Faculty of Science and Technology

MASTER'S THESIS

Study program/Specialization: Master of Science in Computer Science	Spring semester, 2013 Open / Restricted access
Writer: Stephen Michael Jothen	…………………………………………. (Writer's signature)
Faculty supervisor: Hein Meling	
Title of thesis: Acropolis: Aggregated Client Request Ordering by Paxos	
Credits (ECTS): 30 ECTS	
Key words: Distributed Systems Consensus Replicated State Machines	Pages: 79 Enclosure: CD Stavanger, 17 June, 2013

ACROPOLIS

Aggregated Client Request Ordering by Paxos

Stephen Michael Jothen

Contents

Contents i

Abstract iv

Acknowledgements v

1 Introduction 1
- 1.1 Contributions . . . 2

2 Background 3
- 2.1 Model . . . 3
  - 2.1.1 Processes . . . 3
  - 2.1.2 Communication . . . 4
  - 2.1.3 Synchrony . . . 5
  - 2.1.4 Failures . . . 5
- 2.2 Consensus . . . 6
  - 2.2.1 Paxos . . . 7
  - 2.2.2 Replicated State Machines . . . 10
  - 2.2.3 Multi Paxos . . . 10
- 2.3 Related Work . . . 10
  - 2.3.1 Batching . . . 11
  - 2.3.2 Mencius . . . 12
  - 2.3.3 BP Fast Paxos . . . 12

3 Design 14
- 3.1 Concepts . . . 14
  - 3.1.1 Client Logs . . . 15
  - 3.1.2 Batches . . . 16
- 3.2 Replication . . . 18
  - 3.2.1 Batch Triggers . . . 18
  - 3.2.2 Roles . . . 19
- 3.3 Communication . . . 20
  - 3.3.1 Client . . . 20
  - 3.3.2 Replica . . . 21
- 3.4 Agreement Protocol . . . 21

3.5 Algorithm 23
3.5.1 Gaps 25
3.6 Update Protocol 26
3.7 View Change Protocol 27

4 Goxos **32**
4.1 Motivation 32
4.2 The Go Programming Language 33
4.3 Modularization 33
4.4 Concurrency Approach 34
4.5 Network Communication 36
4.6 Liveness 37
4.7 Application 38

5 Implementation **40**
5.1 Client 40
5.1.1 Futures 40
5.1.2 Connections 43
5.2 Replica 44
5.2.1 Intersection 44
5.2.2 Batchpoints 45
5.2.3 Triggers 46
5.2.4 Execution 48

6 Evaluation **50**
6.1 Local Area Network 50
6.1.1 Setup 50
6.1.2 Throughput 51
6.1.3 Latency 53
6.2 Wide Area Network 57
6.2.1 Setup 57
6.2.2 Throughput 58
6.2.3 Latency 60
6.3 Workloads 61
6.3.1 Full Throttle 62
6.3.2 Randomized 62

7 Future Work **65**
7.1 Optimizations 65
7.1.1 Protocol 65
7.1.2 Structures 66
7.1.3 Goxos Architecture 66
7.2 Adaptive Batching 66
7.3 Byzantine Proposer Fast Paxos 67

7.4 Byzantine Client ACROPOLIS 68

8 Conclusion 69

Bibliography 71

Abstract

The consensus problem is one of the most central problems in distributed system. Paxos, an algorithm that solves this problem, can be used to implement replicated state machines (RSMs). By running Paxos for each of the commands received by each of the state machines, the replicas making up the system will maintain the same state. This architecture allows us to create fault-tolerant systems. This thesis introduces ACROPOLIS, which takes its inspiration from the Paxos algorithm and is used to create an RSM. In ACROPOLIS, the clients are responsible for disseminating the request content, while ACROPOLIS itself operates only on content metadata – making it a good candidate in WAN situations. ACROPOLIS also pushes the proposers in Paxos out to the clients, removing some of the leader bottleneck associated with some Paxos-variants. Initial results show that ACROPOLIS provides good time to execution latency – the time from a replica receiving a request to just before it gets executed – an indicator that it will perform well in situations where the replicas are spread over large distances.

Acknowledgements

I'd like to thank my family, for their support during my Master's degree, as well as my wonderful girlfriend Ceren for all her help. Special thanks go out to Georgia as well – for being funny.

Thanks to the distributed systems group for all the weekly meetings and help with some of the problems that sprouted up while developing ACROPOLIS. Thanks to my supervisor, Hein Meling, for always making himself available.

1 Introduction

A common problem in distributed systems is the consensus problem: Many processes in a distributed system are allowed to propose a value, and the system must ensure that only one of these values is eventually selected by every correct process in the system. In other words, no two correct processes in the system can decide on two different values.

This problem has many uses, one of the more prevalent being to implement a replicated state machine (RSM). A state machine is made up of a set of state variables which can be modified by commands given as input to the state machine. The idea behind RSMs is to have many separate state machines. In this way, if one of the state machines crashes, it is still possible for the others to handle commands. This is referred to as the *state machine approach* [23].

The state machine approach and the consensus problem go hand-in-hand: In order to ensure that each state machine making up the RSM receives the inputs in the same order, they need to coordinate among themselves and agree on an ordering for the inputs. This is exactly the consensus problem discussed above.

There are many different algorithms that solve the consensus problem, one of the more popular being the Paxos algorithm created by Leslie Lamport [15, 16]. This algorithm employs three actor types (proposer, acceptor, and learner) who play different roles all working towards the goal of reaching consensus.

The Paxos algorithm is also the basis for the Goxos framework [11]. An inital implementation of this framework was created at the University of Stavanger in the Fall semester of 2012 as a part of the project course. One of the goals of this framework is to be extensible enough to support testing of

many different variants of Paxos – including Byzantine Proposer (BP) Fast Paxos [19].

BP Fast Paxos [19] is a variant of the Paxos algorithm which can handle malicious behaviour. One of the questions posed in the original BP Fast Paxos paper is whether or not it is possible to move the proposers out to the clients – in other words a situation where the clients are the proposers. While the problem seems simple, there are many things that need to be taken into account. This problem led to the creation of a new consensus algorithm, ACROPOLIS.

1.1 Contributions

This thesis contributes the design, implementation, as well as the evaluation of the ACROPOLIS architecture, which is built on the Goxos framework [11]. ACROPOLIS attempts to provide good performance on wide-area networks (WANs). It also removes the bottleneck at the leader, which occurs in Paxos due to a majority of the communication being handled and initiated by the leader.

In Chapter 2, we begin by discussing the background necessary to understand the ACROPOLIS algorithm, including the model in which we design and build ACROPOLIS. We also go into detail about the Paxos algorithm itself. Finally we discuss some of the related works that we took inspiration from.

In Chapter 3, we go into detail about the design of ACROPOLIS. This chapter discusses all of the concepts that we utilize to design ACROPOLIS. In addition to this, Chapter 3 also discusses some of the essential algorithms and sub-protocols associated with ACROPOLIS.

We also include a chapter illustrating the basic design of Goxos. In Chapter 4 we give a brief overview of the Goxos framework [11] which helps to understand how ACROPOLIS fits in, before we delve into the implementation details.

The implementation details are discussed in Chapter 5. In this chapter, we discuss some of the problems we ran into while implementing ACROPOLIS. This chapter includes some code snippets to help the reader better understand some of the difficulties associated with implementing a consensus protocol such as ACROPOLIS.

Finally, in Chapter 6 we evaluate and discuss the ACROPOLIS protocol in two different settings. We talk about how the system performs in these settings, as well as why it performs in the way it does.

2
Background

The aim of this chapter is to provide the reader with all the necessary details in order to understand the further chapters on design, implementation, and evaluation of ACROPOLIS. First we will discuss the model within which we build ACROPOLIS, including the assumptions we make about processes, communication, and fault-tolerance. Then we will discuss the consensus problem in detail, which is at the heart of it all. Finally we will examine other works related to ACROPOLIS.

Some of the content in this chapter related to the Paxos algorithm was co-written with Tormod Erevik Lea as part of the project report in the Fall 2012 semester.

2.1 Model

When discussing distributed systems, it is useful to think about the *model* that a distributed system uses. This refers to the assumptions we make about the processes themselves, the communication medium that the processes use, which types of faults can occur to a process, among other things. This section will explore some of the options that are available when designing a distributed system, as well as explaining what kind of model we will be using in the design and implementation of ACROPOLIS.

2.1.1 Processes

A process is the most fundamental part of a distributed system. Each process in the distributed system is an actor, who has a certain role to play. The

processes are responsible for taking part in the computation of a distributed algorithm which aims to achieve a certain set goal. Usually the distributed algorithm is shared among the processes so that they are all performing the same steps, but it is also possible for different processes to be executing separate algorithms [4].

Processes are usually defined using an event-based model, wherein a process advertises its interest in receiving messages of a certain type, as well as any computation that occurs when such a message is retrieved. This computation can modify the local state of the process, send messages to other processes in the system, or broadcast to the whole system.

In addition to receiving messages from outside sources, other types of events are supported such as timer events which occur once after a set period of time, or periodic timer events which occur every so often ad infinitum.

As we will see in later chapters, ACROPOLIS utilizes this event-based reception of messages as well as timeout events.

2.1.2 Communication

A distributed system is nothing without communication; indeed the definition of a distributed algorithm is for a system made out of a set of processes that work together to achieve a common goal. To achieve this goal, communication between the processes in the system is necessary.

There are, however, many different types of communication possible between processes in a system. There is the unreliable form of communication where processes send a message and forget it, without verifying that the receiving side has actually gotten the message, and there are more advanced communication protocols that guarantee that the receiving side has received the message successfully and without any errors.

In [4], they refer to these forms of communication as *links*. For a distributed system, it is very important that messages that are sent are received at the other end. To do this, there must be an abstraction built on top of the basic communication medium that retransmits messages in the case of failure, checks that the received data is the same as what was sent, or that no duplication occurs. This abstraction is referred to as perfect point-to-point links [4]. Luckily for us, the properties required by perfect point-to-point links are provided by TCP sockets available on nearly every operating system.

In ACROPOLIS we assume that we are communicating over perfect links, which makes implementation simpler in that we don't have to worry about retransmission or many of the other problems that can occur with using a lossy protocol such as UDP.

2.1.3 Synchrony

Distributed algorithms usually have some assumptions about the synchrony of the system, in other words whether operations and communication times can be bounded by a constant amount of time. There are three different types of timing models usually talked about when discussing distributed systems: Synchronous, asynchronous, and partially-synchronous systems [4].

In a *synchronous* system, there is a certain maximum time limit for computations and communication to occur. If you send a message to another process in the system, then in a synchronous system there is a guarantee that the message will get to the receiving node within a period of time. There is also a guarantee that any computations at a node will complete within a set period of time [4].

On the contrary, in an *asynchronous* system there exists no bound on how long it takes for a communication step to occur, or how long it will take to receive a reply to a request. Furthermore, the nodes themselves can have computations that take an arbitrary amount of time to complete [4]. In further sections we will discuss the consensus problem, as well as some algorithms which solve the problem. In an asynchronous system, however, it has been proven that solving the consensus problem with at least a single faulty process is impossible [8]. In order to solve consensus we need some assumptions about the timing constraints of the system.

Real systems do not comply completely with the synchronized model. However, most operations complete within a bounded time. This is referred to as the *partially-synchronous* timing model. For the most part, the timing of a distributed system is synchronous, but there are periods of asynchrony that can occur where the timing guarantees are broken. For example there could be a guarantee about the communication time between processes in the system saying that it takes maximum 10ms to send a request and receive a response. In a partially-synchronous system this time bound can be broken, perhaps due to network congestion causing the reply to be received after 15ms. In a partially-synchronous system there can be periods of asynchronicity but it must eventually become synchronous and the periods where it is synchronous must be long enough that a substantial amount of work can be done [4].

In the case of ACROPOLIS, we assume a partially-synchronous model where periods of asynchronicity can occur, but eventually the system becomes synchronous again.

2.1.4 Failures

One of the main reasons distributed systems are so prevalent in this day and age is due to the fact that they can handle failures in the system. This fault tolerance allows the distributed algorithm to continue on working in spite of

all the bad things that can occur: Network failures, bugs in code, and even malicious behaviour exhibited by clients or servers.

There are, however, many different types of failures that can occur [1]. A failure or crash is defined as being any event that stops the process from executing as normally. This could be due to a power failure, hardware failure, or a bug in the software implementation. In this section we will discuss some of the most important types of failures.

A process is considered faulty if at some point it strays from the actual definition of what the process is supposed to do. This could be caused by things such as incorrect implementation of the algorithm, a bug in the implementation causing incorrect results, or if the process crashes. A process is considered to have crashed if it stops executing the steps of the algorithm, which could occur due to power failure, hardware failure, or a bug [4].

However, there are many models which describe different types of faulty behaviour that can occur. We will now discuss some of these categories and how they relate to ACROPOLIS.

The most common category of faulty behaviour is a crash-stop – which can occur in a process in one of two ways. The first, is the simple crash fault, where a process can crash and never return. In this case, a process is considered correct if it never crashes and continues executing forever. If it crashes then the process is considered faulty [4].

The second fault is a crash-recovery fault which occurs when a process can crash and return to system later. In this case, a process is considered faulty if it crashes and never returns or if it keeps crashing and returning ad infinitum. Otherwise the process is considered correct [4].

A third type of fault that is very important is a Byzantine fault – which can occur due to many things. Byzantine faults are the hardest faults to tolerate, and can arise due to bugs in the implementation or malicious activity from other process in the system. A process in the system can lie about what it has seen, or try and trick another process into performing an action [4].

2.2 Consensus

One of the most important problems in distributed systems is consensus: Multiple separate agents reaching agreement on a single value [28]. The consensus problem rears its head in many situations in distributed systems; chances are if there is any form of synchronization or serialization of commands, then a consensus protocol is needed. One of the most widely used and implemented algorithms that solves the consensus problem is Paxos [15, 16]. Many variants of the original Paxos algorithm exist: Multi Paxos, which extends the original Paxos algorithm to handle multiple commands; Fast Paxos [14], which attempts to decrease the total amount of message delays; and Byzantine Proposer Fast Paxos [19], which handles misbehaving clients.

2.2.1 Paxos

The name Paxos comes from the original paper describing the distributed algorithm, which is expressed in terms of a fictional parliamentary system on the Greek island of Paxos [16]. Many people found this description of the algorithm confusing, which led to Lamport creating a simpler explanation [15].

The Paxos algorithm itself has become quite popular in recent years, with many large companies and users of distributed systems having adopted it for use in their own fault-tolerant architectures. Google in particular has found a great use for Paxos, and has implemented it in their globally-distributed database Spanner as well as in their distributed lock service Chubby [3, 7]. Microsoft in addition has been using Paxos in their Windows Azure Storage architecture, which is a fault-tolerant cloud storage system. Other users of Paxos include Apache in ZooKeeper, a popular distributed configuration management system, and Heroku in Doozer, a distributed data store [5, 10, 20, 21].

As any distributed algorithm, Paxos is made up of a network of replicas. Each replica in the system has the responsibility of performing one or more of three possible roles. The first role is the proposer. The proposer is the only role that has contact with clients to the system. The other roles – the acceptor and the learner – do not communicate directly with clients but only with other replicas. It's also important to note that a replica can assume a subset of any of the three roles, including every role if need be [15].

Another thing to note is that Paxos relies on the use of *quorums* to ensure consistency between the replica set. Quorums are a very powerful tool when it comes developing distributed algorithms. A quorum is defined as being a subset of replicas fulfilling the following two properties: All non-faulty replicas can form a quorum, and any two such quorums intersect in at least one replica. Now, we can begin discussing the actual details of the Paxos algorithm.

The actual Paxos algorithm itself, sometimes referred to as the Synod algorithm, is built out of two phases, where each phase contains two parts. For simplicity's sake we will assume that there is a single leader or coordinator who is responsible for handling the requests from clients and getting the requests accepted. This is required for Paxos to guarantee progress [15]. The properties for Paxos are separated into two categories, liveness and safety. The safety properties state that the algorithm never does anything wrong, while the liveness properties state that eventually something good happens [4]. These properties are as follows, where CS is a safety property and CL is a liveness property [15]:

CS1 Only a proposed value may be chosen.

CS2 Only a single value is chosen.

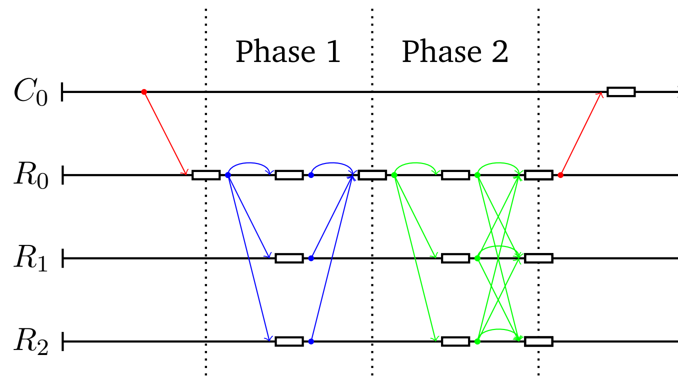


Figure 2.1: A sequence diagram showing the client interaction (sending the request and receiving a reply) as well as the two phases of Paxos that take place. The first phase consists of the leader (R_0) broadcasting a PREPARE*, and then receiving a* PROMISE *from the replicas. The second phase consists of the leader broadcasting an* ACCEPT*, and then an all-to-all* LEARN *broadcast takes place. Note that in this case, each replica takes on all roles.*

CS3 Only a chosen value may be learned by a correct learner.

CL1 Some proposed value is eventually chosen.

CL2 Once a value is chosen, correct learners eventually learn it.

