Christopher Jones

DRAFT

Achieving distributed consensus with Paxos

Part II Project

Trinity Hall

May 16, 2018

Proforma

Name:	**Christopher Jones**
College:	**Trinity Hall**
Project Title:	**Achieving distributed consensus with Paxos**
Examination:	**Computer Science Tripos — Part II, June 2018**
Word Count:	**...**[1]
Project Originator:	Christopher Jones
Supervisor:	Dr Richard Mortier

Original Aims of the Project

...

Work Completed

...

Special Difficulties

None.

[1] This word was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Declaration

I, [Name] of [College], being a candidate for Part II of the Computer Science Tripos [or the Diploma in Computer Science], hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

Chapter 1

Introduction

This dissertation describes the process of researching, developing and evaluating an implementation of the Multi–Paxos distributed consensus algorithm in OCaml.

1.1 Background

Distributed systems must handle many possible errors and failure modes. Unreliability is present in the network where messages can be delayed, re–ordered and dropped and processes can exhibit faulty behaviour such as stalling and crashing. The result of this is that distributed systems can end up in inconsistent states and even unable to make progress.

Consensus is the reaching of agreement in the face of such unreliable conditions. Applications such as transaction systems and distributed databases require consensus in order to remain consistent. Consensus algorithms provide a means of reaching agreement in the face of such unreliability; this is crucial in the design of distributed systems.

Paxos is a consensus algorithm first described by Lamport [5] that allows for consensus to be reached under the unreliable conditions present in distributed systems. The algorithm relies on processes participating in a voting protocol that tolerates the failure of a minority of processes.

Paxos is used internally in large–scale production systems such as Google's Chubby [1] distributed lock service, where it is used to maintain consistency between replicas. Microsoft's Autopilot [4] system for data centre management also uses Paxos, again to replicate data across machines. The generality of Paxos allows it to be used as an underlying primitive for various distributed systems techniques. State Machine Replication [12] is a technique whereby any application that behaves like a state machine can be replicated across a number of machines participating in the Paxos protocol. Likewise,

atomic broadcast [11] can be implemented with Paxos as an underlying primitive.

Over time Paxos has been extended and modified to emphasise different performance trade–offs. Multi–Paxos is the most typically deployed variant which allows for explicit agreement over a sequence of values. Another example, Fast Paxos [7], is a variant that reduces the number of message delays between proposing a value and receiving a response. More recently, Flexible Paxos [3] is a variant that relaxes the requirement on agreement between participants in the voting protocol.

There are also a number of similar algorithms that are not based on Paxos. Viewstamped replication [9] is primarily a protocol for implementing state machine replication that was developed independently of Paxos. Raft [10] is a modern alternative to Paxos that attempts to be more understandable.

1.2 Aims

The aim of this project was to produce an implementation of the Multi–Paxos variant of the Paxos algorithm. This is the variant that is most widely used in production systems and provides a foundation upon which to perform state machine replication. To demonstrate this technique, I implemented a simple key–value store on top of my Multi–Paxos implementation.

From the outset OCaml was to be used as the primary programming language for the implementation of the project. Its powerful type system allows for a number of errors to be rejected at compile time; a quality that is helpful in developing an application with such an emphasis on fault tolerance. OCaml also provides a rich third party ecosystem with libraries for crucial components of the system such as concurrency and RPCs.

In order to check the fault tolerance of the implementation it is necessary to simulate a distributed system on a simulated network. This simulator is used to check that the algorithm provides consensus in the face of a number of failure modes. It is also used to characterise the performance of the implementation in terms of latency and throughput.

Chapter 2

Preparation

This chapter describes the necessary background material required to understand Paxos; first in the simpler case of agreeing on one value and then going on to the Multi–decree case for agreeing on a sequence of values. From this a set of requirements for the project is established. Finally the software engineering aspects of the project are discussed including the choice of tools, build processes and third party libraries.

2.1 Theoretical background

2.1.1 Assumptions of the environment

When considering developing a system with distributed consensus, it is necessary to consider the assumptions made in the environment in which such a system will operate. This is to ensure there is enough functionality embedded in the system to ensure that consensus is reached under the given set of assumptions.

