Using Paxos For Distributed Agreement

Jacob Torrey
Cyrus Katrak

December 12, 2008

Abstract

As trends in application development shift away from a classically centralized approach and towards a massively distributed paradigm, the need for efficiently coordinating agreement between distributed application components becomes clear.

This paper will provide a historical and technical overview of the Paxos algorithm, which provides a mechanism for achieving consensus between processors. Paxos provides basic failure guarantees, and optionally, Byzantine failure guarantees. We will briefly cover previous works that have shown Paxos is correct in its ability to maintain its guarantees, and is optimal in achieving consensus in a minimal number of message transmissions. We will conclude by noting that as a result of this optimality, Paxos has been deployed in a variety of large scale, mission critical distributed systems, and remains the protocol of choice for developers wishing to coordinate distributed state.

1 Introduction

Many modern day computing services are backed by a single, centralized, server that handles all application requests and logic. However, as the number of Internet users grows, and software applications become more complex, the challenge of serving exponentially increasing loads from a single system becomes unsurmountable. Similar to the issue of scale, high availability has become of great concern to many application developers. In many scenarios it is unacceptable for their to exist a single point of failure for mission critical services.

Moving from a centralized to distributed framework is in most cases a non trivial process. Some of the greatest challenges arise when attempting to coordinate agreement of distributed state when messages could be lost corrupted or from colluding processors. Consider the problem of master election in a cluster of computers: nodes in the system need a leader to help make decisions effectively, but picking a master from all the nodes is a difficult task to make sure that everyone is in agreement. With Paxos, once a value has been picked, there is no doubt in the system who was chosen by the majority.

2 History

Before Paxos there was some talk of Byzantine Generals' problem which described a group of generals of many different factions that are trying to coordinate an attack on an enemy using messengers to relay their intentions. However, their is a mutual distrust amoung the generals and the messengers are to be considered unreliable. This problem, proposed by Lamport, Shostak and Pease has been fundemental to success of large scale distributed computation on commodity hardware.

The Paxos (named after the Greek island) algorithm made its debut in 1998 in a paper (submitted in 1990) published in the ACM Transactions on Computer Science 16 titled "The Part-Time Parliament" by Leslie Lamport received very little response when it was initially written. The paper is structured in such a humorous way that almost no one saw that it contained a very well designed algorithm, except for Butler Lampson, who immediately saw its value and gave a number of talks on the subject. It took a number of years before the results from this research became applicable to real world computing. Once it was 'discovered' as a real algorithm, it took only three years until a modification was found to make Paxos safe to use with the presence of Byzantine faults [9][4].

Since then, there has been many improvements to the process such as: Fast-Paxos and Cheap-Paxos used to optimize the number of messages sent to reach agreement in certain conditions, or when it's permissible to give up some of the redundancy requirements. Filesystem and distrbuted storage researchers have focused on Disk Paxos, which provides gaurentees on persistant storage state when there are an arbitrary number of storage devices connected to a single processor [11].

3 Basic Paxos

We start by defining the different agents that take part in a instance of Paxos to reach a consensus [9][1]:

- Client - Requests that value be proposed by a proposer and eventually learned by the learner.
- Proposer - Proposes values to the acceptors.
- Acceptor - Accepts and stores, or rejects proposed values, acts as the 'memory' of the system.
- Learner - Learns the values from a quorum of acceptors, and responds to clients.

It is important to note that the mapping from agents to nodes/processors of little importance. One might find that when implementing the algorithm, a single node may perform any subset or potentially all of the agent roles defined above [9][1].

Guarantees for a single round of Paxos are as follows [1]:

1. Non-triviality - Only values that have been proposed can be learned.
2. Consistency - Only a single value can be learned.
3. Liveness - If some value is proposed, and sufficient processors are non faulty, eventually some learner will learn some value.

