Paxos for System-Builders

Jonathan Kirsch
Advanced Distributed Systems (600.667) Final Report
Professor: Yair Amir
Johns Hopkins University
Spring 2005

1 Introduction

The goal of this document is to clearly and completely specify the Paxos replication algorithm such that system-builders can understand it, and, if they deem it suitable for their applications, implement it.

Unfortunately, many details are simply not specified in the original description of the algorithm[6, 5]. The most important omissions include: how failures are detected, what type of leader election algorithm is used, and how recovering servers learn what has been agreed upon while they were down. These details are non-trivial, and we fill them in.

The remainder of this paper is presented as follows. In Section 2, we define the system model and describe the safety and liveness requirements that Paxos meets. Section 3 presents the Paxos consensus algorithm, which allows a group of servers to agree on a single update for execution. We avoid a rigorous description of the consensus algorithm, as many of the details are directly applied to the replication algorithm, which we do present formally. In Section 4, we show precisely how the replication algorithm extends the consensus algorithm, assigning a global, persistent, consistent ordering to a sequence of updates. This replication algorithm can be used to implement a reliable distributed service via the state machine approach. Section 5 provides discussion, Section 6 concludes.

2 System Model

We assume a system of N servers which communicate by passing messages in a communication network. The servers have unique identifiers in the range 1 to N. Communication is asynchronous and unreliable; in particular, messages can be duplicated, lost, and can take an arbitrarily long time to arrive, but those that do arrive are uncorrupted. Servers can crash and subsequently recover, and they have access to stable storage, which is retained across crashes. Servers do not exhibit Byzantine behavior.

The servers are implemented as deterministic state machines. We assume that all servers begin in the same initial state. A server moves from one state to the next by applying an
update to its state machine; we say that the server executes the update. A clients proposes an update by sending it to a server. The server then introduces the update for execution; we say that the server initiates the update.

Paxos assigns a global, persistent, consistent ordering to client updates. We say that a server orders an update when it learns which sequence number the update has been given in the global order. By (deterministically) executing the same updates in the same order, all servers remain consistent.

2.1 Requirements

We now precisely define the safety and liveness requirements that Paxos meets:

S1 (Agreement). If two servers execute the ith update, then these updates are identical.

S2 (Validity). Only an update that was proposed by a client (and subsequently initiated by a server) may be executed.

L1 (Progress). If there exists a set 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.

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.

2.1.1 Alternative Progress Requirements

We specify two alternative progress requirements to the L1 requirement to gain a deeper understanding of the implications of L1:

Strong L1. If there exists a time after which there is always a set of running servers S, where |S| is at least a majority, 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 set consisting of a majority of servers, and a time after which the set does not experience any communication or process failures, and the members of the set do not hear from any members outside of the set, 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. Note that this problem exists even if the potential leader is stable across consecutive majorities: since the other processes can crash or partition continuously, there can be no guarantees made about communication failures or delays.

Weak L1, on the other hand, reflects the stability required by many group membership algorithms. It requires a stable majority component, which does not receive messages from
servers outside the component. Since the leader election algorithm specified below meets L1, it also meets Weak L1. We note that group communication-based protocols can most likely be made to achieve L1 by passing information from the application level to the group communication level, indicating when new membership should be permitted (i.e. after some progress has been made).

Finally, we note that the Paxos algorithm can actually meet a stronger requirement than L1 during normal-case operation. Once a leader has been elected, the algorithm makes progress as long as the leader remains stable and can communicate with a majority of servers; the other servers in the majority can shift very rapidly, providing a high degree of fault-tolerance.

3 The Paxos Consensus Algorithm

We first present the basic Paxos consensus algorithm, which allows a set of servers to agree upon a single update for execution. We defer discussion of such details as failure detection, leader election, and reconciliation until we present the replication algorithm, since the same mechanisms can be applied to both.

3.1 The Concept

From a high-level perspective, the algorithm proceeds as follows. One server is elected as the leader, who initiates a client update for execution. The other servers accept the proposal, and tell each other that they have done so. A server knows 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.

This simple, two-phase protocol 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 two servers may believe they are the leader at a given time. The algorithm must guarantee that, even if two 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. It is useful to think of the system as proceeding through a series of views, with a view change occurring each time a new leader is elected. Each view has exactly one leader, who 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) that they have accepted.

