Paxos for System Builders

Jonathan Kirsch and Yair Amir

Technical Report CNDS-2008-2 - March 2008
http://www.dsn.jhu.edu

Abstract

This paper presents a complete specification of the Paxos replication protocol such that system builders can understand it and implement it. We evaluate the performance of a prototype implementation and detail the safety and liveness properties guaranteed by our specification of Paxos.

1 Introduction

State machine replication [15, 22] is a well-known technique for implementing distributed services (e.g., information access systems and database systems) requiring high performance and high availability. Typically, the service is implemented by a group of server replicas, which run a protocol to globally order all actions that cause state transitions. The servers begin in the same initial state, and they execute the actions in the agreed upon order, thus remaining replicas of one another.

In this work, we consider the Paxos [16, 17] state machine replication protocol. Paxos is a robust protocol in which global ordering is coordinated by an elected leader. Normal-case operation of Paxos requires two rounds of server-to-server communication for ordering. Although the original Paxos algorithm was known in the 1980's and published in 1998, it is difficult to understand how the algorithm works from the original specification. Further, the original specification had a theoretical flavor and omitted many important practical details, including how failures are detected and what type of leader election algorithm is used. Filling in these details is critical if one wishes to build a real system that uses Paxos as a replication engine.

The goal of this work is to clearly and completely specify the Paxos algorithm such that system builders can understand it and implement it. We provide complete pseudocode, in C-like notation, for our interpretation of the protocol. We specify the leader election algorithm used, in addition to mechanisms for reconciliation and flow control. We clearly specify the safety and liveness guarantees provided by our specification, and we provide intuition as to the environments for which Paxos is particularly well-suited compared to other approaches. In particular, we show the liveness implications of using different leader election protocols.

We implemented our specification and provide performance results for our implementation. Our results show that Paxos is capable of achieving high throughput and low latency without disk writes or when writes are asynchronous, but that it suffers from significant performance degradation compared to other approaches if disk writes are synchronous. We show how aggregation can be used to mitigate this degradation, improving performance by more than an order of magnitude.

The remainder of this paper is presented as follows. In Section 2, we describe relevant related work. In Section 3, we present the system model assumed by Paxos, and we describe the service properties guaranteed by Paxos. Section 4 presents the Paxos consensus protocol, which can be seen as a version of the replication protocol in which only one value needs to be agreed upon. In Section 5, we present the replication protocol, with detailed pseudocode. Section 6 discusses the liveness guarantee of Paxos, both from a theoretical and a practical standpoint. In Section 7, we evaluate the performance of our implementation. Section 8 concludes.

2 Related Work

State machine replication [15, 22] allows a set of servers to remain replicas of one another by ordering the actions that cause state transitions. It is assumed that the actions are deterministic. Servers begin in the same initial state and execute updates according to the agreed upon order.

State machine replication can be used to replicate information systems such as databases. A replicated database should appear to the user as if there is only one copy of the database. Further, concurrent transactions that span multiple sites should be serializable, meaning their execution should be equivalent to some serial execution. The Two-Phase Commit (2PC) protocol [8] is a well-known tool for achieving this property of one-copy serializability. While simple, 2PC is forced to block forward progress if the coordinator of a transaction fails; the other participants need to hold their locks on the database until they can resolve the outcome of the transaction, limiting availability and potential concurrency in the case of failure. The Three-Phase Commit (3PC) protocol [23] overcomes some of the blocking problems associated with 2PC, at the cost of an additional round of communication. 3PC allows a majority of servers to make forward progress in the face of one failure. However, if failures cascade, a majority of servers may still be forced to block in order to preserve consistency. The Enhanced Three-Phase Commit (e3PC) protocol [14] allows *any* majority of participants to make forward progress, regardless of past failures, at the negligible cost of two additional counters.

This paper focuses on Paxos [16, 17]. Paxos is a state machine replication protocol that allows a set of distributed servers, exchanging messages via asynchronous communication, to totally order client requests in the benign-fault, crash-recovery model. Paxos assumes a static membership, and it uses an elected leader to coordinate the agreement protocol. The normal-case operation of Paxos resembles the 2PC protocol, while its robustness is simlar to that of e3PC. For each update, the leader constructs an explicit write quorum consisting of a majority of servers. If the leader crashes or becomes unreachable, the other servers elect a new leader; a view change occurs, allowing progress to safely resume in the new view under the reign of the new leader. Paxos requires at least $2f+1$ servers to tolerate f faulty servers.

Several papers have been published which have attempted to specify and clarify the Paxos algorithm since its original presentation by Lamport. De Prisco, Lampson, and Lynch [21] specify and prove the correctness of the protocol using Clocked General Timed automata to model the processes and channels in the system. Their specification makes use of a failure detector for leader election, and they provide a theoretical analysis of the performance of Paxos. Dutta, Guerraoui, and Boichat [4] use two main abstractions to modularize the functionality of the Paxos protocol: a weak leader election abstraction (implemented by a failure detector) and a round-based register abstraction. They also show four variants of Paxos that make different assumptions and are thus useful in different settings. Lampson [18] presents an abstract version of Paxos and then shows how the different versions of Paxos (i.e., Byzantine Paxos [5] and Disk Paxos [10]) can be derived from the abstract version. Li, et al. [19] specify Paxos using a register abstraction; the protocol consists of how reads and writes to the Paxos register are implemented. A concurrent work by Chandra, et al. [6] describes the authors' experience in using Paxos to build a fault-tolerant database. They highlight the systems-related problems they encountered, many of which were not addressed by previous specifications of Paxos.

The current work takes the same practical approach as [6] and seeks to remedy the shortcomings of previous specifications by including complete pseudocode for those components needed to implement Paxos in practice. Rather than approaching Paxos from a theoretical point of view, or using abstractions to modularize the protocol, we aim to give the reader the tools needed to build a useful *system* that uses Paxos as a replication engine. This requires a precise specification of message structure, protocol timeouts, how messages are recovered, etc. To the best of our knowledge, our work is the first to provide such a systems-based specification. At the same time, we highlight important theoretical differences, related to liveness, between our specification and previous work in Section 6. Finally, we provide the most extensive experimental evaluation of Paxos to date and describe the practical implications of our results.

Other existing state machine replication protocols (e.g., COReL [12] and Congruity [1]) operate above a group communication system, which provides services for membership and reliable, ordered delivery. COReL has a similar message pattern to Paxos (using end-to-end acknowledgements and disk writes per update) but uses the local total order provided by the underlying group communication system to assign a tentative ordering (whereas Paxos uses an elected leader). Both Paxos and COReL allow any majority of servers to make forward progress, regardless of past failures. However, they differ in the degree of network stability required as a prerequisite for ordering. We discuss this in more detail in Section 6.

Congruity leverages the safe delivery service of the Extended Virtual Synchrony (EVS) semantics [20] to remove the need for end-to-end acknowledgements and per-server disk writes on a per-update basis. Congruity trades higher normal-case performance for lower worst-case availability. If all servers crash before any of them installs the next primary component, then forward progress is blocked until one server (directly or indirectly) completes reconciliation with all other servers. As originally specified, this blocking behavior can also result if all servers partition such that they deliver all installation attempt messages, but they all deliver the last attempt message during a membership transition. This partition vulnerability can be removed, and we show how to do so in a related paper [2].

3 System Model

We assume a system of N replication servers, which communicate by passing messages in a communication network. Communication is asynchronous and unreliable; messages may be duplicated, lost, and can take an arbitrarily long time to arrive, but those that do arrive are not corrupted. The network can partition into multiple disjoint components, and components can subsequently remerge.

We assume a benign fault model for servers. Servers can crash and subsequently recover, and they have access to stable storage, which is retained across crashes. We assume that all servers follow the protocol specification; they do not exhibit Byzantine behavior.

Servers are implemented as deterministic state machines. All servers begin in the same initial state. A server transitions from one state to the next by applying an *update* to its state machine; we say that the server *executes* the update. The next state is completely determined by the current state and the update being executed. Paxos establishes a global, persistent, total order on client updates. By executing the same updates in the same order, the servers remain consistent replicas of each other.

Clients introduce updates for execution by sending them to the Paxos servers. Each update is distinguished by the identifier of the initiating client and a client-based, monotonically increasing sequence number. We assume that each client has at most one outstanding update at a time; a client c only submits an update with client sequence number $i_c + 1$ when it has received the reply for update i_c.

The correctness of the Paxos protocol is encapsulated in the following consistency requirement:

DEFINITION 3.1 S1 - SAFETY: *If two servers execute the i^{th} update, then these updates are identical.*

To ensure that meeting Safety is non-trivial, we also use the following requirement:

DEFINITION 3.2 S2 - VALIDITY: *Only an update that was introduced by a client (and subsequently initiated by a server) may be executed.*

We also require that each update be executed at most once:

DEFINITION 3.3 S3 - AT-MOST ONCE: *If a server executes an update on sequence number i, then the server does not execute the update on any other sequence number $i' > i$.*

Since no asynchronous, fault-tolerant replication protocol tolerating even one failure can always be both safe and live [9], Paxos provides a liveness, or progress, guarantee, only under certain synchrony and connectivity conditions. We defer a discussion of these conditions until Section 6. We note, however, that Paxos meets requirements S1, S2, and S3 even when the system is asynchronous; that is, it does not rely on synchrony assumptions for safety.

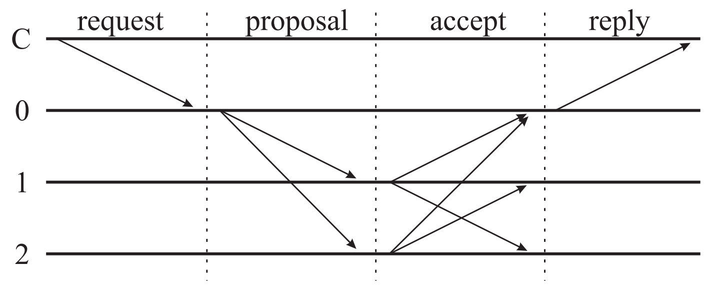

Figure 1: Paxos normal-case operation. Client C sends an update to the leader (Server 0). The leader sends a PROPOSAL containing the update to the other servers, which respond with an ACCEPT message. The client receives a reply after the update has been executed.

4 The Consensus Protocol

4.1 The Concept

From a high-level perspective, the consensus protocol proceeds as follows. One server is elected as the *leader*; the leader coordinates the protocol by *proposing* a client update for execution. The other servers *accept* the leader's proposal and tell each other that they have done so. A server learns that an update has been agreed upon when it learns that a majority of servers have accepted the corresponding proposal; the server can then execute the update. The server that originally received the client's update sends a reply to the client after executing the update. The protocol is depicted in Figure 1.

This simple, two-round protocol (which we call the *Proposal phase*) is the heart of the consensus algorithm. Some complexity needs to be introduced, however, to ensure agreement in the face of crashes and recoveries. The leader may crash, resulting in the election of a new leader, or more than one server may believe they are the leader at a given time. The algorithm must guarantee that, even if multiple leaders propose different updates, at most one update is agreed upon for execution.

We first define a total ordering on the reigns of different leaders by assigning each reign a unique number, which we call a *view number*. The system proceeds through a series of views, with a *view change* occurring each time a new leader is elected. Each view has exactly one leader, which makes at most one proposal in the context of this view (although the proposal may be retransmitted if necessary).[1] Since multiple proposals (from different views) may be made, we define a total ordering on all proposals by attaching the corresponding view number to each one. We say that a proposal with a lower (resp. higher) view number occurred *earlier* (resp. *later*) than one with a higher (resp. lower) view number.

