Paxos Playground: a simulation to understand a replicated state machine implementation using Paxos

Juan I. Vimberg
Stanford

Abstract

Paxos is probably the most well-known algorithm to achieve consensus. It is extremely popular and serves as one of the building blocks of many modern systems. Yet it is regarded as hard to understand and even harder to implement. This paper makes two contributions to foster a better understanding of the algorithm:

1. It introduces a new tool to visualize Paxos execution and simulate all possible scenarios.

2. It presents a suggested learning path (to be applied on the tool of point one) that fosters intuition by presenting the final system step by step, adding one feature at a time.

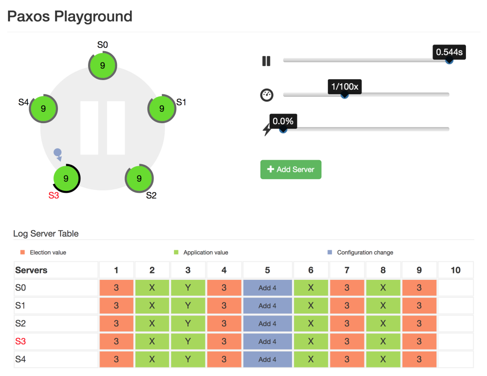
1 Introduction

Many factors contribute to earn Paxos the reputation of a hard algorithm. The original paper[1] is obscure and hard to follow. To the point where the author decided to write a new paper 3 years later named ”Paxos Made Simple”[2] aimed to explain Paxos in simpler terms. This new paper makes for a great entry point for somebody learning about Paxos for the first time. But while it defines Paxos core pretty well, several practical design aspects are left completely undefined. Things like how the system handles concurrent proposals or what technique it uses to implement master election are left to the readers imagination. This are all pieces that a software developer will need to come up with before he can implement a replicated state machine using Paxos. Designwise this flexibility is a positive thing because it allows system architects to tailor Paxos to their particular needs (and indeed over the years dozens of variants of Paxos have surfaced[6]). But at the same time it makes Paxos harder to learn, teach and implement, because in most cases the systems being built end up having little resemblance with the original theoretical description of the protocol. To help solving this issue this paper proposes a progressive approach going from a bare-bones implementation of a replicated state machine using Paxos to a more rich and optimized implementation of the same system.

Finally consensus algorithms are inherently complex because they have many moving pieces and multiple points of failure. At any point in time numerous message, may be interacting with several nodes in different states. Understanding the happy path might not be extremely hard, but considering all the possible issues that could arise with more convoluted scenarios and how the algorithm solves them, is an exercise in imagination. This is what Paxos Playground can help with.

2 Paxos Playground

Paxos Playground is a simulation of a replicated state machine, implemented using the Paxos algorithm, written in Javascript. The UI is heavily based on Raft Scope[4] created by Diego Ongaro to visualize and explain Raft[5] (which in time was created as a simple alternative to Paxos).



Log Server Table

🟥 Election value 🟩 Application value 🟦 Configuration change

Servers	1	2	3	4	5	6	7	8	9	10
S0	3	X	Y	3	Add 4	X	3	X	3	
S1	3	X	Y	3	Add 4	X	3	X	3	
S2	3	X	Y	3	Add 4	X	3	X	3	
S3	3	X	Y	3	Add 4	X	3	X	3	
S4	3	X	Y	3	Add 4	X	3	X	3	

Figure 1: Screen-shot of Paxos Playground

2.1 The functionality

Paxos Playground lets the user choose between different configurations of the same system. Going from the bare essentials to a complete and optimized system. The available configurations are: Simple Strategy, Sync Strategy, Master Strategy, Master Optimized Strategy, Master Optimized + Configuration Strategy. Each strategy is further detailed in section 4. For now it suffices to know that each strategy adds a particular functionality or optimization to the previous one.