The process that the system goes through to accept a value by the quorum is as follows [9]:

1. The client requests a new value from the proposer.
2. The proposer sends a broadcast message to all the acceptors with a new value, the value should be larger than previously proposed values, but if it's not, the acceptors won't accept anyways, forcing the proposer to select another higher value.
3. The acceptors will send a ACCEPT? message to the proposer if the proposed value is greater than any value it's seen (this is to allow for nodes to fail and resume their roles later), if the value is not high enough, the acceptor can either do nothing, or send back a message letting the proposer know the lowest value it will accept.
4. Once the proposer hears back from enough of the acceptors that the value proposed is high enough, the proposer will broadcast a message telling the acceptors to ACCEPT! with the value.
5. The acceptors will then broadcast an ACCEPTED message to all the learners if the value is high enough.

6. Each learner will learn a value only if it hears a message from a majority of acceptors.

7. The learner then can return that value to the client.

4 Byzantine Paxos

The algorithm described above is not resilient against nodes that are intentionally misbehaving, selectively ignoring messages, or colluding. There have been several variants of Paxos that attempt to solve the problem of consensus with additional guarantees[2][3]. Casto and Liskov proposed an alogorithm that provided concencous for a distributed state machine, and would remain correct and lively in the precense of denial of service attacks[2]. Several simple modifications can be made to Basic Paxos to provide modest additional gaurentees. The use of public key cryptography can gaurentee the authenticity of messages [2]. The addition of a VERIFY message sent by each acceptor to every other acceptor between the ACCEPT and ACCEPTED messages also provides the additional guarentee that once an acceptor has accepted a value it can not lie about lie about that value [9].

5 Use Cases

There are very few examples of Byzatine Paxos being employed in large scale. However there are numerous examples of Non Byzantine Paxos being in distributed systems. At Microsoft, Paxos is used to coordinate tasks within the Microsoft Live Search cluster[9]. Google's distributed lock manager Chubby uses Paxos to provide a coarse grained locking primiative for aplications within Google's datacenter [10]. IBM most likely employs a variant of Disk Paxos[11] to run their IBM SAN Volume Controller [9]. The opensource Apache Foundation project ZooKeeper uses Paxos to provide a simple API for dealing with distributed concencous. An early filesystem invented at DEC called Frangipani used Paxos to maintain a consistent view of files across all clients [12].

6 Original Research/Implementations

When looking beyond the current research in the area of Paxos and the Byzantine agreement, we have come up with two additional areas to investigate to add more cryptographic assurances to prevent tampering, making it more difficult to collude to forge packets and disrupt the agreement process. By using bit-commitment one can disallow the acceptors from changing their accepted value after committing to a proposed value. Also, using this system in combination with a distributed version control system in order to reach agreement prior to making commits to the source tree would allow multiple repositories to stay synchronized and keep them from getting poisoned.

Using the bit-commitment protocol to force the acceptors to 'stick' to their word to the proposer. The cryptographically sound application of the bit-commit protocol would alter the Paxos algorithm to function like:

1. The proposer would send an ACCEPT(n)? to the acceptors who would reply to the proposer with a bit-committed value if the value is greater than any previously proposed values.
2. If the proposer hears back from a majority of the acceptors, he will then broadcast the bit-committed values and the proposed value to the learners, and the acceptors will broadcast their keys to the learners.
3. The learners will receive the value from the proposer, the bit-committed values and the keys, they can then unlock the values and see if there is a quorum.

This use of the bit-commitment on the part of the acceptors removes their ability to change their values midway through the process, slowing it down and confusing the proposer and learners. Another advantage of this system is it will reveal to the learners is if the proposers or acceptors is withholding any information, since the proposers will send the bit-committed values from the acceptors, and the acceptors will separately send the keys to unlock their commitments. This process will add a certain 'prisoner's dilemma' to the system, as you have a higher chance of finding missing information if you get a key without the box or vice versa.