With these mechanisms in place, Paxos ensures agreement by introducing another round of communication – the *Prepare phase* – before the Proposal phase. In the Prepare phase, the leader asks the servers to respond with the latest proposal (if any) they have accepted.

[1]In the context of the replication algorithm, we allow multiple proposals to be made within a given view, and we distinguish proposals by pairs consisting of view numbers and sequence numbers, as explained below.

Upon receiving responses from a majority of servers, the leader has either learned that (1) no update has been ordered (if no accepted proposals were reported among the majority) or (2) an update *may* have been ordered (if at least one server reported its corresponding proposal). In the former case, the leader can propose any update for execution, while in the latter, the leader is *constrained* to propose the update from the latest accepted proposal. Besides constraining the updates that can be proposed, the Prepare phase also restricts the proposals that can be accepted. By responding to a Prepare message, a server promises not to accept any proposal from a previous view (i.e., with a smaller view number).

4.2 The Protocol

We now present the consensus protocol in more detail. We assume that a leader has just been elected; we defer discussion of leader election, reconciliation, and other mechanisms until Section 5, where we detail the complete replication protocol.

Upon being elected, the new leader initiates the Prepare phase. The leader *prepares* the proposal it will make by sending a PREPARE message, containing the current view number, to the other servers. The leader waits for responses from $\lfloor N/2 \rfloor$ servers. If it does not receive the necessary responses, and it still believes it is the leader, it tries to prepare a proposal with a higher view number.

A server can respond to a PREPARE message if it has not already responded to one with a higher view number. The server responds by sending a PREPARE_OK message to the leader, containing the view number and the most recent proposal it has accepted, if any. If the leader collects $\lfloor N/2 \rfloor$ PREPARE_OK messages for the current view, the update it can propose is either *constrained* (if one or more PREPARE_OK messages reported an earlier accepted proposal) or *unconstrained* (if no accepted proposal was reported). Note that the leader is unconstrained in this case for the following reason: If any update was ordered, it must have been accepted by a majority of servers, one of which sent a PREPARE_OK (since any two majorities intersect).

The leader sends out a PROPOSAL message to the other servers, containing the current view number and the update being proposed. Again, the update is either the most recent constraining update, or any update if none was reported. A server receiving a PROPOSAL message can accept it, as long as it is has not already shifted to a later view. A server accepts a PROPOSAL by sending an ACCEPT message, containing the current view number. Finally, a server orders an update when it receives the PROPOSAL message and $\lfloor N/2 \rfloor$ corresponding ACCEPT messages.

4.3 Disk Writes

To preserve safety in the face of crashes, servers need to write to stable storage at key places during the protocol. We briefly consider each of these in turn and describe why they are necessary.

Sending a PREPARE_OK. Before responding to a PREPARE message with a PREPARE_OK, a server must write the view number to stable storage. Since the response is a promise not to accept earlier proposals, the server must remember this promise across crashes.

Accepting a PROPOSAL. Before sending an ACCEPT messsage, a server must write the corresponding PROPOSAL message to disk. Since other servers use the ACCEPT in determining whether to order an update, this server must remember the proposal so that it can correctly respond to future PREPARE messages, which in turn will correctly constrain future proposals. If the leader's PROPOSAL is used as an implicit ACCEPT message, it must write to disk before sending the PROPOSAL.

Making the first PROPOSAL in a new view. In the context of the replication protocol, the PREPARE message will contain, in addition to the current view number, a sequence number. For simplicity of the pseudocode, we enforce that the leader never send two different PREPARE messages for the same view. Thus, in our code the leader writes to disk before sending the PREPARE message. We note, however, that to preserve safety, the leader need not sync at this time, provided it syncs its view number to disk at some point before sending a PROPOSAL in the new view.

Initiating a CLIENT_UPDATE. To ensure that updates are consistently named across crashes, either the client or the server must sync the update to disk before initiating it into the system. In our specification, the server assumes this responsibility.

4.4 Protocol Variants

We discuss two variants of the core protocol that reflect different tradeoffs regarding message complexity, latency, and availability.

In the version of the protocol described above, the leader's PROPOSAL messsage is used as an implicit ACCEPT message for the purpose of ordering. This requires that the leader sync to disk before sending the PROPOSAL, resulting in two sequential disk writes during the ordering (one for the PROPOSAL and one for the ACCEPT). One variation is that the leader can send an ACCEPT message as well, in which case it could write to disk in parallel with the other servers. If disk writes are expensive with respect to the overall latency of the ordering, then this results in a reduction of the ordering latency at the cost of one additional incoming message per non-leader (and $N-1$ additional sends by the leader).

Another interesting variation on the core protocol is to have the non-leader servers send their ACCEPT messages only to the leader, which will collect $\lfloor N/2 \rfloor$ responses and then broadcast the ordering decision to all servers. This adds an extra round of latency but reduces the number of messages from $O(N^2)$ to $O(N)$. This version is also less fault-tolerant, since if the leader fails, the update will not be ordered until a new leader is elected. To remedy this, the non-leaders can send to a group of servers, trading off fault tolerance for higher message complexity.

5 The Replication Protocol

5.1 Overview

The Paxos replication protocol extends the consensus protocol to assign a global, persistent, total order to a sequence of client updates. Intuitively, the replication protocol runs multiple instances of the consensus protocol in parallel; the update ordered in the i^{th} instance is assigned the i^{th} sequence number in the global order. A server executes an update after it has executed all previous updates in the global order.

The communication pattern of the replication protocol is almost identical to that of the consensus protocol. The protocol elects a leader, which assigns sequence numbers to client updates. The leader's assignment is encapsulated in a PROPOSAL message, and the non-leaders respond by sending ACCEPT messages. As before, a server orders an update when it collects the PROPOSAL and $\lfloor N/2 \rfloor$ ACCEPT messages. Since the leader might crash, we specify a timeout-based failure detection mechanism, which allows a new leader to be elected when insufficient progress is being made.

The leader remains in power while progress is being made. Therefore, it will usually make multiple proposals in the same view. Thus, we can no longer differentiate proposals by their view number alone, as we could in the consensus protocol; we now distinguish PROPOSAL messages by the combination of their view number and a sequence number. Note that the earlier/later temporal relationship is still solely defined by view number.

We present the replication protocol in the form of a deterministic state machine with three states:

LEADER_ELECTION. A server is attempting to install a new leader. It participates in the leader election algorithm.

REG_LEADER. The leader completes the Prepare phase, after which it can assign global sequence numbers to incoming updates and propose them to the rest of the servers.

REG_NONLEADER. A non-leader forwards client updates to the leader for sequencing and responds to proposals made by the leader.

There are two types of events that produce an action in the state machine. The first is a *message reception* event, which occurs when the transport layer delivers a message to the server application. The server handles the event according to the protocol specification. The second type of event is a *timer* event, which occurs under certain conditions after some time elapses without the server taking some action.

In the remainder of this section, we first present the data structures and message types used in the rest of the protocol. We then present the leader election protocol, followed by the Prepare phase of the algorithm. We then describe the global ordering protocol used during normal-case operation, followed by our mechanisms for reconciliation, client handling, recovery, message retransmission, and flow control.

5.2 Data Structures and Message Types

Each server maintains data structures to track its state during each phase of the protocol. As seen in Figure 2, this state includes variables relating to leader election (Block B), the Prepare phase (Block C), and the global ordering protocol (Block D). The server's *Global_History* is an array, indexed by sequence number, used to maintain PROPOSAL and ACCEPT messages for each instance of the consensus protocol (i.e., for each sequence number). Each server

```
Data Structures:
/* Server State variables */
A1. int My_server_id - a unique identifier for this server
A2. State - one of {LEADER_ELECTION, REG_LEADER, REG_NONLEADER}

/* View State variables */
B1. int Last_Attempted - the last view this server attempted to install
B2. int Last_Installed - the last view this server installed
B3. VC[] - array of View_Change messages, indexed by server_id

/* Prepare Phase variables */
C1. Prepare - the Prepare message from last preinstalled view, if received
C2. Prepare_oks[] - array of Prepare_OK messages received, indexed by server_id

/* Global Ordering variables */
D1. int Local_Aru - the local aru value of this server
D2. int Last_Proposed - last sequence number proposed by the leader
D3. Global_History[] - array of global_slots, indexed by sequence number, each containing:
D4.   Proposal - latest Proposal accepted for this sequence number, if any
D5.   Accepts[] - array of corresponding Accept messages, indexed by server_id
D6.   Globally_Ordered_Update - ordered update for this sequence number, if any

/* Timers variables */
E1. Progress_Timer - timeout on making global progress
E2. Update_Timer - timeout on globally ordering a specific update

/* Client Handling variables */
F1. Update_Queue - queue of Client_Update messages
F2. Last_Executed[] - array of timestamps, indexed by client_id
F3. Last_Enqueued[] - array of timestamps, indexed by client_id
F4. Pending_Updates[] - array of Client_Update messages, indexed by client_id
```

Figure 2: Data Structures

also maintains timers used for failure detection (Block E), as well as variables to ensure each client update is executed at most once (Block F).

The message types used by the replication protocol are presented in Figure 3. The GLOBALLY_ORDERED_UPDATE type (Block H) is used during reconciliation, which is described in Section 5.6. Upon receiving a message, each server first runs a conflict check (Figure 4) to determine if the message should be applied to the server's data structures. The conflict tests are simple; for example, a server only handles a PROPOSAL message if it is in the REG_NONLEADER state. If a message does not cause a conflict, the server applies it to its data structures according to a set of update rules (Figure 5).

```
Message Types:
A1. Client_Update - contains the following fields:
A2.   client_id - unique identifier of the sending client
A3.   server_id - unique identifier of this client's server
A4.   timestamp - client sequence number for this update
A5.   update - the update being initiated by the client

B1. View_Change - contains the following fields:
B2.   server_id - unique identifier of the sending server
B3.   attempted - view number this server is trying to install

C1. VC_Proof - contains the following fields:
C2.   server_id - unique identifier of the sending server
C3.   installed - last view number this server installed

D1. Prepare - contains the following fields:
D2.   server_id - unique identifier of the sending server
D3.   view - the view number being prepared
D4.   local_aru - the local aru value of the leader

E1. Prepare_OK - contains the following fields:
E2.   server_id - unique identifier of the sending server
E3.   view - the view number for which this message applies
E4.   data_list - list of Proposals and Globally_Ordered_Updates

F1. Proposal - contains the following fields:
F2.   server_id - unique identifier of the sending server
F3.   view - the view in which this proposal is being made
F4.   seq - the sequence number of this proposal
F5.   update - the client update being bound to seq in this proposal

G1. Accept - contains the following fields:
G2.   server_id - unique identifier of the sending server
G3.   view - the view for which this message applies
G4.   seq - the sequence number of the associated Proposal

H1. Globally_Ordered_Update - contains the following fields:
H2.   server_id - unique identifier of the sending server
H3.   seq - the sequence number of the update that was ordered
H4.   update - the client update bound to seq and globally ordered
```