In each configuration the user is able to perform the following actions:

- For any server:
  - Make a new request
  - Inspect the state of a node (see whether it has promised a particular proposal or accepted a given value, whether it has a master, etc.)
  - Stop/Start
  - Check the contents of the log
- For any message:
  - Inspect the contents of a message (including Paxos instance number, proposal id, value, etc.)
  - Drop the message
- On the whole simulation:
  - Pause/Play
  - Control the speed of the simulation
  - Control the amount of noise in the network to see how the system reacts to randomly dropped messages
  - Rewind/Replay the simulation

With this controls a creative user could simulate most scenarios a real system running a state machine using Paxos could face, and understand how the system would react.

2.2 The code

2.2.1 The back-end

This module comprises the replicated state machine implementation. It is responsible for executing paxos per se as well as modelling the interaction and messages between the multiple nodes.

The system was modelled following an Object Oriented paradigm. A node is composed of its different roles (as described in ”Paxos made simple”[2]) each role encapsulating its own business logic. Each new configuration is implemented as a Mixin on top of the Node class, so that the code for more advanced configurations wrap and expand the code for previous configurations [1]. Hopefully this design decisions make the code easy to read and understand too.

The communication layer is abstracted so that message handling can be implemented using whatever technology the developer prefers. For example in our particular case, the front-end code simulates remote calls by delaying the delivery of messages (allowing it to render in-flight messages) whereas the tests implement an immediate delivery policy. The beauty of this approach is that a developer could very well implement the communication through RCP and use the back-end code for on a real system.

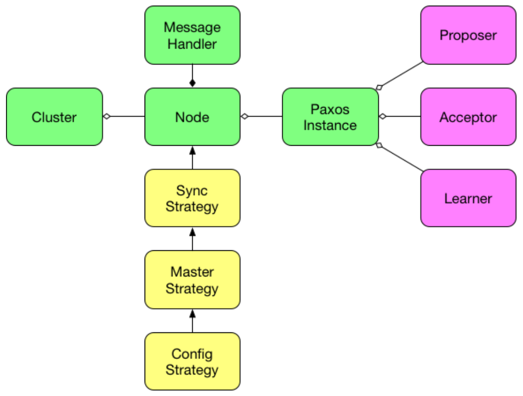


Figure 2: Back-end code design

2.2.2 The front-end

This module is responsible for rendering the state of the system, and controlling the time, speed and network noise of the simulation. It uses Javascript, JQuery, Twitter Bootstrap and a fair amount of SVG manipulation to render and animate the state of all nodes and messages flowing through the system. It was adapted from repository Illedran/raftscope[3] which in time was a fork from the original RaftScope repository[4].

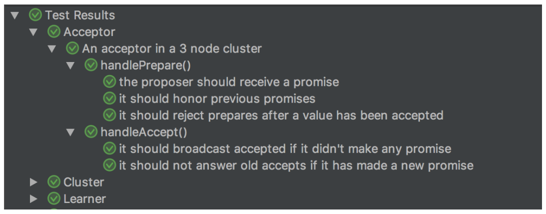
2.2.3 The tests

Last but not least there is a module for tests. I believe it is worth mentioning it because tests might prove useful not only to a developer trying to expand the system, but also to one trying to use the simulation.

[1]The only exception being master and master-optimized configurations which are the same configuration initialized differently

Two different types of tests were written: unit tests and functional tests. Unit tests were written using Mocha in a Behavior-driven development (BDD) fashion. The nice thing about BDD if that a test execution provides a human readable output that could be understood as conditions that the system needs to meet (as shown in *Figure 3*).

On the other hand we have functional tests which where written using Cypress. This tests are just an automated run of several scenarios through the same UI a regular user would use. It might prove useful because it explores (and validates) some of the more contrived scenarios proposed in the next section.



```
▼ ✓ Test Results
  ▼ ✓ Acceptor
    ▼ ✓ An acceptor in a 3 node cluster
      ▼ ✓ handlePrepare()
          ✓ the proposer should receive a promise
          ✓ it should honor previous promises
          ✓ it should reject prepares after a value has been accepted
      ▼ ✓ handleAccept()
          ✓ it should broadcast accepted if it didn't make any promise
          ✓ it should not answer old accepts if it has made a new promise
  ▶ ✓ Cluster
  ▶ ✓ Learner
```

Figure 3: Screen-shot of unit tests

3 A path to comprehension

There is a plethora of implementations of a replicated state machine using Paxos. For the present work I chose to go from the most basic system to a more complex configuration.