Now we can discuss the actual Paxos algorithm itself. The action begins when a client sends a request to the leader, as shown in Figure 2.1.

Phase 1a – PREPARE

The leader initiates the algorithm by broadcasting a PREPARE message containing the leader's current round n.

Phase 1b – PROMISE

When an acceptor receives a PREPARE message, it checks if it has seen any previous PREPARE or ACCEPT messages with a round number greater than n. If it has, then it ignores the message. If it hasn't then it responds to the leader with a PROMISE message, promising not to respond to any PREPARE messages with a lower round number. This PROMISE message contains the round in the original PREPARE message, as well as any round and value that this acceptor previously voted for.

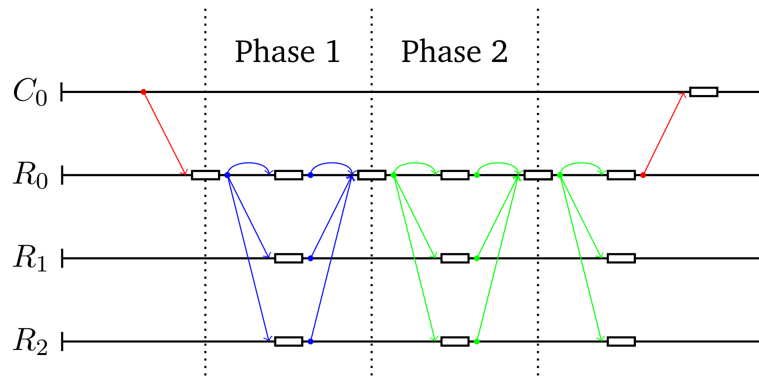


Figure 2.2: A sequence diagram similar to the one in Figure 2.1, except with a distinguished learner (who also happens to be the leader). This optimization can be useful if there are a large amount of replicas in the RSM as it cuts down on the amount of messages. However, it introduces a message delay after the second phase due to having to inform the other learners of a value being chosen.

Phase 2a – Accept

In this phase the leader is receiving Promise messages from the acceptors. When the leader receives a quorum of Promise messages for round n, it sends an Accept message to the set of acceptors containing n and the value associated with the highest previous voted round in the Promise messages, or the original request from the client if no acceptor has previously voted.

Phase 2b – Learn

In the final phase, when an acceptor receives an Accept message with n greater than or equal to any it has previously received in a Prepare message, it broadcasts a Learn message containing n and the value associated with it to the set of learners.

When a learner receives a quorum of these Learn messages, it signifies that the command can be executed.

Optimizations

There are many possibilities for optimizing the Paxos algorithm, including using a single, distinguished learner (as shown in Figure 2.2) who informs all other learners if it receives a quorum. This reduces the number of messages transferred over the network. Also note that a proposer only needs to send its Prepare message to a quorum of acceptors, as long as the quorum is functioning correctly. The same goes for the Accept message broadcast from the proposer.

2.2.2 Replicated State Machines

While a consensus protocol such as Paxos is used to decide upon a single value, it can also be used to implement a replicated state machine (RSM) using the state machine approach [23]. In this approach, we have a distributed system made up of many separate replicas which are connected over a network. The RSM takes input from clients in the form of commands, which modify the state of the system. The RSM then reports back to the client with the result of running the command.

Paxos is required because each replica has to run the commands in the same order, so as to keep the same internal state, thus keeping the state of the whole RSM consistent. It's easy to see that if two replicas execute different commands in a different order (assuming the commands are non-commutable), the state will end up different on both of the replicas, leaving the RSM in an inconsistent state.

In order to solve this we need to extend the Paxos algorithm to Multi Paxos – a variant of Paxos that supports running a sequence of commands instead of just a single command.

2.2.3 Multi Paxos

The way to extend the Paxos protocol to handle multiple commands from clients, and thus implement an RSM is outlined in [15]. The simplest way to do this is to run multiple *instances* of Paxos. In this way, if the leader receives a command, it can just create a new instance and assign the command to it. The same steps used in Paxos are also used within each instance of Multi Paxos.

This also assumes that, since a client request could be received while previous instances of Paxos are still running, there needs to be an additional rule to ensure that the command associated with instance $i + 1$ cannot be executed until all the previous instances 0, 1, ..., i have been decided and executed first [15].

There are also further optimizations that can be used to make Multi Paxos perform better. One of these optimizations is to eschew the first phase altogether. The first phase is only needed to ensure there is a single leader. When a leader is elected, it can run the first phase for an infinite number of instances, and not have to worry about the first phase at all. When the leader receives a request from a client, it can directly send an ACCEPT message to the RSM. This cuts down the number of message delays greatly [15].

2.3 Related Work

In this section we will discuss works that inspired ACROPOLIS as well as works that have similar goals or properties as ACROPOLIS.

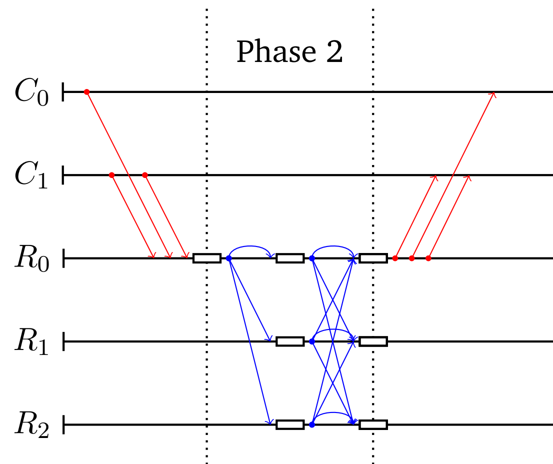


Figure 2.3: An example of how a batching implementation would work. Note that each of the client requests doesn't trigger its own Paxos instance, but instead the leader waits for a sufficient number of requests and puts them into a batch. Then a single instance of Paxos is run for all three requests. We assume the leader is stable in this case, which is why there is no first phase of Paxos.

2.3.1 Batching

Batching is an important optimization for Paxos-based RSMs. It is one of the simplest optimizations to implement, and also provides one of the best performance boosts [22].

Batching works by having the leader collect a series of client requests and *batching* them into a single batched request. The leader can then treat these requests as a single value and runs an instance of Paxos to get the batch executed. To execute a batch, the replica can simply execute each request inside the batch in some specified order [22].

As mentioned in [22], the main difficulties with batching is tuning the parameters needed to batch. How many requests should be in a batch? If the batch size is too big, it may not get filled up. Conversely, if the batch size is too small, then you don't get as much benefit in throughput from batching. One way to solve this issue is to introduce a timeout, where if the batch isn't filled up within a certain time limit, the commands that have been received are still packed up in a batch and agreed upon with Paxos.

The important point to note about the batching, as discussed in [22], is that the leader is the one who is responsible for collecting the requests and getting them accepted. As we will see later, in ACROPOLIS, the client broadcasts to the RSM and the whole replica set does the batching operation.

2.3.2 Mencius

Mencius is a protocol for creating a replicated state machine [17], which shares much the same goals as ACROPOLIS does. Just like ACROPOLIS, Mencius aims to have good WAN performance, and it does this by trying to remove the bottleneck of the leader.

The Mencius protocol takes a lot of its inspiration from Paxos. However, in the Paxos protocol there is a single leader who is responsible for ordering incoming commands and getting them accepted. Conversely, in Mencius the replicas are numbered R_0, R_1, $\ldots$, R_n. Then the leader of the system is chosen based on the instance number. For example, for instance 0, R_0 will be the leader, for instance 1, R_1 will be the leader, and so on.

One of the main goals of Mencius is removing the bottleneck of the leader [17]. By this they refer to the fact that in Paxos, a large proportion of the data has to go through the leader. Client requests all must go through the leader who is responsible for sequencing them. The leader is also responsible for processing quorums of messages from the replicas. By rotating the leader in the fashion done by Mencius, the workload gets spread out and each replica gets its turn in doing the sequencing of client requests, which can lead to greater network utilization [17].

2.3.3 BP Fast Paxos

In the traditional Paxos algorithm, processes are allowed to crash and then recover [15]. However, this crash-recovery model doesn't take into account the real world failures that can occur due to, for example, malicious users of the system.

Byzantine Proposer (BP) Fast Paxos [19] is a Paxos-based consensus protocol for a hybrid failure model where the clients and proposers (also called proxies) can be Byzantine faulty and the servers (acceptors and learners) are crash faulty.

The architecture of the BP Fast Paxos system is split up into multiple layers. The replicas are assumed to be in the same administrative domain, and thus they can trust one another more than they can the clients or proxies who are outside of the domain or at the edge of the domain respectively. The proxy nodes are located at the boundary of the cloud, and interact with the clients who can be malicious. Since the proxies communicate directly with the clients it is possible for them to be compromised. The replicas (acceptors and learners) within the cloud assume that the proxies may be Byzantine faulty due to their interaction with the clients. BP Fast Paxos uses a hybrid failure model due to this dichotomy between the insider and outsider nodes [19].

While keeping servers safe from malicious, possibly compromised proxies, BP Fast Paxos still manages to offer a low latency for client requests. It can

protect from things such as denial of service attacks waged by the clients or the proxies. It does this while tolerating $1/3$ crash faulty servers, and any number of Byzantine faulty proxies.

One of the questions posed in the original [19] paper is whether or not the proposers can be pushed out to the clients. In other words if it is possible for the clients to be proposers themselves. However, Paxos usually requires there to be a single leader who proposes commands in order to ensure progress [15]. If there are multiple proposers proposing at the same time, it is very difficult to figure out how all the proposers can sequence their commands into Paxos instances so that there are no collisions with the other proposers.

This problem with sequencing of commands is one of the problems that lead to the development of ACROPOLIS. With ACROPOLIS, the clients themselves are given the responsibility of disseminating the request content to the RSM which is stored in a fashion that doesn't require the leader to do any sequencing of commands itself.

3 Design

As with most distributed systems, the underlying design of ACROPOLIS is non-trivial. In this chapter we aim to closely examine the structure of ACROPOLIS starting from the basic design of the protocol and all of the structures surrounding the actual algorithm. After having covered the basic protocol and concepts, we will move on to discussing the changes needed to support view change, which occurs when the leader crashes.

3.1 Concepts

Some of the main goals of ACROPOLIS are to be able to solve the consensus problem in situations where replicas are connected over a wide-area network (WAN) in a high-performing manner, as well as to remove the responsibility of the leader to have to sequence incoming requests from clients, which is done in Paxos. This is where most of the intuition for the design of ACROPOLIS came into focus.

Algorithms such as Fast Paxos [14] try to remove the leader from most of the computation by having the clients broadcast to the replica set. However, because there is no leader to apply sequence to the requests, collisions can occur where replicas see the requests in different orders. ACROPOLIS attempts to solve this by using a combination of client request multiplexing and batching to efficiently agree on what all the replicas have seen.

There are many consensus protocols in existence, but most of them concentrate on getting the best possible performance in a local-area network (LAN) setting. With ACROPOLIS we are more interested in creating a consensus protocol that can be used in a WAN setting.

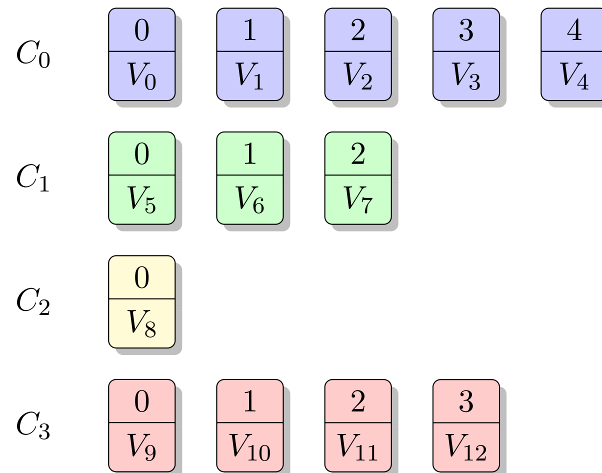


Figure 3.1: A visual representation of the client map. Each C_i is the client ID and everything to the right of the client ID is the client log dedicated to that client. Note that all requests are stored by the sequence number. The bottom field (V_j) is the actual value the client is proposing. In this instance, the clients have sent 5, 3, 1, and 4 commands respectively.

Now we will begin discussing the concepts needed to understand the ACROPOLIS protocol.

3.1.1 Client Logs

Due to the nature of network communication it is difficult to reason about ordering of commands broadcast from clients – specially in cases of heavy load where many clients are broadcasting to the replica set.

In Goxos [11], the client IDs are uniquely generated using a cryptographically secure hash function. This allows us, with very high probability, to consider each client ID as being unique. Therefore, it's possible to multiplex each of the requests into a specific log dedicated to each client in the system. In ACROPOLIS terminology, we call each of these a *client log*.

By doing this, we no longer have to worry about one client's requests interfering or colliding with another client's requests, such as the case in Fast Paxos [14]. To do this, we use the ID in the client request to figure out which client log the request belongs in, and then within each client log we use the sequence number to figure out which position the request should be stored. Note that it is possible for the entries of a client log to be empty in the case the client's message is lost.

Of course, in a system with many clients sending requests, we will have more than a single client log. The set of all client logs is referred to as the *client map*, and is one of the central data structures of ACROPOLIS. The most central data structure, however, is the batch.

3.1.2 Batches

Our main goal is to implement an RSM and get the same commands to run on each of our replicas in the same order. So far we've discussed how to avoid collisions by multiplexing the client requests into separate logs, but this still won't help us because we need to provide an ordering to all of these commands.

Take the case in Figure 3.1 for example. In this case, it would be simple to say we should run all the requests from C_0 ordered by their sequence number, and then all the requests from C_1, and so on for all the clients. Another option would be to order all requests with sequence number 0 ordered by client ID, then all requests with sequence number 1, and so on. Both of these techniques will work, but there is still one problem: We need to communicate to the other replicas that we wish to execute some commands. We can't wait indefinitely to see if there are more requests on the way from a client.

This is where the second important data structure comes in, which we will refer to as the batch. This structure is not to be confused with batches as referred to in [22], however, it is somewhat related.

Before we get into the specifics of a batch, it is useful to describe a client range. There will come a time when we have received a bunch of requests from a client, and they need to be executed. We can use a client range to describe this set of requests, which describes the starting sequence number and stopping sequence number. In other words, a client range is simply an inclusive range $[a, b]$ for a specific client.

A *batch* is then simply a set of client ranges, one for each client we wish to include in the batch. We receive a different amount of requests from each client, so it's highly likely that the batch can look similar to Figure 3.1 with some jagged lines denoting the ranges.

As shown in Figure 3.2, a batch can be constructed for a set of client ranges that the current replica has seen, while further requests (received after the batch was created) are placed outside of the batch. In this case, these commands are V_7 from C_1 and V_{10-12} from C_3. These commands can be included in the next batch. The client ranges for the batch shown in Figure 3.2 are $[0, 4]$, $[0, 1]$, $[0, 0]$, and $[0, 0]$ respectively for clients C_{0-3}. After this batch was made, requests were received from both clients C_1 and C_3.

Now, let's assume that we receive some additional requests from client C_2. These requests will modify the client map, but will only affect the client log for client C_2. Note that there is no modification of batches, a batch is simply a snapshot of ranges inside the client map.

As shown in Figure 3.3, a batch will not necessarily contain a client for every range that the system has seen, but only a range for clients that we have received requests from since the previous batch was created. After all, there is no point in including empty ranges for clients who are idle.

With Figure 3.3, it becomes easier to see how the actual batches can be

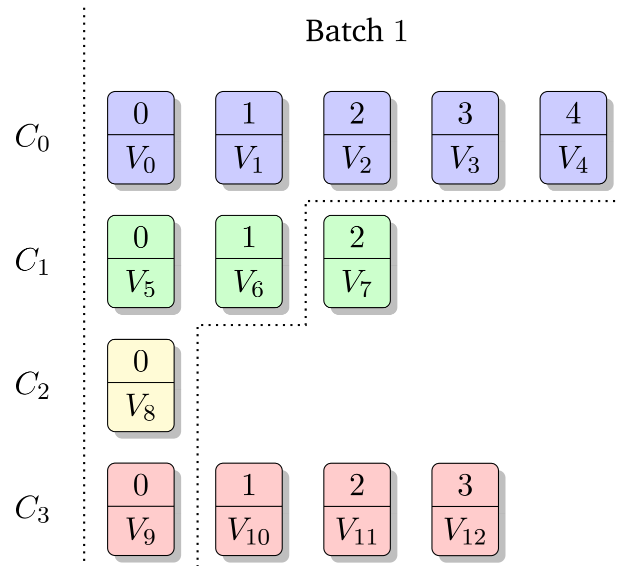


Figure 3.2: A visual representation of a batch, a set of client ranges. The first dotted line on the left is the beginning of each of the client ranges, and the jagged dotted line on the right is the end of each range.

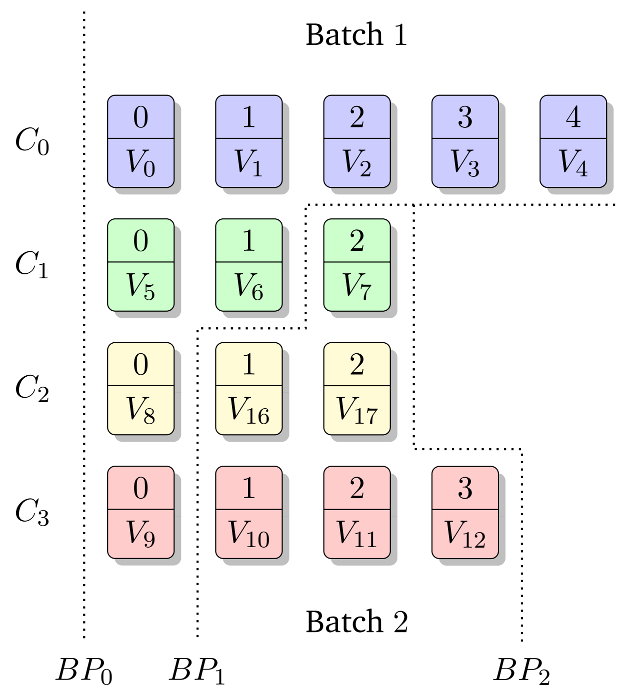


Figure 3.3: Snapshot after two batches have been created. Note that client C_2 has sent two new requests since the last batch operation. In this figure we also show the demarcating batch points (denoted BP_0, BP_1, and BP_2).

