CPSC 625 Project Report: Simple Multi-Paxos Implementation

Kevin Abbott

12/22/2015

Abstract

For my final project, I implemented a simple replicated log that utilizes a version of the Multi-Paxos protocol to maintain consistency between replicas and is able to handle both node failure and recovery. In order to test the performance of the system, I built a simple key-value store application on top of the replicated log.

1 Introduction

A common system in distributed systems is a replicated log, which is simply a replicated list of values or entries. In order to maintain consistency between the replicas, in the context of when entries are chosen and the ordering of the chosen entries, some type of consensus protocol is used when making changes to the log. While there are many different consensus protocols offering different levels of consistency, one of the most common protocols is Paxos, which guarantees strong consistency and is the basis for the system described in this paper.

Section 2 outlines the core Paxos system used in the project. First, the procedure behind Lamports Basic Paxos protocol is covered [6]. While the Basic Paxos protocol gives strong guarantees on how values are chosen, it must be expanded in order to create an efficient system that can choose a series of values. This modified Basic Paxos protocol is the Multi-Paxos protocol and the version implemented in this project is outlined in the next part of section 2 [2]. The final part of section 2 covers how the Multi-Paxos system handles and recovers from node failures.

Section 3 covers the performance of a Multi-Paxos implementation when used as the core of a replicated key-value database. First, the specifics of the key-value application are described and then the performance of the application is discussed. The perfromance is measured in the context of throughput with varying number of client nodes, server nodes, and node failure rates.

Section 4 discusses a number of improvements that can be made on this Multi-Paxos implementation in order to increase the throughput of the system. Finally, section 5 is an overview of related work in the field of consensus protocols.

2 Paxos System

Basic Paxos

At the core of any Paxos system is the Basic Paxos protocol, which is an algorithm for reaching consensus on a single value among a group of unreliable nodes. In Basic Paxos, there are four types of nodes: clients, which request values to be agreed on; proposers, which initiate the Paxos protocol by proposing a clients value to be accepted; acceptors, which react to requests from proposers to accept a value; and learners, which learn of the accepted value from acceptors. For the sake of this project, learner nodes are simply a component of acceptors.

The following outlines the message flow between the nodes during a Basic Paxos instance [6].

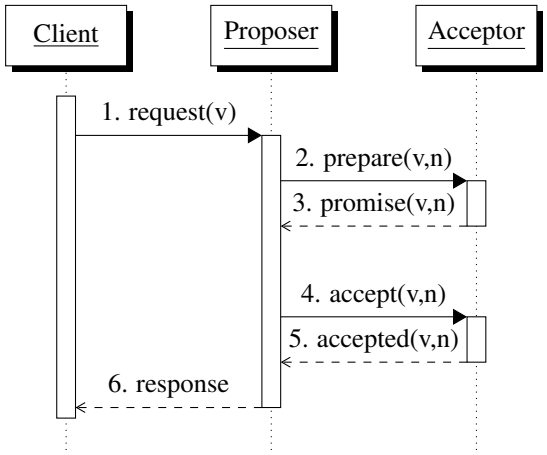

Figure 1: Basic Paxos Message Flow

Stage 1: A clients issues a request to a proposer. This requests acts as a new value for the system to reach consensus on.

Stage 2: The proposer then creates a proposal from the clients request, v, and a proposal number, n, which is larger than any previously used proposal number. The proposer then sends a prepare(v, n) message to the acceptor nodes.

Stage 3: The acceptors then process the prepare request. If an acceptor has already accepted a value, it returns its accepted value and corresponding proposal

number to the proposer. Else, if the new proposal number is larger than any previously seen proposal number, then the acceptor sends an acknowledgement of this to the proposer, which acts as a promise that the acceptor will never accept a new proposal with a lower proposal number.

Stage 4: Once the proposer receives responses from a majority of the acceptors, it processes all the acknowledgements. If any acceptor returns an accepted value, then the proposer replaces its value with the accepted value and its proposal number with the max of its original proposal number and the returned accepted proposal number. The proposer notifies the client that the request has been rejected and now acts as a proposer of the already accepted value. The proposer then broadcasts an accept(v, n) message to all acceptors, where v and n are either the original values or the already accepted values.