A *process* (or networked process) is an instance of the program running on a networked machine in a distributed system. Assumptions of these processes that participate in the system:

- Processes can undergo *crash failures*. A crash failure is defined as a process terminating but not entering an invalid state.
- Processes may recover from crash failures. They can rejoin the system in some valid state.
- Processes operate at arbitrary speeds. This cannot be distinguished by other processes from arbitrarily long delays in the network.

Assumptions of the network over which these processes communicate:

- All processes can communicate with one another.

- The network environment is *asynchronous*. That is, messages may take an arbitrarily long time to be delivered.
- Messages may be re–ordered upon delivery.
- Messages may be duplicated in the network.
- Messages may be dropped from the network.
- Messages are not corrupted or modified in the network.
- A message that is received by one process was, at some point in the past, sent by another process.

The last two assumptions preclude us from handling what are known generally as *Byzantine failures*.

2.1.2 Aim of consensus

With distributed consensus, we wish for a network of processes to agree on some value. It is assumed that processes can somehow propose values to other processes. The goal of distributed consensus, given a number of processes that can each propose some value v, is that one proposed value is chosen. This is the *single–decree* case; only one value is proposed by each process and only one is chosen.

In the *multi–decree* case, agreement is reached over a sequence of values. Each process will propose a sequence of values $v_1, v_2, \ldots, v_n$ and the goal of consensus is to choose one such sequence. This multi–decree case allows for *state machine replication* to be employed.

2.1.3 State machine replication

A desirable goal of distributed computing is to replicate an application across a set of machines so that each *replica* has the same strongly–consistent application state. This technique is referred to as State Machine Replication (SMR); it leads to both increased fault tolerance and higher availability. Multi–decree consensus algorithms provide a means of performing SMR.

Each process participating in the consensus algorithm runs the replicated state machine application and starts in the same state. By treating the values proposed as *commands* to perform state transitions, running a consensus algorithm

Role	Purpose	Number required
Proposer	Propose values to acceptors. Send prepare requests with proposal numbers.	$f + 1$
Acceptor	Decide whether to *adopt* a proposal based on its proposal number Decide whether to *accept* a proposal based on a higher numbered proposal having arriving.	$2f + 1$
Learner	Learn value chosen by majority of acceptors	$f + 1$

Table 2.1: Summary of the roles in single–decree Paxos. In this description the system can tolerate the failure of up to f of each given role.

causes each process to receive the same sequence of commands $c_1, c_2, \ldots, c_n$. Each process thus performs the same sequence of transitions from the same starting state and is therefore maintains a replica of the application.

Before considering how to implement SMR in the multi–decree case, it is useful to examine how Paxos operates in the simpler single–decree case.

2.1.4 Single–decree Paxos

Single–decree Paxos is the variant of the algorithm that allows for a single value to be chosen from a set of proposals and provides a foundation for the multi–decree case that will be considered next. The terminology used here follows Lamport's paper [6] describing the single–decree protocol in simple terms. Processes take the roles of *proposers*, *acceptors* and *learners*, each of which has a designated task in the algorithm. In reality these roles are often co–located within a single process but it is simpler to consider each separately. The purpose of each role and the number of each role required to tolerate f failures is summarised in Table 2.1.

Proposers that wish to propose a value v submit proposals of the form (n, v), where $n \in \mathbb{N}$ is called a *proposal number*. Each proposer may propose one proposal at a time and may only use strictly increasing proposal numbers. Furthermore, each proposer must use a disjoint set of proposal numbers. The Paxos algorithm is divided into a number of stages described below.

Phase 1a (Prepare phase) A proposer wishing to propose a value first sends a `prepare(`n`)` message to a majority of the set of acceptors, where n is the highest

proposal number it has used so far.

Phase 1b (Promise phase) In this phase an acceptor receiving a `prepare(`n`)` message must decide whether or not to *adopt* this proposal number. Adopting a proposal number is the act of promising not to accept a future proposal number n' such that $n' < n$. The acceptor will adopt n if it is the highest proposal number it has received thus far, in which case it will reply to the proposer with a `promise(`n''`,`v`)` message, where n'' is the highest proposal number it has previously accepted and v is the corresponding proposal's value. Otherwise, it can simply ignore the proposer or send a `NACK` message so the proposer can abandon the proposal.

