors do not vote for different values in the same ballot is maintained by letting each ballot have a unique *leader* process that tells acceptors what value they can vote for in that ballot. The same physical processor can execute the leader process for infinitely many ballots.

In ordinary Paxos, the leader of a ballot decides for itself when to begin execution. Our Vertical Paxos algorithms assume a reliable service that does this. We represent the service as a single *Master* process. The master will actually be implemented by a network of processors running a reliable state-machine implementation, such as (horizontal) Paxos.

The ballot b leader process first waits until it receives a *newBallot* message *msg* (one with *msg.type* equal to the string "newBallot") with $msg.bal = b$, which is sent by the master. (The master will send this message in response to some request that does not concern us.) The value of *msg.completeBal* is the largest ballot number that the master knows to be complete.

As in ordinary Paxos, a ballot proceeds in two phases. In phase 1, the leader executes algorithm *VFindSafe* by sending a $1a$ message to acceptors and receiving $1b$ messages in reply. In phase 2, it directs acceptors to choose a value by sending a $2a$ message; it learns that the value has been chosen from the $2b$ messages with which they reply. Recall that *VFindSafe* either returns a single proposable value v safe at b or else returns *AllSafe* indicating that all values are safe at b. In the first case, the leader executes phase 2 and waits until v has been chosen. In either case, the leader then sends a *complete* message informing the master that ballot b is complete. If *VFindSafe*

returned *AllSafe*, the leader waits to receive a client proposal in a *clientReq* message and then begins phase 2 to get that value chosen. In this case, it terminates without waiting to receive the $2b$ messages.

The code that the leader and acceptors execute in the two phases is derived from the following meanings of a message m of each type:

$1a$ Request each acceptor a to set $maxBallot[a]$ to $m.bal$ and report the value of $vote[a][m.prevBal]$. (Acceptor a ignores the message if $m.bal < maxBallot[a]$.)

$1b$ Asserts that $maxBallot[m.acc] \geq m.bal$ and $vote[m.acc][m.voteBal] = m.val$.

$2a$ Requests each acceptor a to set $vote[a][m.bal]$ to $m.val$. (Acceptor a ignores the message if $m.bal < maxBallot[a]$.)

$2b$ Reports that acceptor $m.acc$ has voted for $m.val$ in ballot $m.bal$.

The code of the leader processes is in Figure 2 and that of the other processes is in Figure 3. In PlusCal, an atomic step is an execution from one label to the next, and the identifier *self* is the name of the current process. The leader of ballot number b has process name b. The elements of *Acceptors* are the names of the acceptors, and we assume a set *Clients* of client process names.

In PlusCal, $[f_1 \mapsto v_1, \ldots, f_n \mapsto v_n]$ is a record r whose f_i field $r.f_i$ equals v_i. We represent message passing by a *Send* operation that simply broadcasts a message, which can be received by any process interested in receiving it. For any record r, we let $MsgsRcvdWith(r)$ be the set of messages received by a process that have their corresponding fields equal to the fields of the record r.

4.3 Vertical Paxos II

By having the master keep track of the largest complete ballot number, Vertical Paxos I limits the number of different ballots whose acceptors a leader must query to choose a safe value. In Vertical Paxos II, a leader has to query only acceptors from a single previous ballot. It does this by letting *Ballots* be a subset of the set of natural numbers. The master adds a number b to the set of ballot numbers, a procedure we call *activating* ballot b, when doing so makes b a complete ballot. (Remember that we are assuming that the oracle predicted that the master would add b to the set of ballot numbers, so b is already an element of the constant set *Ballots*.) Ballot numbers are activated in increasing order.

