The Paxos Algorithm, When Visualized, is Actually Pretty Simple

Jackson Lallas and Siddhartha Prakash; https://github.com/sidscrazy/Paxos

1 INTRODUCTION

The Paxos Algorithm has a reputation for being difficult to understand. Perhaps, as Lamport noted, this is due to its initial presentation which included an island metaphor and copious amounts of Greek. It is one of the few algorithms in computer science with the distinction of inspiring work to create a new protocol not to improve on the speed or formal properties of the predecessor but just to be simpler.

Our project aims to bring clarity to the Paxos algorithm by creating a visualization tool that can demystify Paxos and make it easier to understand. The first step was to create a Paxos implementation using multiple processes as nodes on a single machine that could feed in data to a visualization scheme. Our Paxos implementation creates a log of relevant messages which are then ordered and represented visually.

The rest of this paper is ordered as follows: Section 2 briefly discusses the background on Paxos and then details our implementation of multiprocess Paxos and a visual tool. Section 3 shows the results of performance tests on our Paxos implementation. Section 4 details some challenges and limitations that our implementation ran into, and what we would do to extend the project further. Finally section 5 presents the conclusion.

2 PAXOS AND OUR IMPLEMENTATION

Paxos is a protocol for distributed systems to achieve consensus among a network of unreliable and connected nodes on some value. Consensus is a crucial problem in distributed systems because a set of machines performing as a group have to agree on final decision to ensure integrity of the data being processed across different machines. In Paxos each machine has some combination of three roles: proposer, acceptor, and learner. The proposers handle initiating a proposed value as the consensus value that the group together has to agree on, acceptors vote on whether or not to approve proposed values, and learners track the results of the vote to determine whether consensus has been reached. A common optimization is to make all proposers as learners too so that votes go directly back to where they came from and communication is simplified, which we chose to do in this project. For a detailed explanation of the protocol, we strongly recommend reading the original paper by Lamport: "Paxos Made Simple." [1].

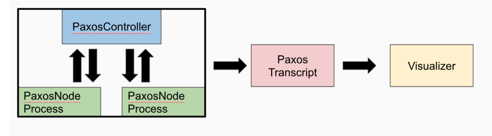

Figure 1: The visualization pipeline. A Paxos simulation produces a live transcript which is given as input to the Visualizer.

Since our goal is to visualize Paxos we created an implementation for consensus among multiple processes instead of multiple machines, such that anyone could run our simulation. We implemented Paxos in c++ with about 800 lines of code. The two core classes to our simulation are PaxosController and PaxosNode. PaxosNode simulates a single machine running Paxos. It includes all information for the state of Paxos on that machine and also simulates crashes with some random, tunable probability.

The PaxosController is used to set up the simulation and initialize node state. It also acts as a virtual network switch between the nodes which greatly simplifies interprocess communication. Without the virtual switch at the PaxosController, nodes would need to establish sockets with all other nodes after creation. We found that the network switch was a reasonable approximation of how communication would actually work in a datacenter running Paxos and greatly simplified messaging between processes.

Since the PaxosController doubles as a virtual switch, it has a global view of all messages sent during a run of single round Paxos. Each message is timestamped before it is sent to the controller from a node. These timestamped messages are logged so that the visualizer can have a transcript of the entire Paxos round and use timestamps for ordering between different events.

Finally, we implemented a PaxosVisualization scheme in python that takes the CSV log file produced by the PaxosController and creates a table-like visualization of the current Paxos run. Each row in the table represents a single message and has the receiver marked. This allows users of the visual tool to track communication between different nodes and see edge cases in practice. We also log crash and resume from crash events so that the user can visualize Paxos under worst case conditions. The visualizer can run at the same time as our Paxos implementation, allowing the user