---

1In the context of the replication algorithm, we allow multiple proposals to be made within a given view, and 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 no update has been ordered (if no accepted proposals were reported among the majority) or that an update may have been ordered (if some servers have accepted 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). For a more detailed description of the intuition behind this, see [5].

3.2 The Algorithm

We now present the consensus algorithm in more detail. We assume that a leader has just been elected, and defer discussion of leader election, reconciliation, and other mechanisms until section 3, where we detail the complete replication algorithm.

Upon being elected, the new leader initiates the Prepare phase of the algorithm. The leader prepares the proposal it will make by sending a PREPARE(viewNum) message to all servers. As described above, the leader waits for responses from a majority of 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 number.

A server can respond to a PREPARE message with the most recent proposal it has accepted, if any, provided it has not already received a message with a higher view number. The response is of the form PREPARE_OK(viewNum, prop), and is sent only to the leader. Note that the prop field may be empty.

If the leader obtains a majority of PREPARE_OK messages for the current view, the update it can propose is either constrained (if an earlier proposal was accepted) or unconstrained (if no proposal was accepted). The leader sends out a PROPOSAL(viewNum, update) message to the servers, where update is the most recent constraining update, or any update if none was reported.

A server receiving a proposal can accept it, as long as it is has not already received a message with a higher view number. A server that accepts a proposal sends an ACCEPT(viewNum, update) message to all servers. A server orders an update when it receives a majority of distinct ACCEPT messages for the same proposal.

3.3 Disk Writes

To preserve the safety requirements described above in the face of crashes, servers need to write to stable storage at key places during the algorithm. 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 accepting a proposal, a server must write the (viewNum, update) pair 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 \texttt{PREPARE} messages, which in turn will correctly constrain future proposals.

**Making the first Proposal in a new View.** Before making a proposal in a newly-established view, the leader must write the view number to stable storage. When a crashed leader recovers, it uses a view number higher than the one written. This ensures that a server will never issue two different proposals with the same view number.

This last disk write is necessary in the general case, where the leader does not process its own \texttt{PREPARE} message before sending it. The leader may never receive its own message, in which case it prepares the view without writing the view number to disk (since it never sent a \texttt{PREPARE\_OK} message).

Note, however, that this scheme may require two sequential disk writes: when the non-leader sends the \texttt{PREPARE\_OK}, and when the leader writes before sending the first proposal. We can remove this additional latency by ensuring that the leader processes its own \texttt{PREPARE} message. The leader can write to disk \textit{after} it sends out the \texttt{PREPARE} message, in parallel with those servers responding with a \texttt{PREPARE\_OK}. Since this write occurs before any proposals are made in the new view, it achieves the same semantics as before, even if the leader crashes before writing to disk.

### 3.4 Variations and Optimizations

We introduce several variations and optimizations to the core algorithm which can be used to reduce message complexity at the cost of slightly higher latency or decreased performance in the face of crashes.

#### 3.4.1 Communication Patterns

1. As presented above, the leader sends its messages to all servers. Note, however, that the leader only needs a majority of \texttt{PREPARE\_OK} responses to prepare the view. The leader can, therefore, send only to a majority of servers to reduce the number of messages being sent. This optimizes for the common case, in which all servers are up and can communicate in a timely manner. If one of the majority crashes, however, the leader will be unable to requisition the necessary responses, in which case it can send to the other servers; the latency of the request will be greater than if it had sent to all servers originally.

2. Currently, a server that accepts a proposal reports this to all of its peers. One variation is for an acceptor to report only to the leader, who will wait for a majority of responses and then broadcast the ordered update to all servers. This adds an extra round of latency into the algorithm, 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 request will not be ordered until a new leader is elected. To remedy this, the acceptors can send to a group of servers, trading off fault-tolerance for a higher message complexity.
4 The Paxos Replication Algorithm

4.1 The Concept

The Paxos replication algorithm extends the consensus algorithm to assign a global, persistent, consistent order to a sequence of client updates. As before, the servers are implemented as deterministic state machines. A server executes an update after it has executed all previous updates in the global order.