Client	BP_1	BP_2
C_0	4	4
C_1	1	2
C_2	0	2
C_3	0	3

Table 3.1: Table describing batch points BP_1 and BP_2.

constructed. We first note that a batch contains all the requests between the previous batch point BP_i and the next batch point BP_{i+1}. In other words, batch B_{i+1} contains all the requests between BP_i and BP_{i+1}. Thus, checking if a batch can be made, all we need to do is see if there are any requests between the two batch points. If there are, then we can successfully batch, otherwise we must wait.

In the specific case of Figure 3.3, we see that our batch points BP_1 and BP_2 look as described in Table 3.1. From this description it is easy to see that the values for C_1, C_2, and C_3 are greater than their previous values in BP_1, thus we can create a batch containing all these clients' requests after BP_1. That's to say for client ranges $[1+1,2]=[2,2]$, $[0+1,2]=[1,2]$, and $[0+1,3]=[1,3]$ for these clients respectively.

3.2 Replication

Now that we have covered the major concepts we need to build ACROPOLIS, we can talk about the most important objective: Replication. The whole goal is to create an RSM that maintains the same state in spite of all the potential problems. In this section we will discuss some of the things that need to be solved in order to use the aforementioned concepts in a replicated manner.

In our architecture we can't rely on the leader to serialize the commands as is done in Paxos [15]. However, we still require a leader to initiate the ACROPOLIS protocol in order to reach consensus. The clients are expected to broadcast their requests to all of the replicas in the system. However, we still need to agree on requests that are to be executed in a batch.

The first key to understanding the ACROPOLIS protocol is to understand how the different types of initiation occur that allow the non-leader replicas to inform the leader with information about which requests they have received. We refer to this initiation as a *batch trigger*.

3.2.1 Batch Triggers

When does the leader get informed about the state at each of the replicas? The operation could conceivably be done for every single client request, or

during periods of inactivity where a replica doesn't receive any command for a certain period of time. However, the most sensible choice is to make all replicas do a batching operation after a set period of time, ad infinitum.

This will solve the problem, but is not a good solution since there can be periods where the replica set is receiving client requests at a very high rate, allowing the batch size to grow enormously. Another drawback is the fact that periodic batching can result in slow response times for the client requests. Imagine if we set the batching period to 1 second. The clients who send a request immediately after the previous batching operation completes will have to wait at least 1 second to receive a response.

The best solution is a hybrid solution where we use the periodic batching solution above, but we also set a maximum size for the batch. If we receive a certain number of requests since the last batching operation, then we should batch immediately and not wait. This can help the efficiency of the system during periods of high activity.

Another possible solution is an extension of the hybrid solution discussed above. Instead of having a constant maximum size for a batch, it can adapt to the current conditions. If we receive a lot of requests very quickly, it could increase to take this into account. If we receive only a single request and then timeout, then the maximum batch size could shrink to decrease the round-trip time in cases of low activity.

3.2.2 Roles

There is one main role for each actor in the system, an acceptor. However, it is possible for an acceptor to be elected as the leader of the system, we will call this replica the leader acceptor or simply the leader.

Leader Acceptor

The leader's main job, as hinted at in previous sections, is to coordinate the information obtained from all the replicas and figure out which commands can be executed. In order to do this, it keeps track of which commands the whole system has, in terms of batches and their associated batchpoints. When the leader finds out that there are commands that can be executed on a majority of the replicas, it is tasked with informing the RSM to run these commands.

Acceptor

The role of the acceptor is very simple. An acceptor receives requests from clients, and appends them to the local client map. It then uses the batch trigger techniques described above to inform the leader that it has new

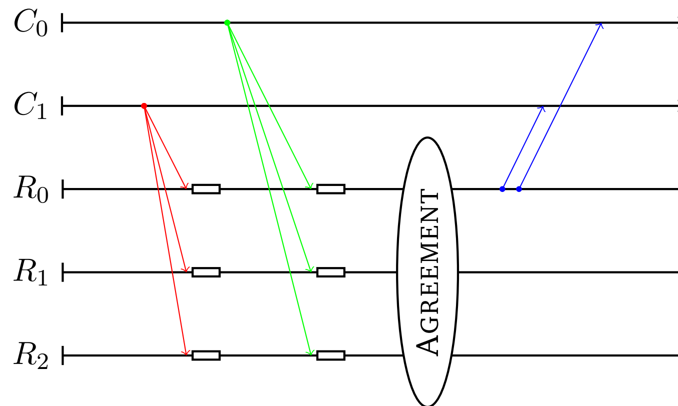


Figure 3.4: A time sequence diagram showing clients C_0 and C_1 broadcasting their requests to each replica in the RSM. In this case, R_0 is the leader of the RSM. These requests trigger a batching operation and the associated agreement protocol. Also shown is the response from the leader back to each of the clients.

commands that should be taken into account. Either the replica will hit the threshold of commands it can accept in a batch, or a timeout will occur.

3.3 Communication

What is the point of a distributed system without communication? The clients of course need to communicate their desire to run a command on the RSM, and the replicas need to communicate between themselves to ensure safety of the system. In this section we will explain the client and RSM communication patterns, as well as some of the potential problems that can occur.

3.3.1 Client

The client communication pattern is very simple: Each client in the system maintains a network connection to all of the replicas in the RSM. When a client wishes to run a command on the system, it broadcasts the request to all of the replicas.

When the agreement protocol comes to an end, and a batch has been decided upon, any client requests in the batch will be executed and then replied to with the response from the application. This is shown in Figure 3.4 as a blue line from R_2. In this case, since R_2 is the leader, it is the only replica that sends the response to the client, to avoid duplicate responses.

3.3.2 Replica

Communication between the replicas that make up the RSM is equally as important as the client-replica communication described above.

The replicas in the RSM are pairwise connected and maintain a full-duplex communication channel, allowing replicas to communicate with one another. The two communication patterns supported are a unicast send to a single replica, and a broadcast to every replica in the system (including the replica broadcasting itself).

Using the unicast and broadcast primitive is essential in creating the actual agreement protocol which we will now go into further detail about.

3.4 Agreement Protocol

Now that we have covered the major concepts and had a discussion about who initiates batching and when, we can begin to discuss how the replicas agree on the requests in a batch.

The agreement phase of ACROPOLIS is really at the heart of it all. This phase is needed to ensure that all the replicas agree on the same set of requests and execute them in the same order.

In this section, we will explain how the basic agreement protocol works, through discussing in-depth the communication that takes place between all of the replicas, in addition to some of the problems that can occur and how we overcome them. First, we will go into detail about what actually occurs when a replica creates a batch, and what it does with it.

There are two main message types that make up the basic ACROPOLIS agreement protocol: BATCHLEARN and BATCHCOMMIT. Each BATCHLEARN message contains a batch ID, signifying which batch this message belongs to and a mapping from client ID to client range telling the leader which ranges this replica has seen. Each BATCHCOMMIT message contains a batch ID and a mapping from client ID to client range signifying which ranges a majority of the replicas have, and thus can be executed.

When a replica initiates batching, the first thing it does is compute the necessary information to send to the leader. The leader needs to know which requests the replica has seen, so the replicas must send a BATCHLEARN message to the leader containing the ranges in the batch.

If we take Figure 3.3 as an example of the state at one of the replicas, and a trigger occurs to initiate batching and sending this information to the leader, the subsequent BATCHLEARN message looks as in Table 3.2. Note that the replica only sends a range for those clients for which it has received new requests from.

On reception of a quorum of BATCHLEARN messages associated with batch B_i, the leader will take the intersection of all the messages received.

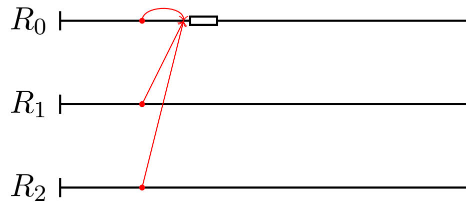


Figure 3.5: The leader must collect a quorum of BATCHLEARN *messages in order to get a picture of what the whole RSM state is.*

Client	Range
C_1	$[2,2]$
C_2	$[1,2]$
C_3	$[1,3]$

Table 3.2: A possible BATCHLEARN *message sent to the leader. Client C_0 isn't included because this particular replica hasn't received any new requests from it since the last batch.*

In other words, for each client range that exists in all of the messages that make up the quorum, it will take the intersection of these ranges to figure out what the quorum has in common. Once it does this, it will know that these client intersections are able to be executed on the quorum.

Once the leader finds the intersection of the client ranges in the quorum of messages, he can broadcast a BATCHCOMMIT message to all the replicas as shown in Figure 3.6.

When a replica receives a BATCHCOMMIT message, it can execute the client ranges contained in the message, as long as it applies a correct ordering to them. Within a BATCHCOMMIT message, there is a mapping from client IDs to ranges. One possible way of applying an ordering to these is to use the rule that you must execute the range from client C_0, then C_1, and so on until C_n. However, there may be missing client IDs if a client hasn't sent a

Client	Range R_0	Range R_1	Intersection
C_1	$[0,3]$	$[1,3]$	$[1,3]$
C_2	$[1,4]$	$[1,2]$	$[1,2]$
C_3	$[0,5]$	$[1,7]$	$[1,5]$

Table 3.3: Computing the intersection of client ranges at the leader. If there are 3 replicas in the system and R_2 is the leader collecting these messages, R_0 and R_1 make up a quorum and the leader can computer the intersection from these.

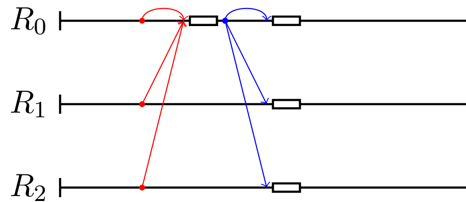


Figure 3.6: The full agreement phase of ACROPOLIS. *Includes initial collection of* BATCHLEARNs *from a quorum of replicas, and the subsequent* BATCHCOMMIT *telling the replicas to execute a set of client ranges.*

request since the last BATCHCOMMIT message. To solve this, we just sort by the client IDs and execute in that order.

Within each client range, the order is set by the range itself. For example, if client C_i has a range $[a, b]$ then we execute requests in the order a, $a + 1$, $a + 2, \ldots, b$.

3.5 Algorithm

In this section we will take a look at a very basic implementation of the ACROPOLIS algorithm in pseudocode. To limit the complexity of the pseudocode, it will just be describing the operations for a single batch. After this section, we will discuss some of the problems that arise with this simple algorithm, and how to modify it to handle these exceptional circumstances.

One thing to note is that the leader acceptor has the same role as all the other acceptors, with the additional role of being the leader. Therefore, the events that occur on a normal acceptor will also occur on the leader as well. In other words, the leader also handles client requests just as the normal acceptors do in Algorithm 2. The leader also handles BATCHCOMMIT messages as well as handling inactivity in the system through using timeouts.

As can be seen in Algorithm 1, the leader acceptor's main responsibility is the handling of the BATCHLEARN messages from all of the acceptors (including itself). On Line 17 the leader checks whether it has received a quorum of these BATCHLEARN messages. If it has then it has a good view of what the state of the whole RSM is. It can then intersect each of the client ranges, and broadcast a BATCHCOMMIT containing the ranges that are common for the whole quorum. This is done on Line 19.

The helper routine on Line 8 is responsible for computing the intersection of the client ranges. It does this first by using another helper routine on Line 1 which first checks that a majority of the responses contain ranges for a particular client. All of these clients that we have a majority of responses for will be used in the computation. This intersection is computed on Line 12 by finding the largest range that all of the messages in the quorum contain.

Algorithm 1 Event handling procedures for the leader acceptor

```
 1 procedure FindMajorityClients(R)
 2     count ← {}
 3     for r ∈ R do                                  ▷ For each client range mapping
 4         for c ∈ r do                              ▷ For each client id
 5             count[c] ← count[c] + 1
 6     return {c | count[c] = ⌊N/2⌋ + 1}             ▷ Majority have seen this client
 7
 8 procedure IntersectRanges(R)
 9     maj ← FindMajorityClients(R)
10     int ← {}
11     for c ∈ maj do                                ▷ For each majority client
12         int[c] ← R_0[c] ∩ R_1[c] ∩ . . . ∩ R_n[c]
13     return int
14
15 event BatchLearn(R)                               ▷ R: Client range mapping
16     MV ← MV ∪ R
17     if |MV| ≥ ⌊N/2⌋ + 1 then                      ▷ Quorum exists
18         i ← IntersectRanges(MV)
19         broadcast BatchCommit(i)
```

Algorithm 2 Event handling procedures for acceptors

```
 1 event Timeout                                     ▷ Inactivity timeout
 2     batch ← GetBatch()
 3     send BatchLearn(batch) to L
 4
 5 event ClientRequest(C_i, V)                       ▷ Request received from client
 6     AddToClientLog(C_i, V)
 7     bs ← BatchLimitHit()
 8     if bs then                                    ▷ If we hit the batch size limit
 9         batch ← GetBatch()
10         send BatchLearn(batch) to L               ▷ Send to leader
11
12 event BatchCommit(R)                              ▷ R: Client range mapping to commit
13     sk ← Sort(R)                                  ▷ Sort client IDs
14     for c ∈ sk do
15         Execute(R[c])                             ▷ Execute the range for this client
```

In Algorithm 2, the main things of interest are what occurs during inactivity as well as the reception of the client requests. When a request is received on Line 5 it is put into the client log structure and then there is a check to see

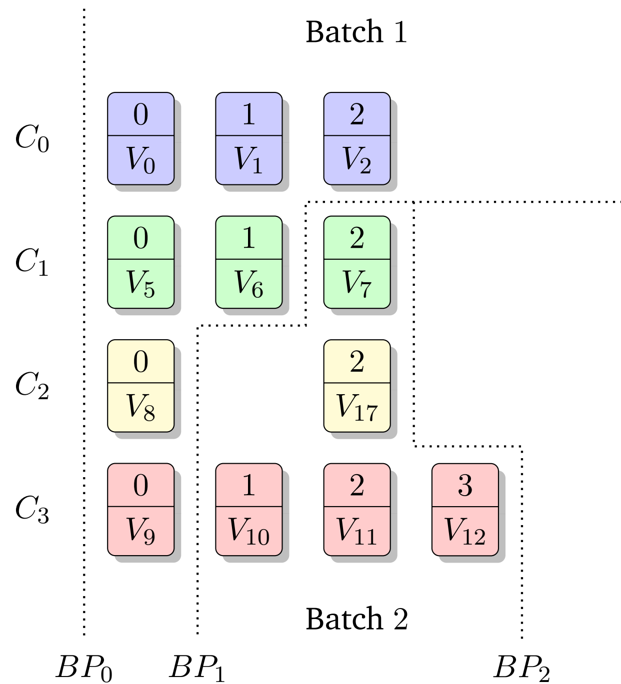


Figure 3.7: Gaps can occur in a client map which need to be taken care of as a part of the agreement protocol. In this case, clients C_0 and C_2 are exhibiting faulty behaviour since this replica has gaps for these clients. Client C_0 did not successfully broadcast requests with sequence numbers $3, 4$ and client C_2 did not successfully broadcast a request with sequence number 1.

if we've seen enough requests since the last time we created a batch. If we have, it means we've hit the limit on the batch size and send a BATCHLEARN to the leader. A similar thing occurs when a timeout event is received on Line 1. In this case, the batch is created due to a period of inactivity which has occurred.

Finally in Algorithm 2, we can receive a BATCHCOMMIT message from the leader which tells us to execute a set of ranges due to these ranges being common between the quorum. This execution occurs on Line 15. Also note that the client IDs in the BATCHCOMMIT must be sorted to ensure that all of the replicas execute in the same order.

3.5.1 Gaps

One problem that hasn't yet been discussed is the problem of gaps. A gap can occur in a client range if it is acting maliciously or the client is faulty. In this case a replica may have a client log similar to the one shown in Figure 3.7.

The protocol can be simply modified to take these gaps into account. A replica will be able to tell if it has a gap in a client range simply by checking

Client	Gaps
C_0	$\{3, 4\}$
C_2	$\{1\}$

Table 3.4: The gap information contained in a possible BATCHLEARN *sent to the leader. The gaps shown are the same as shown in Figure 3.7.*

each entry in the client log. If it finds a gap in the range then it can notify the leader about this gap in the BATCHLEARN message.

The leader, upon receiving a quorum of these BATCHLEARN messages will do the usual consolidation by intersecting the ranges, but it must now take the gaps in the message into account as well. If one of the replicas has a gap in one of the client ranges, we can do one of two things. If the leader himself has the command to fill the gap, he can send this along in the BATCHCOMMIT message. However, if the leader also has the gap, or the leader has a different gap himself, this won't be possible. In this case the leader cannot tell the RSM to execute the gaps. It can do this by including gap locations in the BATCHCOMMIT message.