Node: 0	Node: 2	Node: 4	Node: 3	Node: 1
Prepare: 1.0	Receiver			
Prepare: 1.0		Receiver		
Prepare: 1.0			Receiver	
Receiver		Prepare_Ack: 1.0		
Prepare: 1.0				Receiver
Receiver				Prepare_Ack: 1.0
Receiver			Prepare_Ack: 1.0	
Propose: 1.0, 73.0	Receiver			
Propose: 1.0, 73.0		Receiver		
Propose: 1.0, 73.0			Receiver	
Receiver			Propose_Ack: 1.0, 73.0	
Receiver		Propose_Ack: 1.0, 73.0		
Propose: 1.0, 73.0				Receiver
Consensus: 1.0, 73.0	Receiver			
Consensus: 1.0, 73.0		Receiver		
Consensus: 1.0, 73.0			Receiver	
Receiver	Prepare_Ack: 1.0			
Receiver	Propose_Ack: 1.0, 73.0			
Consensus: 1.0, 73.0				Receiver

Figure 2: Sample Visualization Output for 5 nodes and 1 proposer.

to get a live view of the algorithm. At first we considered running the visualizer after consensus had been reached, but we found that under high failure conditions consensus would take minutes, if occurring at all, so the live simulation was implemented instead.

3 PERFORMANCE ANALYSIS

We tested our Paxos implementation by scaling up the number of nodes and number of proposers and recording the average time to reach consensus in each different environment. These tests were performed on the Stanford myth machines. To see how well our implementation scaled with additional nodes under a single proposer, we tracked average time to consensus for simulations with 5, 10, 15, ..., 40, 45, and 50 nodes. In our proposer tests we kept the number of nodes static to be 15 and record how well our implementation performed with 1-8 proposers.

Furthermore, we ran each of these tests in a low failure and high failure environment. In the low failure environment there was a 5% probability of a failure, and should a failure occur there was a 98% probability that failure would be a network one. The high failure environment had a 20% probability of failure events with the same distribution between network and machine failures. Proposers have a chance to randomly fail every time their state updates. Nodes that are only acceptors check whether or not to fail every time they finish processing a message.

These performance numbers are fairly reasonable in the low failure case, since a single machine is handling all the nodes and routing. Under stable conditions, our simulation is robust to node and proposer increase as no results took longer than 6 seconds to achieve consensus. When failures are common and the system is under high stress, surprisingly we found the introduction of more proposers to be much worse for performance than increasing the number of nodes. We expected performance to generally degrade as the number of nodes or proposers increased, since there would be more opportunities for failure and more communication needed to achieve consensus. These graphs mainly corroborate that expectation, but with some peculiar differences.

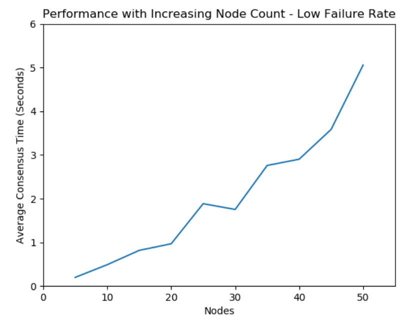

Figure 3: Node Scaling with 5% failure probability.

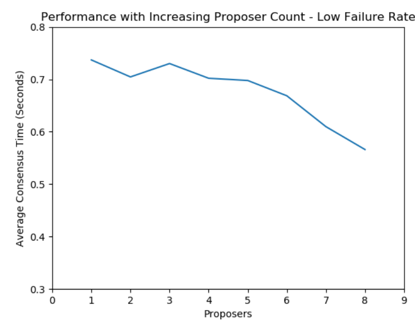

Figure 4: Proposer Scaling with 5% failure probability.

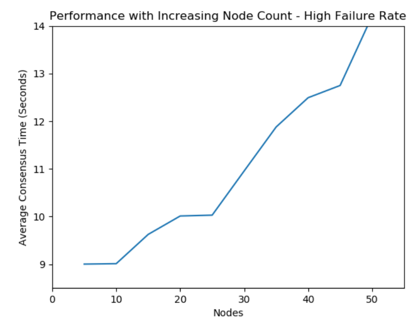

Figure 5: Node Scaling with 20% failure probability.