Intuitively, the replication algorithm runs multiple instances of the consensus algorithm in parallel; the update ordered in the $i$th instance is assigned the $i$th sequence number in the global order.

The communication pattern of the replication protocol is almost identical to that of the consensus algorithm. The protocol elects a leader, who assigns sequence numbers to client updates. The leader proposes the assignments to the rest of the servers, who accept them and report to each other that they have done so. A server orders an update when it learns that a majority of servers have accepted the leader’s assignment. Since the leader might crash, the algorithm provides a failure detection mechanism, which allows a new leader to be elected when insufficient progress is being made. A reconciliation mechanism ensures that all servers are as up-to-date as possible.

Since the leader remains in power while progress is being made, it will usually make multiple proposals in the same view. Therefore, we cannot simply differentiate proposals by their view number, as we could in the consensus algorithm. A proposal is distinguished by the combination of its view number and its sequence number. Note that the earlier/later temporal relationship is still solely defined by view number.

4.2 State Machine

We present the algorithm in the form of a deterministic state machine with five states:

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

Rec_Leader. The new leader has completed the leader election algorithm. It initiates the reconciliation protocol to bring at least a majority of servers up to date.

Rec_NonLeader. A server has completed the leader election algorithm, and was not elected leader. It participates in the reconciliation protocol until becoming up-to-date.

Regular_Leader. The leader is up-to-date and runs the Prepare phase of the algorithm, after which it can assign global sequence numbers to incoming updates and propose them to the rest of the servers.

Regular_NonLeader. A non-leader has finished the reconciliation protocol for the last view it installed. It forwards client updates to the leader for ordering, 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 algorithm specification. The second type of event is a *timer* event. These events occur under certain conditions after some time elapses without the server taking some action.

We present the replication algorithm “chronologically,” from leader election to reconciliation, and finally to normal case behavior. We describe the new data structures and message types required in each stage.

### 4.3 Leader Election

From a high-level perspective, the leader election algorithm proceeds as follows. Servers vote to install a new view if insufficient progress is being made (or, in certain circumstances, upon recovery). The leader of view $i$ is the server where $serverID \equiv i \mod N$. When the leader of the new view receives a majority of votes for the same view, it broadcasts a message to confirm the installation of the view. The algorithm is adapted from the view change protocol developed by Castro and Liskov in [2].

Each server maintains a “loose” membership list, consisting of the set of servers with which this server can communicate. The server uses this list in determining which view to vote for: the server will only vote for a view whose leader is in its membership list. The server votes for these candidates in round-robin fashion.

We now present the algorithm more formally.

#### 4.3.1 Data Structures

- *ServerID* - a unique identifier for this server.
- *CurrentView* - the view number this server is in.
- *Attempt View* - the view number this server is trying to install.
- *MaxView* - the highest view number this server has seen so far.
- *ViewTimer* - how long a server will wait, after receiving a majority, before giving up on the view.
- *MemberList* - a list of servers with which this server can communicate.
- *Timeouts* - how long a server will wait, without hearing from a particular server, before removing it from MemberList.

#### 4.3.2 Message Types

- *View Change* - contains the following fields:
  - *serverID* - unique identifier of this server.
  - *viewNum* - view number being voted for.
• **New View** - contains the following fields:
  
  – `serverID` - unique identifier of this server.
  
  – `viewNum` - view number being established.

• **Alive** - contains the following field:
  
  – `serverID` - unique identifier of this server.

### 4.3.3 Loose Membership

Each server periodically broadcasts an **ALIVE** message (if it has not multicast any other message recently). Upon receiving an **ALIVE** message from server `i`, where `i ∉ MemberList`, the server adds `i` to its MemberList, and resets `i`’s timeout value. If `i` was already present, the server increments `i`’s timeout. This probing timeout mechanism builds trust among servers who are continuously connected. If server `j`’s timeout expires, `j` is removed from MemberList. This scheme is based on a similar scheme presented in [7].

```
Upon receiving ALIVE(serverID)
  if serverID ∉ MemberList
    MemberList ← MemberList ∪ {i}
    Timeouts[serverID] ← default
  else
    Timeouts[serverID] += increment

Upon expiring: Timeouts[i]
  MemberList ← MemberList \ {i}
```

| **Listing 1**: Maintaining Loose Membership |

### 4.3.4 Triggering Leader Election

A server initiates the leader election algorithm in three cases. The first case occurs during system startup. A server realizes that it has never installed a new view, and begins the protocol. The second case occurs during recovery. If a server realizes that it was the leader of the last view it successfully installed, it must abandon this view to preserve safety. Finally, a server initiates the protocol when it suspects the leader. This occurs when the server received a client update, but has not executed it within a certain time period. The server abandons the current view and tries to install a new one.

