Implementation of Multi-Paxos

Calvin Deutschbein, Matthew Shanahan, Brandon Wallace

June 6, 2014

Abstract

Our implementation of Multi-Paxos is an abstraction of the basic Paxos algorithm proposed by Leslie Lamport, designed specifically to support a leader election, which permits efficient, persistent, and consistent data stores among arbitrarily functional proposer and acceptor nodes.

1 Introduction

In order to best maximize performance on data store within the confines of the CAP theorem, Multi-Paxos offers exceptionally high degrees of consistency (in exchange for some losses in availability) and, as such, has many advantages as a networking implementation. Multi-Paxos is the extension of networking pioneer Leslie Lamports Paxos algorithm for distributed consensus that reduces the overhead for a single iteration of consensus checking from passing a minimum of four to a minimum of two messages through use of an elected leader node, consistent across all single Paxos instances, which allows consensus to proceed without the need for passing of Prepare and Promise messages (instead only Accept and Accepted messages are necessarily passed for every iteration).

The benefits of such a design are obvious, and result in a marked improvement over the original Paxos algorithm in terms of flexibility, speed, and usefulness, in a sustained environment.

2 The Algorithm

2.1 Leader Election

Multi-Paxos begins with a round of leader election, which is carried out in the same manner as a basic Paxos iteration.

First, any node may present a proposal to become leader by forwarding Prepare messages with a given proposal number that must be greater than any previously used proposal numbers. Upon reception of a Prepare message with proposal number greater than that of any previously received, an acceptor node returns a Promise message, and updates its internal state to reflect the proposal number.[1] This broadcasts the acceptor's intent to reject all proposals with numbers less than the currently promised value. The combination of Prepare and Promise may be described as the first phase of the algorithm.

When a proposal node receives Promise messages from a majority of nodes, it then broadcasts an Accept message containing the proposal number to all nodes.

Upon reception of an Accept message with valid proposal number (that is, the number to which it is currently promised), a node returns an Accepted message and updates its internal state to reflect the proposers new status as leader for future instances. The message also passes Accepted with the proposers identifier to all other nodes, such that when any node receives a majority of Accepted messages for a leader it knows that this leader has been consistently elected. The Accept and Accepted messages comprise the second and final phase of the algorithm.

While some special cases may arise, such as reception of invalid proposals, any valid leader election will proceed according to the steps outlined above. One special case of importance occurs when a node attempts to become the leader after a valid leader has already been elected, in which case Promise messages contain additional information to identify the leader. In this case, the proposing node learns of the leader for the election it (necessarily) missed. In our implementation, these "Promise" messages contain the value of the current leader rather than the proposal number which the node is promising to listen to. This will cause the recipient of such message to enter "heartbeat" mode, described later.

2.2 Data Store

Following leader elections, the system is prepared to accommodate data store. All data store requests pass through the elected leader, which then submits the the data to all other nodes using methodology similar to the second phase of the leader election, but with the addition to the Accept message of the data to be stored, and the additional requirement that received data be saved to internal state before return of an Accepted message.

Requests to read data are handled similarly, with different types of reads providing varying degrees of consistency assurance. A quick read, which involves an immediate return of a proposer's internal state, provides a lesser degree of consistency than does a slow read, which may involve a complete Paxos instance to ensure that the data returned is the most current available and that it is entirely consistent across all nodes which may be discovered similarly to the original leader election. It can accomplish this by making a proposal with no value and receiving responses from a majority of nodes.

Multi-Paxos requires the further stipulation that each separate write and slow read proceed in their own Paxos instance with their own proposal numbers, so each message must also contain an instance identifier pertaining to the particular Paxos instance in question. The identifiers must be unique to the instance but operate under no additional restrictions; however, using simple incrementation simplifies fail overs.

2.3 Inspiration

The Paxos algorithm was famously introduced in Lamport's 1998 paper *The Part-Time Parliament*[2] and later clarified in his 2001 paper *Paxos Made Simple,*[3] which more clearly presented the algorithm in a presentation after which ours is modeled. Our implementation of Multi-Paxos meets no particular specification—the concept of Multi-Paxos tends to be rather implementationspecific—although it probably most neatly follows the description of Multi-Paxos present on Wikipedia.[4]

3 The Implementation

We implemented Multi-Paxos as an extension of an existing, basic Paxos implementation with three main changes: Generalization to work with the provided ZeroMQ framework, adaptation to work with data store instances, and adaptation to work with leader elections.