Figure 4 presents an extremely counter-intuitive result that more proposers can increase performance in low failure environments. Lamport's paper warns against the opposite effect, where multiple proposers can keep interfering with each other and the algorithm never converges. We believe that this effect is a result of how we handle proposers and may not generalize to other Paxos implementations. In our simulation, if proposer A receives a proposal from some other proposer B, A will behave identically to an acceptor for the lifetime of B's proposal. Since many processes are racing

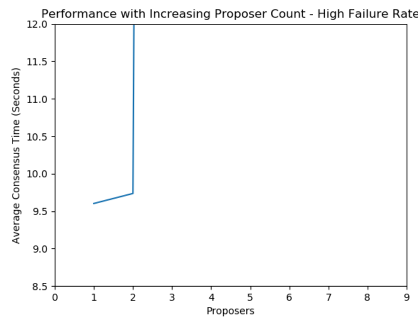

Figure 6: Proposer Scaling with 20% failure probability.

to be the first to get a proposal out and proposers rarely fail, the downward trend makes sense.

Figure 6 demonstrates that environments with many proposers are especially vulnerable to high rates of failure. When there are 3 or more proposers, our system did not reach consensus consistently. While this is consistent with Lamport's warning against many proposers, it is strange that we get a performance boost in the low failure case. We believe that this should be further investigated in other implementations of Paxos, as it is possible that the severe degradation under stress or the performance boost with reliable systems are artifacts of our failure simulation design.

4 CHALLENGES AND FUTURE WORK

Simulating node failure was difficult to design. Our first attempt had all nodes regardless of role attempt to crash after processing a message, and for proposers to attempt to crash after sending prepare or propose messages as well. This seems like a good idea in theory, since maybe work being done by a node is correlated with the chance to crash in production environments. However, as we scaled the number of nodes up we found that our simulations were exploding in length. Since proposers process significantly more messages in the form of acks to prepare and propose messages, as the number of nodes go up the proposer failure probability goes up dramatically. With n nodes and p probability to fail, there is a minimum a 1 - (1 - p)^n probability of failure when trying to achieve consensus, as the proposer needs at least half of the nodes to give a positive acknowledgement for the prepare and propose phases. Concretely with n = 25 and p = .2, there is over a 99% chance for the proposer to fail when trying to reach consensus. As a result, we modified our implementation so that proposers only crash right after sending a prepare or propose message.

Our current implementation is not very optimized. On a simulated machine failure the proposer will lose all state pertaining to prepare acks and current votes that it has received, forcing it to start over at the next round when it comes back online. We also do not have a way to check that a vote has failed and then immediately try again in a new round. For instance, if a proposer receives over n/2 acks that reject a prepare message that proposer should just try again incrementing their round number. Our current implementation only moves on to the next round after a timeout has occurred.

Time permitting, we would also have liked to make a dynamic visualization similar to RaftScope. Though we think our visual tool is an effective way to produce various Paxos transcripts under different conditions, a view of the nodes themselves and packets traversing a network would be a very effective visual aide. Thankfully, our current implementation has all the infrastructure necessary to be plugged into a visualization library to produce such a visual, and that would be the first step if the project were to be extended.

Finally, we wanted to design much more expressive test cases. In our initial design it would have been possible for the user to specify events in the simulation on the granularity of "Proposer 1 will fail after receiving the third prepare - ack." Our simulation in contrast is completely non-deterministic. Failure parameters can be tuned to make some transcripts more likely than others, but the user currently cannot enter some constraint that the simulation must meet.

5 CONCLUSION

It turns out that Paxos is not so bad after all. This project helped us get a much deeper understanding of the Paxos algorithm and hopefully the visualization tool we created can be a first step to making the algorithm more accessible at large. Most of the bugs we encountered from this project were not rooted in the operation of Paxos, but instead classic problems of interprocess communication and distributed systems: managing locks, packet switching, race conditions between various processes, and the like. We believe that the future of Paxos will broadly be convergence on common tenets of implementing the algorithm and understanding how it operates, of which we hope that our project will play some small part in.

REFERENCES

[1] L. Lamport. Paxos made simple. 2001.