### 4.3.5 Choosing AttemptView

As described above, a server votes for candidate views in a round-robin manner. The pseudocode in Listing 2 implements this:
current_id ← AttemptView mod\textit{N}  
\texttt{j} ← \min((\texttt{current}\_id+i)\mod\textit{N} \in \textit{MemberList}, i > 0)  
candidate ← (\texttt{current}\_id+j)\mod\textit{N}  
AttemptView ← \min_{v}(v > \textit{MaxView}, v \equiv \texttt{candidate} \mod\textit{N})  
\textit{MaxView} ← \textit{AttemptView}  
return \textit{AttemptView}

\textbf{Listing 2: GetNextAttempt()}

4.3.6 The Algorithm

Upon triggering the leader election algorithm, a server runs the GetNextAttempt() procedure and broadcasts a \texttt{VIEW\_CHANGE} message for this view. When the leader receives a majority of distinct messages for this view, it sends out a \texttt{NEW\_VIEW} message, which tells the other servers to install the new view. They update \textit{CurrentView} and \textit{AttemptView} accordingly.

Several techniques are employed to ensure synchronization:

- After sending a \texttt{VIEW\_CHANGE} message for view \textit{V}, a server waits until it receives a majority of distinct view change messages for View \textit{V} before starting its ViewTimer. When it receives these messages, it sets the timer to expire after some time \textit{T}. If the view change is not completed by then, it starts the protocol for the next AttemptView, \textit{V‘}. This time, upon receiving a majority of view change messages for \textit{V‘}, the ViewTimer is set to expire after 2\textit{T}, and so on.

- If a server receives a \texttt{VIEW\_CHANGE} message for a view higher than AttemptView, it sends out a \texttt{VIEW\_CHANGE} message for this view and updates AttemptView accordingly, under the following conditions:

  1. The server already suspects the leader
  2. The server’s ViewTimer is not already set

The first condition removes the power of an unstable process once a stable majority has already established a view.

The second condition gives the greatest opportunity to let the view change run to completion. Recall that a process starts its timer when it receives a majority of view change messages for the view it is currently trying to install. Thus, while a process is waiting on the \texttt{NEW\_VIEW} message that will locally establish the new view, it ignores anything from the unstable process.

4.4 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 \textit{execute} these updates, due to gaps in the global sequence.

We provide a “two-pronged” approach to reconciliation. First, we extend the view change protocol to enforce reconciliation whenever possible. The leader of the new view brings itself
up to date, and then performs message retransmissions based on state reported by the other servers. We also provide anti-entropy [4] sessions (both lazy and eager) during normal-case operation to overcome message loss.

4.4.1 Data Structures

- **Ordered[]** - an array of updates, indexed by sequence number, storing the updates this server has already ordered.
- **LocalAru** - the sequence number up to which this server has ordered all updates.
- **TargetAru** - the sequence number up to which this server must reconcile before proceeding.
- **RequestTimer** - signals when to request retransmission.

4.4.2 Message Types

We expand the **VIEW_CHANGE** and **NEW_VIEW** messages from the leader election algorithm to implement the reconciliation mechanism, and introduce two new message types.

- **View Change** - contains the following fields:
  - **serverID** - unique identifier of this server.
  - **viewNum** - view number being voted for.
  - **localAru** - local aru value of this server.

- **New View** - contains the following fields:
  - **serverID** - unique identifier of this server.
  - **viewNum** - view number being established.
  - **(start, end)** - range of sequence numbers to be retransmitted.
  - **maxSender** - identifier of the server responsible for retransmitting the updates from start to end.