3.1 The core

At the core of it all we have the Paxos consensus algorithm (sometimes called ”Basic Paxos”). For brevity sake I am not going to explain Paxos. It is enough to know that each instance of the protocols goes through two phases of messages to decide on a single output value as illustrated in Figure 4.

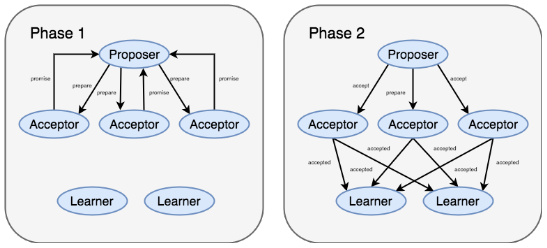


Figure 4: The Paxos protocol

3.2 The simplest replicated state machine

The initial configuration consists of chaining together multiple instances of Paxos. Each new instance decides on the next value the system should accept as part of the log. The log for each node is represented by the table at the bottom of the screen (see Figure 1).

A new user should start by issuing requests to build a mental model on how the two phases of the algorithm works. The suggested method is to stop the simulation on each new phase to inspect the state of the servers as well as the content of the messages.

Once the user has a good grasp on how the system works he can start experimenting with the failure scenarios. First he should try dropping one or two messages from each phase. Since the initial configuration consists of a four node cluster this should be safe and the system should still reach consensus. He could also try stopping one or two servers and the result should still be the same.

Then he could move to a more advanced scenario like having dueling proposers trying to come up with a new proposal. In this scenario two or more nodes try to propose a new value but by the time the first node sends the Accept message (Phase two) another node has gone through Phase one and has convinced a quorum of machines to only accept messages with a Proposal Id higher than the one of Node one messages.

The user will soon notice that a node that might not able to advance to the next Paxos instance, either because it was stopped or it did not get the required messages to achieve resolution in a previous instance. In the present configuration a node left behind is not able to recover and thus the system is render unusable after half of nodes are in this state. The next configuration solves this problem.

3.3 Adding sync

This configuration consists of adding sync messages to recover from the above-mentioned scenario. For clarity sake it was decided to introduce a new set of messages to handle synchronization (namely SyncRequest and CatchUp) instead of trying to leverage the messages already in use for Paxos.

At fixed intervals each node will send a SyncRequest to another random node from the cluster (it might even send it to a node that is down because nodes don’t keep track of their peers state). The SyncRequest message includes the latest Paxos instance number seen by the node sending the message. A node receiving a SyncRequest will check if it knows of a higher instance number and if so it will send a CatchUp message including the highest instance number seen and all the missing log values. In the following sections we’ll see how the message content needs to be expanded to include extra information that needs to be synchronized.

3.4 Adding masters and optimizations

The next two strategies are ”Master” and ”Master Optimized”. The motivation for this strategies is to make certain optimizations to the protocol by selecting a node as Master. The optimizations are the following:

1. **Only allow the master to make requests.** Avoid the dueling proposers scenario by only allowing the master to make proposals.
2. **Use the master as distinguished learner.** Instead of having all learners broadcast the new value to everybody, the master is selected as distinguished learner. Nodes send Accepted messages only to the master and once resolution is achieved the master broadcast the result to all nodes for durability. Effectively reducing the amount of messages needed to learn the new value.
3. **Skip phase one.** Skip phase one for proposals from the master node cutting down the number of messages required to achieve consensus. This is achieved by synthesizing the Prepare and Promise messages on each node during the creation of a new Paxos instance. This works because by having a master a node can anticipate who is it going to make the next proposal.

The master election is implemented through leases leveraging the Paxos state machine. Any node can propose a vote request (which is a special kind of log entry) the same way he would propose any other value. When nodes reach consensus a new leader gets elected and every node advances to the next paxos instance. Nodes promise to only answer messages from the current master for the duration of the lease.