Each BATCHLEARN message in addition to the client ranges can also have a gap map which maps from client ID to a set of gaps for a specific client. In the case of Figure 3.7, this map would include entries for C_0 as well as C_2. When the leader receives a quorum of these messages, it can take the union of all the gap sets for each client ID in the quorum. The leader can then check and see if it can fill any of these gaps itself. If so, these requests can be sent along as a part of the BATCHCOMMIT message. If it can't, then it has to include the gap unions it has in the BATCHCOMMIT so that the replicas don't execute these gaps.

Then, upon receiving a BATCHCOMMIT message, a replica can execute the ranges as normal, except if there is information about a gap, which must be skipped for execution.

3.6 Update Protocol

There is a problem when the leader sends the BATCHCOMMIT message to the RSM. This is the case where the leader sends a BATCHCOMMIT message that contains a range that a replica doesn't currently have, or doesn't have a portion of it.

Consider the case where there are three replicas in the system, R_0, R_1, and R_2. Suppose R_0 is the leader and receives two BATCHLEARNS, one from R_0 and one from R_2 (R_0 receives a message from itself). The leader will use these two replicas' views of the state and find the intersection of what they both have. However, this doesn't take into account the state at replica

R_1, who could not be as up to date as the other replicas that made up the majority.

This is a problem, because the BATCHCOMMIT cannot be executed until the replica has all the commands defined by the client ranges in the message. Thus, we need a sub-protocol that can handle this case and bring the replica up to date so that it can execute the batch.

To do this we introduce two new message types, UPDATEREQUEST and UPDATEREPLY. An UPDATEREQUEST is a request from one replica to another asking for certain client commands to be filled. The corresponding UPDATEREPLY message contains the requested client commands.

3.7 View Change Protocol

In this section we will discuss the changes needed to the basic ACROPOLIS protocol above if we wish to support view changes.

As with Paxos and many other distributed systems, there is always the possibility of faulty behaviour. We require a number of different replicas in ACROPOLIS for a specific reason – fault tolerance. If a replica crashes, as long as we have a quorum of replicas alive and working, the system should still work as advertised.

In ACROPOLIS, if a replica crashes, it doesn't matter as long as the replica is not the current leader of the RSM. If it is the leader, then we have to do a *view change* where a new leader is elected and is informed about the current state of the RSM. This is done in a manner similar to how view change is usually implemented in Paxos. In Paxos, when a view change occurs, replicas send information about their state (including slots that aren't fully decided) to the new leader who can complete these slots and continue the execution of the RSM. However, there is a bit more non-trivial work needed to be done by ACROPOLIS.

To illustrate the pitfalls that occur when we use the exact same view change procedure as Paxos, an example is in order. Suppose we have an RSM containing 4 replicas. Also suppose all batches up to i have been successfully completed – the leader has received the desired BATCHLEARN messages, and sent the BATCHCOMMIT messages, and at least one of the replicas has received this message. Now suppose in batch $i + 1$, the leader receives the BATCHLEARN messages but then crashes. Then there are a few possibilities.

If any of the remaining replicas received a BATCHCOMMIT for batch $i + 1$, then the new leader can simply rebroadcast this BATCHCOMMIT to the RSM to ensure that everyone got this message.

However, if none of the replicas have received a BATCHCOMMIT for batch $i + 1$ but they have sent BATCHLEARNs to the leader, there is a problem. In this case it is possible for the leader to have executed something and crashed before sending the BATCHCOMMIT message to the RSM. The new leader is

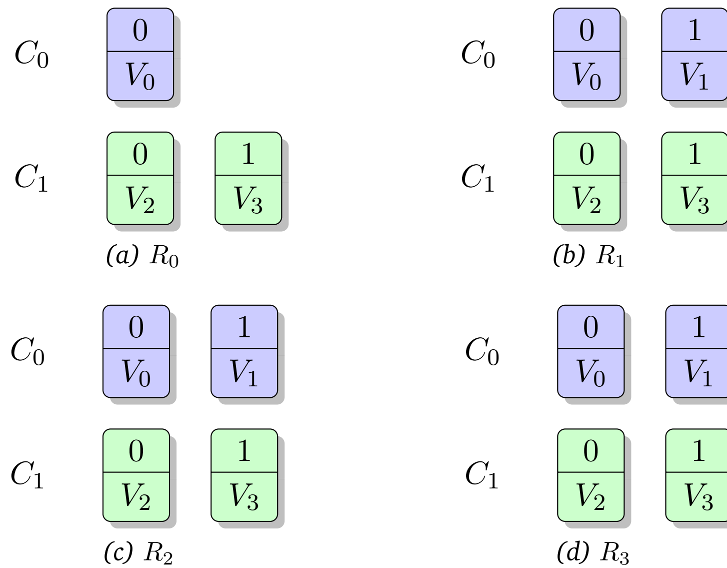


Figure 3.8: The batch information sent from each of the four replicas to the leader (R_0).

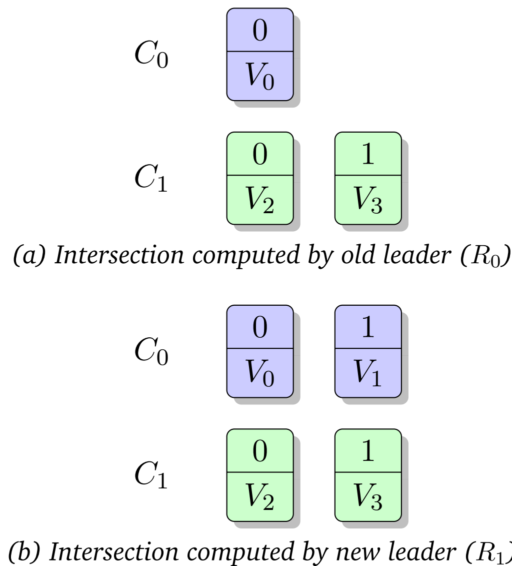
then tasked with figuring out what the leader *could* have possibly executed. It is possible the leader didn't execute anything at all, but this is unknown.

One potential solution – which doesn't end up working – is to do the same as what is done in Paxos. The new leader requests the BATCHLEARN messages that each of the replicas has sent, and when it receives a quorum of them, it can do the usual intersection and send a BATCHCOMMIT for the intersection. This doesn't necessarily work if the new leader receives a different quorum than the previous leader. In this case the new leader could have a different intersection, possibly resulting in a different ordering than the old leader.

Take for example Figure 3.8. Suppose that R_0 is the leader, and he gets a quorum of 3 messages from R_0 (itself), R_1, and R_2. From these three messages he is to do an intersection. In this case the intersection is the same as what R_0 itself has in Figure 3.8(a). R_0 will propagate this intersection to the whole RSM by broadcasting a BATCHCOMMIT. However, suppose that R_0 gets its own BATCHCOMMIT message, and then crashes, and this occurs before it has a chance to send the BATCHCOMMIT to any of the other replicas.

At this point, a new leader will be elected. Suppose that R_1 is the new leader. R_1 will request the last BATCHLEARN messages for any outstanding batches, and any BATCHCOMMITs if possible. If it sees a BATCHCOMMIT for an outstanding batch, then it can do a retransmission of the message since this implies a previous leader having committed. If it doesn't see any BATCHCOMMITs then it has to use the supplied BATCHLEARN messages from the remaining replicas R_1, R_2, and R_3.

The new leader R_1 will wait for the BATCHLEARN messages for the out-



(a) Intersection computed by old leader (R_0)

(b) Intersection computed by new leader (R_1)

Figure 3.9: The intersections computed by the old leader and the new leader, which is then sent to the RSM in the form of a BATCHCOMMIT *message.*

standing batch. When it receives them, it could take the intersection and send a BATCHCOMMIT. But note that if it does this in the case of Figure 3.8, it will commit a different intersection as shown in Figure 3.9.

If the old leader executed the intersection shown in Figure 3.9(a) and crashed before propagating this information to any of the remaining replicas, the new leader will send a BATCHCOMMIT for its intersection shown in Figure 3.9(b). The old leader will have executed V_0, V_2, V_3. The new leader will have executed V_0, V_1, V_2, V_3. Thus, the system can get into an inconsistent state. How can this problem be solved?

To solve this problem we need to understand better why the problem occurs in the first place. To make the solution easier to understand we introduce a conversion from batches to sequences. The sequences are not used in the implementation, and are merely a tool to simplify the explanation of the problem.

To convert a batch to a sequence is simple. The sequence starts out as an empty vector, and for each of the clients in the batch, we add this client's requests to the end of the sequence. In the case of Figure 3.9(a) this would yield the sequence $(V_0, _, V_2, V_3)$. For Figure 3.9(b), however, the sequence would be (V_0, V_1, V_2, V_3). This formulation highlights the problem: The original leader executes the former sequence and the sequence contains a missing request. By hopping over the missing request, the leader allows the future leader (when the older leader crashes) to possibly receive a different quorum resulting in the latter sequence. This sequence contains the previously missing request, resulting in inconsistency. One might argue

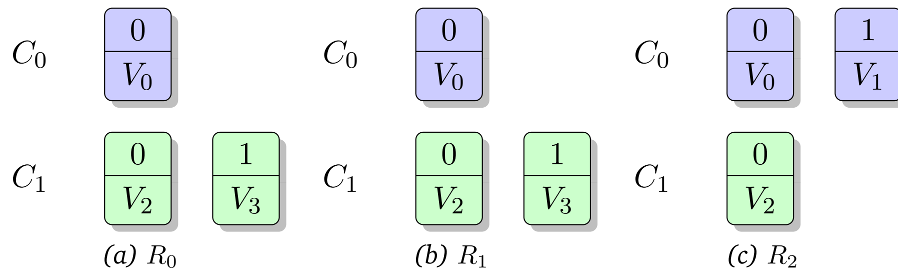


Figure 3.10: The problem case for which we need a larger quorum size. The batch information sent from each of the three replicas to the leader (R_0).

that if the old leader crashed, this won't be a problem. However, because of asynchrony we cannot distinguish between a slow old leader and a crashed one.

The first step to coming to a solution is to not allow the replicas to hop over missing requests like this. This can be done by simply having the leader send a BATCHCOMMIT for a sequence where no missing requests exist. This can be computed by going through the batch by each request, one by one, adding requests to the sequence until a gap is found. At the gap the leader must stop and no further commands can be executed. In this case the leader would be able to commit for V_0 only since the next request is missing. Note that we could also do a round-robin execution pattern where we execute the first request for every client, then the second, and so on. This would result in V_0 and V_2 being accepted before the gap stops us.

The second thing that needs to be done to solve the view change problem is to increase the size of the quorum. We will now need $3f + 1$ replicas to handle f failures. This is needed for a similar reason to why Fast Paxos [14] needs a larger quorum.

To see why this is the case, consider the following example shown in Figure 3.10. Suppose that the leader (R_0) receives a quorum of BATCHLEARNS, executes something, and then crashes before informing the others about this with a BATCHCOMMIT message.

The new leader has to figure out what was possibly executed in this case, but with just the information provided by R_1 and R_2, the new leader can't be sure what was executed. It could be that the leader had the same as R_1, as shown in Figure 3.10(a). However, it could just as well have been that the leader had the same as R_2. In the former case, the leader would have executed (V_0, V_2, V_3) and in the latter case (V_0, V_1, V_2). So using a normal quorum of $2f + 1$ replicas is not enough, and we need to increase it to $3f + 1$. This is similar to how in Fast Paxos the leader could have committed a value and crashed, and during the view change the new leader receives multiple different values in the PROMISE messages [14].

To explain how to solve this problem, we will switch back to using

R_0	A	B	C	_
R_1	A	_	C	D
R_2	A	_	C	_
R_3	A	_	_	D
R_4	A	_	_	D

Table 3.5: The quorum of sequences received by a potential new leader in the case where $3f + 1 = 7$. The _ represents a missing value in the sequence.

R_1	A	_	C	D
R_2	A	_	C	_
R_3	A	_	_	D
R_4	A	_	_	D

Table 3.6: The sequences left after throwing out R_0's sequence due to not having at least $f + 1 = 3$ B values. Note that this also causes the amount of Cs to go below the $f + 1 = 3$ threshold needed for it to stay in the prefix.

sequences to make the concept simpler. Suppose we let $f = 2$, in which case we need $3f + 1 = 7$ replicas to support two failures. Furthermore suppose that the leader crashes, and the new leader receives a quorum of 5 BATCHLEARN messages containing the sequences shown in Table 3.5.

What we can do is count the values in each column of Table 3.5. We start with the first column, and see that we have 5 A's, which is greater than $f + 1 = 3$, so this value could have been executed by the previous leader. We then move on to the next column, and we see there are 4 missing values, which is greater than 3 as well. However, there is a single B in R_0's sequence, but this could not have been executed by the previous leader, so we throw R_0's sequence away, leaving us with Table 3.6.

We move on to the next column in the new table after throwing R_0's sequence away. We now see that previously there were 3 Cs, enough to say that the previous leader could have executed C. However, with the updated table, there are only 2 Cs, which is not enough for the leader to have executed it. When we get a count for a value less than $f + 1 = 3$ we stop the algorithm.

In this case, we have found that the maximum prefix that the leader could have executed is $(A, _)$. This can then be converted back into a batch form and sent as a BATCHCOMMIT to the rest of the RSM while maintaining consistency with the previous crashed leader.

4
Goxos

The Goxos framework [11] – a framework for building Paxos-based RSMs – was designed during the Fall 2012 semester and improved as a part of this master's thesis. This framework forms the core of ACROPOLIS, providing it with abstractions to implement all of the necessary features.

In this chapter, we will discuss the Goxos framework in more detail, including some of the interesting design goals that we had. In addition, we will discuss some of the implementation details concerning the Go programming language which was used to implement Goxos. This content is adapted from a project report completed in the Fall 2012 semester co-written with Tormod Erevik Lea [11].

4.1 Motivation

The Goxos framework was initially created to provide a simple base upon which many of the Paxos variants can be built on. It is also meant to be used as a test-bed for testing and benchmarking the different algorithms and optimizations. This means that we need to provide the abstractions necessary to expressively create these Paxos variants. Things like networking, inter-process communication (IPC), client handling, failure detection, and leader election need to be abstracted away.

These goals go hand in hand with one of the Tidal News project's [27] goals of researching possible fault-tolerant publish-subscribe (pub-sub) architectures. In cases like this, the Goxos architecture can be put to good use for research and testing.

4.2 The Go Programming Language

The Goxos framework is created using the Go programming language, designed and created in 2007 by Google, but didn't become publically available until 2009 [24]. Go is licensed under a BSD-style open-source license, allowing anyone to view and modify the code [25].

Go is a compiled language, and aims to be very efficient – both in terms of developer efficiency as well as the generated machine code. Go also provides very easy to reason about concurrency primitives. While other languages use locking primitives such as mutexes, semaphores, or condition variables to synchronize between threads, Go takes inspiration from communicating sequential processes (CSP) [9].

In CSP, there are many processes executing at the same time, and they can communicate with one another through the use of channels [9]. Go takes the same approach, but refers to these processes as *goroutines* which communicate and synchronize with one another by sending messages through *channels* [24].

4.3 Modularization

As software grows and gets more complicated, one of the best techniques to manage this is by using a modular approach [18]. By doing this, you benefit greatly by separating your concerns which allows you to reason more easily about what the system is actually doing. In addition to this, a modularized system aids in debugging and testing since modules can be debugged and tested separate from one another. However, the amount of modularity has to be limited, because over-modularization can be a problem.

When it comes to frameworks that are based on RSMs, performance is one of the most important attributes. By creating a modular framework, it allows easy testing of performance of different optimizations to the Paxos protocol as well as easy testing of different Paxos variants. This can be done simply by plugging in different modules with optimizations or other features and doing a performance analysis.

There are three main modules that make up the Goxos framework, each of which is subdivided further into separate files. The three modules are the Paxos module, the network module, and the liveness module. The Paxos module is responsible for handling all of the Paxos communication. For example, the Paxos module handles all of the PREPARE, PROMISE, ACCEPT, and LEARN messages that are received over the network. The network module is responsible for the sending and receiving of messages over the network. Finally, the liveness module is responsible for doing failure detection of other replicas as well as leader election, which is does by using the network module to send and receive HEARTBEAT messages. A basic overview of how

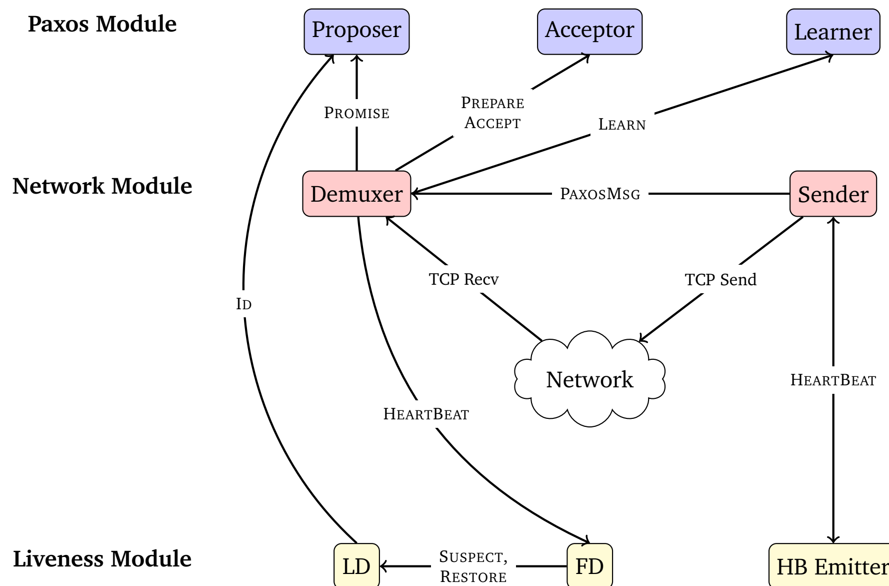


Figure 4.1: A partial view of the architecture of a single node of Goxos where each separately colored layer is a module and each entity is a separate actor in the system, sending and receiving messages. Each arrow represents a communication channel between two modules.

the system is designed is shown in Figure 4.1.