- **Ordered** - contains the following fields:
  - **serverID** - unique identifier of this server.
  - **seq** - sequence number of the ordered update.
  - **update** - the update that was ordered.

- **Request** - contains the following fields:
  - **serverID** - unique identifier of this server.
  - **nackList** - sequence numbers of the updates being requested for retransmission.
4.4.3 Reconciliation: Leader

From a high-level perspective, the leader’s reconciliation protocol proceeds as follows. The protocol begins when the leader receives a majority of VIEW_CHANGE messages for the new view. Each message contains the local aru value of the sender. The leader uses this knowledge to compute start and end, the low and high water marks, respectively, of the majority. This range defines the messages that must be retransmitted. The leader also computes maxSender, which is the server, among the majority, with the most knowledge (i.e. with localAru = end). To avoid unnecessary retransmissions, the leader gives preference to itself if multiple servers (including itself) have localAru = end. These actions are found in Listing 3.

<table>
<thead>
<tr>
<th>Upon receiving a majority of VIEW_CHANGE messages:</th>
</tr>
</thead>
<tbody>
<tr>
<td>start ← min(localArui, ∀i ∈ VC_set)</td>
</tr>
<tr>
<td>end ← max(localArui, ∀i ∈ VC_set)</td>
</tr>
<tr>
<td>if localAru = end</td>
</tr>
<tr>
<td>maxSender ← ServerID</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>maxSender ← id of server with localAru = end</td>
</tr>
<tr>
<td>Send NEW_VIEW(serverID, AttemptsView, (start, end), maxSender)</td>
</tr>
</tbody>
</table>

Listing 3: Initiate_Reconciliation()

After initiating the reconciliation protocol, the leader runs a reliable protocol (via unicast) with maxSender to bring itself up to date. It sets the RequestTimer, which signals the leader to request, using negative acknowledgements, the updates it still needs. If the leader is unable to communicate with maxSender and is still not reconciled, it re-runs the Initiate procedure, this time excluding maxSender’s view change message from the VC_set. Eventually, the leader’s localArui value meets the latest end value, at which point the leader is reconciled. Note that, in the extreme case, the leader will continue to reduce the VC_set until the set consists only of the leader’s own message. The leader multicasts all messages from start to end, and shifts to the Reg_Leader state. It will continue to respond to REQUEST messages to overcome message loss.

4.4.4 Reconciliation: Non-Leader

A non-leader enters the reconciliation phase after receiving the leader’s NEW_VIEW message. If the server is the maxSender, it retransmits the appropriate messages to the leader, and shifts to the Reg_NonLeader state, since it is already up-to-date. Otherwise, the server stores the end value as TargetAru, and sets the RequestTimer, signaling when it should request those messages it still has not ordered. If, at any point, a server is unable to communicate with the leader, it aborts the reconciliation protocol and returns to the leader election phase. When the server’s localAru reaches TargetAru, the server is reconciled, and shifts to the Reg_NonLeader state. These actions are summarized in Listing 4.
Upon receiving \texttt{NEW\_VIEW} \\
\hspace*{1em} \textbf{if} \ ServerID = maxSender \\
\hspace*{2em} Retransmit Ordered[start] to Ordered[end] to leader \\
\hspace*{2em} Shift$_{\text{to\_Reg\_NonLeader}}$() \\
\hspace*{1em} \textbf{else} \\
\hspace*{2em} \textbf{if} \ end < TargetAru \\
\hspace*{3em} TargetAru \leftarrow \text{end} \\
\hspace*{3em} Set RequestTimer \\
Upon expiring: RequestTimer \\
\hspace*{2em} Compute nackList \\
\hspace*{2em} Send \texttt{REQUESTnackList} \\
\hspace*{2em} Set RequestTimer \\

Upon receiving \texttt{REQUEST} \\
\hspace*{2em} Retransmit each update in nackList

\textbf{Listing 4:} Reconciliation: Non-Leader

\subsection{4.4.5 Anti-Entropy Sessions}

We employ both eager and lazy “anti-entropy” sessions to overcome message loss during normal-case operation. In the eager scheme, when a server orders an update “out of order,” it attempts to reconcile the updates it is missing with the leader. If the leader has not ordered the missing updates yet, the server chooses a random node from its MemberList with which to reconcile. In the lazy scheme, a server chooses a node at random with which to reconcile, even if it may not realize it missed anything.

