Paxos Made Experimentally

Mark Zbikowski
Vince Zanella
Ryan McElroy
University of Washington

1. Introduction

One of the major trends in systems design is moving from the client-server network paradigm to so-called 'cloud computing'. Rather than having a small allotment of extremely expensive, high performance servers, it is often much more efficient to host a service on a large cluster of commodity computers. As long as the applications running on this type of system are scalable, the clusters can be relatively cheaply grown to enormous clusters of tens of thousands of compute nodes. For a system like this to work, however, not only must the application be scalable, but the system software that provides the infrastructure must provide reliability and consistency guarantees, as well as be scalable itself.

To build a system that meets the above description, several key components are required. One of them is a sub-service that can be used by several replicated machines to reach consensus on the current state. As one can imagine, when the system hardware is a large cluster of inexpensive machines connected over an unreliable network, failures are guaranteed to occur. Paxos is a distributed protocol that enforces a consensus to be reached by the participating machines before a new state is agreed upon. It is well-described in several papers, and our implementation borrows from the designs presented in them[1][2][3][4][5]. It allows new values to be proposed, and these values will only be accepted if a majority of the participants agree on the value. This allows progress to be made in the presence of n/2 machine failures in a group of n machines. This service can be applied to a variety of applications. It can be used to keep replicated databases or distributed hash tables consistent, in holding new master elections, or in Google's case to grant course-grained locks in distributed applications[1]. In this paper we describe our implementation of Paxos using a language known as Repy, which is a variant of python. For testing and evaluation we also implemented a distributed hash table that utilizes Paxos to keep its replicated partitions consistent. We evaluated the correctness of our system by adding and deleting entries from the hash table while machines in the distributed system were taken offline and verifying liveness was maintained. We evaluated performance and scalability by adding clients to the system to increase contention and by adding more nodes to the set of machines running Paxos.

2. Implementation

The Paxos protocol was implemented in a variant of the Python programming language called Repy. The language is a subset of Python that runs in a sandboxed execution environment for protection of the host. This allows programs written in Repy to be deployed on the research testbed called Seattle. The Seattle testbed is a cooperative network of donated compute resources, where donating resources from a machine allows one to acquire resources on other machines in the network. We had originally planned to deploy our Paxos and DHT implementation on this network, but unfortunately it was experiencing technical problems and it was impossible to acquire a stable set of resources. Instead, testing was done on workstations in the Computer Science labs at the University of Washington.

The Paxos algorithm has three different types of members: proposers, acceptors, and learners. In our implementation, each machine that is a Paxos node has one instance of each of the three members. A distinguished proposer is elected, and all client requests to propose a new value are directed to this distinguished proposer. The distinguished proposer maintains authority through heartbeat messages; a new distinguished proposer is elected if a machine failure or network partition is discovered due to a loss of heartbeats. When a Paxos node does not receive a distinguished proposer heartbeat message in a timeout period, it issues a new decree to propose itself as the new distinguished proposer. Most Paxos nodes will come to this realization at approximately the same time and will all issue proposals to become the distinguished proposer themselves. The Paxos algorithm handles the multiple proposals, and depending on the random, unique proposal numbers, one of the machines will have their proposal accepted and learned and will become the new distinguished proposer. All clients using the Paxos instance will then have their proposal requests directed to this machine.

For handling messages, a message processor was developed that allows message receipt and processing to be fully asynchronous. An early implementation had a synchronous message queue that was drained every 10ms. The message processor eliminated the need for a queue by allowing each type of Paxos member to register callbacks with the message processor, and any time a message was received, the processor would pass it to all of the callback routines registered for that message type.

To ensure robustness to machine failures, acceptor and learner members maintain a log to use during restart and recovery. Acceptors log all of the promises they have made and all of the values they have accepted. This prevents an acceptor from accepting a proposal number for a given decree that is less than some proposal number it accepted before it failed. Learners use their log to repopulate their state history so that upon recovery they can answer queries about proposals they learned before they crashed.

The external API presented to the client allows them to request new proposals of arbitrary values, to query learned values, and to use an RSVP system to receive notifications when their proposed values are learned. The RSVP system registers client-provided routines to be dispatched when their requested proposal is learned by the system.