Phase 2a (Accept phase) Upon receipt of a `promise(`n`,`v`)` message from a majority of acceptors, the proposer replies to each with an `accept(`n'`,`v'`)` message, where n' is the highest proposal number returned by the acceptors in the promise phase and v' is its corresponding value.

Phase 2b (Commit phase) An acceptor receiving an `accept(`n`,`v`)` message from a proposer will decide whether to commit the proposal for v. If the acceptor hasn't made a promise to adopt a proposal number higher than n, then it will commit v, otherwise it will ignore this message or send a `NACK` to the proposer.

Figures 2.1 and 2.2 show two example message flows between proposers and acceptors in the single Paxos protocol.

Once this process is completed, a majority of the acceptors will have chosen the same proposed value. Learners are required to each learn the value chosen by the majority. A number of different methods can be employed to achieve this. The simplest is to have acceptors chosen broadcast chosen values to the learners.

2.1.5 Multi–decree Paxos

Single–decree Paxos can be naively extended by allowing proposers to propose values one at a time. However, this is wasteful as it requires that proposers send `prepare` messages for each proposal they wish to make. Van Renesse et al [13] describes Multi–Paxos with an emphasis on use for state machine replication that forms the basis of the description that follows. In this paper the roles of the processes are changed to emphasise the purpose of the system for SMR. *Replicas*

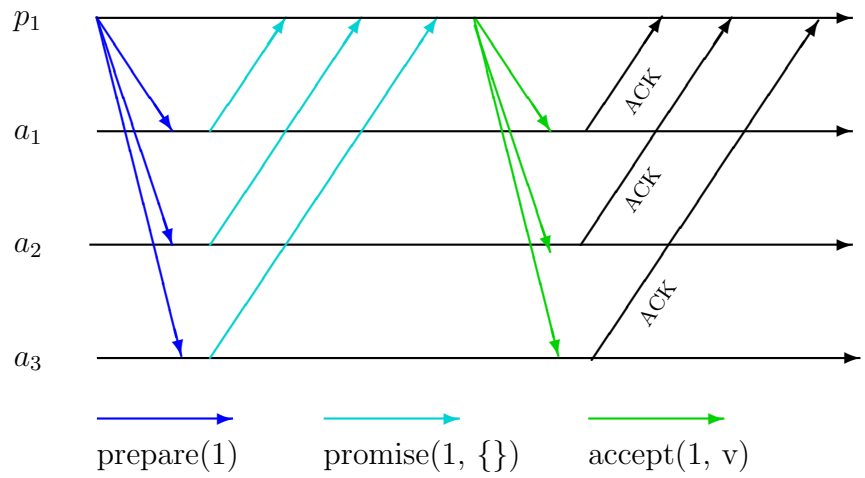

Figure 2.1: Timing diagram showing a single proposer p_1 committing a value v with three acceptors a_1, a_2 and a_3