\section{4.5 Prepare Phase}

Once the leader is reconciled, it runs the Prepare phase of the algorithm to gather information about any update beyond its LocalAru value that \textit{may} have been ordered. As in the consensus algorithm, the leader collects responses from a majority of servers, and uses the responses to constrain the updates it may propose for each sequence number.

\subsection{4.5.1 Data Structures}

- \textit{MaxAccepted[]} - an array of proposals, indexed by sequence number, reflecting the latest proposal accepted by this server for each sequence number in the global order.
- \textit{Prepare Timer} - signals when the leader should retransmit a Prepare message.

\subsection{4.5.2 Message Types}

- \textit{Prepare} - contains the following fields:
  - \textit{serverID} - unique identifier of this server.
  - \textit{viewNum} - the view number being prepared.
localAru - local aru value of the leader.

- **Prepare__Ok** - contains the following fields:
  - `serverID` - unique identifier of this server.
  - `viewNum` - view number associated with this message.
  - `propList` - list of accepted proposals beyond Prepare.localAru.

In the original Paxos algorithm, a server can respond to a `PREPARE` message if it has not responded to one from a later view. While we could apply the same rule without violating safety, we modify the rule to match the semantics of our leader election and reconciliation mechanisms. In particular, a server may respond with a `PREPARE__OK` message if:

1. It has not moved past Prepare.viewNum
2. It is reconciled up to Prepare.localAru

The following procedure encapsulates this behavior:

```
Upon receiving PREPARE
  if Prepare.viewNum < AttemptView
    return;
  else if Prepare.viewNum = CurrentView
    Send last PREPARE__OK
  else
    ***Sync CurrentView ← Prepare.viewNum
    Construct propList from MaxAccepted[]
    Send PREPARE__OK(CurrentView, propList)
```

*Listing 5: Prepare_Handler()*

The leader resends the `PREPARE` message if it is unable to garner a majority of `PREPARE__OK` responses before the PrepareTimer expires. To guarantee the progress requirement, the leader cannot be permitted to retransmit indefinitely; otherwise, there could be a situation in which only a majority of servers are running, and all but the leader move past the current view. They will not respond to the `PREPARE` message, but will be unable to install a new view.

The leader handles a `PREPARE__OK` response as detailed in Listing 6.

```
Upon receiving PREPARE__OK
  if viewNum = CurrentView and from distinct server
    For each proposal Prop in propList
      if Prop.viewNum > Constraints[Prop.seq].viewNum
        Constraints[Prop.seq] ← Prop
```

*Listing 6: Prepare_Ok_Handler()*
4.6 Proposal Phase

The leader can begin to propose updates for ordering after it receives a majority of PREPARE_OK messages. If it receives any updates before this, it stores them in a queue. Non-leaders forwards client updates to the leader. The leader reinitiates proposals for any constrained sequence numbers, and can propose any update for unconstrained sequence numbers.

4.6.1 Data Structures

- *Constraints[]* - an array of updates, indexed by sequence number, reflecting the constraints placed on which updates can be proposed for a given sequence number.

- *UpdateQueue* - FIFO queue for incoming client updates.

4.6.2 Message Types

- **Proposal** - contains the following fields:
  - *serverID* - unique identifier of this server.
  - *viewNum* - the view number in which the proposal is being made.
  - *seq* - the sequence number for this proposal.
  - *update* - the update being proposed.

- **Accept** - contains the following fields:
  - *serverID* - unique identifier of this server.
  - *viewNum* - the view number of the associated proposal.
  - *seq* - the sequence number of the associated proposal.
  - *update* - the update being accepted.

- **Client_Update** - contains the following field:
  - *Update* - the update being initiated by the client.

The Proposal mechanism is detailed in Listing 7.

In the original Paxos algorithm, a server may accept a proposal as long as it has not responded to a PREPARE message from a later view. We bring this rule in line with our leader election algorithm by allowing a server to accept a proposal as long as it has not moved past the proposal’s view number. The accepting server stores (on stable storage) the latest proposal it has accepted for each sequence number in MaxAccepted[]. This action is detailed in Listing 8.

