Scaling Leader-Based Agreement Protocols for State Machine Replication

Presented By

Ellis Michael

in Partial Fulfillment of the Requirements for Graduation With a Dean's Scholars Honors Degree in Computer Science

Supervisor: Lorenzo Alvisi

Second Reader: Manos Kapritsos

Abstract

State machine replication is a technique used to guarantee the availability of a system even in the presence of faults. Agreement protocols are often used to implement state machine replication. However, the throughput of many agreement protocols, such as Paxos, does not scale with the number of machines available to the system. Systems whose throughput does scale often provide weaker consistency guarantees than state machine replication's linearizability. These weaker guarantees, such as eventual consistency, make the job of the application programmer much more difficult.

Kapritsos and Junqueira previously proposed a protocol which uses an agreement protocol as a primitive and does scale. In this thesis, I describe my novel implementation of this protocol as well as the optimizations that were necessary to achieve scalability. The results of my experiments with this system show modest but promising evidence that the throughput of this system can, indeed, scale linearly with the number of machines.

Acknowledgments

First, I thank my family for a lifetime of encouragement. Next, I thank Lorenzo Alvisi for his guidance and Manos Kapritsos for his help and his patience.

Lastly, I thank Ruta Maya Coffee and Stone Brewing Co., the makers of products with which I pursued both optimal caffeination and the Ballmer Peak while working on this thesis.

Contents

Chapter 1

Introduction

State machine replication [1] is often used to guarantee the availability of a service even in the presence of faults, and is ubiquitous both in the distributed computing literature and in industry.

In today's world where large-scale web applications are commonplace, scalability is key. In order for systems to be of practical use to these applications, they need to be able to accommodate massive throughput requirements. While we cannot expect a system with fixed resources to handle an arbitrarily large workload, we would like for its throughput to scale (hopefully linearly) with the resources available to it (in this case, the number of machines).

Agreement protocols, such as Paxos [2], are commonly used to implement state machine replication. Instead of scaling, however, Paxos and other agreement protocols slowdown as the number of replicas in the system increases. Systems that do scale well often do so at the cost of consistency. They enforce much weaker con-

sistency guarantees which make using the system more difficult for the application programmer.

In this thesis, I describe an approach originally proposed by Kapritsos and Junqueira [3] to solve the scalability problem while maintaining linearizability. Their approach is a meta-protocol that uses a leader-based agreement protocol as a primitive. It divides the incoming client requests among multiple different clusters, all of which are running the agreement protocol to order those requests. The ordered requests are then forwarded on to execution nodes which execute commands and reply to the clients. Their approach is only useful as long as ordering is the bottleneck in the system, but for many applications, execution can be quite fast, especially when compared to many messages required by agreement protocols.

I then describe my novel implementation of their protocol, as well as a handful of optimizations. I also present the results of experiments with my implementation that demonstrate the scalability of the protocol.

Chapter 2

Background

2.1 State Machine Replication

A state machine is an *abstraction*. Informally, a state machine consists of some mutable state and *commands* which *clients* of the state machine can perform on that state [1, 2]. The state machine receives those commands, performs some operations to transform its state, and sends the response back to the clients. The state machine executes commands atomically (or *linearizably*). That is, the execution is equivalent to some sequential execution which respects the real-time ordering of the commands [4, 5]. Also, the commands that the state machine executes are *deterministic*.

An archetypal example of a state machine is a web server. The web server exposes a public API, which clients can make requests against. The web server receives client requests, processes them, and returns responses.

Often, we would like our state machines to tolerate faults. After all, in the case of

a web server, hardware often fails, and we want our server to be up for long periods of time. One approach is to have the server log to disk all of the commands it receives. Whenever the server fails, we could simply restart it and have it recreate its local state from the log. This approach is called replication in time. While it can be an effective way of tolerating failures, replication in time leaves the service unavailable for long periods of time while the server is restarting and reloading its local state.

Another approach to making a state machine fault tolerant is replication in space. Here, instead of having one state machine, we have a group of state machines. When one state machine fails, other state machines are up and running and able to take the failed state machine's place. These state machines are known as *replicas*. These replicas coordinate their actions so that their states stay consistent over time and they reply to clients' requests in a consistent manner. For instance, suppose we have a set of replicas running a banking service, and clients' balances aren't allowed to go below $0. If one account's balance is $100, and two separate clients concurrently make a request to the replicas to withdraw $100, the system should approve exactly one of the withdrawals and send an insufficient funds notification to the other.

In order to keep the replicas in a consistent state, it is sufficient for them to execute the exact same set of commands in the exact same order [1]. As long as the commands are deterministic, each of the replicas will go through the exact same state transitions, and if multiple replicas reply to the same command, they will send back the same result. Typically, we say that each replica has a sequence of *slots* in which to place commands, and as long as replicas place the same commands in the same slots, they will stay in consistent states.