The master lease timer is started by the time he sends the vote proposal whereas other nodes only start it by the time the node learns about the new master. This guarantees that, in normal operation, the master's timer is going to be the first one to end enabling him to renew the lease using the optimized protocol instead of having to resort to a full protocol election.

The benefit of using the log for master election is that we ease understanding by reusing a familiar concept. But this comes at the cost of incurring in possible delays. For example in a worst case scenario node 0 could propose himself as leader and then die before completing phase two of the election. In this case a new node will have to drive the proposal forward and vote for node 0 even when by the time of resolution node 0 might be down.

Upon restart a new node will try to propose himself as leader. If the rest of the nodes already have a leader and a lease promising not to answer to any other node his request will simply be ignored. If by the time the current leader renews the lease the resuming node is up to date he'll be able to respond correctly and recognize the leader.

In this configuration it might prove insightful to play around with scenarios in which the master node fails at different stages of the protocol.

Another interesting aspect to analyze is how user requests interfere with election proposals. Consider what is the worst case scenario and how the system recovers from it and returns to its steady state.

3.5 Considering configuration changes

The next and final strategy takes into account cluster configuration changes, that is how to add or remove nodes from the system. Once more we rely on Paxos to achieve consensus on the proposed change. Just like we did with master election a new type of log entry is added for configuration changes. The protocol to agree on such configuration change is just the same as for any other message (including optimizations when a master is elected). The only difference is that upon resolution the new operation takes place, either by adding a new node to the cluster or by removing an existing one.

Removing a node is straightforward, any node could be removed at any given time and providing all the other nodes are healthy and up to date the system should still be fully functional. The only exception is removing a master node. In which case nodes will trigger a new election after learning the master is gone.

Adding a new node on the other hand is a little more involved. For starters the new node starts up a clean slate. Meaning it is on Paxos instance zero and has its log empty. The node will have to wait for an appropriate *CatchUp* message to get up to speed with the rest of the cluster. Afterwards the process is exactly the same as for a node being resumed in the ”master-optimized” strategy. In this strategy the CatchUp message needs to be expanded to include information about the current cluster configuration. Using the log for cluster configuration changes ensures us that if a node has seen the latest entry from the log the he knows which is the latest cluster configuration.

4 Future work

Some improvements and optimizations were purposely left out of the implementation to make the system more easy to understand and reason about. For example when using the ”master-optimized” strategy consensus is reached in a single round trip. Yet another message is sent to all nodes after consensus is reached have them learn the value. This message could be piggybacked on future requests (as proposed by Lamport at "Paxos Made Simple"[1]). But doing that would require a new property on the Accept message to store this information, and giving it a new responsibility that it did not have on previous configurations.

There are certain areas where the simulation could be expanded and improved. It would be interesting for example to provide UI controls to simulate network partitions just by drawing lines between the nodes and preventing any message from crossing the line.

Another useful improvement would be to mark why and how a node handles a message. Right now such logging is happening on the browser console but is hidden to the end user. Such information has proven essential for writing the code and I think users can benefit from such insight too.

Finally an easy addition would be to let the user configure each node to decide which roles the node should play in the simulation (proposer, acceptor and/or learner). It should be a straightforward improvement because the back-end is already written so that each node receives a set of roles during creation. This would help a user understand how the different roles interact on the system and measure how the amount of nodes playing particular roles impact the system.

The simulation is hosted on Github pages at https://jivimberg.github.io/paxos-playground/src/main/html/. The source code is publicly available under the ISC license at: https://github.com/jivimberg/paxos-playground. Pull requests and contributions are very much welcome.

References

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

[2] Lamport, L., et al. Paxos made simple. *ACM Sigact News 32*, 4 (2001), 18–25.

[3] Nardelli, A. Raftscope fork. https://github.com/Illedran/raftscope. Accessed: 2017-12-12.

[4] Ongaro, D. Raftscope. https://github.com/ongardie/raftscope. Accessed: 2017-12-12.

[5] Ongaro, D., and Ousterhout, J. K. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference* (2014), pp. 305–319.

[6] Van Renesse, R., and Altinbuken, D. Paxos made moderately complex. *ACM Computing Surveys (CSUR) 47*, 3 (2015), 42.