3.1 Set Requests

Set requests are handled exclusively through the elected leader. If no elected leader exists, or if the leader appears to be experiencing failure, a new leadership election takes place and then the set request is handled by the newly elected leader. The leader then selects a new, unique instance identifier and passes Accept messages with the instance identifier, a starting proposal number, and the data store requested. Upon reception of this Accept message, a node replies with Accepted containing instance identifier and proposal number and then updates its internal state in line with the data store. When the leader receives majority Accepted messages, it updates its internal state in accordance with the data store request, confident that enough acceptors have agreed upon the value he asserted with the original ”Accept” message. The request is then valid and consistent.

Each successive iteration of the Paxos algorithm is considerably shorter than the first, due to the lack of need for proposal and promise messages. Because each node is aware of the leader—who can only proclaim himself to be the leader after receiving an accepted message—proposer nodes will not try to initiate proposals directly, but instead would be able to forward requests directly to the leader. In our implementation, we only attempt to initiate proposals as the leader, so that message passing among propopsers is limited.

Note that the set request results in either a consistent write (if leader exists and majority nodes are accessible or if no leader exists but leader election and store completes while majority nodes are accessible) or no write (if majority of nodes are inaccessible or if no leader exists and sufficient nodes fail to prevent a new election or the data store while running the algorithm).

3.2 Get Requests

For get requests, we elected to pursue fast read implementation. We wanted to avoid overburdening the network with an unreasonable number of Paxos instances just for the purpose of reads, and to also minimize get request latency, both of which are achieved by fast read. In addition, fast reads still have provide the assurance of eventual consistency, so clients can still take advantage of the highest degree of consistency with multiple get requests, allowing client ownership of overhead and get costs. This design choice seems to best satisfy anticipated demands. In light of this, get requests simply return whatever value is stored to the read location immediately, without consulting the network (which would require as much work as a leader election), so the client is simply issued a copy of local internal state. As a connected node will be in distributed consensus with other nodes, this will provide consistent read from connected nodes, eventually consistent read from presently disconnected nodes, and still around reads in the case of network failure, network partition, or failure of a majority of the nodes.

3.3 Heartbeats

Meanwhile, non-leader proposers follow a different pattern of behavior. While they receive updates about decisions made by acceptors at the request of the elected leader, they—along with the acceptors—send out periodic "Heartbeat" messages, indicating that they would like to know whether the elected leader is still alive and properly functional. Failing the receipt of three "Alive" messages, a non-leader proposer would initiate a new leader-election, reverting to the the "full-Paxos" scenario, beginning with a proposal and hoping for a promise which meets the proposal number. Acceptors who decide that the leader is not online will simply prepare themselves for a proposal, and interpret it as non-redundant. In this way, our system can be fault-tolerant. Should a leader fail, it should only take a few moments for one of the non-leader proposers to detect the event, and to begin electing a new leader. Given that heartbeats are sent sporadically, we had little trouble with leader re-election even in the absence of an exponential-backoff system.

Once a fallen leader returns and becomes again operational, it will need to identify that he is no longer the leader—this is accomplished by comparing his last accepted proposal number with those of the acceptors' promised values. For example, the returned ex-leader will likey try to send an "Accept" message to acceptors, containing the proposal id which he was promised before he "went down," and the instance number representing the number of prior accept messages he sent before he crashed. The acceptors will then send back a "Reject" message, indicating that there is a new leader, and the ex-leader can enter "heartbeat" mode himself.

4 Example Scripts

1. Our first example script demonstrates what is at the heart of Multi-Paxos: leader election, in the event of a failure. During otherwise normal operations, we are unconcerned with the failure of proposers who are not the leader. However, when the leader fails, there is considerable work to be done to maintain consistency and to be able to handle incoming requests. As initiated by the `after 22` block at the bottom of the following initializing script, when the elected leader (in this case, proposer1) fails, its failure is noticed soon thereafter by each of the other nodes. The first proposer to do so—necessarily , in this case—initiates a new proposal, receives a promise from the acceptors who were informed of the leader's failure, and eventually sends out an accept. This value contributes nothing to the data store, but simply helps to quickly establish a new leader.

```
start proposer1 --proposer --proposer-names proposer1,proposer2 --acceptor-names
acceptor1,acceptor2,acceptor3,acceptor4
start proposer2 --proposer --proposer-names proposer1,proposer2 --acceptor-names
```

```
acceptor1,acceptor2,acceptor3,acceptor4
start acceptor1 --acceptor
start acceptor2 --acceptor
start acceptor3 --acceptor
start acceptor4 --acceptor
after 22 {
stop proposer1
set proposer2 key 31
}
```

Assuming this proposer is successful—meaning that he receives an accepted message from a majority of acceptors (a guarantee in this situation)—he will assume the leader position. Since there is no value to commit to the data store, the first thing proposer2 does is notify other proposers (none) and acceptors (four) of his election. Then, he assumes the role of the leader, accepting requests and sending them straight to acceptors in an Accept message. Without the basic Paxos algorithm to fall back on, it would be considerably more difficult to handle leader failures. The following diagram explains how the detection of leader failure allows our Multi-Paxos implementation to continue handling requests.