Stage 5: The acceptors then process the accept(v, n) message. If n is greater than or equal to the largest proposal number seen, the acceptor sets its accepted proposal number and accepted value accordingly. The acceptors respond to the proposer with the maximum seen proposal number.

Stage 6: Once the proposer receives accept responses from a majority of acceptors, it can either accept or reject the value. If any of the acceptors respond with a proposal number greater than its proposal number, the proposer knows its value has been rejected and it starts over from the prepare phase with a new proposal number. Otherwise, the proposer knows its value has been chosen and it notifies the client if the chosen value is still the original client request.

The Basic Paxos protocol provides the guarantee that a majority of the system will eventually agree on a single accepted value, but it gives no guarantee on the liveness of the system (i.e. how long it takes for a value to be accepted) or how long it takes for every acceptor to learn the accepted value. In order to make a usable Paxos system, which has the goal of quickly reaching consensus on a series of values, a number of constraints and heuristics are added on top of the Basic Paxos protocol to create the Multi-Paxos Protocol [2].

Multi-Paxos

At a high level, Multi-Paxos provides consistency on the values of a replicated log, where each entry in the log is a request that has been accepted using a separate instance of an optimized Basic Paxos protocol. The main difference between the Paxos protocol used in Multi-Paxos and Basic Paxos is that there is a single proposer which acts as a leader node. In Basic Paxos, there is a liveness issue which results from a value never being accepted because multiple proposers keep retrying their proposals with increased proposal numbers. This issue is fixed by only allowing one proposer node at a time. The following outlines the version of the Multi-Paxos protocol implemented for this project.

Stage 1: A client issues a request to the leader, which includes a value and a request id. The request id is simply the clients IP and a request number that increments for every new request. In order to maintain the invariant that request ids are monotonically increasing, the client keeps a log on disk of all requests and whether they have been accepted.

If the clients leader information is outdated and it tries to send a request to a non-leader node, then that node rejects the request and informs the client of the new leader. If a client attempts to contact a node that is offline, it will eventually timeout and contact a different node. If a client sends a request to the leader it will wait for the leaders response before submitting a new request. This is to ensure the linearizability of the clients requests. If after some amount the client has not received a response from the proposer, it contacts the leader to check its state. If the client is unable to contact the leader, it assumes the leader is down and retries its request with a new node until it learns the current leader.

Stage 2: When the leader receives a request from a client, it first checks whether it has already processed that request. The leader maintains an in memory copy of the log where each entry is a value, proposal id, request id, and whether the request has been chosen. If the leader has already seen the request and it has been chosen, it notifies the client that the request is accepted and returns the results of the request. If the request has not been accepted, the leader simply continues the Paxos instance for the request.

If the leader has not previously seen the request, it begins a new Paxos instance for the request. The leader first finds an open index in its log that is larger than any previously used index and makes an entry in the log for the request. It then sends every acceptor node a propose message that contains the value, proposal id, request id, log index, and the index of the first unchosen entry in its log. Since there is only one proposer, the proposal id is just a round number, where every time the leader changes the round increments, and the node ID of the leader. By sending the index of the first unchosen entry in the log, the acceptors are able to learn if there log is stale. This allows for every node in the system to learn the chosen value more quickly than in Basic Paxos.

Stage 3: The acceptors process the proposal via the same method as Basic Paxos, except they keep track of the specific index of the proposal. At this stage, though, one heuristic is added to allow the system to reach complete consistency more quickly. The acceptors are able to utilize the index of the first unchosen entry in the proposers copy of the log in order their logs. Since a leader

uses the same proposal number for every request during the round that it is leader, the acceptors know that if there are any unchosen entry indices less than the leaders first unchosen index and the highest proposal number seen at these indices is equal to the current proposal id, then the acceptors can simply accept the value there. The acceptors can correctly do this since the current proposer was the one that proposed the respective entry and it has it marked as chosen. Whenever an acceptor accepts an entry for an index it stores the entry on disk, which allows the system to recover from node failures.

Stage 4: Once the leader receives proposal responses from a majority of acceptors, it continues similarly to how it would in Basic Paxos. The main difference being that if it receives something different than the original request, it doesnt simply inform the client that the request is rejected. Instead, it starts the request over at a new larger index. This is why only one client request can be processed at a time, since otherwise a request could be rejected and moved to an index that is larger than a subsequent client request. This would break the lineralizability requirements of the client requests.