Figure 3: Message Types

```
boolean Conflict(message):
  case message:
A1.   View_Change VC(server_id, attempted):
A2.     if server_id = My_server_id
A3.       return TRUE
A4.     if State $\neq$ LEADER_ELECTION
A5.       return TRUE
A6.     if Progress_Timer is set
A7.       return TRUE
A8.     if attempted $\leq$ Last_Installed
A9.       return TRUE
A10.  return FALSE

B1.   VC_Proof V(server_id, installed):
B2.     if server_id = My_server_id
B3.       return TRUE
B4.     if State $\neq$ LEADER_ELECTION
B5.       return TRUE
B6.   return FALSE

C1.   Prepare(server_id, view, leader_aru):
C2.     if server_id = My_server_id
C3.       return TRUE
C4.     if view $\neq$ Last_Attempted
C5.       return TRUE
C6.   return FALSE

D1.   Prepare_OK(server_id, view, data_list):
D2.     if State $\neq$ LEADER_ELECTION
D3.       return TRUE
D4.     if view $\neq$ Last_Attempted
D5.       return TRUE
D6.   return FALSE

E1.   Proposal(server_id, view, seq, update):
E2.     if server_id = My_server_id
E3.       return TRUE
E4.     if State $\neq$ REG_NONLEADER
E5.       return TRUE
E6.     if view $\neq$ Last_Installed
E7.       return TRUE
E8.   return FALSE

F1.   Accept(server_id, view, seq):
F2.     if server_id = My_server_id
F3.       return TRUE
F4.     if view $\neq$ Last_Installed
F5.       return TRUE
F6.     if Global_History[seq] does not contain a Proposal from view
F7.       return TRUE
F8.   return FALSE
```

Figure 4: Conflict checks to run on incoming messages. Messages for which a conflict exists are discarded.

```
Update_Data_Structures(message):
   case message:
A1.   View_Change V(server_id, view):
A2.      if VC[server_id] is not empty
A3.         ignore V
A4.      VC[server_id] ← V

B1.   Prepare P(server_id, view, leader_aru):
B2.      Prepare ← P

C1.   Prepare_OK P(server_id, view, data_list):
C2.      if Prepare_OK[server_id] is not empty
C3.         ignore P
C4.      Prepare_OK[server_id] ← P
C5.      for each entry e in data_list
C6.         Apply e to data structures

D1.   Proposal P(server_id, view, seq, update):
D2.      if Global_History[seq].Globally_Ordered_Update is not empty
D3.         ignore Proposal
D4.      if Global_History[seq].Proposal contains a Proposal P'
D5.         if P.view > P'.view
D6.            Global_History[seq].Proposal ← P
D7.            Clear out Global_History[seq].Accepts[]
D8.      else
D9.         Global_History[seq].Proposal ← P

E1.   Accept A(server_id, view, seq):
E2.      if Global_History[seq].Globally_Ordered_Update is not empty
E3.         ignore A
E4.      if Global_History[seq].Accepts already contains ⌊N/2⌋ Accept messages
E5.         ignore A
E6.      if Global_History[seq].Accepts[server_id] is not empty
E7.         ignore A
E8.      Global_History[seq].Accepts[server_id] ← A

F1.   Globally_Ordered_Update G(server_id, seq, update):
F2.      if Global_History[seq] does not contain a Globally_Ordered_Update
F3.         Global_History[seq] ← G
```

Figure 5: Rules for updating the Global_History.

```
Leader_Election():
A1. Upon expiration of Progress_Timer:
A2.   Shift_to_Leader_Election(Last_Attempted+1)

B1. Upon receiving View_Change(server_id, attempted) message, V:
B2.   if attempted > Last_Attempted and Progress_Timer not set
B3.      Shift_to_Leader_Election(attempted)
B4.      Apply V to data structures
B5.   if attempted = Last_Attempted
B6.      Apply V to data structures
B7.      if Preinstall_Ready(attempted)
B8.         Progress_Timer ← Progress_Timer*2
B9.         Set Progress_Timer
B10.        if leader of Last_Attempted
B11.           Shift_to_Prepare_Phase()

C1. Upon receiving VC_Proof(server_id, installed) message, V:
C2.   if installed > Last_Installed
C3.      Last_Attempted ← installed
C4.      if leader of Last_Attempted
C5.         Shift_to_Prepare_Phase()
C6.      else
C7.         Shift_to_Reg_Non_Leader()

D1. bool Preinstall_Ready(int view):
D2.   if VC[] contains $\lfloor N/2 \rfloor + 1$ entries, v, with v.attempt = view
D3.      return TRUE
D4.   else
D5.      return FALSE

E1. Shift_to_Leader_Election(int view):
E2.   Clear data structures: VC[], Prepare, Prepare_oks, Last_Enqueued[]
E3.   Last_Attempted ← view
E4.   vc ← Construct_VC(Last_Attempted)
E5.   SEND to all servers: vc
E6.   Apply vc to data structures
```

Figure 6: Leader Election

5.3 Leader Election

As described above, Paxos is coordinated by a leader server, which assigns sequence numbers to client updates and proposes the assignments for global ordering. The servers use a *leader election* protocol to elect this leader; we use a protocol similar to the one described in [5], adapted for use in benign environments.

From a high-level perspective, servers vote, or *attempt*, to install a new view if insufficient progress is being made – that is, when their Progress_Timer expires. Each view is associated with an integer; the leader of view i is the server such that My_server_id $\equiv i \bmod N$, where N is the total number of servers in the system. When a server receives a majority of votes for the view it is trying to install, we say that the server *preinstalls* the view. When a server completes the Prepare Phase for the preinstalled view (see Section 5.4), we say that the server *installs* the view. The server tracks its last attempted view and its last installed view with the Last_Attempted and Last_Installed variables, respectively (Figure 2, Block B).

The complete leader election protocol is listed in Figure 6. When a server's Progress_Timer expires, it invokes the Shift_to_Leader_Election procedure (Block E), clearing its view-related variables and sending a VIEW_CHANGE message to the other servers. The server waits for one of two cases to occur. First, if the server receives a VIEW_CHANGE message for a view higher than the one it attempted to install, then the server jumps to the higher view and sends out a corresponding VIEW_CHANGE message (lines B2-B4). On the other hand, if a server collects a majority of corresponding VIEW_CHANGE messages from distinct servers for the view it attempted to install (i.e., when it preinstalls the view), the server sets its Progress_Timer. If the server is the leader, it triggers the Prepare phase. In addition, each server periodically transmits a VC_PROOF message, which contains the value of its Last_Installed variable. Upon receiving a VC_PROOF message for a later view, a server in the LEADER_ELECTION state can immediately join the installed view.

The protocol relies on two techniques to ensure synchronization:

1. A server sets its Progress_Timer to twice its previous value. This is essentially a probing mechanism, allowing more time for the leader election protocol to complete. It also gives a newly established leader more time to complete the global ordering protocol.

2. A server only jumps to a higher view if two conditions both hold. First, the server must already suspect the current leader to have failed. This condition removes the power of an unstable server to disrupt progress once a stable majority has already established a view. Second, the server's Progress_Timer must not already be set (see Figure 4, line A6). This condition gives the leader election protocol the greatest opportunity to run to completion, without being interrupted by an unstable server. Recall that a server starts its Progress_Timer when it receives a majority of VIEW_CHANGE messages for the view it is currently attempting to install. Thus, while a server is waiting for progress to occur, it ignores anything from the unstable server.

5.4 Prepare Phase

The Prepare phase is run after the leader election algorithm completes; it is the critical mechanism used to ensure safety across view changes. The Prepare phase is run once for all future instances of the consensus protocol (i.e., for all sequence numbers) that will be run in the context of the new view. This key property makes normal-case operation very efficient, requiring only two rounds of communication. The newly elected leader collects ordering information from $\lfloor N/2 \rfloor$ servers, which it uses to constrain the updates it may propose in the new view. Pseudocode for the Prepare phase is listed in Figures 7 and 8.

Upon preinstalling the new view, the new leader shifts to the Prepare phase, sending to all servers a PREPARE message. The PREPARE message contains the new view number and the leader's *Local_Aru* value, which is the sequence number through which it has executed all updates. In the new view, the leader will make proposals for sequence numbers above its Local_Aru. Thus, to preserve safety, the leader must learn about any accepted proposals for these sequence numbers. By collecting information from $\lfloor N/2 \rfloor$ servers, the leader is guaranteed to learn about any update that may have been ordered.

Upon receiving a PREPARE message, a server that has preinstalled the leader's new view builds a PREPARE_OK message and sends it to the leader. Each PREPARE_OK message contains the new view number and a *data_list*. The data_list is a (possibly empty) list of PROPOSAL and/or GLOBALLY_ORDERED_UPDATE messages, and is constructed as seen in Figure 8, Block A: For each sequence number above the leader's Local_Aru, the server includes either (1) a GLOBALLY_ORDERED_UPDATE, if the server has globally ordered that sequence number or (2) the latest PROPOSAL message that the server has accepted for that sequence number, if any. After responding to the PREPARE message, the server shifts to the REG_NONLEADER state (Figure 9, Block B).

```
A1. Shift_To_Prepare_Phase()
A2.   Last_Installed ← Last_Attempted
A3.   prepare ← Construct_Prepare(Last_Installed, Local_Aru)
A4.   Apply prepare to data structures
A5.   data_list ← Construct_DataList(Local_Aru)
A6.   prepare_ok ← Construct_Prepare_OK(Last_Installed, data_list)
A7.   Prepare_OK[My_Server_id] ← prepare_ok
A8.   Clear Last_Enqueued[]
A9. **Sync to disk
A10.  SEND to all servers: prepare

B1. Upon receiving Prepare(server_id, view, aru)
B2.   if State = LEADER_ELECTION /* Install the view */
B3.      Apply Prepare to data structures
B4.      data_list ← Construct_DataList(aru)
B5.      prepare_ok ← Construct_Prepare_OK(view, data_list)
B6.      Prepare_OK[My_server_id] ← prepare_ok
B7.      Shift_to_Reg_Non_Leader()
B8.      SEND to leader: prepare_ok
B9.   else /* Already installed the view */
B10.     SEND to leader: Prepare_OK[My_server_id]

C1. Upon receiving Prepare_OK(server_id, view, data_list)
C2.   Apply to data structures
C3.   if View_Prepared_Ready(view)
C4.      Shift_to_Reg_Leader()
```

Figure 7: Prepare Phase

```
A1. datalist_t Construct_DataList(int aru)
A2.   datalist ← ∅
A3.   for each sequence number $i$, $i > aru$, where Global_History[$i$] is not empty
A4.      if Global_History[$i$].Ordered contains a Globally_Ordered_Update, G
A5.         datalist ← datalist ∪ G
A6.      else
A7.         datalist ← datalist ∪ Global_History[$i$].Proposal
A8.   return datalist

B1. bool View_Prepared_Ready(int view)
B2.   if Prepare_oks[] contains $\lfloor N/2 \rfloor + 1$ entries, p, with p.view = view
B3.      return TRUE
B4.   else
B5.      return FALSE
```

Figure 8: Prepare Phase Utility Functions

When the leader collects $\lfloor N/2 \rfloor$ PREPARE_OK messages (i.e., when it *prepares* the new view), it is properly constrained and can begin making proposals. It shifts to the REG_LEADER state (Figure 9, Block A).

5.5 Global Ordering

Once it has prepared the new view, the leader can begin to propose updates for global ordering (Figure 10). If the leader receives any updates before this, it stores them in its *Update_Queue*. Non-leader servers forward client updates to the leader. The leader creates and sends new PROPOSAL messages using the *Send_Proposal()* procedure (Figure 11, Block A). Recall that if the leader's PROPOSAL is used as an implicit ACCEPT for ordering purposes, the leader must sync to disk before sending the PROPOSAL. For a given unordered sequence number, the leader proposes either (1) the last known accepted proposal (if the sequence number is constrained) or (2) the first update on the Update_Queue, if any.