2.2 Fault Tolerance

We would like distributed systems to be robust even in the presence of failures. Since no system can possibly maintain availability if all of its processes fail, we typically define the types of failures we wish to tolerate and then describe the number of such failures our protocol can tolerate. One common failure model is *crash-fail*, in which processes can at any point crash but otherwise exhibit correct behavior. The weakest of all possible failure models is the *Byzantine* failure model in which faulty processes can fail in arbitrary, even adversarial, ways. The main protocol described in this thesis uses an agreement protocol as a primitive and can be adapted to any failure model. However, my implementation uses Paxos (briefly described in Section 2.5) and assumes the crash-fail model. Unless otherwise noted, the crash-fail model should be assumed throughout this thesis.

2.3 Network Models

A *synchronous* network is one in which message delays are bounded. An *asynchronous* network is one in which message delays are arbitrary. All work in this thesis assumes an asynchronous network.

2.4 Consensus

Consensus is a fundamental problem in distributed computing. It involves multiple processes proposing values and reaching agreement on a single decided value out of

the proposed values. A protocol that solves consensus must satisfy the following four properties.

- **Validity** If all processes that propose a value propose v, then all correct processes eventually decide v
- **Agreement** If a correct process decides v, then all correct processes eventually decide v
- **Integrity** Every correct process decides at most one value, and if it decides v, then some process must have proposed v
- **Termination** Every correct process eventually decides some value

Here, a *correct process* is one which never fails.

Many important and well-known problems can be formulated as special instances of consensus. For instance, deciding whether or not a distributed transaction in a database system should commit is a special instance of consensus with the extra requirement that if any process proposes an abort, all processes should decide to abort. There exist algorithms to solve consensus in a synchronous network. However, it is impossible to solve consensus in an asynchronous network in which message delays are unbounded and processes can fail [6].

2.5 Paxos

While Paxos does not solve Consensus, it comes very close. The Paxos protocol assumes an asynchronous network and is always safe. That is, if a correct process

decides v, then some process must have proposed v, and no other correct process ever decides a value other than v. Paxos can tolerate f faults, where the number of processes is at least $2f + 1$, and still be live as long as there is sufficient synchrony in the network. In many practical applications, there is almost always sufficient synchrony in the network, and Paxos is almost ubiquitous in industry.

Paxos (or any "agreement protocol") can be used to implement state machine replication by having the processes run multiple (often concurrent) instances of the protocol, one for each slot. Processes propose a command in an open slot in response to a client request, and when a decision is reached, they place that command into the permanent slot order.

Chapter 3

The Scalability Problem

Suppose that you have a service running with replicated Paxos state machines. Further suppose that executing clients' commands happens relatively quickly (e.g. all of the commands are processed in $\mathcal{O}(1)$ time). Your service becomes very popular, and you notice that your throughput isn't what you would like it to be and that client response times are starting to spike. You might try increasing the number of replicas in the system and the number of machines that those replicas are running on. If you did that, however, you would discover that your system gets slower and its throughput decreases instead of increases. This may seem counter-intuitive since you're essentially increasing the resources available to what could fairly be described as a parallel computation.

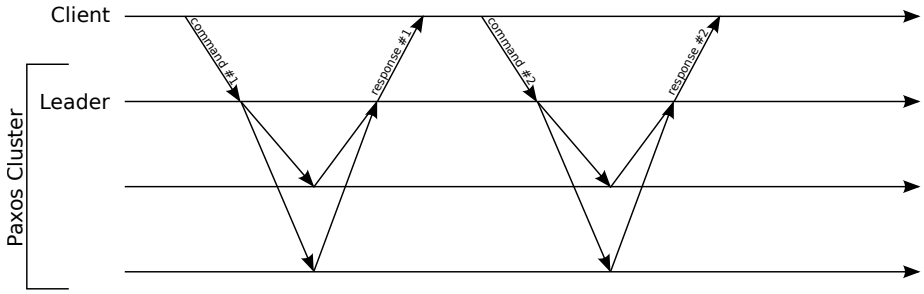


Figure 3.1: Best normal case operations for Paxos. The client sends the command to the current leader, the leader sends a message to all the replicas, waits for replies from a majority of the replicas, and then responds to the client.

3.1 Why Paxos Doesn't Scale

To understand why Paxos gets slower as you increase the number of replicas, we need to understand the message pattern of the normal case command-response loop. The best case of this loop (the case where the client originally sends its command to the current leader and no other replica preempts the leader) is shown in Figure 3.1. There, we see two iterations of the normal case operations. This is what Lamport calls the second phase of the Paxos algorithm [7]. The leader sends messages to all of the replicas and waits for responses from at least a majority of the replicas, including itself. Only after receiving those responses does the leader respond to the client.