While Figure 4.1 might suggest that Goxos and the Paxos module go hand in hand, it is not true. In the case of ACROPOLIS, the Paxos module can be replaced with the ACROPOLIS module which handles a different set of messages such as shown in Figure 4.2. This simple method of extending the Goxos framework to handle different Paxos variants or possibly even other distributed algorithms altogether is one of the shining examples of how important modularization is for an RSM framework.

There are many benefits to designing software while taking a very modular approach, including the ease of which the system can be extended in the future by adding more Paxos variants and optimizations. One of the trademarks of RSM frameworks is the fact that many things are happening at the same time which tends to require the use of concurrency. By keeping a very modular design, it helps to reason about the code in a way that would be very difficult in a flat file-based architecture.

4.4 Concurrency Approach

Paxos requires many things to be occurring at around the same time. The protocol must be running while the leader elector and failure detector are

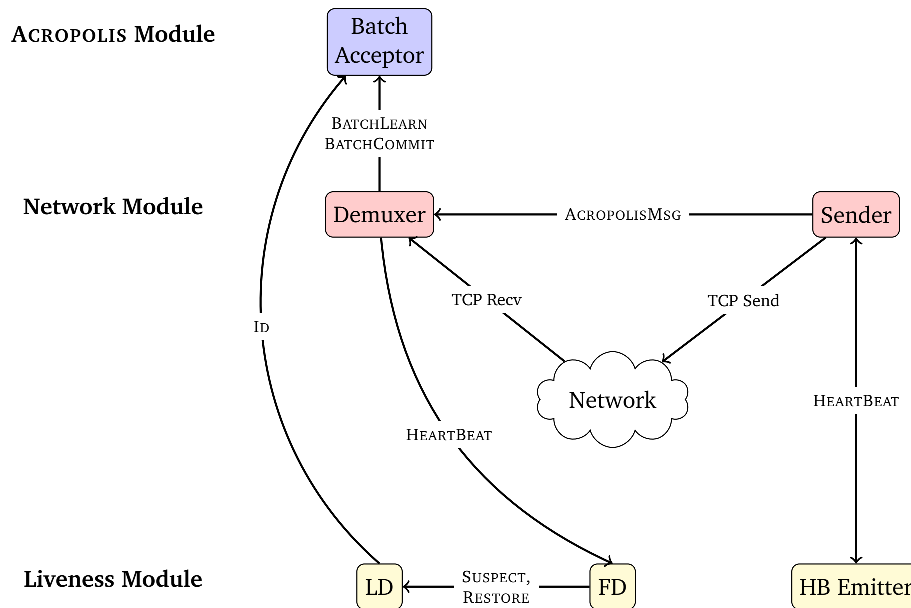


Figure 4.2: A partial view of the architecture of a single replica of Goxos when running with the ACROPOLIS *module instead of the Paxos module.*

running. It needs to be handling the connections between replicas as well as from clients to replicas. As well as being necessary, modern processors can benefit from concurrency due to the prevalence of multi-core computing these days.

The Go programming language provides us with goroutines, which are basically very lightweight threads. We can spawn off different goroutines for different modules in our system, and the Go scheduler will take care of everything for us.

Due to the need for concurrency in our framework, we also need a way to coordinate between the different goroutines. For example, the Paxos actor goroutines displayed in Figure 4.1, as well as the ACROPOLIS actor goroutines in Figure 4.2, need to communicate with the network goroutines in order to send and receive messages from other replicas and clients.

The Go programming language gives us the ability to efficiently and easily communicate and synchronize between the different goroutines in the system by using channels. A channel is a method of performing inter-process communication. When two goroutines are spawned off, they can both be given a reference to a channel. One of the goroutines can put data onto the channel, which can then be retrieved by the other goroutine. The channel abstraction allows easy concurrency without having to worry about the traditional concurrency methods such as locks or semaphores.

It should be noted that this approach of using concurrent threads or

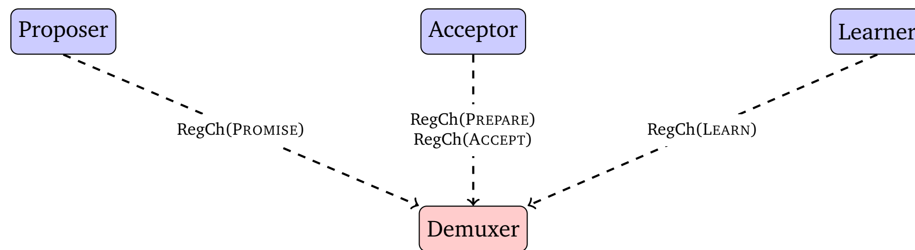


Figure 4.3: The Paxos modules must register their interest in certain types of messages upon start up. Then when a message of one of these types is received over the network, the network module knows which modules are interested and can deliver them to the correct location. In this case, the dotted lines represent a method call on the demuxer.

goroutines and channels to communicate between them is not a new concept, but instead a renewal of an old concept. These ideas are based on communicating sequential processes originally described in the 70s [9].

4.5 Network Communication

Each Goxos replica needs to communicate with the outside world, whether it be with external clients or to other replicas within the RSM. The sending and receiving of information is one of the most important parts of any distributed system.

The network communication is split into two different modules: One module to handle traffic for inter-replica communication, and one for handling per-client connections. The inter-replica communication is handled by the network module, which consists of *sender* and *demuxer* submodules. Client communication is handled by the *client handler* module.

But how do these messages get to the correct destination? Using Go's reflection capabilities allows each module to register interest in a certain type of message when it is initialized. This is shown in Figure 4.3 where the Paxos modules register their interest with the demuxer. The same thing would occur in ACROPOLIS where the batch acceptor would register interest for BATCHLEARN and BATCHCOMMIT message types.

Before the messages can be delivered, however, they need to be sent on the network. To send them on the network, they have to be serialized to some binary format. This serializing and deserializing is handled by the sender and demuxer modules. When a message is to be sent onto the network, it is given to the sender module, who then encodes the message using Go's own serialization format called *gob* encoding [24, p. 363]. The demuxer then reads these serialized values, deserializes them, and passes them to any interested parties.

This works fine for the inter-replica communication, but client communication is sufficiently different to warrant a separate design, and this is done by the client handler module. This module is responsible for receiving requests as well as sending replies back to the client.

Each client in the system must have a unique ID so that Goxos can differentiate between requests from different clients. To do this, when a client starts up it generates a unique ID using a cryptographic hash function approach similar to the one seen in [13]. Once this is done, the client does a handshake with Goxos, and assuming all of this goes as planned, then a new goroutine can be spawned to handle the network communication for this client.

Client communication could use the same approach for the inter-replica communication whereby each request from the client is encoded using the gob encoding. However, this assumes that each client will be built using the Go programming language as well, which isn't necessarily true. It makes more sense to take a language-agnostic approach where the client can be written using whichever languages are available. To support this in Goxos, the network communication between Goxos and any clients must use a serialization library that is available for many different languages. One such library is Google's protocol buffers. Libraries for serializing to and from protocol buffers are widely available for many different languages, so the client can be written in any language that supports protocol buffers.

For Paxos, the client only connects to the leader of the RSM, since the leader is responsible for sequencing the commands from clients, it would make no sense to send the command to the whole RSM. It does this by connecting to the first replica in the configuration file, and waiting to see if it gets a redirect reply from this replica telling it to connect to a different replica which is currently the leader. However, in Fast Paxos [14] as well as ACROPOLIS, the client needs to connect to every replica in the system so that it can broadcast to the RSM.

4.6 Liveness

While the original descriptions of Paxos [15, 16] assume that there is a single coordinator or leader of the system who is responsible for handling client requests and getting them accepted, there is no description of how the leader is actually selected.

The failure detection Goxos employs is based on heartbeats. One of the central parts of the liveness module is the heartbeat emitter, which is responsible for – as the name implies – sending heartbeats out onto the network to every node. It does this periodically using a timeout. These HEARTBEAT messages are used by the failure detector module to tell whether or not a replica may have crashed.

The failure detector (FD) is responsible for detecting failures of replicas in the RSM, which is computed based on the aforementioned HEARTBEAT messages which are sent and received. The algorithm used for failure detection is based on the eventually perfect failure detector, also known as $\Diamond\mathcal{P}$ described in [4, p. 53]. The FD module is responsible for informing the leader detector (LD) module when a replica hasn't been heard from within a certain amount of time. It does this by sending SUSPECT messages to the LD module when the FD suspects a replica may have crashed. It is possible, however, that the FD suspects a replica has crashed when it actually hasn't. This can occur due to a temporary increase in network latency. In this case the FD will issue a RESTORE message to the LD module telling it that a replica has come back online.

The leader detector module uses an algorithm based on the monarchical eventual leader detection algorithm explained in [4, p. 57]. This algorithm implements Ω, a leader detector scheme where *eventually* all replicas trust the same node as being the leader. The way it works is, if the leader detector gets a SUSPECT message for the current leader of the system, it will elect a new leader. The new leader will be selected based on its rank (which is predefined based on the ID of the replica), and the highest rank node that is currently alive will be the one elected.

Other modules in the framework can notify the leader detector that they are interested in receiving information about which nodes are elected. In particular, the Paxos or ACROPOLIS modules are required to know who the current leader of the system is, so that they are aware of what their current role in the system is.

4.7 Application

A replicated state machine framework is usually implemented with a certain application in mind. The RSM itself is used to ensure consistency between each of the replicas without having to worry about any application-specific details. Some implementations of RSMs take a monolithic design approach where the application is built into the actual RSM code. This may have some performance benefits, but also suffers from the application code being tightly coupled to state machine code – which can limit the flexibility of the framework [18].

Goxos achieves the decoupling between application and RSM by making Goxos a library. Anyone interested in creating an application which has RSM backing can simply include the Goxos library in their code and write their application using the interface provided by the Goxos library. In Go terminology, the application must implement an interface containing an `Execute([]byte)` method. This interface can be passed to the actual Goxos server who can call this method on the application whenever a command is decided. Note that

the method takes a byte array as its parameter. This furthers the decoupling between the application and the replica, since the application must interpret the byte array however it wants while the application doesn't need to worry about this application-specific data that has been decided upon by the RSM.

5 Implementation

In this chapter, we will discuss some of the implementation details that went into building ACROPOLIS. Since Goxos is made of two distinct parts, the client and the server, we will start by discussing the implementation of the client. Then we will discuss some of the non-trivial details surrounding the implementation of the server.

5.1 Client

The client-replica interaction in ACROPOLIS is sufficiently different to warrant talking about some of the interesting difficulties that were run into during development. The two problems had to do with making the client interaction asynchronous as well as the maintenance of connections.

5.1.1 Futures

Many Paxos-based RSMs use a synchronous client-server communication model [12, 13]. This means that the client sends a request, and awaits a reply from the RSM before it sends any further requests. This works well for Paxos since a request from the client should be tied to an instance and the agreement protocol will initiate right away. Once the agreement protocol is concluded, the reply can be sent back to the client.

This synchronous client-server model (shown in Listing 1) doesn't make any sense for ACROPOLIS, since the request reception at the replicas is decoupled from the actual agreement protocol. The reception of a request doesn't necessarily initiate the agreement protocol – unless the maximum batch

size has been reached. Otherwise the agreement protocol of ACROPOLIS is initiated by a timeout due to inactivity in the system.

```
1 func (c *Conn) SendRequest(req []byte) ([]byte, error) {
2   var request = c.genReq(req)
3   var resp gxc.Response
4
5   // Send request ...
6   // Block until response ...
7
8   return resp.GetVal(), nil
9 }
```

Listing 1: The method signature for sending requests used by Paxos clients. The response is returned directly from the method. This synchronous method of sending requests is not applicable for ACROPOLIS.

If we have N clients in the system and we assume the maximum batch size is greater than N – which should usually be the case – each client will only get to send a single request per timeout. This is due to the fact that each client will send a request, and wait for a reply to it. The batch, however, will never fill up so a timeout will eventually be triggered, producing a response to the requests. This will continue ad infinitum, hindering the performance of the system.

The solution that ACROPOLIS takes is to use *futures* [2]. A future represents a parallel computation that returns a result. In Goxos, when a client sends a request, the send procedure returns a result – the response from the RSM. To make this work with ACROPOLIS, the send procedure doesn't return the response, but instead returns the future of the response. Furthermore, it stores the future in a map which maps from sequence numbers to futures so that the response can be inserted into the future later. A code snippet showing the creation of a future for a request is shown in Listing 2.

Representing futures in Go turns out to be very simple: A future is nothing more than a buffered channel of size one. When a client sends a request, instead of waiting for the response from the RSM, a future is returned by the client library. When the client wants to access the future and obtain the actual response, it is a normal receive operation on the channel. This is shown in Listing 3.

The client futures are created by the sending procedure, but how are they filled with the actual response? The client library, when the connections are made to the replicas, also spawns a goroutine handling data coming from the replica. This data is deserialized as a response, which has a sequence number associated with it – the same sequence number in the initial request.

Listing 4 shows how the worker goroutine handles each connection to the replicas and receives responses which it then delivers to the future so that the client can access it.

```go
type ResponseFuture chan ResponseInfo

func (a *AcropolisConn) SendRequest(req []byte) (error, ResponseFuture) {
  var request = c.genReq(req)

  // Make buffered channel of size 1
  future := make(ResponseFuture, 1)

  // Register this future for later access
  a.futures[request.GetSeq()] = future

  // Send request ...

  // Return future rather than wait for response
  return nil, future
}
```

Listing 2: The interface for sending requests in ACROPOLIS*, used by clients. Note the return value is different from Listing 1. While this one returns a future for a response not yet received, while Listing 1 waits for the response right after sending.*

```go
// Get future for this command sent to the RSM
err, future := g.SendRequest([]byte(cmdString))

// Access future to get value - may block if not ready yet!
resp := <-future
```

Listing 3: How futures are actually used on the client-side. A future is returned, which can then be accessed. This may cause a blocking operation if the response isn't yet asynchronously received by the client library.

```
// Read response from socket
var resp gxc.Response
err := read(conn, &resp)

// ...

// Get future
future := c.futures[resp.GetSeq()]
senttime := c.send[resp.GetSeq()]

// Fill it with the response
future <- ResponseInfo{
  Value:    resp.GetVal(),
  SendTime: senttime,
  RecvTime: time.Now(),
}
```

Listing 4: A portion of the code involved with the worker goroutine for connections to the RSM. The response is read off the connection, and then the sequence number is used to look up the future associated with initial request. This is filled with the response value, as well as information about when it was sent and received.

5.1.2 Connections

In regular Paxos-based RSMs, the client only needs to maintain a connection to the current leader of the system. This is due to the fact that the leader is responsible for attaching the commands to an instance number and beginning the Paxos protocol in order to get the command accepted by the state machine. The leader is also responsible for replying to the client with a response once the command has been executed. So, there only needs to be a single communication channel between each client and the RSM.

In Goxos, a client connects to the first IP address in the configuration file. It could be that this IP address represents the current leader replica. In this case, the client connection is created and requests can be sent through the connection. It could be possible that the IP is not the leader, in which case the client will receive a redirect notification from the replica telling it who the current leader is and the IP of that replica. The client can then successfully connect to the leader replica and begin issuing requests.

However, for ACROPOLIS, the client communication pattern is a bit different. Each client has to broadcast their requests to the RSM, since the client is responsible for disseminating the data to each of the replicas.

In order to modify Goxos to handle broadcasts from clients, the *client handler* module had to be changed to allow connections to all replicas even if the replicas are not the leader.

Replica	Batch 0	Batch 1
R_0	$[0,10]$	$[11,15]$
R_1	$[0,8]$	$[9,12]$
R_2	$[0,5]$	$[6,10]$

Table 5.1: Two consecutive batches showing the ranges of a single client at each of the three replicas making up the RSM.

5.2 Replica

Now we can discuss some of the more complex tasks involved in implementing ACROPOLIS from the point of view of the replica.

5.2.1 Intersection

One of the most important roles that the leader has in ACROPOLIS is computing the intersection of all of the received client ranges. The most obvious way to do this is to have each of the replicas send the client ranges for all clients from whom they have received new requests from since the last trigger.

However, this can lead to problems if one of the replicas receives a client's requests before the other replicas in the RSM. Take, for example, the following table containing range information for two consecutive batches shown in Table 5.1.

Note that the leader can do a successful intersection operation on each of the ranges for this client in batch 0. The intersection will be computed as $[0,5]$ since that is what all of the replicas are guaranteed to have seen. However, on progression to the next batch a problem occurs. The computation of the intersection for this batch will produce the empty set, $\emptyset$. The ranges for R_0 and R_1 have $[11,12]$ in common, but intersecting this with the final range from R_2 produces $\emptyset$ since they are totally disjoint.

Because the intersection computation for batch 1 produces $\emptyset$, nothing for this client will get executed. This doesn't, however, mean that the client will never get any commands executed in the future. It is possible in batch 2 that the intersection produces a valid range instead of the empty set. However, because of the empty set in batch 1, it is possible for some of the commands to be lost, and the client will have to retransmit them with new sequence numbers to get them accepted.

One method to solve this is to use implicit ranges instead of explicit ranges. When the leader sends a BATCHCOMMIT message, it is informing the RSM what the actual intersection of the ranges were. The replicas can store this information and send ranges starting from the end of the range in the previous batch. In Table 5.1, the intersection for batch 0 is computed as $[0,5]$, which the leader will propagate to the other replicas. So, when the

other replicas trigger the next time, they will know to start their ranges from $5 + 1 = 6$ instead of what they sent in the previous batch.