Stage 5: The acceptors process the accept request for a specific index via the same method as Basic Paxos. The only difference is that they return their first unchosen entry index with their accept response, which will allow them to update any stale entries they have.

Stage 6: Once the leader receives accept responses from a majority of the acceptor nodes, it processes similarly to in Basic Paxos; except if the request is rejected, the proposer doesnt restart the request at the same index. It restarts the request at a new larger index.

Further, for every acceptor node, if first unchosen index returned with the accept response is less than the proposers first unchosen index, then the proposer is able to update the acceptor by sending it the chosen value at the respective index. The acceptor then acceptor then updates its log and returns its new first unchosen index and the proposer responds to that. This process helps the replicas more quickly reach equal states and continues until the acceptors first unchosen index is greater than or equal to the proposers.

If the proposer is able to mark the clients request as chosen, it informs the client and is able to process a new request from the client.

Node Failure

Since Paxos relies on the decision of a majority of the nodes to make progress, it is able to handle transient, fail-stop node failures; specifically, given a static set of 2f+1 nodes, Paxos can tolerate f nodes to fail. The system implemented in this project is able to handle both acceptor and proposer node failure and recovery. It is able to recognize when a node fails, because a part of Multi-Paxos is that every node sends every other node a heartbeat message periodically. If a node does not receive a heartbeat message from another node after some amount of time, it assumes the node has failed and handles it accordingly.

Acceptor Node Failure and Recovery: When an acceptor node fails, the system does not have to do anything. As long as a majority of nodes are up, the system can simply continue to make progress as usual. This is due to the core Paxos protocol only requiring a majority of nodes to accept a value. If there are less than a majority of nodes online, the system is unable to make any progress and simply waits for acceptor nodes to come back online before handling any more requests.

When an acceptor nodes comes back online, it first loads all of the accepted values it knows about from disk. This puts the acceptor into a state similar to where it was when it failed. The node is then ready to start processing new requests. Due to the heuristics added in this Multi-Paxos implementation, where the first unchanged index value is passed along with the request messages, the acceptor node will eventually learn about all the values that were accepted when it was offline.

Proposer/Leader Node Failure and Recovery: Since there is only one proposer in the system, if that node fails, the system must choose a new leader in order to make any progress. In order recognize the need for leadership change, in every heartbeat message a node adds who it thinks the current leader is and which round it thinks the system is on. Whenever an acceptor node loses connection with the leader to the point that it thinks the leader is offline, it attempts to initiate a leadership change. It does this by changing its heartbeat message to indicate that the highest numbered node, that the acceptor knows is online, is the new leader and that the round number has increased. Once a majority of nodes indicate a new leader and agree on that new leader, the system changes leadership. If the old leader is still online and there were just network issues, it will cease trying to be leader once it receives a majority from the heartbeats. Further, the old leader will not be able to make any successful proposals, because its proposal number will always be less than that of the new leader, since the round number has increased

When a node learns that it is the new leader, it first goes through a proposal initiation in order to learn the state of the system. While the system would work if it simply started processing client requests, this would result in a number of throughput issues since the proposer continues to restart client requests as it learns of already accepted values at indices. To prevent this, when a new leader takes over, it first tries a series of fake requests by sending proposals at increasing indices with a proposal number of 0 until it receives a proposal response indicat-

ing the fake proposal is selected. This is now the index where it begins processing client requests.

When the leader changes, any unchosen requests still in the system will eventually be overwritten since the new leader always has a higher proposal number as the round number has incremented. The unchosen client requests of the old leader will eventually be processed once the client timeouts on the requests and tries it again with the new leader.

3 Evaluation

In order to test my Multi-Paxos system, I deployed my implementation on the Zoo cluster. In this setup, each node is a separate machine in the system that acts as an acceptor node with one of them acting as both an acceptor node and the current leader handling client requests and proposals.

On top of the core Paxos system, I built a key-value store application that supports simple put, get, and remove commands. In this application, every node knows the log of accepted database commands; but, out of simplicity, only the current leader maintains the actual key-value database. So when a client issues a command and it is chosen for a given index, it is only applied to the database once a command has been applied for every smaller index. Then, once the command is applied to the database, the leader is able to return the result of the command to the client. Whenever the leader changes, the new leader reconstructs the state of the database while initializing its log.