It is pretty easy to see what is going on with our system that is slowing down. As we increase the number of replicas in the system, the number of messages the leader has to send per request increases. The cost of sending and receiving messages is often much higher than the cost of local computation in distributed systems. If we neglect local processing entirely and assume that the leader replica can send or receive messages at a fixed rate, then as we increase the number of replicas, n, the

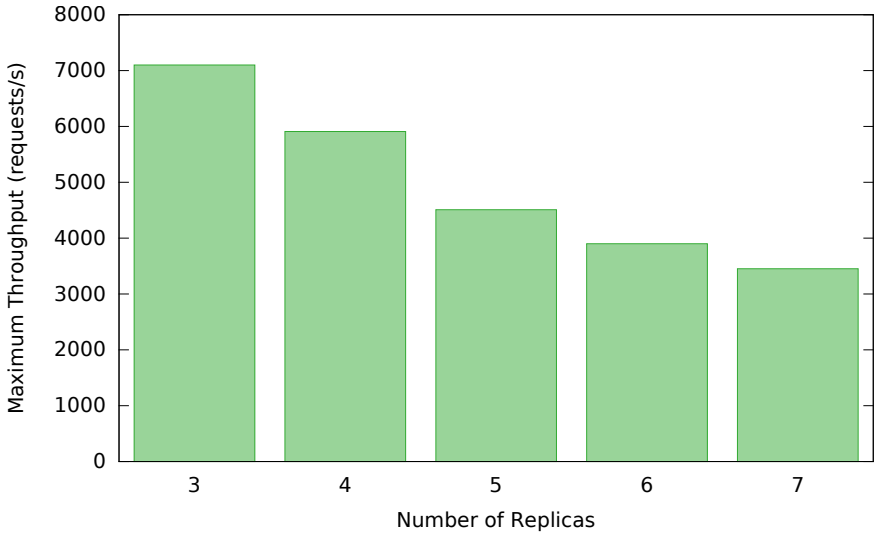


Figure 3.2: Performance of a single Paxos cluster as the number of replicas in the cluster increases. Each replica was run on a separate machine.

throughput of the system, t, should be described by the following:

$$t \sim \frac{1}{n}$$

And in fact, the data in Figure 3.2 show just that relationship.

It is worth pointing out that in the experiments in Figure 3.2, the leader of the system always waits for replies from of a majority of replicas. This means that as the number of replicas increases, the fault-tolerance of the system increases. Waiting for a majority of replicas, however, is necessary since in Paxos, proposals must be accepted by a majority to be decided.

3.2 Approaches to the Scalability Problem

Various systems have gotten around the scalability problem, but they typically do so by sacrificing the total ordering of all client commands. Google's Megastore [8], for instance, partitions up its processes' state space into disjoint entity groups. Commands executed on items within a certain entity group will have strong consistency semantics, but Megastore maintains looser consistency semantics across entity groups. Amazon's DynamoDB [9] does away with linearizability entirely and provides only *eventual consistency* for writes and optionally provides "strong consistency" for reads. Apache's Cassandra [10], while boasting linear scalability, also embraces eventual consistency. Eventual consistency only guarantees that two processes with the same set of writes will have the same state and that any write received by one process will eventually be received by all processes.

While these approaches have certainly been able to achieve scalability in practice, they all do so at the expense of consistency. Programming in the eventual consistency setting is very difficult, and many nonintuitive situations can occur. For instance, if I have a group messaging application, it is entirely possible under eventual consistency for people in the group to see messages out of order. This could result in seemingly impossible orders, such as seeing an answer to a question before seeing the question itself. Also, if I had an application with a user management system, under eventual consistency, one process could see some user's actions before seeing that the user was created. It would be up to the application programmer to handle these cases

Others have noted how hostile the eventually consistent model is and argue for stronger consistency semantics such as *causal consistency* [11]. However, not even

causal consistency can guarantee the safety of global invariants of a system, such as account balances staying above $0 in a banking application. Linearizability, the consistency guarantee of normal Paxos replicated state machines, is strong enough to be able to guarantee global invariants.

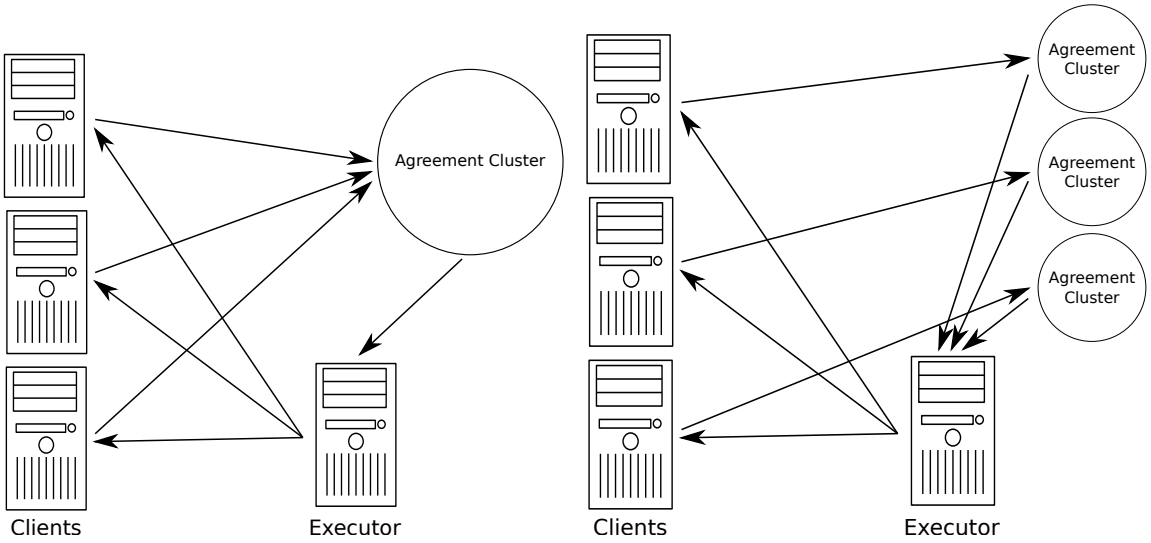
It is worth noting that many systems do not opt for eventual consistency only for scalability reasons. Many do away with consistency because they want to preserve the availability of the system, even in the case of network partition. So, because of the famous (or perhaps infamous) CAP theorem [12, 13], systems designers often reason that providing stronger consistency semantics is impossible and therefore embrace eventual consistency. Systems such as Eiger [14] demonstrate that stronger consistency properties are practical, even in the face of partition. However, a detailed discussion of coping with network partition is beyond the scope of this thesis.