In this algorithm, the master's *newBallot* message m for ballot b has $m.prevBal$ equal to the largest currently activated ballot number (which is less than b). As in Vertical Paxos I, the ballot b leader performs phase 1 and, if *VFindSafe* returns a single value, it performs phase 2; and it then sends its *complete* message. It does all this assuming that b is the next ballot to be activated. If the master has not activated any ballot since sending its *newBallot* message to b, then it activates b and sends an *activated* message to the leader. Upon receipt of this message, the leader performs phase 2 if it has not already done so. If ballot b is not activated, then the actions performed have no effect because b is not a ballot number (so it doesn't matter what the value $vote[a][b]$ is for a process a). The leaders' code is in Figure 4 and the master's code is in Figure 5. For coding convenience, ballot 0 is not performed. Note that the leader's **while** loop in Vertical Paxos I has been eliminated, since the leader assumes that *prevBal* is the next lower ballot number.

```
process (BallotLeader ∈ Nat)
  variables completeBal, prevBal, safeVal = AllSafe ; {
b1: with (msg ∈ MsgsRcvdWith( [type ↦ "newBallot", bal ↦ self] )) {
      completeBal := msg.completeBal ; }
    prevBal := self − 1;
b2: while ((safeVal = AllSafe) ∧ (prevBal ≥ completeBal)) {
b3:   Send( [type ↦ "1a", bal ↦ self , prevBal ↦ prevBal]) ;
      either with (msg ∈ {m ∈ MsgsRcvdWith([type ↦ "1b",
                                            voteBal ↦ prevBal]) :
                                  m.val ≠ None}){
                safeVal := msg.val }
      or      await ∃ Q ∈ RQuorums(prevBal) : ∀ a ∈ Q :
                    MsgsRcvdWith([type ↦ "1b",
                                  bal ↦ self,
                                  voteBal ↦ prevBal,
                                  val ↦ None,
                                  acc ↦ a]) ≠ {};
      prevBal := prevBal − 1; };
b4: if (safeVal ≠ AllSafe) {
      Send( [type ↦ "2a", bal ↦ self, val ↦ safeVal] ) ;
b5:   await ∃ Q ∈ WQuorums(self) :
              ∀ a ∈ Q :
                MsgsRcvdWith([type ↦ "2b", acc ↦ a, bal ↦ self])
                    ≠ {} ; };
b6: Send( [type ↦ "complete", bal ↦ self ] ) ;
b7: if (safeVal = AllSafe) {
      with (msg ∈ MsgsRcvdWith( [type ↦ "clientReq"] )) {
        Send( [type ↦ "2a", bal ↦ self, val ↦ msg.val] ) ; } } }
```

Figure 2: The leader processes of Vertical Paxos I.

```
    process (MasterProc = Master)
      variable completeBallot = 0, nextBallot = 0 ; {
m1:   while (TRUE) {
        either {Send( [type ↦ "newBallot", bal ↦ nextBallot,
                       completeBal ↦ completeBallot] ) } ;
               nextBallot := nextBallot + 1; }
        or     with (msg ∈ MsgsRcvdWith( [type ↦ "complete"] )) {
                 if (msg.bal > completeBallot) {
                   completeBallot := msg.bal; } } } }

    process (Acceptor ∈ Acceptors)
      variables vote = [b ∈ Ballots ↦ None] ; maxBallot = 0 ; {
a1:   while (TRUE) {
        either with (msg ∈ MsgsRcvdWith([type ↦ "1a"])) {
                 if (msg.bal ≥ maxBallot) {
                   maxBallot := msg.bal ;
                   Send( [type ↦ "1b", acc ↦ self,
                          bal ↦ msg.bal, val ↦ vote[msg.prevBal],
                          voteBal ↦ msg.prevBal] ) } }
        or     with (msg ∈ MsgsRcvdWith( [type ↦ "2a"] )) {
                 if (msg.bal ≥ maxBallot) {
                   maxBallot := msg.bal ;
                   vote[msg.bal] := msg.val ;
                   Send( [type ↦ "2b", acc ↦ self,
                          bal ↦ maxBallot, val ↦ msg.val] ) } } } }

    process (Client ∈ Clients) {
c1:   with (v ∈ PValues) {
        Send( [type ↦ "clientReq", val ↦ v] ) } } }
```

Figure 3: The master process and the acceptor and client processes of Vertical Paxos I.

```
process (BallotLeader ∈ Nat \ {0})
  variables prevBal, safeVal ; {
b1: with (msg ∈ MsgsRcvdWith( [type ↦ "newBallot", bal ↦ self] )) {
      prevBal := msg.prevBal ;
      Send( [type ↦ "1a", bal ↦ self, prevBal ↦ prevBal] ) ; }
b2: either with (msg ∈ {m ∈ MsgsRcvdWith([type ↦ "1b",
                                          voteBal ↦ prevBal]) :
                        m.val ≠ None}){
            safeVal := msg.val }
    or      await ∃ Q ∈ RQuorums(prevBal) : ∀ a ∈ Q :
                  MsgsRcvdWith([type ↦ "1b",
                                bal ↦ self,
                                voteBal ↦ prevBal,
                                val ↦ None,
                                acc ↦ a]) ≠ {};
b3: if (safeVal ≠ AllSafe) {
        Send( [type ↦ "2a", bal ↦ self, val ↦ safeVal] ) ;
b4:     await ∃ Q ∈ WQuorums(self) :
                  ∀ a ∈ Q :
                    MsgsRcvdWith([type ↦ "2b", acc ↦ a, bal ↦ self])
                        ≠ {} ; };
b5: Send( [type ↦ "complete", bal ↦ self, prevBal ↦ prevBal] ) ;
b6: await HasRcvd( [type ↦ "activated", bal ↦ self] ) ;
b7: if (safeVal = AllSafe) {
        with (msg ∈ MsgsRcvdWith( [type ↦ "clientReq"] )) {
          Send( [type ↦ "2a", bal ↦ self, val ↦ msg.val] ) ; } } }
```