In fact, the replica doesn't even need to send an explicit range in this form, since the start of the range is no longer needed. Each replica can simply inform the leader of the end of the range for all clients that it has received from since the previous batch. Using this approach, the actual intersection computation is simple: Find the minimum endpoint for each of the *implicit* ranges given in all the BATCHLEARN messages for each client.

5.2.2 Batchpoints

One of the other interesting problems solved is the batchpointing done by the leader. This also ties in with the previous problem of computing the intersection. The leader needs to keep track of batchpoints in order to have a picture of what the global RSM state is.

The batchpoints are also used to ensure that the servers are moving in lock step with one another, and to check that the potentially new batchpoint location is valid – that it is not the same as the previous batchpoint.

Batchpoints are computed passing the information about the intersection to the batchpointer structure. This structure is dedicated to handling and keeping track of all of the batchpoints.

```
1 func (bp *BatchPointer) TryBatchPoint(nbp map[string]uint32)
2     (bool, RangeMap) {
3   // ...
4 }
```

Listing 5: The signature for the `TryBatchPoint` method, the central method in computing batchpoints.

When actually computing the batchpoints, there are two cases that can occur: The first case, when the batchpoint to be computed is BP_0, the first batchpoint. The second case is when we are computing BP_i for some $i > 0$.

```
1 if len(nbp) > 0 {
2   bp.BatchPoints = append(bp.BatchPoints, nbp)
3   return true, makeRangeMap1(nbp)
4 } else {
5   return false, nil
6 }
```

Listing 6: The first case when computing a batchpoint. In this case, all that is done is check that the intersection given to us (`nbp`) allows us to extend the batchpoint, and add the batchpoint to the list of batchpoints. Finally the ranges are returned that describe the newly created batchpoint.

Note that in Listing 6 the value returned from `TryBatchPoint` is a range map, expressing the ranges for the batch. This is due to the fact that the ranges are what is required to send in the BATCHCOMMIT message. Now we can look at the second case, where there exists a previous batchpoint. In this case, we have to use the previous batchpoint and the potentially new batchpoint to create the ranges.

```
// Get the previous batchpoint
last := bp.BatchPoints[len(bp.BatchPoints)-1]
grtr := make(BatchPoint)
// Whether or not a batchpoint should be created
rmable := false

for cid, hs := range nbp {
  phs, exist := last[cid]
  if !exist {
    // New client, should batchpoint
    grtr[cid] = hs
    rmable = true
  } else if hs > phs {
    // Old client, but intersection contains greater than previous
    grtr[cid] = hs
    rmable = true
  } else {
    // Old client, with no growth
    grtr[cid] = phs
  }
}

// If we detect that we should batchpoint, then do it
if rmable {
  bp.BatchPoints = append(bp.BatchPoints, grtr)
  return true, makeRangeMap2(last, grtr)
} else {
  return false, nil
}
```

Listing 7: The second case, where previous batchpoints exist. We must check that at least one of the clients in the intersection expands the batch. In other words, it ensures that the new batchpoint is strictly greater than the previous.

5.2.3 Triggers

One of the most essential parts of ACROPOLIS is the batch trigger. There are two ways for a batch to be triggered: Either a timeout occurs due to inactivity, or the maximum batch size was reached. While it seems trivial to implement, there were some difficulties in implementing them in Go.

The main problem with using batch triggers is that there are two types, and it is possible for one of them to trigger right after the other. For example,

if the maximum batch size is reached, and we create a batch, then it is still possible for the timeout trigger to occur right after. This is a problem, since the timeout trigger should only be used after a period t milliseconds of inactivity in the system, and since we just created a batch there is no inactivity.

What would make more sense is that the inactivity timeout should only occur at least t milliseconds after the last batch trigger occurred. This way, if a batch is triggered due to a maximum size being reached, then a timeout cannot occur for another t milliseconds after this.

Implementing the naive approach where we don't have to worry about when the timeout triggers occur is simple: Maintain a counter which counts how many requests we have received. If this hits the maximum size of a batch, then we do a batching operation. We also maintain a timer which expires after t milliseconds. When this happens, we try to do the same batching operation – if there are requests available to be batched.

Go provides a ticker structure, which creates a separate goroutine and returns a channel which it uses to send a tick value every so often. By reading from this channel synchronously, we can use it to perform an operation every t milliseconds. However, there is no safe way to reset the ticker so that the problem discussed above doesn't occur. To solve this problem we use plain timers in Go.

```
1  func (ba *BatchAcceptor) Start() {
2    ba.registerChannels()
3    ba.resetTimer()
4
5    go func() {
6      for {
7        select {
8        // ...
9        case <-ba.timeoutChan:
10         ba.handleBatchTimeout()
11       // ...
12       }
13     }
14   }()
15 }
16
17 func (ba *BatchAcceptor) resetTimer() {
18   ba.timeoutChan = time.After(ba.batchTimeout)
19 }
```

Listing 8: The reset timer procedure which uses Go's `time.After` functionality. The goroutine in `Start` is for handling incoming messages, in this case the messages are received from the created timer channel.

As shown in Listing 8, the `resetTimer` method overwrites the previous

timer channel. The timer channel can be read from, to figure out when to do a batching operation. Note that these timers are one-shot timers that only go off once. Thus in the `handleBatchTimeout` method we need to reset the timer which will create a new channel and goroutine.

```
1 func (ba *BatchAcceptor) handleBatchTimeout() {
2   batcher := ba.batcher
3   // Try batching
4   batch := batcher.GetBatch()
5
6   if batch != nil {
7     // Batch was available, send a learn to the leader
8     ba.send(BatchLearnMsg{
9       Id:      ba.id,
10      BatchId: batch.Id,
11      HSS:     batch.HSS,
12    }, ba.leader)
13  }
14
15  // Start timer from beginning to avoid timeout right after
16  ba.resetTimer()
17 }
```

Listing 9: The batch timeout procedure, which is responsible for checking if a batch is available, and if so sending a BATCHLEARN *message to the leader. Note the end, where it resets the timer to avoid the problem discussed in this section.*

5.2.4 Execution

Arguably the most important part of ACROPOLIS is the execution of the commands. While seemingly simple, there are some snags that must be discussed. Recall that the execution occurs when a replica receives a BATCHCOMMIT message from the leader. The leader sends the ranges to be executed in this message, in the form of a mapping from client ID to client range.

```
1 type ClientRange struct {
2   Start uint32
3   Stop  uint32
4 }
5
6 type RangeMap map[string]ClientRange
```

Listing 10: The client range structure as well as the range map structure which is sent from the leader to the RSM in the BATCHCOMMIT *message.*

The client range and the range map structures are shown in Listing 10. Note that the range map structure is a type synonym for a Go map with the

key being a string (the client ID) and the value being a client range. One thing that must be kept in mind when dealing with Go's maps is that they are not sorted. So the order by which you loop over the map is non-deterministic [26]. This causes problems if you want to loop over each client ID in the map and execute the range for that client.

```
1 func SortedKeysRange(m RangeMap) []string {
2   ids := make([]string, 0)
3   for cid, _ := range m {
4     ids = append(ids, cid)
5   }
6   sort.Strings(ids)
7   return ids
8 }
```

Listing 11: Procedure for getting a sorted list of keys for a range map.

Solving the problem is done by first creating a sorted list of the keys in the map as shown in Listing 11. Once this sorted list is created, we can loop over each item in the sorted list, and use it to index into the map to get the correct ranges ordered by client ID.

```
1 func (br *Batcher) ForEach(ranges RangeMap, fn func(client.Request)) {
2   ids := SortedKeysRange(ranges)
3
4   for _, id := range ids {
5     rng := ranges[id]
6     for i := rng.Start; i <= rng.Stop; i++ {
7       req := br.GetRequest(id, i)
8       fn(req)
9     }
10  }
11 }
```

Listing 12: Method which accesses the ranges in the range map in sorted order, passing each client request in the range to a callback function.

Once the sorted list from the range map's keys is created, executing the ranges can be done simply by looping over the sorted list, accessing the map with each of the lists elements. This gets us the correct range ordering, and then we can use the range to access the client requests in the range, passing each one to a callback function that does the actual execution of the command. This is shown in Listing 12.

This sorting of range maps is necessary to ensure that each of the replicas executes the ranges in the same order, which is likely not the case by default since maps in Go are unordered [26].

6 Evaluation

In this chapter we will do an evaluation of the ACROPOLIS protocol, mainly in two different settings: One setting is on a local area network (LAN), where replicas are located closely together, allowing very quick communication times. The other setting is a wide-area network (WAN) where the replicas are very distant from one another – in this case located at various points around the world. In both the LAN and WAN evaluations, the payload size of client requests varies between around 10 and 15 bytes.

6.1 Local Area Network

In the first setting, we will evaluate the ACROPOLIS algorithm in the LAN setting. This setting is where most traditional RSMs are used. In our case, the replicas are located on different machines connected directly to one another over a single switch.

6.1.1 Setup

These measurements and experiments are performed on the machines in the Linux lab at the University of Stavanger. Each of the machines has the same hardware, as shown in Table 6.1. Depending on the measurement, there are between four and six of these machines in use, with each machine running a single ACROPOLIS replica.

The clients making the requests are also running on the same machines, though different machines than where the replicas themselves are located. The amount of clients varies depending on the experiment.

Operating System	CentOS 6.4
Processor	Intel 1.86GHz Core2 Duo
Network Card	Intel 1GBps PRO/1000

Table 6.1: Hardware and software specifications for the machines used to run the replicas and clients.

In experiments where only four replicas were required, the replicas were run on the `pitter1`, `pitter2`, `pitter3`, `pitter5` servers. In situations where we require 5 or 6 replicas, `pitter7` or `pitter9` were also used respectively.

The clients were run on different machines. For example, if we wanted to run 30 clients, then we would run 10 clients on three separate machines.

6.1.2 Throughput

One of the most important metrics when it comes to RSMs is the throughput of the system, which is defined as being the total amount of requests which are executed on the server per second. For these experiments we use 4 replicas. The minimum replicas we test is 4 due to the fact that it is the minimum number we need in order to handle a single fault in the system.

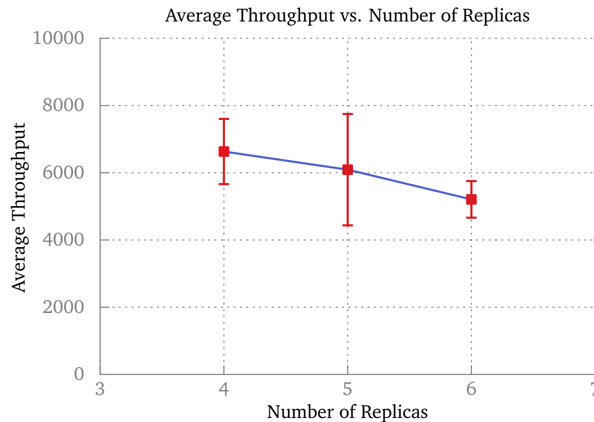


Figure 6.1: Average throughput of the RSM in comparison to the number of replicas making up the RSM. The higher the number of replicas, the more fault tolerance. Average throughput computed with 45 clients each sending 2000 commands. Any ramp up or ramp down values were thrown away to give a better idea about the typical throughput.

The first experiment was to figure out the relationship between throughput and the number of replicas. The more replicas in the system, the more fault tolerance we have, but usually this causes the performance of the system to degrade due to the increased amount of messages needed. This is

shown in Figure 6.1. The average throughput is computed by computing the number of commands executed for each second during the run, and then averaging this value over the number of seconds. In this case we see the average throughput is 6632, 6092, and 5208 commands per second for 4, 5, and 6 replicas respectively. As can be seen in Figure 6.1, it is a linear decrease in the throughput of the system as we add more and more replicas.

One of the questions that came up while evaluating ACROPOLIS is how tweaking some of the parameters, such as the batch size, would affect the throughput of the system. This is answered with Figure 6.2. To test the affect the batch size has on the throughput, we run the system with 4 replicas and 30 clients, where each client sends 2000 commands in total. We then record the average throughput for different batch sizes.

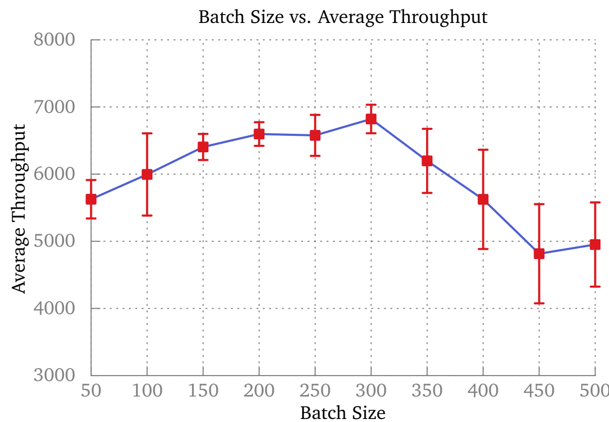


Figure 6.2: A plot of the average throughput of the RSM versus the size of the batches. Run with 4 replicas, 30 clients, each client sending 2000 commands in total.

In Figure 6.2 we see that the average throughput does seem to be dependent on the size of the batch. Furthermore, there seems to be a sweet spot in batch sizes, which makes sense. The larger the batch size, the longer the replicas will have to wait to actually fill the batch, which will delay the ACROPOLIS agreement protocol and thus lower the throughput to an extent. If you decrease the batch size too much, then the replicas will reach consensus and execute the commands faster, but the sizes of the batches – and thus the sizes of the actual intersection – will be smaller, decreasing the throughput. Therefore, the sweet spot for batch sizes should be calculated for different settings.

In the case of Figure 6.2, the sweet spot is at a batch size of 300 requests. With this batch size we got around 6821 commands per second throughput.

We can also take a look at one specific replica's throughput in the RSM, specifically for the case where we attempt to maximize the throughput by

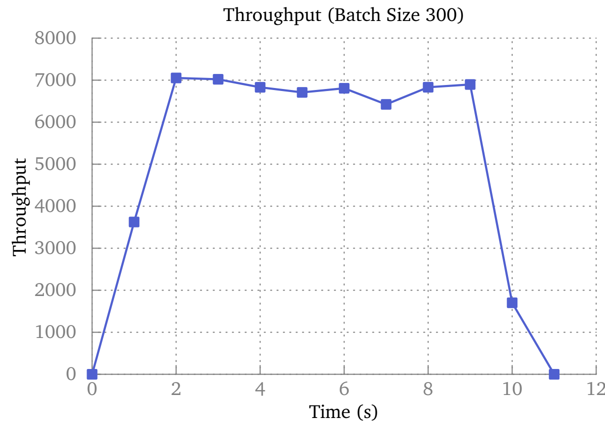


Figure 6.3: A plot of the throughput at one of the replicas, showing the stability of the system in terms of the number of commands executed per second.

tweaking the batch size. We found in Figure 6.2 that for 4 replicas, the optimal batch size is around 300. Figure 6.3 shows the throughput for one of the replicas, with the initial ramp up occurring from 0 seconds to 1 second. The throughput then reaches a maximum of around 7000 commands a second, and hovers around this point for around 8 seconds before ramping back down due to the clients finishing their run.

One of the other tweakable parameters in ACROPOLIS is the batch timeout period. We attempted to tweak this to figure out if there were any optimal timeout periods, but in a LAN setting where the clients are sending commands full throttle, the batch size will always fill up and no timeout triggers will actually occur.

6.1.3 Latency

Besides throughput, one of the most interesting metrics is the latency of a request. Also referred to as the round trip time (RTT), it measures the total amount of time from when a client sends a request to when it receives a response for the request.

If we take a look at the reported RTTs shown in Figure 6.4, we can see the usual pattern expected with ACROPOLIS. The times form spikes in the plot, which occur due to the fact that the RTT will be higher for requests which get received at the start of a batch. Requests which come near the end of a batch will have lower RTTs. This is due to the fact that these requests are the ones that are closer to the actual batch trigger occurring. The requests which come right after the batch trigger will have to wait for the replica to receive enough requests to fill up the batch.

A lot of information can be gleaned from Figure 6.4. If we look at the

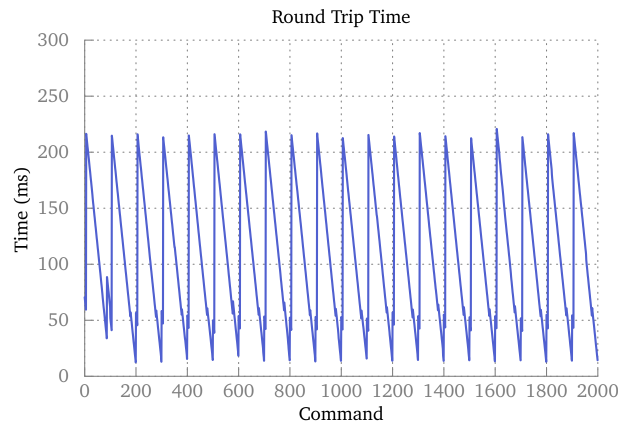


Figure 6.4: The round trip times reported for a run with 6 *replicas and* 5 *clients. This shows the RTTs for one of the* 5 *clients.*

lower valleys, these are RTTs for the requests that actually trigger the batch. The upper spikes are the RTTs for the requests that are at the beginning of a new batch. From the spikes we can even tell the batch size, in this case 100, since the period between spikes is 100 commands. Also, the lower valleys shown give us a clue about the actual time it takes the agreement protocol to reach consensus. Since these valleys are at a height of about $10 - 15$ms, this is approximately the time it takes for the leader to collect the required amount of BATCHLEARNS from the other replicas and commit the batch.