Upon receiving a PROPOSAL message from the current view, a server constructs an ACCEPT message, syncs the accepted PROPOSAL to disk, and sends the ACCEPT to all servers.

```
A1. Shift_to_Reg_Leader()
A2.   State ← REG_LEADER
A3.   Enqueue_Unbound_Pending_Updates()
A4.   Remove_Bound_Updates_From_Queue()
A5.   Last_Proposed ← Local_Aru
A6.   Send_Proposal()

B1. Shift_to_Reg_Non_Leader()
B2.   State ← REG_NONLEADER
B3.   Last_Installed ← Last_Attempted
B4.   Clear Update_Queue
B5.   **Sync to disk
```

Figure 9: Shift_to_Reg_Leader() and Shift_to_Reg_Non_Leader() Functions

```
Global Ordering Protocol:
A1. Upon receiving Client_Update(client_id, server_id, timestamp, update), U:
A2.   Client_Update_Handler(U)

B1. Upon receiving Proposal(server_id, view, seq, update):
B2.   Apply Proposal to data structures
B3.   accept ← Construct_Accept(My_server_id, view, seq)
B4.   **Sync to disk
B5.   SEND to all servers: accept

C1. Upon receiving Accept(server_id, view, seq):
C2.   Apply Accept to data structures
C3.   if Globally_Ordered_Ready(seq)
C4.      globally_ordered_update ← Construct_Globally_Ordered_Update(seq)
C5.      Apply globally_ordered_update to data structures
C6.      Advance_Aru()

D1. Upon executing a Client_Update(client_id, server_id, timestamp, update), U:
D2.   Advance_Aru()
D3.   if server_id = My_server_id
D4.      Reply to client
D5.      if U is in Pending_Updates[client_id]
D6.         Cancel Update_Timer(client_id)
D7.         Remove U from Pending_Updates[]
D8.   Last_Executed[client_id] ← timestamp
D9.   if State ≠ LEADER_ELECTION
D10.     Restart Progress_Timer
D11.  if State = REG_LEADER
D12.     Send_Proposal()
```

Figure 10: Global Ordering Protocol

ACCEPT messages are very small, containing only the view number and the sequence number; it is not necessary to include the update (or even a digest of the update), since a leader will make at most one (unique) proposal for a given view and sequence number.

When a server collects the PROPOSAL and $\lfloor N/2 \rfloor$ ACCEPT messages for the same view and sequence number, it orders the update (Figure 11, Block B), advancing its Local_Aru if possible (Figure 11, Block C). A server executes an update with a given sequence number when its Local_Aru reaches that sequence number. The server to which the client originally sent the update sends a reply to the client.

5.6 Reconciliation

The original Paxos algorithm fails to specify a reconciliation protocol for bringing servers up to date. As a result, the algorithm may allow a server to order updates quickly, without allowing the server to *execute* these updates, due to gaps in the global sequence. This is a serious problem, since there is little use in continuing to order without being able to transition

```
A1.  Send_Proposal()
A2.    seq ← Last_Proposed + 1
A3.    if Global_History[seq].Globally_Ordered_Update is not empty
A4.      Last_Proposed++
A5.      Send_Proposal()
A6.    if Global_History[seq].Proposal contains a Proposal P
A7.      u ← P.update
A8.    else if Update_Queue is empty
A9.      return
A10.   else
A11.     u ← Update_Queue.pop()
A12.   proposal ← Construct_Proposal(My_server_id, view, seq, u)
A13.   Apply proposal to data structures
A14.   Last_Proposed ← seq
A15.   **Sync to disk
A16.   SEND to all servers: proposal

B1.  bool Globally_Ordered_Ready(int seq)
B2.    if Global_History[seq] contains a Proposal and $\lfloor N/2 \rfloor$ Accepts from the same view
B3.      return TRUE
B4.    else
B5.      return FALSE

C1.  Advance_Aru()
C2.    $i$ ← Local_Aru +1
C3.    while (1)
C4.      if Global_History[$i$].Ordered is not empty
C5.        Local_Aru++
C6.        $i$++
C7.      else
C8.        return
```

Figure 11: Global Ordering Utility Procedures

to the next state. This section addresses this problem. For clarity of presentation, we do not provide pseudocode for reconciliation.

Our mechanism is built upon a peer-to-peer, sliding-window, nack-based, selective repeat protocol. Reconciliation is triggered in three ways. First, if a server receives a PROPOSAL message for a sequence number more than a threshold amount above its Local_Aru, it attempts to reconcile with a random server, trying a different server if the selected server is unavailable or indicates that it cannot complete reconciliation up to the desired sequence number. Second, if a server globally orders an update "out of order" (i.e., the ordering does not increase the server's Local_Aru), the server again tries to reconcile with a random peer. Finally, we use periodic anti-entropy sessions [11] to recover missed updates. Each server periodically sends out a message containing its Local_Aru. If a server receives a message containing a sequence number higher than its own Local_Aru, the server sets a timer and records the sequence number as a potential target for reconciliation. If the timer expires and the server's Local_Aru is still less than the target, the server initiates a reconciliation session. The timer is set to avoid triggering reconciliation unnecessarily, when "missing" messages are actually in the server's buffer.

In addition to the above reconciliation mechanism, we also modify he Prepare phase such that a non-leader server only sends a PREPARE_OK message if the leader's Local_Aru, contained in the PREPARE message, is at least as high as this server's Local_Aru. If the leader is trailing behind, the non-leader server tables the PREPARE message and performs pair-wise reconciliation with the leader to bring it up to date.

```
Client_Update_Handler(Client_Update U):
A1.  if(State = LEADER_ELECTION)
A2.     if(U.server_id != My_server_id)
A3.        return
A4.     if(Enqueue_Update(U))
A5.        Add_to_Pending_Updates(U)
A6.  if(State = REG_NONLEADER)
A7.     if(U.server_id = My_server_id)
A8.        Add_to_Pending_Updates(U)
A9.        SEND to leader: U
A10. if(State = REG_LEADER)
A11.    if(Enqueue_Update(U))
A12.       if U.server_id = My_server_id
A13.          Add_to_Pending_Updates(U)
A14.       Send_Proposal()

B1. Upon expiration of Update_Timer(client_id):
B2.  Restart Update_Timer(client_id)
B3.  if(State = REG_NONLEADER)
B4.     SEND to leader: Pending_Updates[client_id]
```

Figure 12: Client Update Handling

5.7 Client Handling

In a practical Paxos implementation, we must ensure that each client update is not executed more than once. One approach to achieving this goal would be to allow updates to be ordered without restriction, checking at execution time if an update has already been executed and, if so, ignoring it. We take a different approach: once an update is bound to a sequence number, we prevent the leader from binding the update to any other sequence number (unless it learns that the update could not have been ordered with the first sequence number).

In the following discussion, we refer to the clients that send updates to a given server as that server's *local clients*. Each server assumes responsibility for updates received from its local clients. Note that a leader will receive updates from other clients, as well, since non-leaders forward updates from their local clients to the leader. By assuming responsibility, a server will continue to take action on behalf of the client (as described below) until the update is globally ordered. Each server stores the updates for which it is currently responsible in its *Pending_Updates* data structure.

During normal-case operation, a non-leader server that receives a new update from a local client adds the update to Pending_Updates, writes it to disk, and then forwards the update to the leader (see Figure 12, lines A6-A9, and Figure 13, Block B). Since pending updates are stored on disk, a server will re-assume responsibility for a pending update upon recovery. In addition, the server sets a timer on globally ordering the update. If the timer expires, the update may have been lost on the link to the leader, and the server resends the update to the leader. The server cancels the timer when it executes the update (Figure 10, line D6).

When the leader receives an update, it attempts to place the update on its Update_Queue (Figure 13, Block A). Each server maintains two additional data structures to ensure the Update_Queue is managed correctly. The *Last_Executed* data structure stores the timestamp of the last executed update from each client; the leader does not enqueue an update that has already been executed (Figure 13, lines A2-A3). The *Last_Enqueued* data structure stores the timestamp of the last enqueued update from each client for the current view; the leader does not enqueue updates that have already been enqueued (Figure 13, lines A4-A5). If the

```
A1. bool Enqueue_Update(Client_Update U)
A2.   if(U.timestamp $\leq$ Last_Executed[U.client_id])
A3.     return false
A4.   if(U.timestamp $\leq$ Last_Enqueued[U.client_id])
A5.     return false
A6.   Add U to Update_Queue
A7.   Last_Enqueued[U.client_id] $\leftarrow$ U.timestamp
A8.   return true

B1. Add_to_Pending_Updates(Client_Update U)
B2.   Pending_Updates[U.client_id] $\leftarrow$ U
B3.   Set Update_Timer(U.client_id)
B4.   **Sync to disk

C1. Enqueue_Unbound_Pending_Updates()
C2.   For each Client_Update U in Pending_Updates[]
C3.     if U is not bound and U is not in Update_Queue
C4.       Enqueue_Update(U)

D1. Remove_Bound_Updates_From_Queue()
D2.   For each Client_Update U in Update_Queue
D3.     if U is bound or U.timestamp $\leq$ Last_Executed[U.client_id] or
          (U.timestamp $\leq$ Last_Enqueued[U.client_id] and U.server_id $\neq$ My_server_id)
D4.       Remove U from Update_Queue
D5.       if U.timestamp $>$ Last_Enqueued[U.client_id]
D6.         Last_Enqueued[U.client_id] $\leftarrow$ U.timestamp
```

Figure 13: Client Update Handling Utility Functions

leader enqueues the update, it adds the update to Pending_Updates if it originated from a local client.

It is easy to see that, within an established view, the leader will enqueue each update at most once, and all forwarded updates will eventually reach the leader (assuming the link between a non-leader and the leader does not repeatedly drop the update). Care must be taken, however, to ensure correct behavior across view changes. While in the LEADER_ELECTION state, a server enqueues updates from its local clients, and, if necessary, takes responsibility for them. We now consider the state transitions that might occur across view changes, and we show how updates are managed in each case.

1. A server that shifts from LEADER_ELECTION to REG_NONLEADER may have updates in its Update_Queue; these updates should be forwarded to the new leader. The server is only responsible for updates from its local clients. These updates are already in Pending_Updates, and thus when the Update_Timer on these updates expires, the server will now correctly forward them to the new leader (Figure 12, Block B). As a result, all updates can be removed from the Update_Queue (Figure 9, line B4).

2. A server that shifts from LEADER_ELECTION to REG_LEADER must ensure that pending updates from its local clients are placed in the Update_Queue, so that they may be proposed in the new view. However, some of these pending updates (in addition to updates from non-local clients left over from a previous view) may already be bound to a sequence number, in which case they will be replayed as constrained updates in the new view; these updates should not be enqueued. After completing the Prepare Phase, the server (i) enqueues any pending updates that are not bound and not already in the queue (Figure 13, Block C) and (ii) removes any bound updates from the Update_Queue (Figure 13, Block D).

3. A server that shifts from REG_NONLEADER to LEADER_ELECTION will have an empty

```
Recovery:
A1. if no data exist on stable storage /* Initialization */
A2.   Create stable storage files
A3.   Last_Attempted ← 0
A4.   Last_Installed ← 0
A5.   Local_Aru ← 0
A6.   Last_Proposed ← 0
A7.   Progress_Timer ← default timer value
A8.   **Sync to disk
A9. else
A10.  Read State from stable storage
A11.  Read Last_Attempted and Last_Installed from stable storage
A12.  Rebuild Global_History from message log
A13.  For each pending update U
A14.     Add_to_Pending_Updates(U)
A15.  Compute Local_Aru from Global_History[]
A16. Shift_to_Leader_Election(Last_Attempted + 1)
```

Figure 14: Recovery Procedure

Update_Queue but may have pending updates, which will be handled when the server shifts in one of the above two cases.

4. A server that shifts from REG_LEADER to LEADER_ELECTION may have both local updates and non-local updates in its Update_Queue. Local updates are handled via the Pending_Updates mechanism, while other updates will either be retained (if they are not bound and the server shifts to REG_LEADER) or discarded (if the server shifts to REG_NONLEADER).