Performance

The system implemented in this project is able to handle a variable number of both server nodes and client nodes. Since the only useful metric to the system is its throughput, I analyzed the performance of the system by looking at its throughput while changing the number of servers and clients. Figure 2 shows the results of these tests in client requests processed per second. Here, a client request is simply a put command at increasing indices, and it's considered processed once the client receives an indication that the request was chosen and applied to the leader's copy of the database.

Overall, the actual throughput values of the system are less important than the general trends. Being able to process between 180 and 370 requests per second is not representative of a Multi-Paxos system and is more indicative of my specific implementation, the example client application, and the workload type. Instead, only the relationship between throughput and number of servers and clients is a useful metric of Paxos in general. Specifically, the trend that an increase in number of server nodes causes a significant decrease in throughput. This is an expected result of the overhead of increasing the number of replicas. Since every node in the system sends every other node a heartbeat messages, as the number of server nodes grows the number of heartbeat messages in the system grows exponentially. As a result, the system spends more and more time sending and processing heartbeat messages instead of handling client requests. Ultimately, this limits the size of a Paxos cluster. For my project, I was not even able to get the system to work with 7 nodes without issues arising from the number of heartbeat messages, and instead I could only compare 3 and 5 server nodes.

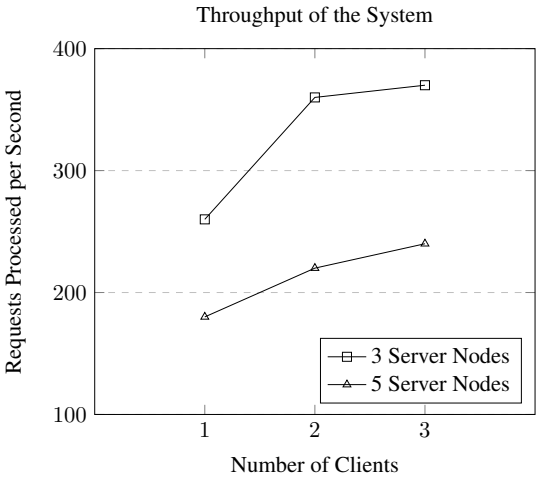

Figure 2: Graph of System Throughput

4 Future Work

There are a number of ways in which the system can be improved. Currently, whenever nodes come back online they must learn all the requests that have been accepted while they were offline. When a node is offline for a short period of time, this is not a big performance hit for the system; but if a node is offline for a long period of time, it will have a lot of values to learn and potentially congest the system. One fix to this is to implement regular snapshotting on the log. In this type of system, after so many indices have been processed, the system could simply create a snapshot of all the accepted values and store this on disk. In this case, when an acceptor node comes back online, it would first load its stored log, then query another node for any missed snapshots, then it could finish updating itself the regular way with messages to the leader.

The system also has a lot of congestion whenever a new leader initializes, because it has no concept of the state of the log and must either get the state using a series of fake requests or learn the state once it attempts the first client request. One fix to this is to have the new leader initialize its log with that of the acceptor node on

the same machine. For simplicity, while the proposer nodes shares the same machine as an acceptor node, the current implementation treats these as separate entities. If a proposer is able to load a state of the log locally and then get corrections on it once it tries to propose new values, this would prevent a lot of congestion in the system.

The classic Paxos protocol also assumes a static membership set and transient node failures, and this assumption is used for this project. In a real Paxos implementation, though, node failures may not be transient and there may be a desire for dynamic membership. Dynamic membership would require an update to every nodes membership list and a change in quorum size. This could be handled via a Paxos instance on these values followed by some Leadership and round change.

Overall, Paxos is meant to provide strong consistency on the replicated log, but for some applications strong consistency is not always needed. In the case of the key-value store, this means allowing clients to read stale data. There are many considerations to make when allowing clients to read stale data, but the system could easily be adapted to do so. Currently, only the leader maintains an actual copy of the database, but if every acceptor node also maintained the database, then any node could process read commands. Any command that alters the state of the database, though, would still need to be processed by the leader, so the decision to do this would depend on the workload. In a write heavy workload, there is no benefit to adding this feature and it would only increase the memory requirements of the system. In a read heavy workload, however, the throughput of the system could increase with this improvements if it is acceptable to read stale data.