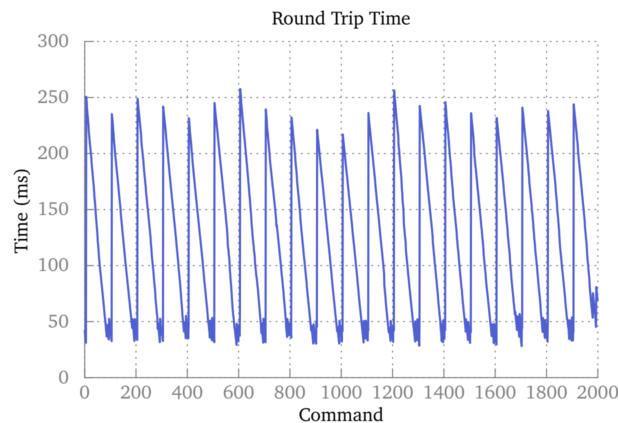


Figure 6.5: The same setup as in Figure 6.4 except with 15 *clients, each making* 2000 *requests to the RSM.*

If we take a look at the same setup except with 15 clients as shown in Figure 6.5, we see a few differences. The first difference one can note is that the spikes are not as uniformly placed as in Figure 6.4. Instead of them being

at a fairly uniform height, they are a bit less uniform. The same can be said of the valleys. The reason for this is due to the increased load on the replicas, who are handling more requests per second than with just 5 clients.

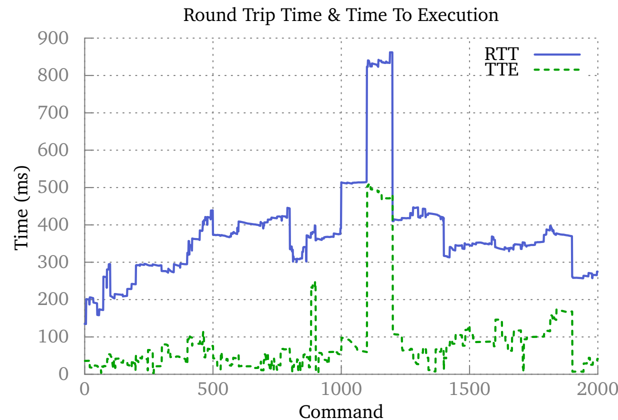


Figure 6.6: The same setup as in Figures 6.4 and 6.5, but with the time to execution (TTE) metric included in addition the round trip time (RTT).

Increasing the number of clients to 30 yields even more interesting results as shown in Figure 6.6. This figure also introduces the time to execution (TTE) metric. While the round trip time (RTT) measures the period of time between sending a request and receiving a response as observed by a client, the TTE metric measures the time from which a replica receives a request to the time just before it executes the request. We do not include the time it takes for the application to execute the request in the TTE because this depends on the type of application the RSM is running.

In Figure 6.6, we see the TTE is, in most cases, drastically lower than the RTTs reported. The large spike at around the 1200th command is due to a previous batch having a gap in it, which had to be filled by using the update protocol discussed in Section 3.6. This caused the times for this batch, as well as further batches to increase.

There is one question to be asked. What is making the RTT and TTE times so different? It makes no sense that the difference between the RTT and TTE is so high. So what is actually happening between the TTE and RTT being recorded? At the replica side, after the TTE is recorded, we give the command to the server to execute, generate a response, and respond to the client. The most likely explanation for this difference is due to a bottleneck in the server module or client handler module.

We can understand a bit more about the difference between the TTE and RTT by looking at box plots [29] of these values for varying amounts of client load. In Figure 6.7 we show a set of box plots for an increasing amount of client load on the RSM. From this we can see that the bottleneck starts to be

a hindrance at around a load of 20 clients.

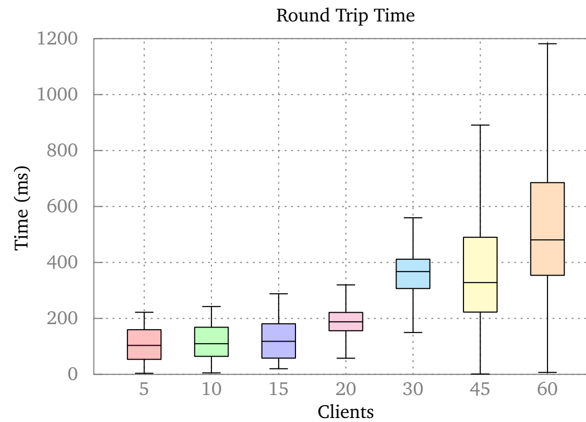


Figure 6.7: A set of box plots showing the variations in round trip time depending on the number of clients in the system. This was run with 6 replicas, and a varying amount of clients from 5 to 60.

However, we can also take a look at the box plots for the TTE shown in Figure 6.8. These plots confirm that the bottleneck problem starts occurring at around a load of 20 clients. However, it also shows that the median time to execution metric is relatively stable. The TTE median measurements starting at a client load of 5 and increasing are 24ms, 18ms, 15ms, 34ms, 43ms, 41ms, and 46ms. While the median does increase, it doesn't increase as drastically as the RTT does in Figure 6.7.

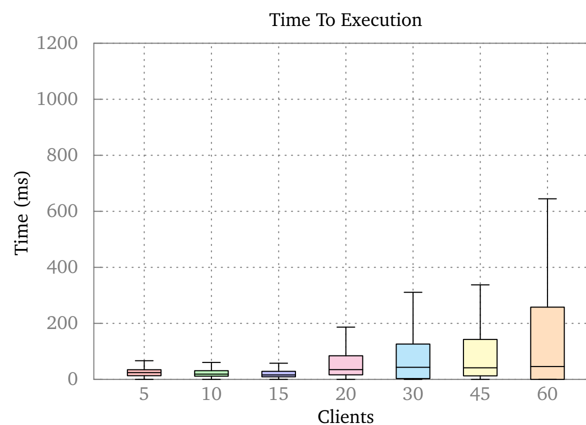


Figure 6.8: Box plots for the time to execution, where the number of clients varies from 5 to 60. Each client sends 2000 commands upon startup to each of the 6 replicas in the RSM.

After running a linear regression test on the data used to make Figure 6.8,

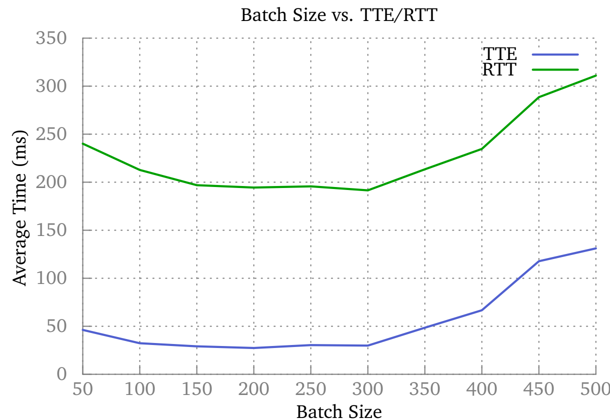


Figure 6.9: A plot of the round-trip time (RTT) versus the time to execution (TTE). Run with 4 *replicas,* 30 *clients, each client sending* 2000 *commands in total. The reported RTT and TTE times are averaged for each of the batch sizes. The batch size is varying from* 50 *to* 500 *commands.*

we found that the client load is a significant predictor of the TTE with the equation $T = 0.029c + 36.122$ where c is the client load and T is the predicted TTE for that client load. The p-value for this test is $2.2 \cdot 10^{-16}$, which is very small. This tells us that there is a strong correlation between the dependent variable T and the independent variable c.

One final interesting thing to look at is the relationship between the TTE/RTT and the batch size, as shown in Figure 6.9. This figure reinforces what we previously saw in Figure 6.3 where we looked at the average throughput of the system for different batch sizes. This is backed up in this figure as well, as we can clearly see that the sweet spot is around a batch size of 300. This emphasizes the fact that the performance of ACROPOLIS is highly dependent on tweaking the batch sizes for different network settings.

6.2 Wide Area Network

The second – and most important – setting is the WAN setting. In this setting, we use Amazon EC2 virtual machines located at different areas around the US. While traditional RSMs are usually deployed in a single data center and connected over a LAN, more and more we are seeing consensus algorithms being used in a WAN setting [7].

6.2.1 Setup

The measurements and experiments in this WAN setup are made using Amazon EC2 instances. An EC2 instance is basically a virtual machine. Each

instance is located in a different part of the continental United States. In the case of EC2, there are 3 availability zones in the US: Oregon (OR), Northern California (CA), and Northern Virginia (VA). The real latency between these locations is shown in Figure 6.10.

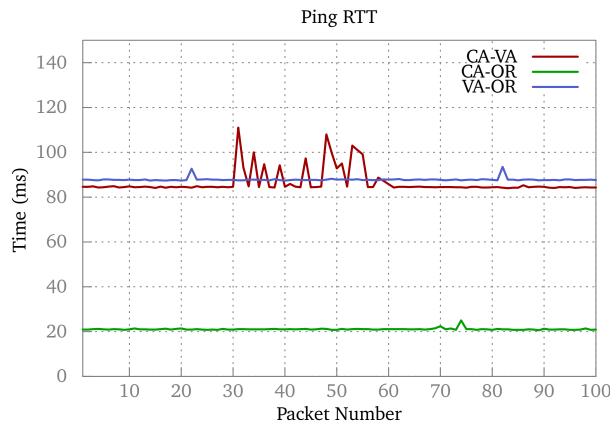


Figure 6.10: The actual RTT (ping) from one of the replicas located in Northern California to the two others. Not to be confused with request RTT in ACROPOLIS. *These are generated by pinging the other replicas.*

The times shown in Figure 6.10 are intuitive: Northern California and Oregon are relatively close together, while the distance to Northern Virginia is quite large.

Amazon has different classes of instances, in this case we are using the `t1.micro` instance tier. This tier allows us free use of any number of EC2 instances for 750 instance hours. While this instance tier doesn't have the best performance or bandwidth, they are still useful for getting an idea of how ACROPOLIS would perform in a WAN setting.

The WAN replica setup is a bit different compared to the LAN setup. We would like to run experiments with up to 6 different replicas, but with only 3 availability zones in the US, it isn't possible to run all 6 replicas at different locations. In order to solve this, we place more than a single replica at each of the 3 locations. For example, if we want 6 replicas, then we put 2 replicas at each location. If we want 4 replicas, then one of the locations will have 2 replicas, and the rest will have 1 replica.

6.2.2 Throughput

The first metric we measured in a WAN setting was the throughput of the system. There were no really surprising results when looking at the throughput for a single replica as shown in Figure 6.11. In this case there were 10 clients sending commands to the 6 replicas in the RSM. As expected in a

WAN setting, the throughput is quite a bit lower than in a LAN setting. This is explained by the higher latencies between the replicas.

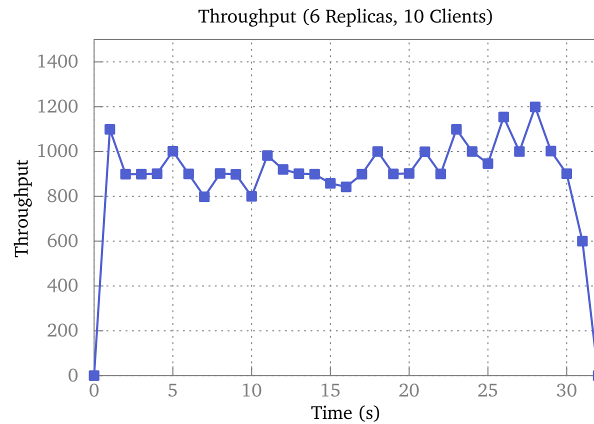


Figure 6.11: The throughput of our RSM in a WAN setup with 10 *clients and* 6 *replicas. This shows the throughput at a single replica.*

Also not surprisingly, the number of replicas affects the throughput of the system. This is shown in Figure 6.12. With four replicas, we achieve the highest average throughput with just over 1500 commands per second. Both Figure 6.12 and the earlier Figure 6.1 illustrate that as you increase the number of replicas in the RSM, the throughput will decrease due to the larger quorum size as well as the increased amount of communication between replicas.

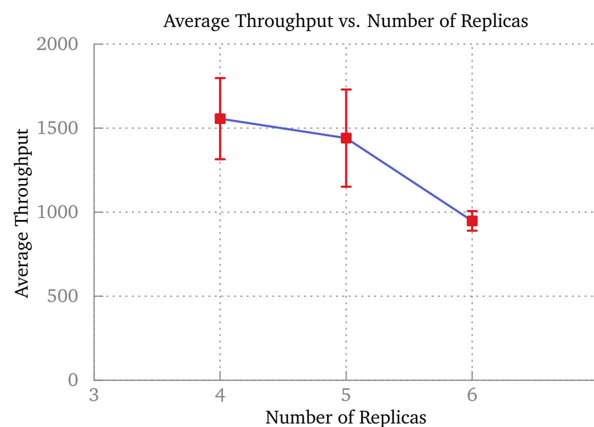


Figure 6.12: A comparison between the number of replicas in the RSM versus the average throughput with that many replicas. We run 10 *clients sending* 2000 *commands each.*

We can also look at how the batch size affects the throughput of the

system as we did in the previous section where we talked about the LAN setting. This is shown in Figure 6.13. Compared to the LAN comparison between batch size and throughput in Figure 6.2, Figure 6.13 shows that in a WAN setting, a smaller batch size produces a greater throughput. As we see in the plot, the average throughput is maximal with a batch size of 50.

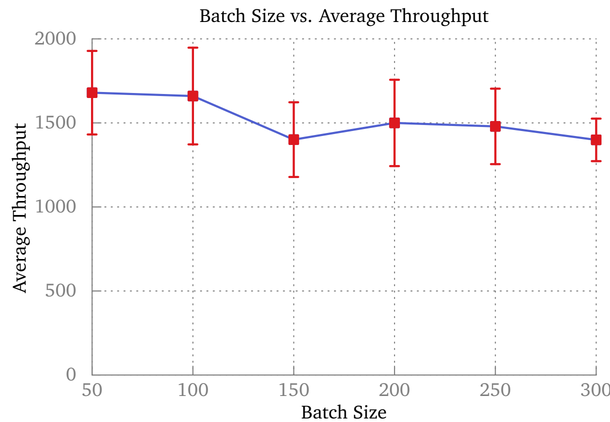


Figure 6.13: The average throughput of the RSM in comparison to the batch size. These measurements were taken with 4 *replicas and* 10 *clients sending* 2000 *commands each.*

Why is it the case that a small batch size is better for WAN environments? Since the replicas aren't exchanging the commands themselves in the agreement protocol, it cannot be due to this. One explanation is that the smaller batch sizes allow the clients to fill the batch quicker, since in a WAN environment the client-replica communication is not as fast as in a LAN. Another contributing factor could be that the update protocol is coming into play. With large batch sizes, it is possible for the replicas to get more out of sync, resulting in some replicas requesting updates which ultimately slow down the average throughput of the RSM.

6.2.3 Latency

The second metric we will cover for ACROPOLIS in a WAN setting is the latency of the system, in terms of the round trip time (RTT) and the time to execution (TTE).

Figure 6.14 shows the RTT and TTE for a particular client as it is sending 2000 commands to the server. There are an additional 9 clients also sending 2000 commands at the same time, making a total of 10 clients. Similar to with the LAN setting, we see that the RTT is significantly larger than the TTE, due to the bottleneck discussed earlier. It is very useful to look at the RTT and TTE data in the form of box plots and cumulative density functions, which we will do next.

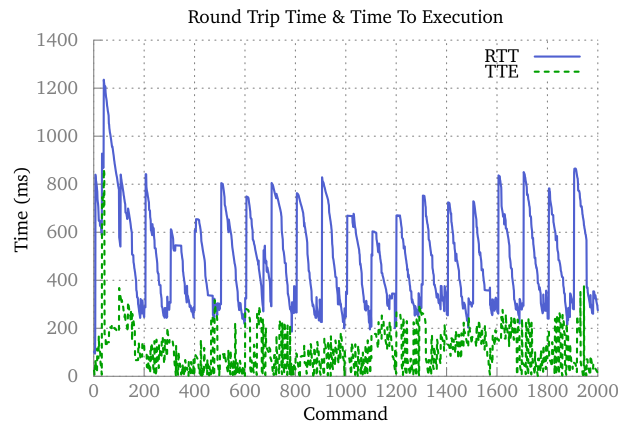


Figure 6.14: The round trip time and time to execution for a single client in a WAN setting. In this case, there were 10 *clients sending* 2000 *commands to each replica.*

The box plots shown in Figure 6.15 show how big the bottleneck can be, with the median RTT for 60 clients being around 4 seconds. In a WAN situation the bottleneck still seems to manifest itself at around a load of 20 clients, the same as in the LAN experiments. But let's look at the TTE boxplots as well, which show that this isn't actually the fault of the protocol.

The TTE box plots shown in Figure 6.16 show the time taken between a command being received and the command actually being decided and executed. For all of the client loads except 60 we see that the median load is below 200ms, and in most cases is below 100ms. The median TTE for 45 clients, for example, is 84ms. In our WAN setup, the highest actual latency (ping time) is to Northern Virginia, which has around an 86ms average ping time. This suggests that, if the bottleneck in the response back to the client is fixed, ACROPOLIS can provide RTTs that are reasonable in WAN situations.

6.3 Workloads

One thing that needs to be evaluated is how different types of workloads affect the sizes of the batches that can be committed. This is discussed in Section 3.7, where we explain why more needs to be done than just computing the intersection of the client ranges. This section aims to evaluate the difference between doing a simple intersection, and the more advanced rule required to support view change. We will do this by evaluating two different types of workloads.