To ensure that updates are properly queued, the Last_Enqueued data structure is reset upon shifting to the LEADER_ELECTION state. Updates from local clients are only received once, and thus only new, unexecuted updates from local clients are enqueued.

5.8 Recovery

We provide a simple recovery mechanism for allowing crashed servers to return to correct operating status. As seen in Figure 14, a recovering server that has no information stored on stable storage initializes its data structures and creates a file for maintaining its log and other protocol meta-state. If the recovering server finds these files, it rebuilds its data structures from the log. In both cases, the server shifts to the LEADER_ELECTION state, where it attempts to install a new view.

We note that there are some cases in which a crashed server can return to normal-case operation without installing a new view. For example, if the server was a non-leader server, it might recover to find that updates are still being ordered in the same view. In this case, the server could begin responding to PROPOSAL messages, but it would need to recover missing updates before it could begin executing.

5.9 Message Retransmission

We employ several message retransmission mechanisms to overcome message loss during the course of the algorithm.

1. Messages handled with a *receiver-based* mechanism are only retransmitted upon reception of some "trigger" message:

- PREPARE_OK - sent in response to a PREPARE message.
- ACCEPT - sent in response to a PROPOSAL message.

2. Messages handled with a *sender-based* mechanism are retransmitted by the sender after a timeout period:

    - PREPARE - retransmitted when the view is not prepared within a timeout period.
    - PROPOSAL - retransmitted if the associated update has not been globally ordered within a timeout period.
    - CLIENT_UPDATE - retransmitted by the originating server if it has not been globally ordered within a timeout period.

3. The leader election messages are sent periodically.

5.10 Flow Control

In a practical implementation, one must avoiding putting too many updates into the pipeline such that messages are lost. Our implementation imposes a sending window on the leader, limiting the number of unexecuted proposals that can be outstanding at a given time. The leader sets the size of the window dynamically, gradually increasing the window until it detects a loss (via a PROPOSAL retransmission), at which point it cuts the size of the window in half. In addition, we impose the requirement that a non-leader server only responds to a PROPOSAL message if it is within its receiving window – that is, if the sequence number of the PROPOSAL is within a threshold amount of the server's Local_Aru. This throttles the leader such that it can proceed with $\lfloor N/2 \rfloor$ other servers.

We note, however, that this is not a true flow control mechanism when there are more than ($\lfloor N/2 \rfloor$+1) connected servers. While the system can proceed with ($\lfloor N/2 \rfloor$+1) up-to-date servers, a real system should keep all connected servers as up-to-date as possible. Thus, in addition to the mechanism described above, we employ a loose membership service. Each server periodically sends an ALIVE message, which contains the server's Local_Aru. The leader only sends a new proposal when (1) there is space in the window and (2) all connected servers have reported a Local_Aru value within some threshold range of the beginning of the window.

As we discuss in greater detail in Section 6, Paxos is theoretically able to make forward progress as long as there is a stable set of connected servers, even if other servers come and go very rapidly. We wish to make clear that the flow control mechanism described above may not meet this property, since it attempts to slow down if more than ($\lfloor N/2 \rfloor$+1) servers are present. An unstable server may repeatedly "fool" the leader into waiting for it, preventing progress from being made. It is easy to modify our implementation to remove this mechanism and meet the theoretical guarantee, at the cost of potentially leaving some servers far behind during normal-case operation (they would essentially proceed at the speed of reconciliation but may always lag behind). This issue highlights the tension between allowing a majority of servers to make progress and providing a useful service in practice.

6 Liveness: Theory and Practice

Throughout this work, we have referred in passing to the *liveness* of Paxos. The liveness of a protocol reflects its ability to make forward progress. The liveness of a protocol impacts its availability, since while a protocol that is forced to block may still allow read queries to be handled, it disallows write updates. Blocking is thus one of the major costs associated with maintaining replica consistency.

Given sufficient network stability, Paxos allows any majority of servers to make forward progress, regardless of past failures. That is, the combined knowledge of any majority of servers is sufficient to allow the servers to act in a way that respects previous ordering decisions such that safety is preserved. The liveness of Paxos, then, is primarily dependent on its *network stability requirement*. The network stability requirement defines the properties of the communications links between some subset of the servers in the system. For example, a network stability requirement might describe which servers are connected, how many servers are connected, and whether or not messages can pass between them (and with what delay).

It is not entirely straightforward to specify the network stability requirement of the Paxos protocol, in part because it is difficult to define what exactly "the Paxos protocol" is. Many of the important details, some of which play a large role in determining the liveness of the overall protocol, were not originally specified. Even in our own treatment of Paxos, presented in this work, we noted that there were several variants of the protocol that could be used. Thus, in our specification of the liveness of Paxos, we take these variants into consideration and explain their impact.

6.1 Normal-case Operation

We assume a system with $2f + 1$ servers in which a leader has been elected. To order an update, the leader must be able to send a PROPOSAL to f other servers, which must currently be in the leader's view. These servers will then send an ACCEPT messsage back to the leader, at which point the leader will globally order the update. Thus, if our goal is for the leader to globally order a single update, it must remain connected with a majority of servers (including itself) long enough for them to receive the PROPOSAL and for the leader to receive the ACCEPT messages.

In practice, the reason to have $2f + 1$ servers is for availability and performance; all servers should globally order the updates if possible. To see how this goal impacts the network stability requirement, we must look at the two possible communication patterns that we specified in Section 4.4. First, upon receiving a PROPOSAL, each server can send its ACCEPT message to all other servers. If the ACCEPT messages do not contain the update itself, then a server must be connected to the leader to receive the PROPOSAL and then connected to f accepting servers to globally order the update. If the ACCEPT messages contain the update, a server need not be connected to the leader at all to order the update, provided it can receive ACCEPT messages from $f + 1$ servers. The second approach allows the leader to send notification of the ordered update to the other servers, either as a separate message or by piggybacking the information on its next PROPOSAL. In this case, we need only pairwise connections with the leader to keep a majority of servers up to date; the other servers do not need to be connected to each other at all.

In our implementation, we chose the approach in which the ACCEPT messages are very small (i.e., they do not contain the update itself) and are sent all-to-all. In this setting, the leader must be able to communicate with a majority of servers, where the servers other than the leader can come and go very rapidly.

Observe, however, that we must take care to distinguish the theoretical network stability requirement for *ordering* (as described above) from the practical network stability requirements for *execution*. In practice, we are primarily interested in the ability of the servers to execute new updates (i.e., to globally order new updates with no holes). There seems to be little value in having servers order updates very quickly if execution must be delayed due to gaps in the global sequencing. In our implementation, servers maintain a window, based on their current Local_aru value; a server ignores (or tables) a PROPOSAL message with a sequence number beyond the end of the window. In order to continue making progress at full speed, a majority of servers must be able to increase their Local_aru values, by executing and globally ordering without holes. This means that they must be continuously connected. While in theory the members of the majority can switch very rapidly (with the exception of the leader), this only allows continual progress if reconciliation occurs sufficiently quickly so that it is as if they were connected to the majority for each proposal.

Thus, in practice, we essentially need a *stable set* of at least ($\lfloor N/2 \rfloor + 1$) servers. We define a stable set as follows:

DEFINITION 6.1 STABLE SET: *A stable set is a set of processes that are eventually alive and connected to each other, and which can eventually communicate with each other with some (unknown) bounded message delay.*

We highlight the fact that the definition of a stable set differs from the definition of a *stable component*, which is the network stability requirement needed by group communication based replication protocols (e.g., COReL [12, 13] and Congruity [1, 3]). A stable component is defined in [7] as follows:

DEFINITION 6.2 STABLE COMPONENT: *A stable component is a set of processes that are eventually alive and connected to each other and for which all the channels to them from all other processes (that are not in the stable component) are down.*

The stable set allows servers in the set to receive messages from servers outside of the set, whereas the stable component requires an isolated majority. Thus, our specification of Paxos requires less stability than group communication based protocols during normal-case operation, which require a stable component for the membership algorithm to complete. Paxos allows servers outside the majority to come and go without impacting overall system liveness.

6.2 Leader Election

The leader election algorithm presented in Section 5.3 requires a majority of servers to be able to communicate with each other. The algorithm minimizes the ability of an unstable server to disrupt a stable majority set: once a majority is in the same view, no unstable server can cause a view change until a majority of servers decide to switch views. Thus, the network stability requirement is for a majority of servers to be able to communicate with each other; that is, our leader election component requires a stable set.

We wish to make clear that the choice of leader election algorithm has a significant impact on the overall liveness of the system. We show this by comparing our own network stability requirement to that of the leader election failure detectors used in [21] and [4].

In the specification of Paxos presented in [21], the leader election failure detector is implemented as follows. Each server periodically sends an I-AM-ALIVE message to each other server. Each server maintains a list of the servers it currently believes is alive, removing a server if no I-AM-ALIVE message is received within a timeout period. The failure detector returns the identifier of the server, from the set of those believed to be alive, with the highest identifier. We can formally specify the network stability requirement of this protocol as:

Stable Set with Partial Crash/Partition Isolation. Let S be a stable set that can communicate with bounded delay Δ, and let max_stable_id be the maximum server identifier in S. Let $unstable\_crash$ be the set of servers that perpetually crash and recover, and let $unstable\_partition$ be the set of servers that are eventually alive but whose communication with at least one member of S is not always bounded at Δ. There is a time after which (1) S exists and (2) the members of S do not receive any messages from servers in $unstable\_crash \cup unstable\_partition$ whose identifiers are greater than max_stable_id.

Observe that this network stability requirement does not permit certain unstable servers to repeatedly crash or recover, or to come in and out of a network partition, since this might perpetually prevent the stable servers from agreeing on the identity of the leader.

A similar issue arises with respect to the leader election algorithm specified in [4]. This failure detector is based on an implementation of the Ω failure detector in the crash-recovery model with partial synchrony assumptions. The algorithm guarantees that, given a global stabilization time (i.e., a time after which correct processes stop crashing, remain always up, and can communicate with bounded message delay), there is a time after which exactly one correct process is always trusted by every correct process. The algorithm chooses this process as the leader. As specified, each server maintains a *trustlist*, containing the servers from which it has received an I-AM-ALIVE message within a timeout period. When an I-AM-ALIVE message is received from a currently untrusted server, the server is added to the trustlist and its timeout incremented. When a server recovers, it sends a RECOVERY message to the other servers. Upon receiving a RECOVERY message, the recovering server is added to the trustlist, and a record of the recovery is taken. The leader is chosen as the server from the trustlist with the lowest number of recoveries and the highest server identifier. The network stability requirement of this protocol is:

Stable Set with Partial Partition Isolation. Let S be a stable set that can communicate with bounded delay Δ, and let max_stable_id be the maximum server identifier in S. Let $unstable\_partition$ be the set of servers that are eventually alive but whose communication with at least one member of S is not always bounded at Δ. There is a time after which (1) S exists and (2) the members of S do not receive any messages from servers in $unstable\_partition$ whose identifiers are greater than max_stable_id.

The use of RECOVERY messages prevents unstable servers that repeatedly crash and recover from perpetually preventing agreement on a leader. However, it does not ensure agreement if certain unstable servers repeatedly partition away (without crashing) and then re-merge with the group. In this case, the stable servers would not increment the recovery counters for the unstable servers, allowing some stable server to elect one of the unstable servers at "the wrong time," preventing agreement.

We observe that there is a scale of network stability requirements resulting from the three leader election protocols. The failure detector specified in [21] requires the strongest stability from the network, requiring the stable set to be isolated from particular servers that repeatedly crash and recover or partition. The failure detector specified in [4] requires strictly less stability, requiring only isolation from particular servers that repeatedly partition. The protocol in our own specification is not vulnerable to disruption by any unstable servers. However, in certain cases, our specification requires a longer time to settle on a leader in the stable set than [4], since our elections are not based on which servers are believed to be alive.