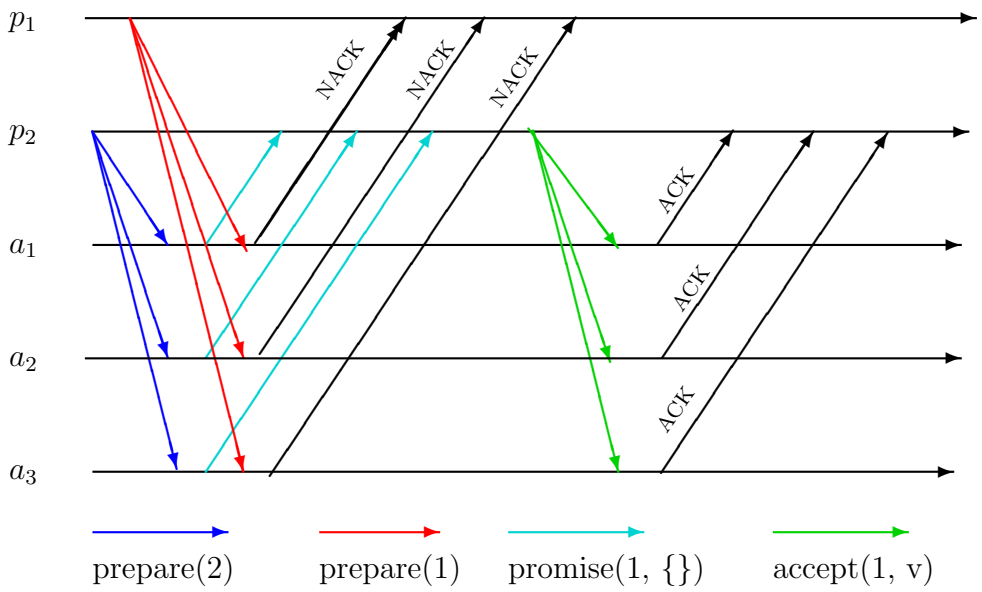

Figure 2.2: Timing diagram showing a proposer p_1 being preempted by acceptors a_1, a_2 and a_3 having already adopted a higher proposal number submitted previously by p_2

Multi–Paxos role	**Single–decree equivalent**	**Purpose**	**Number required**
Client	N/A	Send commands to replicas and receive responses	N/A
Replica	Learner	Receive requests from clients. Serialise proposals and send to leaders. Receive decisions and apply to the replicated application state. Handle reconfiguration of set of leaders	$f + 1$
Leader	Proposer	Request acceptors adopt ballots. Secure adoption of ballots in phase 1 of synod protocol Secure acceptance of ballots in phase 2	$f + 1$
Acceptor	Acceptor	Adopt the highest ballot they have received Accept pvalues of ballot they have adopted Fault tolerant distributed memory of pvalues	$2f + 1$

Table 2.2: Summary of the roles in Multi Paxos. In this description the system can tolerate the failure of up to f of each given role. Clients do not explicitly participate in the protocol and so there is no requirement on any number being live at any given time.

are introduced to implicitly act as the learners and maintain the replicated state machine. *Clients* are introduced as a means to externally submit commands to the replicas. Proposers are referred to as *leaders* as each takes the lead over some set of *ballots*. The new roles and their correspondence to the single–decree roles are summarised in Table 2.2.

Clients

The purpose of clients is to allow for commands to be sent externally to the system which can then be formed into proposals internally. This allows the system to behave in a manner more like that of a typically deployed distributed system

(with a client / server architecture) and provides a degree of failure transparency.

A command c takes the form (κ, cid, op), where κ is a unique identifier for the client, cid is a unique identifier for the command and op is the operation to be performed. Clients broadcast a `request(`c`)` message to the replicas and each is issued a `response(`cid`,`$result$`)` when consensus is reached and it has been applied to each replica's application state.

Replicas

Replicas receive commands and attempt to serialise them by converting each command c into a proposal (s, c), where $s \in \mathbb{N}$ is a *slot number*. The slot number describes the ordering of the sequence in which commands should be committed; this is not to be confused with the proposal number n in the single–decree protocol.

Different replicas may form different sequences of proposals and so broadcast a `propose(`s`,`c`)` message to the leaders and await a `decision(`s'`,`c'`)` message. The resulting decision may differ in its slot number and so the replica may have to re–propose the command. Upon receipt of decisions the replica will apply the associated operation to the application state, maintaining the replicated application.

Ballots and pvalues

Ballots are the structure over which leaders and acceptors operate. Ballots provide indirection, allowing leaders to secure agreement with acceptors for a number of proposals with different slot numbers, or for a given slot to be targeted by different ballots. This allows for a leader to perform the first phase of the synod protocol independently of having received any new proposals. It also increases the ability to execute the protocol concurrently for a number of slots at once.

Each ballot is uniquely identified by a *ballot number*. These are either pairs (r, λ) consisting of a round number $r \in \mathbb{N}$ and a leader identifier λ, or a specially designated least ballot number $\perp$.

Pvalues associate a ballot number with a proposal. They are triples (b, s, c) consisting of a ballot number, a slot number and a command. These are analogous to to the (n, v) pairs used in the single–decree case, except now by

encoding a leader's identifier directly in the ballot number each leader will have a disjoint set of ballots.