Chapter 4

Dividing The Workload

We would like to somehow have our system running Paxos replicated state machines scale as we add more machines to the system. Ideally, we would like the system to scale linearly. That is, if n machines can process r client requests per second, we would like $2n$ machines to be able to process $2r$ requests per second, all without sacrificing the fault-tolerance of the system or the linearizability of the commands. At first, it might be surprising that this is even possible. After all, Lamport himself claimed that the Paxos algorithm is essentially optimal in that the second phase of the algorithm has the minimum cost among all algorithms for reaching agreement with the possibility of process failures [7]. And, indeed, it is optimal in a sense, but what is also true is that it is not strictly necessary that *every* process in the system reach agreement about every slot in the global ordering of commands.

I will describe a protocol which runs on top of an existing agreement protocol such as Paxos and allows the entire system to scale. I will organize this description



(a) A simple example of separating ordering from execution. Clients send requests to the agreement cluster, which orders those requests and passes them on to the execution node. The execution node will execute those requests in order and then send the responses back to the clients.

(b) Splitting order form execution allows the possibility of multiple agreement clusters, each accepting client commands, ordering them, and passing the order on to the execution node.

Figure 4.1: From splitting order and execution to splitting up the workload

by describing each of the principles that were applied in order to arrive at the final protocol.

4.1 Separating Order from Execution

First of all, this protocol makes use of the fact that we can separate ordering from execution [15]. Paxos and other agreement protocols are only used to reach agreement about the order in which commands should be executed. It is not strictly necessary to have the same machines also execute those commands. Instead, we can have clients

send requests to nodes running the agreement protocol, and once those agreement nodes reach agreement about a certain slot, they forward the command and its slot number on to the execution node(s). The execution node executes the commands according to the ordering from the agreement cluster and then sends the responses back to the clients. Figure 4.1a shows this system design.

Here, it is important to note that, just as in the normal case for Paxos replicated state machines, the execution node could get a decision for some slot numbered i before getting decisions for all slots numbered less than i. In this case, the execution node simply adds the command to a backlog and waits for more decisions from the agreement cluster. Once it receives decisions for and processes all slots numbered less than i, only then can it execute the command in slot i and send the response to the appropriate client.

4.2 Using Multiple Clusters

It might be unclear what separating ordering from execution accomplishes. It may seem as if we're in the exact same state as before, only this time with a new machine in the system that all commands must be routed through, increasing latency. However, separating ordering from execution means that agreement nodes need not know about the decisions for every single slot. In order for any replicated state machine to respond to a client command in some slot, it has have to know about all of the decisions for smaller slot numbers. Since the processes in the agreement cluster are no longer responding to clients, they need not know about all of the decisions made. What

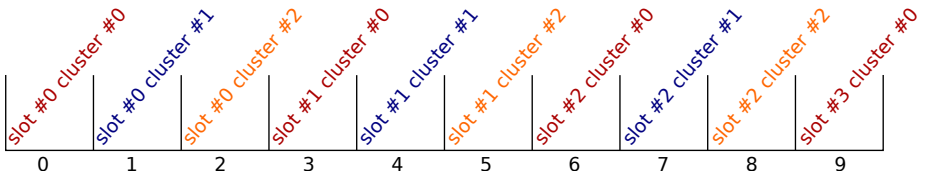


Figure 4.2: A view of the global slot order, seen by the execution node(s). Here, there are three clusters total.

this allows us to do is split up the workload among multiple clusters, all running the agreement protocol, as shown in Figure 4.1b.

Each cluster will now receive only a subset of the incoming client commands. The agreement cluster will order them normally and pass on the commands and their ordering to the execution node(s). This establishes multiple partial orders over all commands issued to the system. In order to achieve linearizability in a way that is scalable, we need to combine the partial orders into a total order in such a way that minimizes coordination between clusters.

The approach we take is fairly straightforward. We say that each agreement cluster has its own command slots as before, but also that there is a *global slot order*, which takes commands from each of the cluster's slot orders in a round-robin fashion. More formally, if there are N clusters total, the ith slot in cluster j's slot order corresponds to slot $i \cdot N + j$ in the global slot order (assuming clusters and slots are both indexed from 0).