The preceding discussion should make it clear that, while it is generally stated that Paxos requires less stability than group communication-based replication protocols, such a claim only makes sense in the context of a complete specification of the Paxos algorithm and all its components.

6.3 Liveness Properties

We now present the liveness properties provided by our specification of Paxos:

Paxos-L1 (Progress). If there exists a stable set consisting of a majority of servers, then if a server in the set initiates an update, some member of the set eventually executes the update.

Paxos-L2 (Eventual Replication). If server s executes an update and there exists a set of servers containing s and r, and a time after which the set does not experience any communication or process failures, then r eventually executes the update.

To gain a deeper understanding of the implications of Paxos-L1, we specify the following two alternative progress requirements:

Strong L1 (Majority Set). If there exists a time after which there is always a set of running servers S, where $|S|$ is at least $(\lfloor N/2 \rfloor + 1)$, then if a server in the set initiates an update, some member of the set eventually executes the update.

Weak L1. If there exists a stable component consisting of a majority of servers, and a time after which the set does not experience any communication or process failures, then if a server in the set initiates an update, some member of the set eventually executes the update.

Strong L1 requires that progress be made even in the face of a (rapidly) shifting majority. We believe that no Paxos-like algorithm will be able to meet this requirement. If the majority shifts too quickly, then it may never be stable long enough to complete the leader election protocol. Weak L1, on the other hand, reflects the stability required by many group communication based protocols, as described above. It requires a stable majority component, which does not receive messages from servers outside the component. Since the leader election protocol specified in Section 5.3 meets Paxos-L1, it also meets Weak L1. We note that group communication based protocols can most likely be made to achieve Paxos-L1 by passing information from the application level to the group comunication level, indicating when new membership should be permitted (i.e., after some progress has been made).

7 Experimental Results

We implemented the Paxos replication protocol described in Section 5. The code is available for download at http://www.dsn.jhu.edu. In this section we evaluate our implementation to provide some intuition as to the level of performance that one can achieve.

As mentioned in Section 6, it is difficult to define exactly what is "the Paxos algorithm." This problem is even more prominent when one tries to evaluate its performance. Should an evaluation of Paxos allow the use of IP multicast, or should links be point-to-point? Should an implementation use aggregation or some other packing technique that can increase throughput? Is such an algorithm still "Paxos?" What type of flow control or reliability mechanisms should be used?

Our approach is to find a balance such that we can evaluate our implementation in a useful way. We first present results without any aggregation at all. The throughput results (measured in updates per second, as is standard) can be thought of as the number of proposals that can be ordered per second. We then present results for a version of our implementation that uses aggregation. These results show that aggregation can result in significantly higher throughputs, without paying a high cost in latency. In fact, latency is actually lower in many cases.

To evaluate the performance of our implementation with respect to other system-related issues, we benchmark two versions of our implementation, which we refer to as Paxos-clean and Paxos-complete. Paxos-clean implements the essential normal-case operations of the Paxos protocol, without additional overhead needed to build a full system. All communication is point-to-point via UDP, and no reconciliation mechanisms are implemented. Flow control consists of a window at the leader, which limits the number of outstanding proposals at any given time. We tuned this window for each configuration to achieve the highest performance without losing any messages. Paxos-complete, on the other hand, incorporates reconciliation, a dynamically adjusted flow control window, and garbage collection. Servers sync updates from their local clients to disk to ensure they are not lost across crashes. They also use IP-multicast for messages sent to all servers.

7.1 Network Setup

Our experimental setup consists of a cluster of twenty 3.2 GHz 64-bit, dual processor Intel Xeon computers, connected via a Gigabit switch. We tested our implementation on configurations ranging from 4 to 20 servers, varying the number of clients initiating write updates.

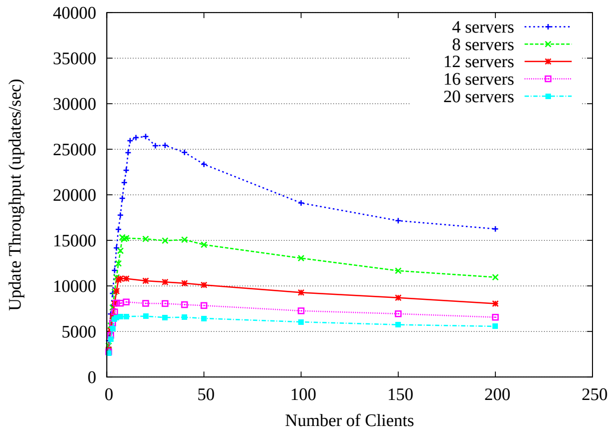

Figure 15: Paxos-clean Throughput, No Disk Writes

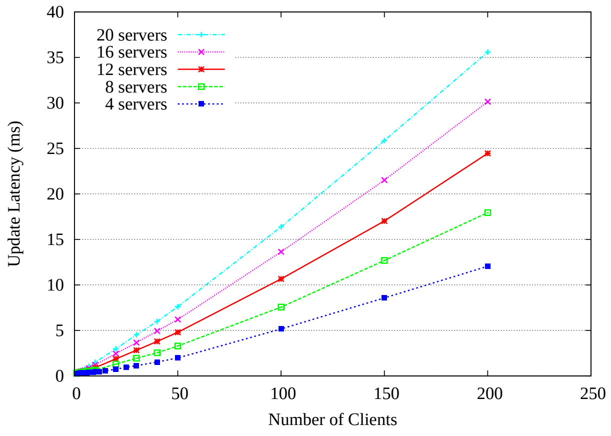

Figure 16: Paxos-clean Latency, No Disk Writes

Each update is 200 bytes long, representative of an SQL statement. Clients are spread as evenly across the servers as possible. Clients connect to the server on their local machine via TCP. Each client has at most one outstanding update at a time, waiting for the reply from its last update before initiating a new update.

7.2 Memory Tests (No Disk Writes)

We first evaluated the performance of our Paxos implementation without the use of disk writes; that is, all operations were performed in memory, and no information was logged on stable storage. While such an implementation is not resilient to crashes, it shows the type of performance that can be achieved strictly when considering the messaging and processing overhead associated with normal-case operation.

Figure 15 shows the throughput achieved by Paxos-clean in configurations ranging from 4 to 20 servers. The results are the average of two trials, where there servers order 300,000 updates in each trial. In all configurations, we observe a steady increase in throughput until the system reaches its saturation point (i.e., when the leader becomes CPU-limited), at which point throughput levels off. We achieve a maximum throughput of 26,401 updates per second when using 4 servers, with a plateau around 6000 updates per second when using 20 servers. In all cases, we observe a slight drop-off as the number of clients continues to increase past the saturation point. This is most noticeable in the case of 4 servers, since the overhead associated with processing more clients is only spread over the 4 servers. We also note that the slope of the lines decreases as more servers are added, since each server must process more incoming messages per update. For the same reason, the number of clients required to bring the system to its saturation point decreases as the number of servers increases.

Figure 16 shows the latency of Paxos-clean without disk writes. Each client first outputs its own average latency and the number of updates it initiated. We then plot the latency as the weighted average taken across all clients. In all configurations, latency increases very slightly until the system reaches its saturation point, at which point latency begins to increase linearly. The slope of the increase is impacted by the size of the flow control window used (14, 9, 6, 4, and 3 for configurations of size 4, 8, 12, 16, and 20, respectively). This

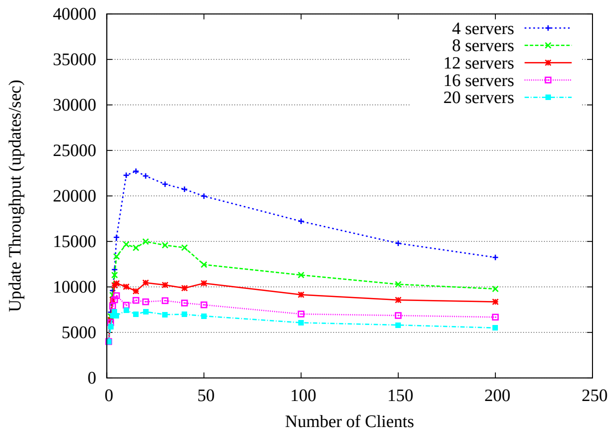

Figure 17: Paxos-complete Throughput, No Disk Writes

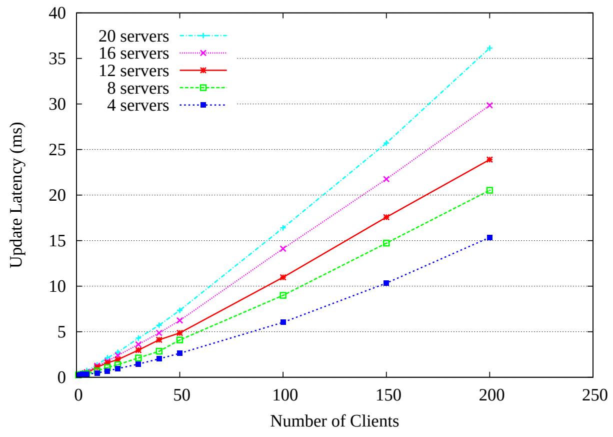

Figure 18: Paxos-complete Latency, No Disk Writes

reflects the amount of pipelining that can be used, which affects how long an update will wait in the leader's queue before being initiated in a PROPOSAL message.

Figures 17 and 18 show the same tests for Paxos-complete. Paxos-complete achieves slightly lower throughput compared to Paxos-clean in the 4-server and 8-server configurations (with maximum throughputs of 22,116 and 14,980 updates/sec compared to 26,401 and 15,330 updates/sec, respectively). This reflects the overhead associated with dynamically adapting the flow control window (which results in triggering some timeouts) and the associated reconciliation, as well as additional overhead needed for garbage collection. Paxos-clean and Paxos-complete show roughly the same throughput in the case of 12 servers, and Paxos-complete achieves slightly higher throughput than Paxos-clean in the 16-server and 20-server configurations. This shows the benefit of using IP-multicast in larger configurations, reducing the number of system calls needed for sending.

Both results indicate that Paxos can achieve very low latency when a relatively small number of clients is used, remaining below 5 ms at 30 clients for all configurations, below 8 ms at 50 clients for all configurations, and roughly 35 ms with 200 clients.

7.3 Synchronous Disk Writes

We next evaluated the performance of our implementation when using synchronous (forced) disk writes. The use of synchronous writes is necessary to preserve safety across crashes. As described in Section 4, Paxos requires all servers to write to disk on each update. To minimize the latency associated with writes, we use the variant of Paxos in which the leader sends an ACCEPT message, paying the cost of one extra message for the ability to write to disk in parallel with the other servers.

Our stable storage consists of a log file, where proposals, ordered updates, local data structure information, and pending updates are stored. The log file is written synchronously before a server sends an ACCEPT message, and asynchronously when a server globally orders an update.