We require ballot numbers be totally ordered so that acceptors can compare them when choosing whether to adopt or accept. Letting $\mathcal{B}$ denote the set of all ballot numbers, we define the relation $\leq_{\mathcal{B}} \in \mathcal{B} \times \mathcal{B}$ which satisfies the following two conditions:

$$\forall\, (n, \lambda)\,, (n', \lambda') \in \mathcal{B}.\ (n, \lambda) \leq_{\mathcal{B}} (n', \lambda') \iff (n \leq n') \vee (n = n' \wedge \lambda \leq_{\Lambda} \lambda')$$
$$\forall\, b \in \mathcal{B}\ .\ \bot \leq_{\mathcal{B}} b$$

This implies that we require leader identifiers be equipped with a total order relation $\leq_{\Lambda}$ as well.

Quorums

Unlike replicas, acceptors do not each attempt to store their own consistent copies of decisions. Instead, it is required that at least a majority of the acceptors have in their memory a committed proposal. This is so that if any minority of the acceptors crash then there is still at least one acceptor with memory of the decision. A subset of the acceptors $Q \subseteq \mathcal{A}$, where $|\mathcal{A}| = 2f + 1$ for some $f \in \mathbb{N}$, is called a *majority quorum* or just a *quorum* if $|Q| = f + 1$.

Hence at each phase of the synod protocol leaders broadcast their message to all acceptors and wait for a quorum of responses. This guarantees that if any minority of the acceptors fail in the future then at least one will have retained the result of the current round of the protocol.

The Synod Protocol

The synod protocol is undertaken by the leaders and acceptors in order to decide the slot to which each command is committed. The protocol proceeds in two phases similarly to the single–decree case except now leaders and acceptors operate over pvalues.

Acceptors maintain a ballot number in their state their represents the last ballot number they have *adopted*. Adopting a ballot number represents a promise not to accept any pvalues that do not have that ballot number, ensuring acceptors do not accept conflicting pvalues. Acceptors *accept* a pvalue by adding it to the

set of pvalues they have already accepted. The algorithm requires that if two acceptors $\alpha, \alpha' \in \mathcal{A}$ have accepted (b, s, c) and (b, s, c') respectively then $c = c'$.

In the absence of receiving any `propose(`s`,`c`)` messages from replicas, leaders attempt to secure an initial ballot with ballot number $(0, \lambda_k)$, where λ_k is the identifier of the k^{th} leader. They do this by broadcasting a `phase1a(`b`)` message in the same format as of that below.

Phase 1a Leaders attempt to secure adoption of a ballot b by broadcasting a `phase1a(`b`)` message to the acceptors.

Phase 1b Acceptors receiving a `phase1a(`b`)` compare b to the highest ballot number b' they have adopted thus far. This is initially $\perp$ so acceptors will always adopt the first ballot number they receive. If $b' \leq_{\mathcal{B}} b$ then the acceptor will adopt this new ballot number, that is set $b' := b$. In either case, the acceptor will reply with a `phase1b(`b'`, pvals)` message, where pvals is the set of pvalues previously accepted.

If a leader receives `phase1b(`b'`, pvals)` from a majority quorum of acceptors and if $b = b'$ then it will proceed with phase 2 of the protocol. Otherwise the leader abandons the proposal, increments the round number of its ballot number and tries again. Likewise if a leader receives a response where $b' \neq b$ then it should restart the protocol with a higher round number.

Phase 2a Leaders then attempt to commit a given pvalue (b, s, c) by broadcasting a `phase2a(`b, s, c`)` message to the acceptors.

Phase 2b Acceptors receiving a `phase2a(`b, s, c`)` will compare b to their adopted ballot number b'. If $b = b'$ then they will accept the pvalue (b, s, c); otherwise they do not change their state. In either case they reply with a `phase2b(`b'`)` message.

Leaders will again wait for receipt of a `phase2b(`b'`)` from a majority quorum of acceptors. If $b = b'$ then they know the acceptor has accepted the pvalue and so broadcast a `decision(`s, c`)` message to the set of replicas so they can commit command c to slot s. Otherwise if $b \neq b'$ then a leader knows the acceptor has since adopted a different leader's ballot number and so should restart the protocol with a higher round number.

2.2 Requirements

2.2.1 System requirements

As previously noted, there are a number of papers that describe Paxos and its variants. Each presents the algorithm differently and nearly all of them present it theoretically, with little concern for functionality required in software. Implementation details are often entirely overlooked. Hence it is necessary to recognise both the theoretical requirements from the literature as well as identify additional functionality required to realise the algorithm as functioning software.