Since only the execution node needs to know the decisions for all slots in the global slot order, the system can make progress with little to no coordination between the agreement clusters. This means that, as long as the execution node has the capacity to keep up with all of the decision notifications it receives and is not the bottleneck

of the system, the entire system should scale linearly as more identical agreement clusters are added.

4.3 Collocation

Recall that in Figure 3.1, the leader was the sole bottleneck of the system. In the best case, each follower node in a Paxos agreement cluster only needs to send and receive two messages per request. It would be inefficient to have machines in our system only running a single follower process. Instead, we can *collocate* leaders and followers from different clusters on the same machine, in a way similar in spirit to Mencius [16]. For instance, if we wanted three clusters with three agreement nodes each, we could use three machines total; each machine would have the leader process from one cluster and follower processes from the other two clusters. This more evenly distributes the load across all machines.

We do not require that the number of nodes in each cluster be equal to the number of cluster or machines or even a multiple thereof. If we want N clusters, each with r nodes each, we can use N machines, locating a leader on each of the machines as well as $r - 1$ followers, from as many different clusters as possible. The simplest strategy would be that machine i would have on it the leader of cluster i as well follower node in clusters $i + 1$ through $i + r - 1$, wrapping around to cluster 0 of course for cluster numbers greater that N.

4.4 Maintaining Liveness

We would like to maintain the liveness of the system even in the case of extreme load imbalance. Even if only one cluster is receiving requests, we would still like to guarantee that those requests will eventually be processed by the execution node. Without some mechanism in place to maintain the liveness of the system, the execution node could wait indefinitely for the other clusters to fill in the holes in its slot order.

Therefore, from time to time, the leader of each cluster sends a message to a node in the other clusters informing them of the largest slot number for which leader's cluster has decided a command. The other clusters will then propose NoOps for each of slots up to that limit that do not yet have a command. That way, the execution node can always make progress. An important note here is that these NoOp decisions can be batched together and treated as a single "command" by the agreement protocol. In the terminology of "Paxos Made Moderately Complex" [17], these NoOps can be grouped together as a single pvalue.

Chapter 5

Implementation and Optimization

I implemented the multi-cluster approach described in Chapter 4 in Java using Paxos as my agreement protocol. The end goal of implementing the system was to demonstrate the approach's scalability. A number of different implementation details and optimizations were necessary to achieve this.

5.1 Dealing with Load Imbalance

One major hurdle that the multi-cluster system must deal with is load imbalance, where one agreement cluster is receiving many more requests than the other agreement clusters. Ideally, there would be no load imbalance. This could be achieved easily enough by placing a load balancer in front of the clusters. If there are sufficiently many clients, having each client connect to a random agreement node from a random cluster would also be an easy way to ensure a more or less balanced load. Nevertheless, we would like the system to still make progress even if one cluster is

receiving all of the clients' commands, while the other clusters are receiving none. The mechanism described in Section 4.4 will guarantee the liveness of the system. However, the method and parameters used to determine when to send NoOp notifications have performance implications for the system, even when the load on the system is balanced

In my testing, I discovered that if the NoOp notifications are to be sent at some fixed rate, that rate has to be fairly large compared to the latency of the system. I usually used one hundred milliseconds, whereas the base latency of the system was usually around one millisecond. If I made the notifications more frequent, the system's overall performance degraded somewhat as the number of clusters increased. This meant, however, that when the load on the system was imbalanced, latencies spiked above one hundred milliseconds. I discuss alternatives to the fixed rate notifications in Section 8.3.

5.2 Client Backlogs

In order to allow each client to connect to multiple agreement nodes, perhaps in different clusters, and still have its commands processed in FIFO order, client backlogs on the execution node are necessary. Each client numbers its commands, and each of those commands ends up in some slot in the execution node's global slot order. If the execution node sees some command i from client c in a slot it is currently processing but hasn't processed all of the commands from c with command numbers smaller than i, it keeps processing slots in its global slot order until it has processed

all commands from client c with command numbers smaller than i. It then pulls command i from c's backlog and processes it, replying to c with the result.

5.3 Dealing with Failures

We want the meta-protocol to be as fault-tolerant as the agreement protocol it is using. In the case of Paxos, this means that it should always be safe and be live when there are fewer than f failures and sufficient network synchrony. When the multi-cluster protocol is using Paxos agreement clusters, each of which can tolerate up to f faults, the entire system can, in fact, tolerate more than f faulty agreement nodes, as long as there are f or fewer faults in each cluster. When there are more than f faulty nodes in an agreement cluster, however, that cluster may not be able to reach agreement for any new slots, and the entire system could be stalled.

From a practical perspective, this is why placing follower nodes from different clusters on the same machine is a good idea. This way, when there are more machines than nodes per cluster, we are at least keeping failures in the same cluster as statistically independent as machine failures. If we collocated agreement nodes from the same cluster on the same machine, then a single "failure" could cause both nodes to be faulty.