Figure 4: The leader processes of Vertical Paxos II.

```
    process (MasterProc = Master)
      variable curBallot = 0, nextBallot = 1 ; {
m1:   while (TRUE) {
        either {Send( [type ↦ "newBallot", bal ↦ msg.bal, ;
                       prevBal ↦ curBallot] ) ;
                nextBallot := nextBallot + 1; }
        or      with (msg ∈ MsgsRcvdWith( [type ↦ "complete"] )) {
                  if (msg.prevBal = curBallot) {
                    Send( [type ↦ "activated", bal ↦ msg.bal] );
                    curBallot := msg.bal ; } } } }
```

Figure 5: The master process of Vertical Paxos II.

5 Vertical Paxos and Other Primary-Backup Protocols

Primary-backup replication protocols are common in practical distributed systems. Niobe [9], Chain Replication [10], and the Google File System [3] are three examples of such protocols that have been deployed in systems with hundreds or thousands of machines. While these protocols are seemingly unrelated, the first two can be viewed as Vertical Paxos algorithms. The Google File System does not provide consistency, so it is not an instance of Vertical Paxos; but it could be made consistent by using Vertical Paxos.

Niobe follows the Vertical Paxos II protocol closely. Each replica forms a read quorum, while the entire replica set constitutes a write quorum. Reconfiguration uses a global-state manager, with the configuration numbers corresponding to the ballot numbers in Vertical Paxos II. Niobe incorporates certain simple optimizations of Vertical Paxos. For example, when the primary removes a faulty backup from the configuration, no change of configuration number or state transfer is needed. This is because in Vertical Paxos with a single write b-quorum, the master can at any time remove acceptors from that b-quorum. Niobe also allows the primary to handle query operations locally, as described in Section 3.

Chain Replication imposes an additional chain structure on a replica group. An update arrives at the head of the chain and propagates down the chain to the tail. An update message carries both the phase 2a message and phase 2b messages along the chain. When it arrives at the tail, it contains the phase 2b messages telling the tail that its vote makes the update chosen. This allows Chain Replication to use the tail, instead of just the head, to process queries locally. The chain structure also makes it easy to perform reconfigurations that add a replica to the tail or remove the head or tail. Like Niobe, Chain Replication uses an external master for reconfiguration and follows the Vertical Paxos II protocol, maintaining one active configuration.

The Google File System (GFS) implements the abstraction of reliable distributed files and is optimized for append operations. A file consists of a sequence of *chunks*, each replicated on a (possibly different) set of servers. A master tracks the composition of each file and the locations of its chunks. GFS superficially resembles Vertical Paxos I in having multiple configurations active concurrently, each operating on a different chunk. However, GFS does not guarantee state-machine

consistency—even for operations that involve only a single chunk. GFS can be made consistent, mainly by changing how operations to an individual chunk are implemented. This can be done by making each chunk a separate state machine implemented with Vertical Paxos. While we do not know the precise algorithm used by GFS, we expect that this change would not seriously degrade its performance.

There are quite a few different primary-backup protocols. We believe that each one that guarantees state-machine consistency can be described, and its correctness demonstrated, by viewing it as an instance of Vertical Paxos. Vertical Paxos may also lead to interesting new algorithms, including ones that do not use a single write quorum for each ballot and are thus not traditional primary-backup protocols.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] Bernadette Charron-Bost and André Schiper. Uniform consensus is harder than consensus (extended abstract). Technical Report DSC/2000/028, École Polytechnique Fédérale de Lausanne, Switzerland, May 2000.

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In Michael L. Scott and Larry L. Peterson, editors, *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating Systems Principles*, pages 29–43, New York, NY, USA, 2003. ACM.

[4] Leslie Lamport. The PlusCal algorithm language. URL `http://research.microsoft.com/users/lamport/tla/pluscal.html`. The page can also be found by searching the Web for the 25-letter string obtained by removing the "-" from `uid-lamportpluscalhomepage`.

[5] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[6] Leslie B. Lamport and Michael T. Massa. Cheap Paxos. United States Patent 7249280, filed June 18, 2004, issued July 24, 2007.

[7] Butler W. Lampson. The ABCDs of Paxos. `http://research.microsoft.com/lampson/65-ABCDPaxos/Abstract.html`.

[8] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *OSDI'04: Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation*, pages 105–120, Berkeley, CA, USA, 2004. USENIX Association.

[9] John MacCormick, Chandramohan A. Thekkath, Marcus Jager, Kristof Roomp, Lidong Zhou, and Ryan Peterson. Niobe: A practical replication protocol. *ACM Transactions on Storage (TOS)*, 3(4), 2008.

[10] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 91–104, Berkeley, CA, USA, 2004. USENIX Association.