Figures 19 and 20 show the throughput and latency of Paxos-clean using synchronous disk writes. The results are the average of two trials, where the servers order 1500 updates

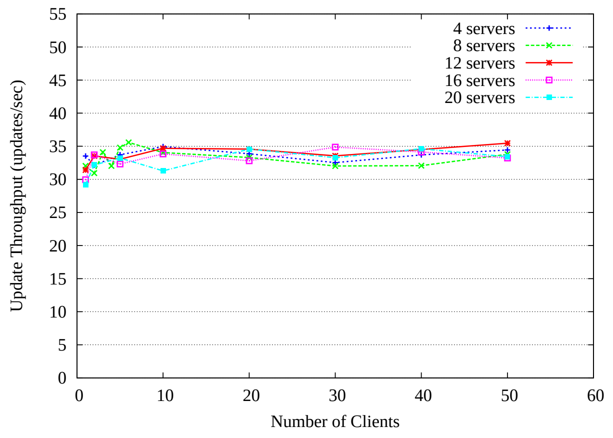

Figure 19: Paxos-clean Throughput, Synchronous Disk Writes

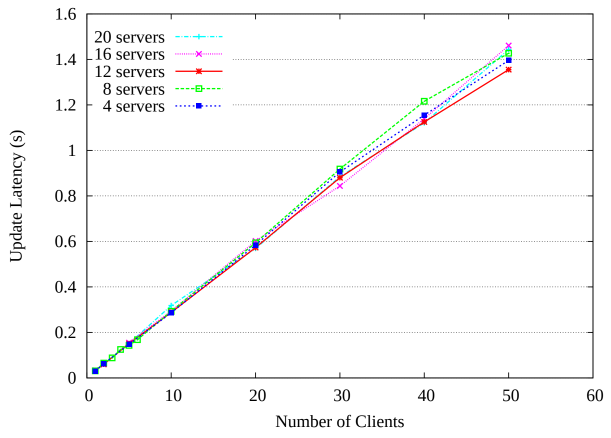

Figure 20: Paxos-clean Latency, Synchronous Disk Writes

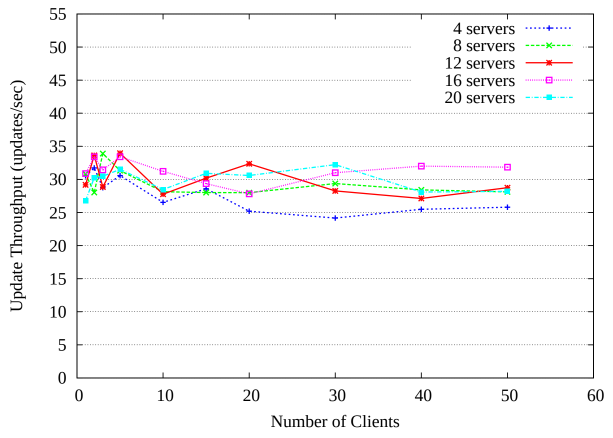

Figure 21: Paxos-complete Throughput, Synchronous Disk Writes

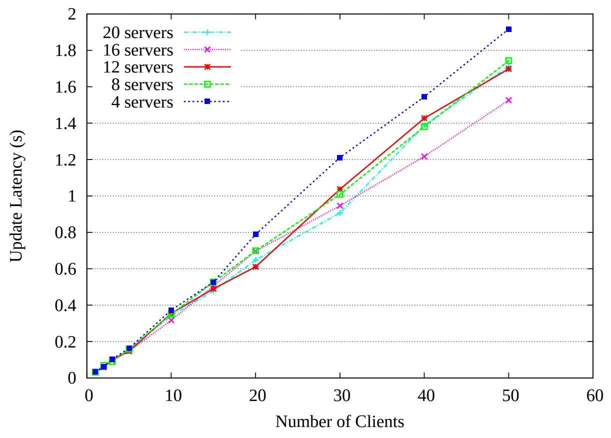

Figure 22: Paxos-complete Latency, Synchronous Disk Writes

in each trial. As seen in Figure 19, Paxos-clean achieves a throughput between 33 and 35 updates per second in all of the configurations tested, reaching saturation at 2 clients in all cases. The throughput is limited by the speed at which a single server can sync to disk. As seen in Figure 20, the update latency increases linearly after the system reaches saturation, reaching one second at roughly 35 clients.

Figures 21 and 22 show the results for Paxos-complete. Recall that servers in Paxos-complete sync updates from their local clients to disk on origination. Paxos-complete achieves between 25 and 35 updates per second in all configurations tested, which is slightly lower than the throughput of Paxos-clean, with slightly higher latency.

7.4 Asynchronous Disk Writes

Since the price of faster disk technology (e.g., flash disks, battery-backed-up RAM disks) continues to decrease, it is useful to consider the performance of Paxos when disk writes are less costly. For this reason, we tested our implementation using asynchronous disk writes.

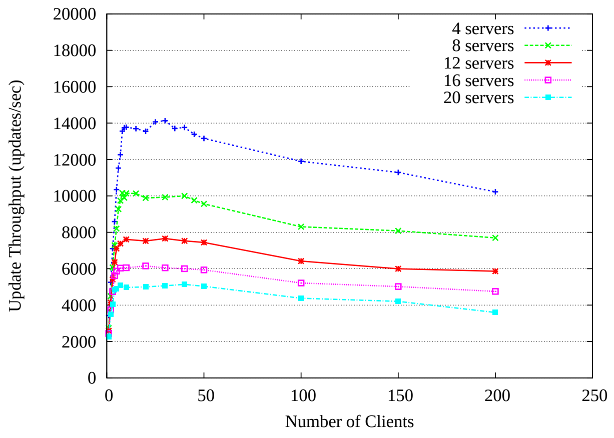

Figure 23: Paxos-clean Throughput, Asynchronous Disk Writes

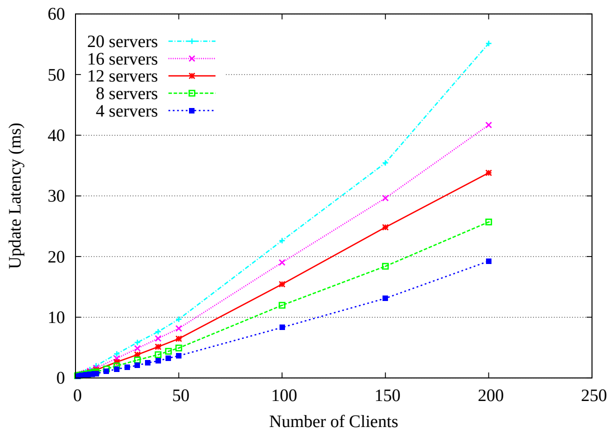

Figure 24: Paxos-clean Latency, Asynchronous Disk Writes

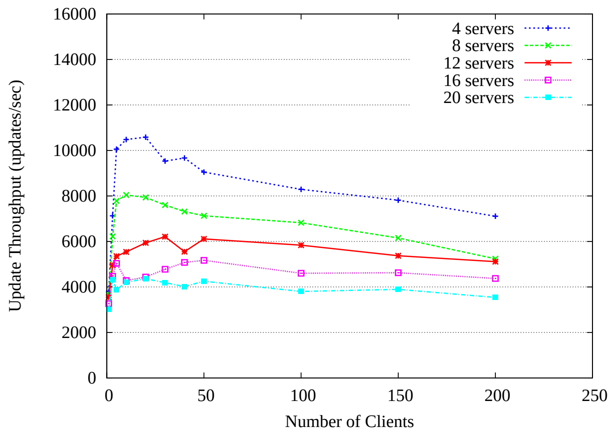

Figure 25: Paxos-complete Throughput, Asynchronous Disk Writes

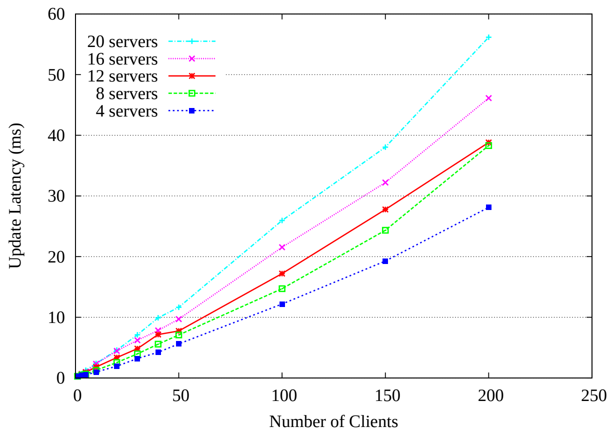

Figure 26: Paxos-complete Latency, Asynchronous Disk Writes

Figures 23 and 24 show the update throughput and latency of Paxos-clean when using asynchronous writes. We observe the same general trend as the tests without disk writes, but achieve lower plateaus, reflecting the overhead associated with the writes.

Figures 25 and 26 show the same trends for Paxos-complete. Paxos-complete achieves slightly lower throughput and slightly higher latency than Paxos-clean, reflecting the additional overhead of dynamically adapting the window and performing reconciliation.

7.5 Scalability

Finally, we tested the scalability of our implementation, without disk writes, when the number of clients is kept constant. We maintain 25 clients and vary the number of servers.

In Figure 27, we plot the number of updates per second Paxos-clean achieves as a function of the number of servers. For each number of servers, we adjusted the flow control window at the leader to the highest value possible while remaining stable. The more rapid initial decrease in throughput (from 4 servers to 6 servers, and from 6 servers to 8 servers) is caused

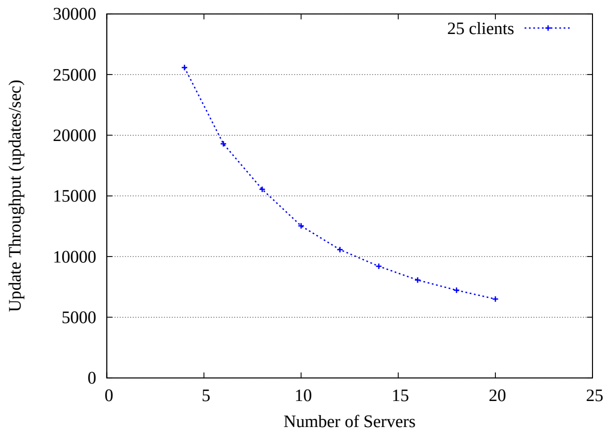

Figure 27: Paxos-clean, Throughput vs. Servers, No Disk Writes, 25 Clients

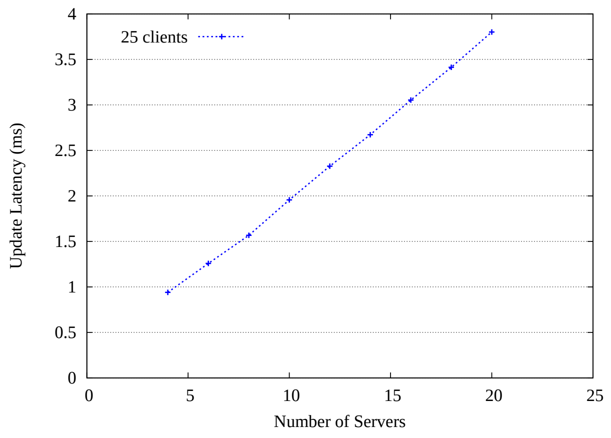

Figure 28: Paxos-clean, Latency vs. Servers, No Disk Writes, 25 Clients

by the need to shrink the flow control window a significant amount between configurations, from 17 to 14, and then from 14 to 10. We contrast this with the smaller decrease between 16 servers, 18 servers, and 20 servers, where the window was decreased from 5 to 4 to 3. Figure 28 shows the average latency achieved by Paxos-clean as a function of the number of servers. Figures 29 and 30 show the same trends when running Paxos-complete. The curves show slightly more variability than those for Paxos-clean (i.e., they are not as smooth), reflecting the use of the dynamic window and the need for retransmissions as loss is created.

We observe that, in Paxos-clean, as the number of servers is doubled, the maximum throughput achieved is cut roughly in half, implying a linear decrease in performance. This is contrary to what one might expect, given that doubling the number of servers results in four times as many messages (since communication among all servers in the Accept phase is all-to-all). Note, however, that doubling the number of servers only requires the leader to processes twice as many ACCEPT messages per update, not four times. Thus, the decrease in maximum performance is a function of the number of messages processed, not the total number of messages in the network. This is validated by the fact that Paxos-complete shows the same trend: although the number of message sends is reduced by using IP-multicast, the number of message receives is the same for Paxos-clean as for Paxos-complete.