In order to tolerate execution node failures, the execution nodes should be replicated as well. They should all receive decisions from the agreement clusters and all respond to clients. Since the execution nodes all process commands in the global slot order in a deterministic fashion, they should all go through the exact same sequence

of state transitions and send the same responses to the same commands. The client simply ignores any duplicate responses.

One might worry that duplicating the execution nodes increases the cost of each request and negates the advantage of splitting up the workload. It is true that it increases the overhead per request since we now have to send each decision in the agreement cluster to multiple execution nodes, but we can always increase the number of clusters to counteract the decrease in throughput. It is important to note here that the number of execution nodes is only relevant to maintaining the fault-tolerance of the system. The number of execution nodes is not the lever we use to increase the throughput of the system, so the overhead becomes a constant factor when analyzing how the system scales. As long as the execution nodes have the capacity to deal with the messages they need to send and receive, the system should still scale linearly with the number of agreement clusters. In a sense, the fault tolerance of the execution nodes is orthogonal to the scalability of the agreement protocol.

5.4 Agreement Protocol Interface

In developing the infrastructure for this project, I wanted to make the interface between the multi-cluster meta-protocol and the underlying agreement protocol as clean as possible in order to facilitate modularity. My implementation used Paxos, but the meta-protocol was implementation agnostic. It could use any leader-based agreement protocol implemented in terms of the following interface:

- propose(command)

- proposeNoOps(slotNumberLimit)
- getNewDecisions()
- isLeader()

Of course, my implementation of the meta-protocol assumed non-Byzantine faults and would have to be modified to be Byzantine fault tolerant, even if a Byzantine fault tolerant agreement protocol such as PBFT [18] were used.

What this common interface allows is for someone who wants to see how well their favorite agreement protocol scales to sit down and have a running system in an afternoon, provided they have a thorough knowledge of the protocol or a good reference implementation.

5.5 Paxos Optimizations

When implementing multi-cluster Paxos, I started off with the "Paxos Made Moderately Complex" [17] implementation of Paxos. A number of optimizations had to be made to the protocol, however, to prevent performance degradation in the multi-cluster case. Perhaps most important was allowing the current leader of the cluster to change the slot number of a proposal. Ordinarily, clients send commands to any node in the Paxos cluster, and that node then proposes the command in its first empty slot by sending the proposal to the current leader. If the leader already has a proposal for that slot, it ignores the new proposal. Only when the original node gets a decision notification for that slot number does it see that its command was overwritten and then repropose the command. I allow the leader to alter incoming

proposal messages from nodes, moving the command to the leader's first empty slot. While this could result in the same command being decided for multiple slots, that could happen anyway if the client retries the command because of a timeout. The execution node simply ignores duplicate commands. In the normal case, however, the command will only be decided for a single slot. This change saves quite a few messages and prevents high latencies due to many commands taking multiple rounds of being overwritten and reproposed.

Chapter 6

Evaluation

I tested my final system thoroughly using a variety of experimental setups. Each setup consisted of a complete configuration of the system, including the number of clusters, the number of nodes per clusters, the number of execution nodes, and the number of machines being used as clients. For each setup, I ran a sequence of trials, each time exponentially increasing the number of active requests each client machine had at any given time. I ran each trial for a fixed period of time, usually a couple of minutes, and recorded the throughput and latency of the system, throwing away the result and restarting the trial if there was too much variability. These trials produced a throughput-latency graph for each experimental setup, two of which can be seen in Figure 6.1. The horizontal asymptote in this graph is the latency of the unloaded (or lightly loaded) system, while the vertical asymptote is the maximum throughput of the system.

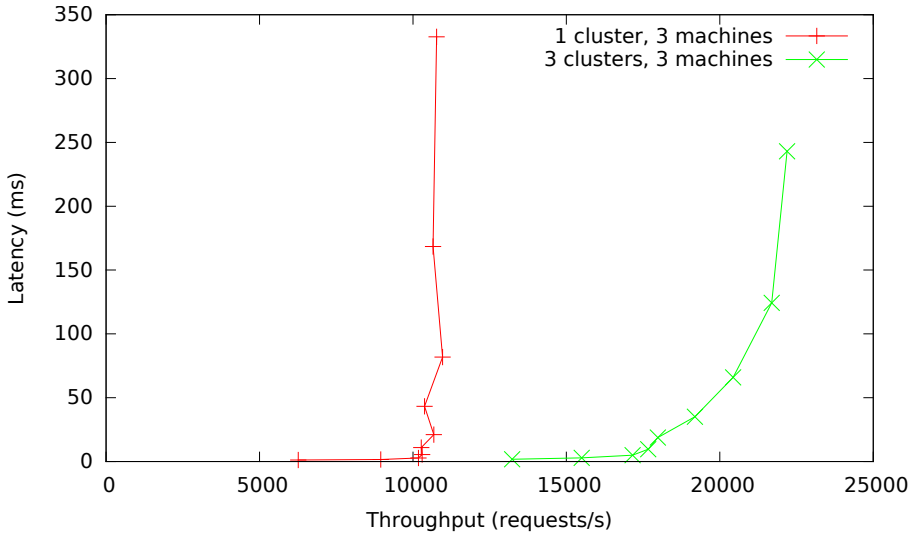


Figure 6.1: Running one cluster on three machines as compared to running three clusters on the same three machines.