A *requirements analysis* of the final system was therefore performed. Information was collected from the literature about how the algorithm worked at a high–level before considering the practical functionality. This is gathered into the final system requirements listed below.

General requirements

- The ability to pass the executable file command line arguments for initialisation.
- Ability for each process to be nominated to, and start up in, a given role.
- Configuration files that contain the network addresses of each of the processes participating. Each process must be able to parse such a configuration file.
- The ability to log important information to disk or direct to `stdout` / `stderr`. The ability to log different information (e.g. debugging info versus error info) to different files.

Messaging subsystem

- Provide an abstraction so processes can send messages without concern of the underlying network.
- Send messages as required by the Multi–Paxos algorithm. Ability to serialise and deserialise arguments and results.
- Mask failures in the network from processes.

Application requirements

- A key–value store application that can be modelled as a state machine. This needs to have commands to create key–value pairs, update key–values pairs, read a value for a given key and delete a key–value pair for a given key.
- The store must be strongly consistent across all replicas.

Consensus algorithm requirements

- Client processes that can send commands to replicas and receive responses.
- Replicas that each maintain a strongly consistent copy of the application state. They must receive requests from clients and each attempt to propose their own serialisation of the requests to leaders.
- Leaders and acceptors take part in the synod protocol to produces a consistent serialisation of the commands proposed by replicas.

2.3 Software engineering

This section gives detail on the software development pratices and related work that was undertaken before development commenced. The methods, third party libraries, tools and test strategies that were employed to manage the project and improve productivity are discussed.

2.3.1 Starting point

The starting point of this project is the same as that described in the Project Proposal reproduced in Appendix C.

2.3.2 Methodology

The method by which the software was developed was to split the system into its logical constituents and develop each in turn, with as much independence from the each other as possible. To facilitate this, having been informed by the requirements, I deemed it necessary to separate the messaging system from the actual operation of Multi–Paxos itself.

This messaging system was the first module developed and was incrementally equipped with functionality that facilitated the rest of the program. The next logical division was to separate the operation of clients and replicas from that

of leaders and acceptors. Clients, replicas and associated functionality were developed first followed by that of leaders and acceptors in two development phases.

2.3.3 Libraries

OCaml has a large ecosystem of third party libraries available. They are used where it is necessary to avoid writing large bodies of code for common tasks. This project uses the OPAM[1] package manager to install, update and manage the libraries used.

A library of particular interest to this project is that which provides RPCs (Remote Procedure Calls). Cap'n Proto[2] was chosen as it has good support and documentation for its OCaml bindings[3].

Lwt[4] is a monadic concurrency library chosen for this project. It leverages the type system to express the promise of a computation as a *monad* of type `'a Lwt.t`. This library was chosen because it has thorough documentation and interoperates with Cap'n Proto.

This project also uses Core[5] on top of the standard library, providing useful functions and enriching others already present. Yojson[6] was used for serialisation into JSON. OUnit[7] was used to manage unit testing for this project, discussed in §2.3.6. Uri[8] was used to format URIs.

In the case of each library used their software license was available either through OPAM or the library's Github repository. YoJson required reproduction of their licensing information be distributed with the project. Appendix A gives details of each library's software license.

[1] https://opam.ocaml.org
[2] https://capnproto.org/
[3] https://github.com/capnproto/capnp-ocaml, https://github.com/mirage/capnp-rpc
[4] https://github.com/ocsigen/lwt
[5] https://github.com/janestreet/core
[6] https://github.com/mjambon/yojson
[7] http://ounit.forge.ocamlcore.org/
[8] https://github.com/mirage/ocaml-uri

2.3.4 Tools

Choice of tools is important for being productive in a language. MacVim[9] was used as a text editor. Merlin was used to provide code–completion and edit–time type inference features within MacVim.

Git was used for source control and versioning, with branches made for each new feature under development and merged back into the master. Backups were made to a remote repository on Github and periodically to Google Drive. All of the development work was undertaken on my personal machine with the option of fallback onto the MCS machines.