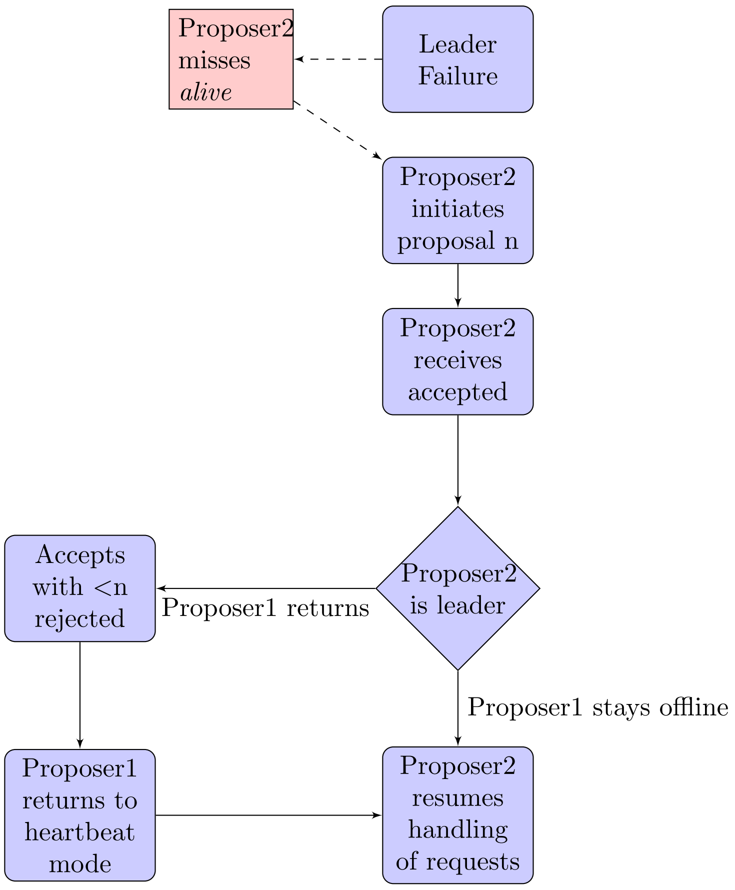
Proposer2 misses *alive*
Leader Failure
Proposer2 initiates proposal n
Proposer2 receives accepted
Proposer2 is leader
Proposer1 returns
Accepts with <n rejected
Proposer1 stays offline
Proposer1 returns to heartbeat mode
Proposer2 resumes handling of requests

Figure 1: Proposer failure

2. Our second example demonstrates how our system performs in the (unfortunate) event of a network partition. When nodes are separated from each other, they attempt to continue handling requests on their own. For one half of the partition, this is of no particular challenge: the proposer who already knows he is a leader will continue sending requests to his acceptors. Given that we take for granted a node's knowledge of the number of acceptors, we assume that he has no difficulty achieving a majority of Accepted messages, and he will continue to update his data store.

In the other half of the partition, however, a bit more work is involved. First, each acceptor will notice that proposer1 appears offline. Then, proposer2 will initiate a new leader election and will win. Finally, he will be able to send Accept requests to nodes. Once each half of the partition has added a few values to its store, we permit the second half of the network to come "online" again in the eyes of the first half. This causes a whole host of problems—the acceptors in the second half begin to receive Accept requests from proposer1, who is using an older proposal number. Therefore, these acceptors know to reject his messages and inform him that he is no longer the current leader. Then, he defers to proposer2, who will begin to handle all incoming requests. This does not, however, satisfy the problem of two, inconsistent data stores.

```
start proposer1 --proposer --proposer-names proposer1,proposer2 --acceptor-names
```

```
acceptor1,acceptor2,acceptor3,acceptor4
start proposer2 --proposer --proposer-names proposer1,proposer2 --acceptor-names
acceptor1,acceptor2,acceptor3,acceptor4
start acceptor1 --acceptor
start acceptor2 --acceptor
start acceptor3 --acceptor
start acceptor4 --acceptor
after 10 {
split half proposer2,acceptor3,acceptor4
after 20 to proposer1 {join half} }
```

For our network-wide key-value store, any overlapping keys which have conflicting values once a partition has been detected will auto-resolve to accept the value established with the lower proposal/instance number combination. In this case, therefore, the first half of the network does not have to overwrite any of its stored values, but it does lose its leader when the second half of the network comes online. The opposite is true for the second half of the network.