For monitoring and debugging purposes, we implemented an event-logging tool. Each Paxos machine hosts a web server, and all Paxos events are logged to the server. This allows easy tracking of events when the system is deployed over the Seattle testbed. Any of the Paxos nodes' logs can be seen by visiting the web site the node is hosting.

To test our Paxos implementation in a real-world application that would utilize it, we built a distributed hash table. The DHT has a top-level that exposes an interface to its clients and then has multiple partitions, each storing a subset of the keys into the hash table. The top level of the DHT passes (key, value) insert requests and key delete requests to the appropriate partition. For reliability, both the top level and the partitions of the hash table are replicated. To maintain consistency across the replicas, Paxos is used to ensure each replica executes the same set of insert and delete operations. This guarantees that when any operation requested by the client is accepted by a quorum of the Paxos nodes, each replica of a partition of the hash table will be consistent with the other replicas. It is possible

that a client's requested operation will fail to reach consensus among the Paxos nodes, but it is not possible for a key to be added or deleted in one replica of a hash table partition but not in another one of the replicas. As long as n/2+1 machines of the Paxos instance remain functional and connected in the network, consensus can be reached and progress can be made in servicing client requests.

3. Results

The graphs in figures 1-9 depict the performance characteristics of the distributed hash table under different configurations and loads.

The two variables are the number of nodes participating in the system and the number of clients pushing new values into the distributed hash table. In each setup, the hash table had five partitions replicated across the nodes. Nodes consisted of UW CSE undergraduate lab computers (hostnames colin, romieu, aliakc, stelian, amlia, maxk, jgarzik, hch, garloff, ajk, tori, ralf, mhw, philb, and simon). The system was programmed in the Repy language, a Python derivative that is part of the Seattle distributed systems testing environment.

The first three graphs show performance with three nodes in the distributed hash table responding to one, two, and four clients each sequentially pushing 100 values into the hash table (ie, in the four client case, each of the four clients were running simultaneously, and each client sequentially set values in the hash table). The next three graphs show the system performance set up with seven nodes and one, two, and four clients. The final three graphs show the performance of the hash table when 15 nodes are participating, again with one, two, and four clients pushing new values into the hash table.

In each of the graphs, the x-axis denotes the trial number (out of 100) and the y-axis denotes the time from the proposal of the value to the acceptance of that value. Fewer than 100 trials denotes lost set commands; these may have failed for a variety of reasons, including temporary network failures leading to lost requests or RSVP packets, timeouts, and so on.