A network simulator was used to simulate the operation and performance of the implementation. Mininet[10], the simulator used in this project, uses process based virtualisation thats allows for arbitrary processes to be run on hosts on a virtual network. Mininet provides a complete Python API for setting up such simulations and performing experiments. Mininet required use of VirtualBox[11], virtualization software that allowed Mininet to be used on my own machine.

2.3.5 Build system

JBuilder was used as the build system for this project. It was chosen for its minimal configuration files (called *jbuild* files), automatic linking of OPAM packages and generation of `.merlin` files. It is also able to automatically run the Cap'n Proto code generator upon building.

2.3.6 Testing

Unit testing was performed in order to check the operation of functions at a fine–grained level within the program. OCaml's static type system ensures the compiler will catch a number of errors resulting from improper calls to functions and so unit tests were designed as a complement to this; tests focussed on correctly typed arguments to functions yielding appropriate results. The test suites were executed after successful builds of the project.

[9] https://github.com/macvim-dev/macvim
[10] http://mininet.org/
[11] https://www.virtualbox.org/

2.4 Summary

This chapter discussed the theoretical basis for implementing Multi–Paxos and state machine replication before going on to describe a set of requirements for an implementation of the algorithm, finishing with a description of the software engineering tools and techniques employed. With this in place it is possible to go on to describe the implementation of the algorithm and its underlying systems in Chapter 3.

Chapter 3

Implementation

This chapter describes the implementation. It begins with a high level overview of the structure of the program and initialisation. This is followed by an examination of the data structures that are used and goes on to describe the messaging system. Finally, each of the roles in the system is described in turn.

3.1 High level architecture

This section describes the architecture of the program. Each process participating in Multi–Paxos is an instance of the same executable configured to take the *role* of either client, replica, leader or acceptor. These roles each have a different function in the system within their *role module*, which encapsulates all the functionality associated specifically with a given role. Each of these modules will need to send and receive messages of some form; this is abstracted over by a *messaging system* which exposes an interface via which each can send messages or maintain a server that receives messages. Figure 3.1 shows communication paths between each of these subsystems.

Each process requires additional information when initialised, such as

1. The role it should take.
2. The address at which it should start a server to receive messages.
3. The set of addresses of all the participating processes with which it must communicate.

Items (1) and (2) are provided as comand line arguments when the program is executed. If the number of processes is large, then supplying a list of addresses to each process would prove tedious so (3) is addressed by using a configuration file, the relative path to which is provided as a command line argument.

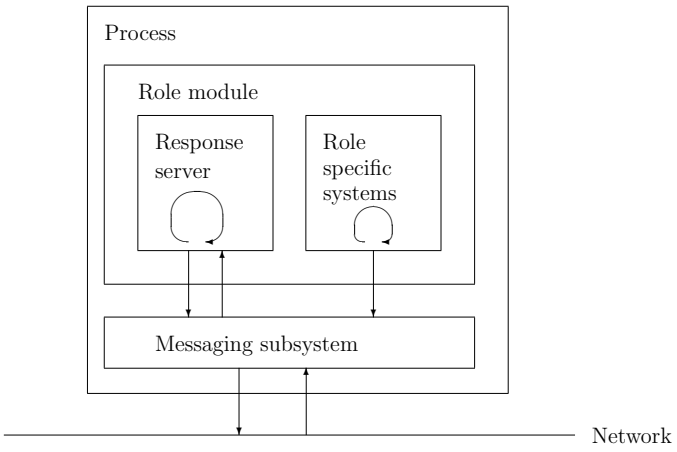

Figure 3.1: High level subsystems of a process

Configuration files are formatted as JSON[1], chosen since it is semi-structured data that can be written by hand and also parsed using Yojson. The file contains, for each role, a list of all of the addresses of each process in that role. Not every process needs to know the address of every other node, it is only necessary that each file parsed by each process contains the addresses of the nodes they are required to communicate with. In practice, it is more maintainable to have a single file used by each process.

The addresses discussed above are IPV4 IP address and port number pairs which are used both to send and receive. Internally a different addressing format is used, described in §3.3, where the addresses provided at initialisation ar mapped directly onto the internal representation.

3.2 Data structures

Before proceeding to the implementation of the systems described above, it is necessary to examine the data structures that will be used.