Of late, version control systems have moved from a central repository system to a distributed network of local source trees where developers can check in there changes and then push certain changes to other developers who can select which changes to apply. While this helps foster an organic & ad hoc development environment, it does pose a problem to keep a central repository where end users can get the code. The Paxos algorithm applies itself perfectly to this problem, by letting the proposer propose a commit to make to the central repository, and each individual developer's repository is an acceptor, which accepts if the patch does not cause any conflicts. If enough acceptors see no problem with the patch, then it can be committed to the central repository. This also solves the problem of a single developer introducing a patch that breaks the other's code, this could be extended by adding other checks to the acceptors such as virus checking and regression testing. With those checks in place, it would be very difficult for a rouge developer to introduce a malicious patch into the main repository.

7 Conclusion

What started off as a hypothetical and somewhat humorous system for an ancient govermental system has become a powerful and practical system for reaching agreement in a distributed system. The many improvements made on the

Paxos algorithm to make it both more efficient and better tailored to certain application domains have shown its flexibility. With the advent of a Byzantine version of Paxos, it is now feasible to implement a system that is resiliant to intentional failures and collusion. By extending this with the cryptograpic procotol of bit-commitment, an even higher level of assurance is allowed. Applying this algorithm to the real-life problem of synchronizing distributed VCS repositories in a safe way. Paxos is an excellent stepping stone and a valuable tool as more and more applications out-grow their single server beginnings.

8 Bibliography

1. Lamport, Leslie. (December 2001). "Paxos made simple". SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory), 32:1825.

2. Practical Byzantine Fault Tolerance and Proactive Recovery

3. Martin, Jean-Philippe and Alvis, Lorenzo. "Fast Byzantine Paxos". Department of Computer Science, University of Texas, Austin. `http://www.cs.utexas.edu/ftp/pub/techreports/tr04-07.ps.gz`.

4. Lamport, Leslie (May 1998). "The Part-Time Parliament". ACM Transactions on Computer Systems 16 (2): 133169. doi:10.1145/279227.279229, `http://research.microsoft.com/users/lamport/pubs/pubs.html#lamport-paxos`.

5. Pease, Marshall; Robert Shostak, Leslie Lamport (April 1980). "Reaching Agreement in the Presence of Faults". Journal of the Association for Computing Machinery 27 (2), `http://research.microsoft.com/users/lamport/pubs/pubs.html#reaching`.

6. Lamport, Leslie; Robert Shostak, Marshall Pease (July 1982). "The Byzantine Generals Problem". ACM Transactions on Programming Languages and Systems 4 (3): 382401. doi:10.1145/357172.357176, `http://research.microsoft.com/users/lamport/pubs/pubs.html#byz`.

7. Amir, Yair and Kirsch, Jonathan. (September 2008). "Paxos for System Builders". Large-Scale Distributed Systems and Middleware. `http://www.cs.jhu.edu/~jak/docs/paxos_for_system_builders.pdf`.

8. Wikipedia Authors. (2008, November 19). "Byzantine fault tolerance". In Wikipedia, The Free Encyclopedia. Retrieved 21:09, December 11, 2008, from `http://en.wikipedia.org/w/index.php?title=Byzantine_fault_tolerance&oldid=252814034`.

9. Wikipedia Authors. (2008, November 29). "Paxos algorithm". In Wikipedia, The Free Encyclopedia. Retrieved 21:10, December 11, 2008, from `http://en.wikipedia.org/w/index.php?title=Paxos_algorithm&oldid=254782153`

10. Burrows, Mike. (November 2006). "The Chubby lock service for loosely-coupled distributed system". Google Research Publication. `http://labs.google.com/papers/chubby.html`.

11. Gafni, Eli and Lamport, Leslie. (2003). "Disk Paxos". Distributed Computing. `http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#disk-paxos`.

12. Thekkath, Chandramohan; Mann, Timothy and Lee, Edward. (1997). Frangipani: A Scalable Distributed File System. ACMSYOSP. `http://www.thekkath.org/papers/frangipani.pdf`.