5 Related Work

Paxos, especially any implementation that maintains consistency across multiple values, is often considered an extremely complex and difficult to implement system. As a result, a significant amount of work has been done to explain Lamort's algorithm more cleanly, optimize the algorithm, or modify the algorithm to make the system more easily implementable [2].

Beyond explanation or simple modifications, a lot of work has been done in implementing the algorithm for large scale systems. Google uses Paxos as the base for two of its systems: Chubby and Spanner. Chubby is a distributed lock service that relies on Paxos to keep replicas consistent in case of failure [1]. Spanner utilizes Paxos within its operation to shard data across servers [4].

While Paxos is widely used, there a number of alternative approaches to maintaining consensus in a distributed state machine; these include Raft [8], ZooKeeper's Atomic Broadcast (ZAB) [5], and other systems like Conflict-free Replicated Data Types [9], Viewstamped Replication [7], or the Chandra-Toueg consensus algorithm [3].

Raft was created to be a more understandable alternative to Paxos. Whereas Paxos is primarily concerned with the mechanics of consensus, Raft is oriented around the practiacl challenges of building a replicated log system. The biggest difference between Raft and Paxos is that Raft makes leadership a core part of the protocol and requires electing a leader before any proposed change, as well as making most of the consensus functionality concentrated in the leader. This simplifies the consensus algorithm since leader election becomes the first phase of the two phase consensus, as opposed to an entirely separate concern, as it is in Paxos.

ZooKeeper's Atomic Broadcast (ZAB) is the consensus protocol used by ZooKeeper to achieve fault tolerance by replicating the configuration repository. As a result, Zab is designed for primary-backup systems, whereas Paxos is designed for state machine replication. Paxos ensures that the same sequence of client requests are executed in some order. Zab, though, wants replicas to agree on incremental state updates and must ensure that updates are executed in the correct order. It does this by requiring strong leadership constraints and stronger synchronization during consensus and recovery.

6 Conclusion

In the context of a single value, the Basic Paxos protocol offers some strong guarantees. It is guaranteed that as long as a majority of nodes are online and functioning then the system will be able to make progress and eventually agree on a single value. While the Paxos protocol can be used to create a strongly consistent replicated log where each entry is a Paxos instance, it is normally optimized in the Multi-Paxos protocol by adding a few constraints and heuristics in order to make the system more efficient and available. The main difference is that there is a single leader node that acts as the sole proposer. This alone eliminates the main liveness issue of Basic Paxos, since there are no longer multiple proposers constantly vying to get a value accepted at a specific index in the log. On top of this, any number of heuristics or design choices can be made to increase the usefulness of the system. In the context of this project, node failure handling and recovery, and leadership change are implemented; but functionality to speed up failure recovery, or allow for weaker consistency in reads is not, even though these are use functions for a Multi-Paxos implementation. This ambiguity in Multi-Paxos functionality is why is not very well defined and difficult to implement, whereas the Basic Paxos protocol is clearly outlined with very specific guarantees.

Overall, the goal of this project was to make a sim-

ple replicated log that utilizes the Paxos consensus protocol in order to maintain consistency between replicas. While the outcome of this project is usable, it is far from deployable and there are much better already existing Paxos implementations. Instead, this project was an opportunity to explore the Basic Paxos protocol and research the various optimizations used to make it efficient.

References

[1] BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 335–350.

[2] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing* (2007), PODC '07, ACM.

[3] CHANDRA, T. D., AND TOUEG, S. Unreliable failure detectors for reliable distributed systems. *J. ACM 43*, 2 (Mar. 1996), 225–267.

[4] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[5] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks* (Washington, DC, USA, 2011), DSN '11, IEEE Computer Society, pp. 245–256.

[6] LAMPORT, L. Paxos made simple. In *ACM SIGACT News* (2001), ACM.

[7] OKI, B. M., AND LISKOV, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1988), PODC '88, ACM, pp. 8–17.

[8] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 305–319.

[9] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems* (Berlin, Heidelberg, 2011), SSS'11, Springer-Verlag, pp. 386–400.