A server orders an update when it receives a majority of ACCEPT messages for its corresponding proposal.
Upon receiving last PREPARE_OK
   For each i > localArn where Ordered[i] ≠ ⊥
     if Constraints[i] = ⊥
       if (UpdateQueue.IsEmpty())
         Waiting_On_Request ← TRUE
         return
       else
         ToPropose ← UpdateQueue.dequeue()
       end
     end
     ToPropose ← Constraints[i].update
     Send PROPOSAL(ServerID, CurrentView, i, ToPropose)
   end
   while(!UpdateQueue.IsEmpty())
     i ← GetNextToPropose()
     Send PROPOSAL(ServerID, CurrentView, i, UpdateQueue.dequeue())
   end
   Waiting_On_Request ← TRUE

Upon receiving CLIENT_UPDATE(U)
   if (Waiting_On_Request)
     i ← GetNextToPropose()
     Send PROPOSAL(ServerID, CurrentView, i, U)
   else
     UpdateQueue.Add(U)

Listing 7: Proposal Mechanism

Upon receiving PROPOSAL
   if viewNum < AttemptView
     return;
   if viewNum > MaxAccepted[seq].viewNum
     ***Sync MaxAccepted[seq] ← Prop
     Send ACCEPT(ServerID, viewNum, seq, update) to all.

Listing 8: Proposal_Handler()
4.7 Recovery

There are three cases to consider in the context of the Paxos recovery mechanism:

1. A server that has not successfully installed any view beyond the default view will shift to the Leader_Election state.

2. A server that realizes it was the leader of the view in which it crashed must shift to the Leader_Election state, so that it will never issue two conflicting proposals in the same view. This forces the server to re-run the Prepare phase before making any new proposals.

3. A server that was not the leader in the view in which it crashed can resume activity in the Reg_NonLeader state. If progress has been made beyond this view, the server will jump to the later view, while if the server suspects the leader, it will shift to the Leader_Election state.

4.8 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.
   - Ordered - sent in response to a New_View or Request message.

2. Messages handled with a sender-based mechanism are retransmitted by the sender after a timeout period:
   - Prepare - retransmitted when the PrepareTimer expires.
   - Proposal - retransmitted if the associated update has not been ordered within a timeout period.
   - Request - retransmitted when the RequestTimer expires.

3. The leader election messages are sent periodically until progress is made:
   - View_Change - sent periodically. The leader stops retransmitting when it receives a majority of messages, while non-leaders stop retransmitting when they receive a New_View.
   - New_View - sent periodically until the view is prepared.
5 Discussion

5.1 Eventual Path Propagation

As described above, neither the leader nor the client’s local server writes an update to disk before initiating it. While such disk writes are not necessary for achieving the Agreement and Validity properties, their absence has several significant practical implications.

First, since no server keeps a record of which updates it has proposed (or even received), there seems to be no efficient way to perform reconciliation on unordered updates. This, in turn, implies that a server will only be able to initiate updates if it is a member of the majority. In particular, this means that Paxos achieves eventual path propagation only on ordered updates. Other algorithms, such as the one described in [1], allow updates to be ordered even if the initiating server never joins a majority, provided that some path to a member of the majority eventually exists.

As a result, it may be beneficial for a server to write to disk upon receiving a client update, attaching a server-based sequence number. This would allow for a vector-based reconciliation mechanism in which the servers exchange messages through the high water mark for each server.

5.2 Latency

Paxos is often described as requiring only two message delays during normal-case operation [5]. Such an analysis, however, does not accurately reflect the way in which the algorithm is used (in the client/server model) in practice. Most clients will not be local to the leader machine; this is especially true in a wide-area setting. This implies that one must also account for the latency of forwarding the update from the client’s local server to the leader. The normal-case latency is, therefore, three message delays, with two delays added during view changes.

6 Conclusion

This document provides a complete description of a Paxos-inspired replication algorithm, from a system-builders perspective. While there are several aspects that require further investigation (such as the reconciliation of unordered messages), we believe the groundwork has been laid to approach these issues from a common and consistent understanding of a fully-specified algorithm.
References