7.6 The Impact of Aggregation

We now present results for the version of our implementation that uses aggregation. Without disk writes, we aggregate in several ways. First, the leader packs multiple updates into a single PROPOSAL message. Second, non-leader servers sends a single ACCEPT message for several PROPOSALS, greatly reducing the number of messages that must be processed (although each ACCEPT is now larger, since it contains a list of the sequence numbers that it covers).

As seen in Figures 31 and 32, aggregation significantly changes the trend of the throughput and latency graphs. Instead of throughput degrading as the number of servers increases, it increases, reaching a maximum of over 40,000 updates per second. Reducing the num-

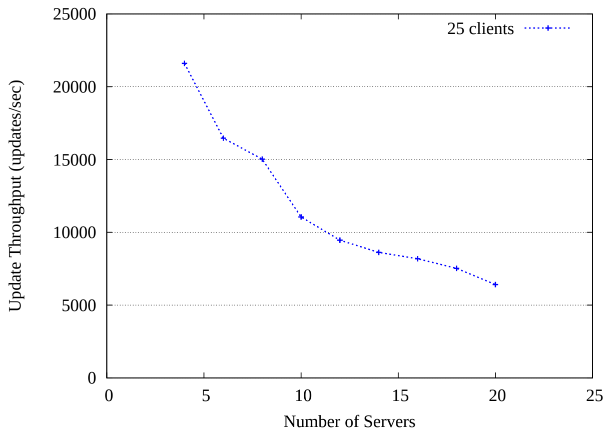

Figure 29: Paxos-complete, Throughput vs. Servers, No Disk Writes, 25 Clients

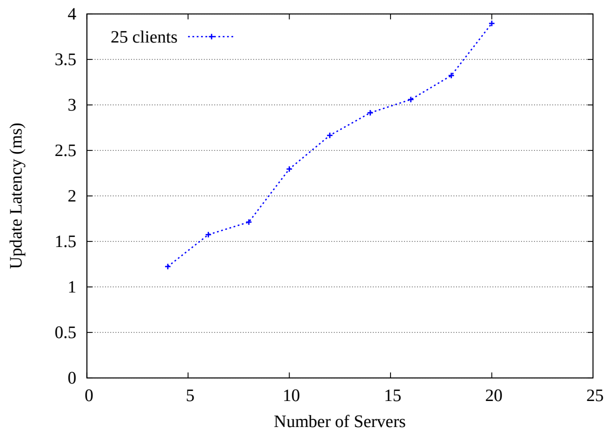

Figure 30: Paxos-complete, Latency vs. Servers, No Disk Writes, 25 Clients

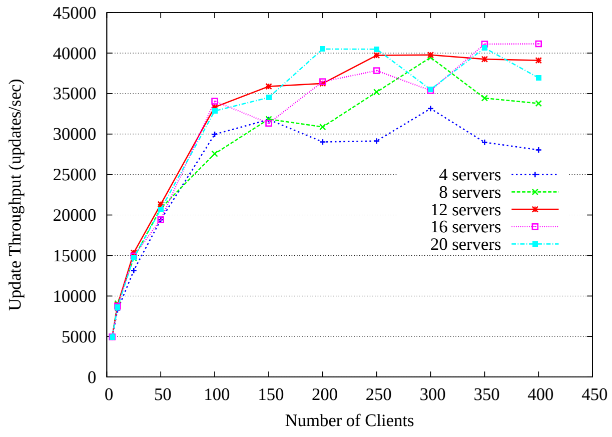

Figure 31: Paxos-complete Throughput, Aggregation, No Disk Writes

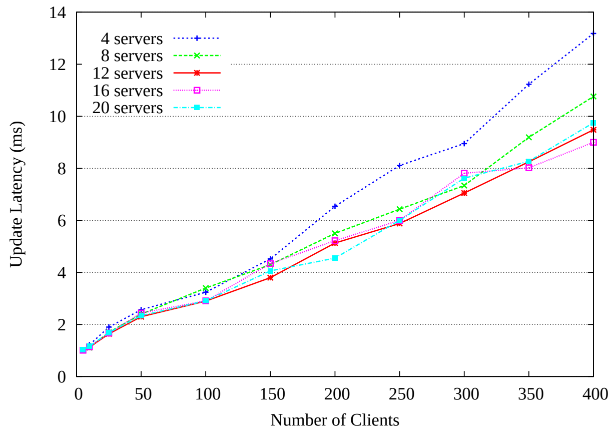

Figure 32: Paxos-complete Latency, Aggregation, No Disk Writes

ber of ACCEPT messages being sent greatly reduces the overhead associated with adding more servers. It also allows more of the CPU to be devoted to processing new UPDATE and PROPOSAL messages, rather than extra acknowledgements. At small numbers of clients, aggregation results in slightly higher latency than when it is not used. However, given that the system can support a much higher maximum throughput, latency is signficantly reduced at higher numbers of clients.

When using synchronous disk writes, we sync several PROPOSAL messages to disk at once, while sending back a single ACCEPT message. We also sync several updates from local clients at once, amortizing the cost of syncing upon initiation over several updates. The results are shown in Figures 33 and 34. Comparing Figures 21 and 33, we can see that aggregation dramatically increases the maximum throughput for all configurations of servers (from roughly 35 updates per second to about 1500 updates per second). Again, this higher throughput results in signficantly lower latency compared to when aggregation is not used.

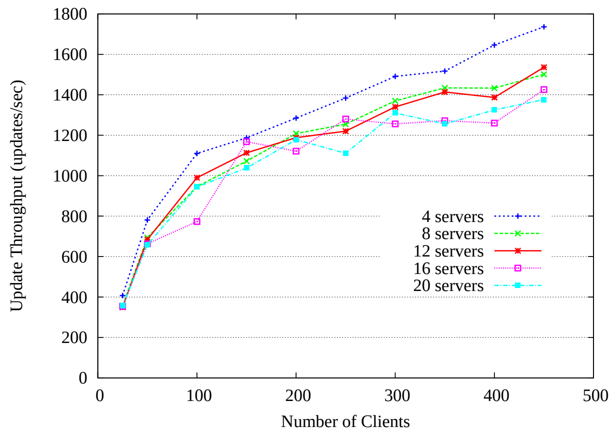

Figure 33: Paxos-complete Throughput, Aggregation, Synchronous Disk Writes

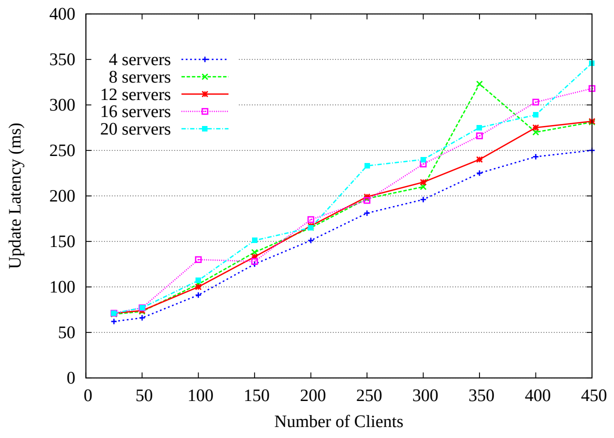

Figure 34: Paxos-complete Latency, Aggregation, Synchronous Disk Writes

8 Conclusions

In this paper, we completely specified the Paxos algorithm in system builder language such that one can implement it. Our analysis of the liveness and performance of Paxos leads us to several conclusions. First, Paxos is a very robust algorithm, allowing any majority of servers to make safe progress regardless of past failures, and requiring only a stable set of connected servers. This makes it suitable for environments in which the network stability required for group membership protocols to complete may not be achievable (i.e., when one may not have a stable component).

Second, when using asynchronous disk writes (or when writing to disk is not required), Paxos is able to provide low update latency when the number of clients is reasonably small; this is due to the fact that only two communication rounds among the servers are needed during normal-case operation. We have also shown that aggregation greatly improves the performance of Paxos when synchronous disk writes are used. With a difference of almost two orders of magnitude, it seems clear that a practical system using Paxos as a replication engine should employ aggregation techniques.

Finally, we comment that a practical implementation of Paxos must address many real issues addressed by group communication based replication protocols such as COReL and Congruity. These include reliability, flow control, and reconciliation. Both COReL and Congruity build reliability and efficient reconciliation into their protocols, making it easier to integrate them into a real system. Paxos, on the other hand, must handle these issues with additional mechanisms not specified in the original algorithm. This work was a first step towards specifying Paxos such that system builders can implement it, which required specifying precisely these missing system mechanisms, and filling in details such as which leader election algorithm is used. We observed that how these details are specified can greatly impact both the theoretical guarantees that can be offered and the practical performance of the replication system.

Acknowledgements

We are grateful to John Lane and John Schultz for the countless hours they have spent working with us to understand the Paxos protocol, its implementation, and its performance. Their questions and comments were invaluable in refining this work and helping to give it context.

References

[1] Yair Amir. *Replication Using Group Communication Over a Partitioned Network*. PhD thesis, The Hebrew University of Jerusalem, Jerusalem, Israel, 1995.

[2] Yair Amir, Jonathan Kirsch, and John Schultz. On the liveness and performance tradeoffs of state machine replication: A comparative analysis of Congruity, COReL, and Paxos. Technical Report CNDS-2008-3. In Preparation, Johns Hopkins University, www.dsn.jhu.edu, 2008.

[3] Yair Amir and Ciprian Tutu. From total order to database replication. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 494–503, Washington, DC, USA, 2002. IEEE Computer Society.

[4] R. Boichat, P. Dutta, S. Frlund, and R. Guerraoui. Deconstructing Paxos. *SIGACT News*, 34(1):47–67, 2003.

[5] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 173–186. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[6] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC '07)*, pages 398–407, New York, NY, USA, 2007. ACM Press.

[7] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[8] K. Eswaran, J. Gray, R. Lorie, and I. Taiger. The notions of consistency and predicate locks in a database system. *Communication of the ACM*, 19(11):624–633, 1976.

[9] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[10] Eli Gafni and Leslie Lamport. Disk Paxos. *International Symposium on Distributed Computing*, pages 330–344, 2000.

[11] Richard A. Golding and Kim Taylor. Group membership in the epidemic style. Technical Report UCSC-CRL-92-13, University of California, Santa Cruz, CA, March 1992.

[12] I. Keidar. A highly available paradigm for consistent object replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[13] Idit Keidar and Danny Dolev. Efficient message ordering in dynamic networks. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 68–76, 1996.

[14] Idit Keidar and Danny Dolev. Increasing the resilience of distributed and replicated database systems. *Journal of Computer and System Sciences*, 57(3):309–324, 1998.

[15] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

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

[17] Leslie Lamport. Paxos made simple. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 32:18–25, 2001.

[18] Butler Lampson. The ABCD's of Paxos. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC '01)*, page 13. ACM Press, 2001.

[19] Harry C. Li, Allen Clement, Amitanand S. Aiyer, and Lorenzo Alvisi. The Paxos register. *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS '07)*, 2007.

[20] Louise E. Moser, Yair Amir, P. Michael Melliar-Smith, and Deborah A. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems (ICDCS '94)*, pages 56–65, 1994.

[21] Roberto De Prisco, Butler Lampson, and Nancy Lynch. Revisiting the Paxos algorithm. *Theor. Comput. Sci.*, 243(1-2):35–91, 2000.

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

[23] Dale Skeen. A quorum-based commit protocol. *6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 69–80, 1982.