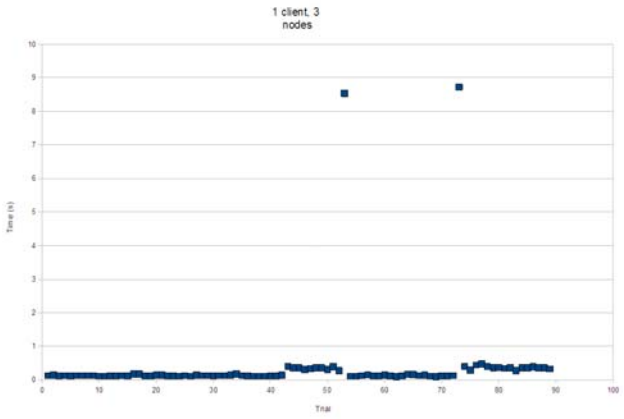

Figure 1: 100 DHT operations with 1 client and 3 Paxos nodes

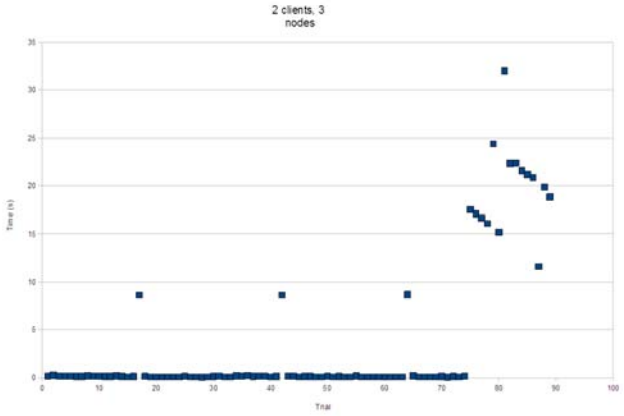

Figure 2: 100 DHT operations with 2 clients and 3 Paxos nodes

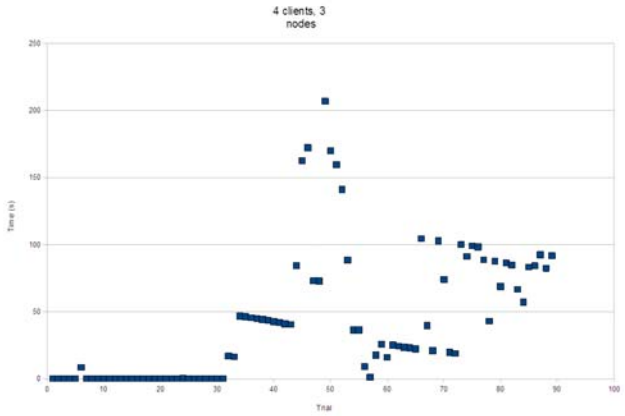

Figure 3: 100 DHT operations with 4 clients and 3 Paxos nodes

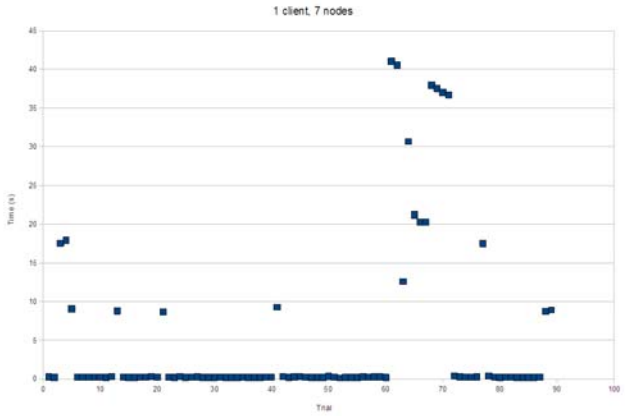

Figure 4: 100 DHT operations with 1 client and 7 Paxos nodes

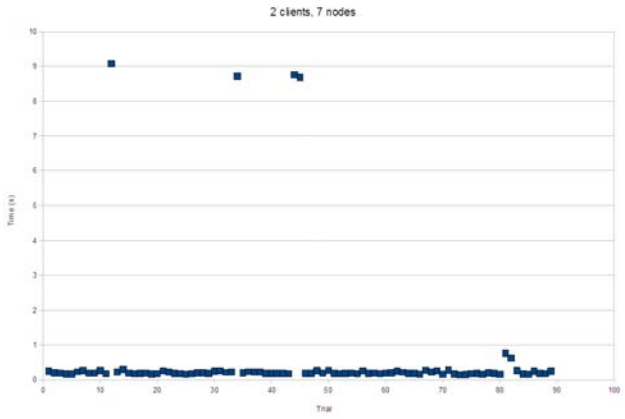

Figure 5: 100 DHT operations with 2 clients and 7 Paxos nodes

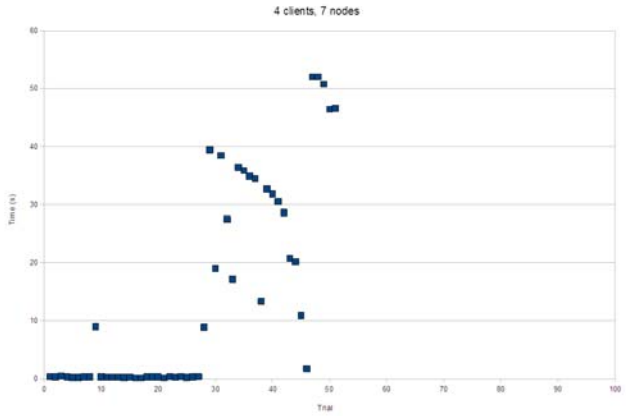

Figure 6: 100 DHT operations with 4 clients and 7 Paxos nodes

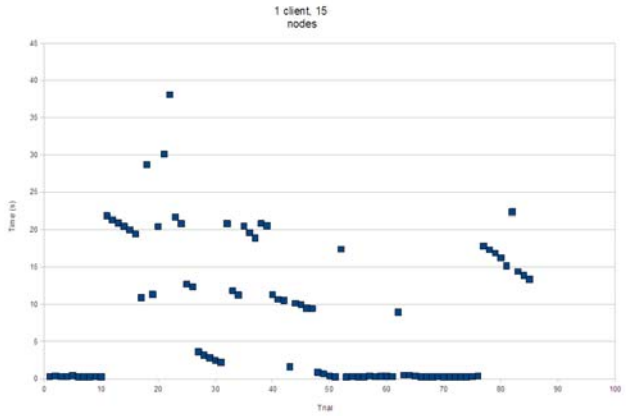

Figure 7: 100 DHT operations with 1 client and 15 Paxos nodes

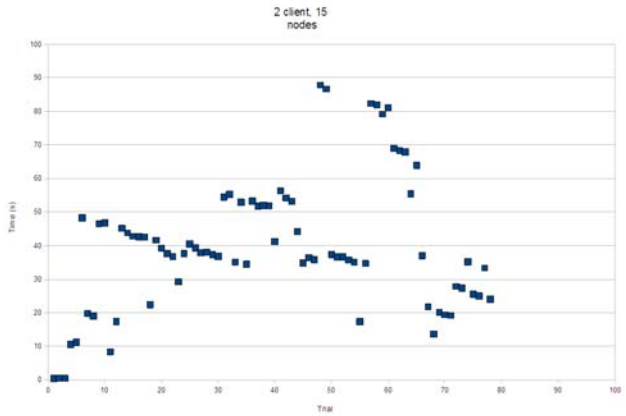

Figure 8: 100 DHT operations with 2 clients and 15 Paxos nodes

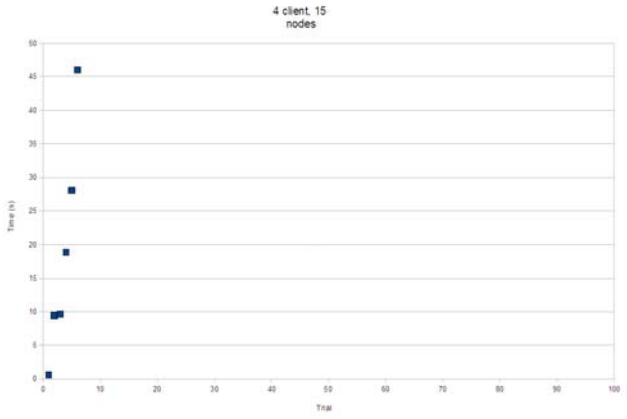

Figure 9: 100 DHT operations with 4 clients and 15 Paxos nodes

In general, the more nodes and the more clients, the higher the average latency, and the more high latency requests there are. In the highly resource-limited environment Seattle provides (even with modified restriction files), this is not entirely unexpected. The one exception to this trend is the two client, seven node case, where performance is better than with a single client.

4. Conclusion

This project involved synthesizing and implementing many of the concepts studied in CSE 551, including system design, concurrency, parallelism, networking, scalability, and distributed system design. It was a great exercise in exploring the complexities of developing a distributed protocol running on faulty machines running over an unreliable network. Furthermore, measuring the performance of the system to determine its response to different system setups and loads forced us to understand the limits of the system we implemented, and provides a solid direction for future improvement. While our experimental results didn't show our implementation to be highly scalable, the Paxos protocol isn't inherently scalable. Our implementation requires on the order of n^2 messages per decree, and the breakdown becomes clear when scaled up to 15 machines.

References

[1] Mike Burrows "The Chubby lock service for loosely-coupled distributed systems" OSDI, November, 2006.

[2] Chandra, Griesemer, Redstone, "Paxos Made Live – An Engineering Perspective". ACM PODC 2007.

[3] Lamport, "Paxos Made Simple". ACM SIGACT News, 2001.

[4] Mazieres, "Paxos Made Practical".

[5] onathan Kirsch and Yair Amir, "Paxos for System Builders", Jonathan Kirsch and Yair Amir, 2008