6.1 Collocation

Figure 6.1 shows the results of two separate experiments using the same three machines. In this case, collocation was able to yield more than a 100% increase in the maximum throughput of the system.

6.2 Scalability

I ran a series of experiments with increasing numbers of clusters of three nodes in which the number of machines was kept equal to the number of clusters. The condensed results are shown in Figure 6.2. Here, we clearly see linear scaling up to seven machines. This result is encouraging and at the very least is good evidence that the approach is sound.

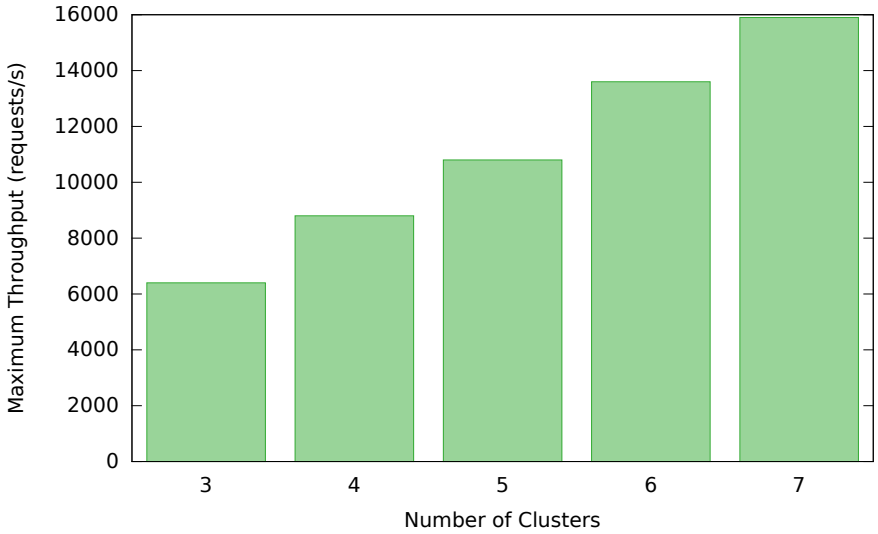


Figure 6.2: Performance of multiple Paxos clusters, where the number of machines available to the system is equal to the number of clusters. Each cluster ran with three nodes, and each machine had exactly one leader and two follower processes.

This series of experiments was run on much slower machines than those in Figure 6.1, which explains the discrepancy in base throughput. The slower machines were the only ones available in large enough quantities to run this experiment. However, all that is important to the outcome is that the system scales properly; the base throughput is mostly irrelevant.

6.3 Latency

Having multiple clusters might impose significant costs in terms of average request latency since decisions sent to the execution node might wait in the backlog for decisions from other clusters. I call the amount of time a command waits in the execution node's backlog the *waiting time* of the command. For each experiment I

ran, I logged the global average waiting time as well as the average waiting time of each cluster's commands.

In my testing I found that even when the load on the system was relatively low (about 20% of maximum), the average waiting time was between one and three tenths of a millisecond across all clusters. As the load on the system increased, the average waiting time only decreased since the volume of requests coming in ensured that the clusters were filling temporary "holes" in the execution node's slot order quickly. While not entirely negligible, this delay is insignificant when compared to the latencies experienced when the system is under much higher load. Of course, the load on the system during these tests was balanced across clusters and agreement nodes. When the load was unbalanced, I saw much higher average waiting times and more variability in waiting times across clusters.

Chapter 7

Related Work

Of course, much of this work was first described by Kapritsos and Junqueira [3]. The main contribution of this thesis is a novel implementation of the protocol and a thorough evaluation of the working system, as well as a number of optimizations.

The idea of separating order from execution was introduced in an earlier paper, though in the context of Byzantine fault tolerant systems [15]. Mencius [16] was a paper which introduced the idea of spreading the load of the leader among Paxos replicas based on the observation that the leader processed an inordinate number of messages per request.

Scatter [19] is a project which provides a scalable distributed hashtable and maintains global consistency. Google's Spanner database [20] is a fairly recent project which aims to provide "external consistency."

Chapter 8

Conclusion

Paxos and similar leader-based agreement protocols for state machine replication do not scale. While many popular systems do achieve linear scaling, they often do so at the cost of consistency. Eventual consistency and other forms of consistency weaker than linearizability impose difficulties for the application programmer.

While the protocol presented and evaluated in this thesis cannot provide availability in the face of network partition, it does provide a scalable system for state machine replication.

8.1 Applications

I believe that this work is most likely to be of value in a data center setting, where the network is almost always reliable and partition (within the data center) is unlikely. Paxos is already being used in systems in data centers and is well understood. This work describes a natural extension to Paxos that allows it to scale, something that is

certain to be necessary if Paxos is to be used to totally order all operations in some of the largest web applications.

As proposed by Kapritsos and Junqueira, an *ordering service* could be implemented using this work [3]. Not only would such a service add to the litany of "XaaS" services out there, it would also be useful for the development of web applications. Instead of each application managing its own replication or relying on its own instance of the scalable agreement system for ordering, having a central service would be much easier to manage.