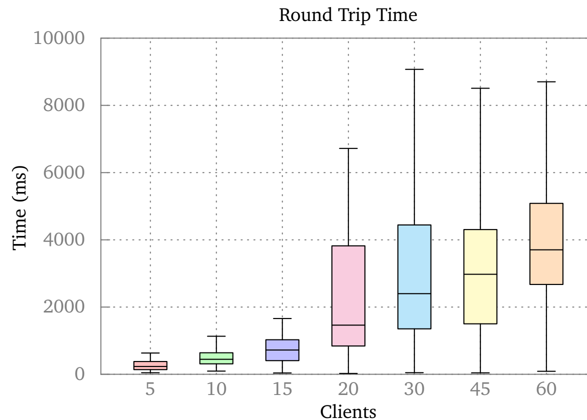


Figure 6.15: Box plot of the round trip time (RTT) for increasing amounts of clients and request load. In this case there are 6 replicas spread out over the Internet.

6.3.1 Full Throttle

In this workload, the clients fire their requests at the RSM full throttle, without waiting any period of time between sending requests. As can be seen in Figure 6.17, there is a lot of difference between the plain intersection and the intersection needed to keep consistency during a view change (VC).

The plain intersection size tends to hover around 100, which happens to be the configured batch size for this run. This can exceed the configured batch size due to the fact that the leader's last batchpoint could be lagging behind. Now, take a look at the VC intersection sizes. In the best case, they are the same as the plain intersection size, but usually they tend to be quite a bit less. This is due to the fact that any jagged edges in the client ranges collected by the leader need to be chopped off. If one client range is quite large, and the others are small, then quite a few potential requests will be lost in the VC intersection operation. This can result in an increased number of protocol messages due to smaller overall batch sizes, and thus a slower protocol.

6.3.2 Randomized

In this workload, the clients emulate real-world behaviour by waiting a uniformly random period of time between sending requests. This type of workload is shown in Figure 6.18. Note the stark difference between this figure and the previous one, Figure 6.17.

While in the previous workload, the plain intersection size was hovering around 100, but with quite a bit of variance, the random workload stays exactly at 100 for the entire duration of the first 100 batches. Similarly, the

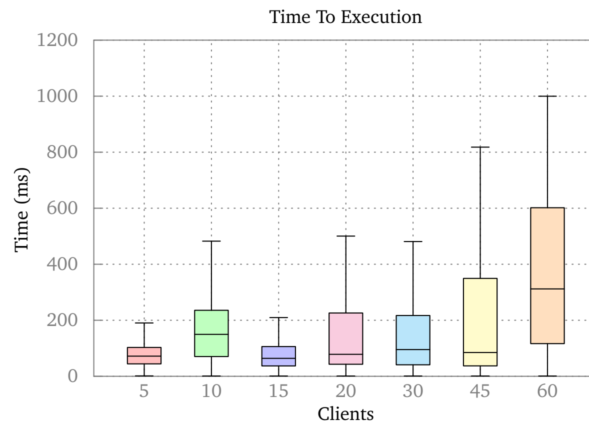


Figure 6.16: Box plot of the time to execution (TTE) for increasing amounts of clients and request load. In this case there are 6 replicas spread out over the Internet. Note the smaller time axis range compared to Figure 6.15, specially as the amount of client load increases.

VC intersection algorithm produces much cleaner results: Instead of the very erratic behaviour with full throttle clients, the random waiting seems to make the VC intersection sizes quite predictable.

One thing to note is that, due to the random waiting period after each client sends a request, this throttles the client a bit, and allows nearly every client to be a part of every batch. This explains why the VC intersection size tends to change by factors of 20. In order to have a batch of exactly 100 requests, the replica would have to receive the same amount of requests from every client. By consulting the log, we find out what is forcing the VC intersection algorithm to max out at 80 requests. What tends to be happening is a replica will receive 5 requests from every client except a few others who only have 4 requests. This forces us to throw away the last request from all the clients who the replica received 5 from. This gives us a total of $4 \cdot 20 = 80$ total requests.

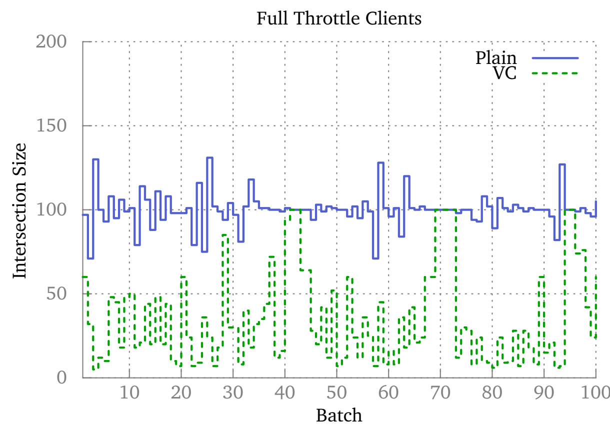


Figure 6.17: A visual description of the intersection sizes when doing a full throttle workload with 20 *clients sending* 2000 *commands. Each client sends at full speed, with no waiting between each request. This figure shows the first* 100 *batches.*

Figure 6.18: A workload where the client waits a random amount of time between sending each request. In this case, there were 20 *clients sending* 2000 *commands. Each client sends a request, and then waits a random period of time between* $0 - 20ms$ *before sending the next. This figure shows the first* 100 *batches.*

7 Future Work

This section will discuss any possible future work that would be interesting to research or look into in more depth. This includes things like optimizations to ACROPOLIS, as well as variants to the basic ACROPOLIS protocol.

7.1 Optimizations

There are various optimizations that should be done. These range from optimizations to the ACROPOLIS protocol or the structures used therein, to changes to the Goxos architecture.

7.1.1 Protocol

One possible optimization has to do with the update protocol described in Section 3.6. Currently, if a replica receives a BATCHCOMMIT message for some ranges that it doesn't have or only partially has, the replica will broadcast a UPDATEREQUEST message to the RSM, requesting some of the client requests that it is missing.

However, it would be better if the replica that is missing the commands were able to choose a random replica to send this UPDATEREQUEST to. This will minimize the number of messages that have to be processed as well as balance the update load among all replicas in the system.

In order to do this, though, the replica who needs an update must know which replicas were in the quorum that the leader received, and thus which replicas are actually able to reply to the update for sure. There could be

another replica also with a gap and it would make no sense to request from this replica.

To solve this problem, the leader could include a list of replicas in the BATCHCOMMIT that made up the quorum. Then the replica with a gap can randomly choose one of these and request an update from them. If the system is large and there are many replicas, it may be wise for the leader to randomly choose one of the replicas that made up the quorum itself and include just this single replica in the BATCHCOMMIT message.

7.1.2 Structures

Some of the structures should be optimized if ACROPOLIS is to be used in the real world. Namely, the client log structure currently stores all of the client requests throughout the lifetime of each replica. This results in quite a lot of memory usage that isn't really necessary.

In addition to the actual client log structures that exist, there is also some duplication in keeping track of all the batchpoints as well as batches.

A technique such as snapshotting [6], a technique that is often used in real world Paxos-based frameworks could be used. When a certain batch B_{i+1} is decided, a snapshot of the system state can be made and all previous batches $B_0, \ldots, B_i$ can be garbage collected.

7.1.3 Goxos Architecture

The architecture of Goxos [11] could be changed to support ACROPOLIS better. Currently, Goxos is built to support Paxos and its variants, but there are various optimizations to Goxos that could be done to help with improving the round trip times. The bottleneck discussed in Chapter 6 is one thing that could be fixed. One possible method of fixing the bottleneck is to batch the responses back to the client. Instead of sending individual responses to the client handler module, all of the requests for each client in a batch could be executed, and be sent back to the client as a group.

7.2 Adaptive Batching

There are penalties that requests have to pay if they are near the beginning of a batch, or if not enough requests are received and we need a timeout to trigger the batching operation.

It should be investigated how to minimize these penalties. One possible approach is to use an adaptive batching method. One such way to do this would be to start the maximum batch size at a certain value, and adjust the value depending on what happens to the current batch.

For instance, if the current maximum batch size is 100 requests, and we actually timeout due to not receiving this amount of requests, then it could be interesting to investigate how the performance of the system can be improved by perhaps adaptively changing this value to half of the current value, 50. Of course it would converge to something similar to the normal Paxos algorithm if it keeps getting halved down to 1 for the maximum batch size, so there would need to be adjustments in both directions: If we timeout, then the maximum batch size should shrink, but if we get enough requests to fill the batch, then it would be smart to increase the maximum batch size as well.

There are a lot of possible methods for dynamically adjusting the size of the batches. One simple method is to increase the batch size by a percent of the current batch size if the current batch size is reached. In other words, if for batch B_{i+1} the size for this batch is $S_{i+1} = S_i + pS_i$ where $0 \leq p < 1$ is the percentage of growth. If, however, we timeout then the current batch size is too big for the current load on the system, and must be lowered using a similar formula such as $S_{i+1} = S_i - pS_i$. However, the growth and reduction of the batch size should be limited to within a certain range so that it doesn't get too high or too low.

Another interesting possibility is to adaptively modify the timeout period and batch size based on the amount of work being done by each replica. For example, if a replica is experiencing a lot of client requests in rapid succession, it could realize this and adaptively increase its batch size or decrease its timeout. Metrics such as the amount of time it takes to execute a request can be taken into account to adaptively tune these settings.

7.3 Byzantine Proposer Fast Paxos

As hinted at in previous chapters, the initial design and development of ACROPOLIS came as a stepping stone to reach the ultimate goal of implementing Byzantine Proposer (BP) Fast Paxos [19]. This variant of Paxos aims to aims to handle misbehaving clients: Clients that are sending different values to each replica, or clients that are trying to perform a denial of service attack on the system.

ACROPOLIS allows BP Fast Paxos to be developed with proposers at the clients – in other words with client-proposers rather than separate clients and proposers. However, ACROPOLIS currently only handles crash faulty clients rather than malicious clients. The next step would be to integrate all of the specifics needed for doing trust changes, as well as supporting using cryptographic signatures or HMACs to verify that the clients are not lying.

7.4 Byzantine Client ACROPOLIS

A preliminary version of ACROPOLIS had the support for detecting equivocating clients through the use of cryptographic hashes. This could be implemented in the current version of ACROPOLIS as well, but there was insufficient time due to a significant change in the protocol at the last minute.

One way to do this would be to simply include a hash along with the client ranges in the BATCHLEARN messages and BATCHCOMMIT messages. This hash value would be computed by hashing all of the client requests for every client range in the message. Thus, the leader could detect whether any of the replicas have seen any equivocation and disconnect the client from the RSM.

8 Conclusion

This thesis presents new research in distributed systems, specifically related to the consensus problem. Solving this problem is the key to implementing replicated state machines (RSMs) using the state machine approach [23]. This research contributes the ACROPOLIS protocol, which is inspired by and related to the Paxos algorithm by Leslie Lamport [15, 16]. ACROPOLIS can be used to implement RSMs where each server is connected over a wide-area network (WAN).

The research into ACROPOLIS started with the design of the protocol itself, which was largely based on trying to solve a problem posed in a paper describing Byzantine Proposer (BP) Fast Paxos, a variant of the Paxos protocol that solves the consensus problem in spite of malicious clients and proposers [19]. The problem was to push the proposer's role in Paxos out to the clients. Usually in Paxos, the proposer is responsible for serializing the incoming client requests, but ACROPOLIS introduces a novel technique to avoid any explicit serialization being done by the proposers.

The design of the ACROPOLIS protocol lead to an initial implementation built on top of the Goxos [11] framework, which was built as a part of our project course in the Fall 2012 semester.

The evaluation of ACROPOLIS shows that it is a viable solution, especially when replicas are connected over WAN networks, as the time to execution (TTE) metric is acceptably low. ACROPOLIS was evaluated on LAN, as well as on WAN using Amazon EC2 with replicas located in each of the availability zones in the US.

This thesis also discusses some of the future work that is available related to ACROPOLIS, including optimizations to the protocol and the algorithms that drive the protocol, as well as some variants to ACROPOLIS that could be

implemented.

ACROPOLIS contributes a new way to look at implementing RSMs through solving the consensus problem, while removing the bottleneck of the leader associated with consensus protocols such as Paxos. Its specialty is achieving consensus over WANs, and can be extended to handle malicious clients, or used to implement Byzantine Proposer (BP) Fast Paxos [19].

Bibliography

[1] A. Avizienis et al., "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 11–33, 2004. DOI: 10.1109/TDSC.2004.2 (cit. on p. 6).

[2] H. G. Baker and C. Hewitt, *The incremental garbage collection of processes*, 1977 (cit. on p. 41).

[3] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06, Seattle, Washington: USENIX Association, 2006, pp. 335–350. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298487 (cit. on p. 7).

[4] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*, 2nd ed. 2011. Springer, Feb. 2011. [Online]. Available: http://amazon.com/o/ASIN/3642152597/ (cit. on pp. 4–7, 38).

[5] B. Calder et al., "Windows Azure Storage: a highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, Cascais, Portugal: ACM, 2011, pp. 143–157. DOI: 10.1145/2043556.2043571. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043571 (cit. on p. 7).

[6] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, ser. PODC '07, Portland, Oregon, USA: ACM, 2007, pp. 398–407. DOI: 10.1145/1281100.1281103. [Online]. Available: http://doi.acm.org/10.1145/1281100.1281103 (cit. on p. 66).

[7] J. C. Corbett et al., "Spanner: google's globally-distributed database," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'12, Hollywood, CA, USA: USENIX Association, 2012, pp. 251–264. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387905 (cit. on pp. 7, 57).

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," in *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*, ser. PODS '83, New York, NY, USA: ACM, 1983, pp. 1–7. DOI: 10.1145/588058.588060. [Online]. Available: http://doi.acm.org/10.1145/588058.588060 (cit. on p. 5).

[9] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978. DOI: 10.1145/359576.359585. [Online]. Available: http://doi.acm.org/10.1145/359576.359585 (cit. on pp. 33, 36).

[10] P. Hunt et al., "ZooKeeper: wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, ser. USENIXATC'10, Boston, MA: USENIX Association, 2010, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855851 (cit. on p. 7).

[11] S. M. Jothen and T. E. Lea, "Goxos: A Paxos implementation in the Go Programming Language," 2012 (cit. on pp. 1, 2, 15, 32, 66, 69).

[12] J. Kirsch and Y. Amir, "Paxos for System Builders," Johns Hopkins University, Tech. Rep. CNDS-2008-2, Mar. 2008, 35pp (cit. on p. 40).

[13] J. Kończak et al., "JPaxos: state machine replication based on the Paxos protocol," Faculté Informatique et Communications, EPFL, Tech. Rep. EPFL-REPORT-167765, Jul. 2011, 38pp (cit. on pp. 37, 40).

[14] L. Lamport, "Fast Paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, Oct. 2006. DOI: 10.1007/s00446-006-0005-x. [Online]. Available: http://dx.doi.org/10.1007/s00446-006-0005-x (cit. on pp. 6, 14, 15, 30, 37).

[15] —, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, Dec. 2001 (cit. on pp. 1, 6, 7, 10, 12, 13, 18, 37, 69).

[16] —, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998. DOI: 10.1145/279227.279229. [Online]. Available: http://doi.acm.org/10.1145/279227.279229 (cit. on pp. 1, 6, 7, 37, 69).

[17] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for WANs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08, San Diego, California: USENIX Association, 2008, pp. 369–384. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855767 (cit. on p. 12).

[18] S. McConnell, *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004 (cit. on pp. 33, 38).

[19] H. Meling, K. Marzullo, and A. Mei, "When you don't trust clients: byzantine proposer fast paxos," in *ICDCS*, IEEE, 2012, pp. 193–202 (cit. on pp. 2, 6, 12, 13, 67, 69, 70).

[20] K. Rarick. (2011). Introducing Doozer, [Online]. Available: `http://doozer-sdec2011.heroku.com/` (cit. on p. 7).

[21] K. Rarick and B. Mizerany. (2011). Go at Heroku, [Online]. Available: `http://blog.golang.org/2011/04/go-at-heroku.html` (cit. on p. 7).

[22] N. Santos and A. Schiper, "Tuning paxos for high-throughput with batching and pipelining," in *Proceedings of the 13th international conference on Distributed Computing and Networking*, ser. ICDCN'12, Hong Kong, China: Springer-Verlag, 2012, pp. 153–167. DOI: `10.1007/978-3-642-25959-3_11`. [Online]. Available: `http://dx.doi.org/10.1007/978-3-642-25959-3_11` (cit. on pp. 11, 16).

[23] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990. DOI: `10.1145/98163.98167`. [Online]. Available: `http://doi.acm.org/10.1145/98163.98167` (cit. on pp. 1, 10, 69).

[24] M. Summerfield, *Programming in Go: Creating Applications for the 21st Century (Developer's Library)*, 1st ed. Addison-Wesley Professional, May 2012. [Online]. Available: `http://amazon.com/o/ASIN/0321774639/` (cit. on pp. 33, 36).

[25] The Go Project. (2012). The Go Programming Language, [Online]. Available: `http://golang.org/` (cit. on p. 33).

[26] —, (2013). The Go Programming Language Specification, [Online]. Available: `http://golang.org/ref/spec` (cit. on p. 49).

[27] The Tidal News Project. (2013). The Tidal News Project, [Online]. Available: `http://www.tidal-news.org/` (cit. on p. 32).

[28] J. Turek and D. Shasha, "The many faces of consensus in distributed systems," *Computer*, vol. 25, no. 6, pp. 8–17, Jun. 1992. DOI: `10.1109/2.153253`. [Online]. Available: `http://dx.doi.org/10.1109/2.153253` (cit. on p. 6).

[29] Wikipedia, the free encyclopedia. (2013). Box plot, [Online]. Available: `http://en.wikipedia.org/wiki/Box_plot` (cit. on p. 55).