Admittedly, our system is not perfect in the presence of network partitions. Our implementation has difficulty handling the recovery from partitions regarding consistency. While we can begin to accept requests again quickly, we sometimes provide outdated values in the face of partitions. These values are, however, not entirely incorrect, and so do not provide an undue burden to any applications which might use our implementation, because our fast-read design choice means that we never guarantee immediate consistency with availability anyway. Our values can be seen, in the worst case, as representing permanently-pending eventual consistency.

5 Issues, Challenges, and Lessons

The biggest challenges with the implementation of Multi-Paxos are leader election and elegant, clear differentiation of Paxos instances. Fortunately, existing Paxos implementations are well suited to both leader election and incremented Paxos instances. However, leader election requires the modification of values to either be a leader to be elected or to include no value and accept proposer as leader. Naturally, as we favored lightweight and clear implementation, we simply allowed the majority selected proposer to become the elected leader. For individual Paxos instances, we differentiated by the introduction of instance identifiers to the Paxos message structure. We went back-and-forth deciding how best to handle instances, and agreed upon an explicitly clear value being passed which includes both the monotonically increasing proposal number, and the instance number which resets upon a new leader's election. This allows nodes to first screen incoming messages to forward them to the correct internal state machine (of which one necessarily existed for each Paxos instance) and then Paxos to proceed internally to the state machine as it had in our reference Paxos implementation.

At a more abstract level, it was certainly challenging designing a system with simultaneously so much freedom—in terms of implementation choices and even primary algorithms—but that needs to meet highly-specific consistency goals. It required thinking through and fully understanding basic Paxos at a very low level to allow us to begin to abstract it and make it useful in Multi-Paxos. Even so, because Lamport did not explicitly design a Multi-Paxos system in his paper, we were left to do much of the thinking ourselves yet again. This proved a worthwhile challenge, because we gained a better understanding of how groundbreaking new algorithms for consistency (like Byzantine Generals/Paxos) must have been—if only we had been handed Multi-Paxos in the same way we were handed Paxos, this implementation would have been considerably easier!

Another sticking point, for us, was the decision between fast- and slow-read. As Paxos doesnt specify a single read message, the data store specification to support simple get requests required understanding not just of the engineering of Paxos, but rather understanding of the nature of various Paxos functions, with their varying strengths, in light of the aims of our project. While we selected Paxos because of its strength in consistency, and we felt fast-read offered a reasonable degree of consistency without the punishingly slow performance incurred by initializing entire Paxos instances for each individual read. While slow reads certainly provide stronger consistency guarantees, especially in light of the discussion in class regarding the impracticality traditionally associated implementation of Paxos on systems with any reasonable amount of traffic.

As a result, we took away a few key lessons which really enhanced our understanding of the challenges with distributed systems. For one, even implementing a protocol or algorithm with a lot of specificity is difficult and requires a considerable amount of forethought and planning. Designing such an algorithm from scratch still seems, to us, to be nearly impossible. Moreover, we realized that without careful planning, it becomes impossible to freely decide on many of the choices we faced regarding our implementation. For example, the consequences of how we chose to elect a leader affected nearly every aspect of our Multi-Paxos code, and as such, we were forced to redo quite a bit after we decided to allow arbitrary leader-election (which we felt was more robust) rather than continue with our initial design of lexicographic leader precedence ordering.

Finally, through testing, we gained an appreciation for the intracacies of many of the distributed systems we take for granted all the time. Sure, we had had in-class discussions regarding how, for example, Amazon might manage its shopping cart service, but having tried to implement a very basic consistent key-value store across a small number of nodes, we came to truly appreciate how much work must be involved in implementing an Internet-scale distributed system.

References

[1] Amber on rails. *Paxos/Multi-Paxos Algorithm.* `http://amberonrails.com/paxosmulti-paxos-algorithm/`.

[2] Lamport, Leslie. *The Part-Time Parliament.* 1998. `http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf`

[3] Lamport, Leslie. *Paxos Made Simple.* 2001. `http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf`

[4] Wikipedia. *Paxos.* `http://en.wikipedia.org/wiki/Paxos_(computer_science)#Multi-Paxos`