8.2 Limitations

Firstly, it is worth mentioning again that this approach is only of use when ordering is the bottleneck. When execution is the bottleneck in the system, increased ordering throughput becomes useless.

Throughout my experiments on the system, I discovered that the throughput of the entire system is limited by the throughput of the slowest cluster (i.e. the total throughput was the number of clusters times the throughput of the slowest cluster). Therefore, the approach outlined in this thesis should ideally be used in homogeneous environments where the machines on which the system is deployed are identical.

I was only able to test the scalability of my system up to seven clusters due to the limited number of identical machines I had access to. While I was able to see linear scaling up to seven clusters, the total throughput of the seven cluster system was modest at best, and without seeing scalability up to several hundred thousand

requests per second, it is tough to make definitive statements about the scalability of the system.

8.3 Future Work

The protocol in this thesis assumes a fixed number of clusters for the lifetime of the system. Moreover, it also assumes a fixed number of processes; there is no mechanism to remove or replace failed processes. Techniques exist to dynamically reconfigure Paxos [21] and could be adapted to reconfigure the system running the scalable meta-protocol with Paxos. However, it would be nice to be able to reconfigure the number of active clusters based on the current load on the system. One way to achieve this would be to have epoch markers at a fixed interval in the global slot order and have the execution nodes agree on the cluster configuration for every epoch.

Sending push NoOp notifications from cluster to cluster at a fixed interval has the advantage that it is simple to implement, but it forces a trade-off between low coordination overhead and high latencies. When the interval is set to a low value, the cost of sending these messages increases significantly as the number of clusters increases, limiting scalability. When the interval is set much higher, requests might experience high latencies when there is an unbalanced load on the clusters or when there is not sufficient load on the whole system. A more complex approach could possibly circumvent this trade-off. For instance, a cluster could try to detect when it is continually falling behind and instead of proposing only enough NoOps to catch up, it could propose enough to stay caught up for a decent amount of time. In a

production system, perhaps machine learning could be of use. The aforementioned dynamically reconfiguring the number of clusters could also help solve this problem.

Lastly, this prototype, while modular, was written only with Paxos. While Paxos is probably the most well-known and widely used agreement protocol, implementing the meta-protocol with other agreement protocols would prove the wider applicability of the meta-protocol. In particular, showing linear scaling with a Byzantine fault tolerant protocol, such as PBFT [18] or Zyzzyva [22], would obviate the usefulness of the meta-protocol.

Bibliography

[1] Fred B. Schneider. Replication Management using the State Machine Approach. *ACM Computing Surveys*, 22(4):299–319, 1990.

[2] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[3] Manos Kapritsos and Flavio P. Junqueira. Scalable agreement: Toward ordering as a service. In *Proceedings of the Workshop on Hot Topics in System Dependability*, pages 1–6, Vancouver, British Columbia, Canada, 2010.

[4] Leslie Lamport. The Mutual Exclusion Problem—Part I: A Theory of Interprocess Communication. *Journal of the ACM*, 33(2):313–326, 1986.

[5] Leslie Lamport. On Interprocess Communication. *Distributed Computing*, 1(2), 1986.

[6] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[7] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News*, 32(4):51–58, 2001.

[8] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data Systems Research*, pages 223–234, Asilomar, California, USA, 2011.

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazons Highly Available Key-value Store. In *Proceedings of the Symposium on Operating Systems Principles*, pages 205–220, Stevenson, Washington, USA, 2007.

[10] Avinash Laksham and Malik Prashant. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[11] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Dont Settle for Eventual Consistency. *Communications of the ACM*, 57(5):61–68, 2014.

[12] Seth Gilbert and Nancy Lynch. Perspectives on the CAP Theorem. *Computer*, 45(2):30–36, 2012.

[13] Eric A. Brewer. Towards Robust Distributed Systems. In *Proceedings of the Symposium on Principles of Distributed Computing*, page 7, Portland, Oregon, USA, 2000.

[14] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, pages 313–328, Lombard, Illinois, USA, 2013.

[15] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of the Symposium on Operating Systems Principles*, pages 253–267, Bolton Landing, New York, USA, 2003.

[16] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the Symposium on Operating System Design and Implementation*, pages 369–384, San Diego, California, USA, 2008.

[17] Robbert van Renesse. Paxos Made Moderately Complex. *ACM Computing Surveys*, 47(3):1–36, 2011.

[18] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the Symposium on Operating System Design and Implementation*, pages 1–14, New Orleans, Louisiana, USA, 1999.

[19] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in Scatter. In *Proceedings of the Symposium on Operating Systems Principles*, pages 15–28, Cascais, Portugal, 2011.

[20] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Googles Globally-Distributed Database. In *Proceedings of the Symposium on Operating System Design and Implementation*, pages 1–14, Hollywood, California, USA, 2012.

[21] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a State Machine. *ACM SIGACT News*, 41(1):63–73, 2010.

[22] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of the Symposium on Operating Systems Principles*, pages 45–58, Stevenson, Washington, USA, 2007.