3.2.1 Identifiers

Unique identifiers are required to identify each process in the system. To avoid having a central authority distribute these identifiers a method where any process can generate their own unique identifer is used. Universally Unique

[1]https://tools.ietf.org/html/rfc7159

```
module type APPLICATION = sig
  type state = (int, string) List.Assoc.t
  type operation = Nop
                 | Create of int * string
                      ⋮
                 | Reconfigure of Uri.t list

  type result = Success
              | Failure
              | ReadSuccess of string

  val initial_state : state
  val apply : state -> operation -> state * result
        ⋮
end
```

Figure 3.2: Signature of the key value store module

Identifers[2] (UUIDs) are used as they are a well established standard and have support in the OCaml Core library. UUIDs are also totally ordered and so satisy the condition that we have such an ordering on the identifers of leaders.

3.2.2 Key value store

The key–value store is the application to be replicated. Each replica process maintains an instance of its state. The application need not maintain any synchronisation logic, it need only behave as a state machine. The state is represented by an *association list* that maps integer keys to string values. Operations each have their semantics described in Table 3.1. The `Reconfigure` command does not affect application state but is used by replicas to perform reconfigurations, described in §3.4.2. The application follows a state machine pattern in that if each replica starts in the same state and applies the same sequence of operations the resulting state is the same.

When a command has been committed to a slot by the consensus algorithm, the command's operation is applied to the state. This returns a new state and a result. The result is the value that is returned to the client; either `Success`, `Failure` or `ReadSuccess(V)`. The signature of the key value store is shown in Figure 3.2.

[2]https://tools.ietf.org/html/rfc4122

Command	**Argument**	**Semantics**
Nop		No operation, no change to state Returns Success
Create	`(K,V)`	If key not present, add new `(K,V)` pair to the state and return `Success`. Otherwise return `Failure`.
Update	`(K,V)`	If key present, update pair with key `K` to value `V`. Otherwise return `Failure`
Read	`K`	If key present read value `V` associated with key `K` and return `ReadSuccess(V)`, else return `Failure`.
Remove	`K`	If key present remove pair with key `K`, else return `Failure`.
Reconfigure	`uris`	Treated as a `Nop` by the application. Reconfigures the leaders to those with addresses in `uris`

Table 3.1: Set of operations that can be applied to the application state, along with their corresponding arguments and their semantics. Note that Create and Update commands are separate, each with their own success and failure semantics in order to reduce ambiguity of how the application operates.

```
type t = Bottom
       | Number of int * leader_id
```

Figure 3.3: Types of ballot numbers.

```
type t
val bottom : unit -> t
val init : leader_id -> t
val succ_exn : t -> t
```

Figure 3.4: The interface exposed by the Ballot module. These are the types of functions that can be used to generate ballot numbers. Note the naming of the function `succ_exn` implies it can throw an exception, which occurs when calling the function on `bottom`.

3.2.3 Ballots

The definition of ballot numbers from §2.1.5 lends itself to representation by an algebraic datatype. Ballots are hence the tagged union of Bottom (representing the least ballot $\perp$) or a pair consiting of an integer and a leader identifer. The type definition is listed in Figure 3.3.

Figure 3.4 shows the interface exposed by the Ballot module. Note that the concrete type of a ballot number isn't exposed, only the abstract type `t`, hiding the internal representation of the type. This allows the type system to prevent errors arising from improper use of ballots.

Functions are supplied to safely manipulate ballots outside the module. A given leader can generate an initial ballot number with their identifer. Subsequent ballot numbers can be generated by calling a successor function, which increments the round number each time. By exposing such an interface we prevent errors such as using negative round numbers from being able to compile. The module provides functions to test structural equality and the ordering of ballots. Also in the Ballot module are functions for serialisation and deserialisation for use by the messaging system.

```
type non_blocking_message = ClientRequestMessage of command
                          | ProposalMessage of proposal
                          | DecisionMessage of proposal
                          | ClientResponseMessage of command_id *
                              result

val send_non_blocking : non_blocking_message -> Uri.t -> unit Lwt.t

val send_phase1_message : Ballot.t -> Uri.t ->
  (Core.Uuid.t * Ballot.t * Pval.t list, string) Result.result Lwt.t

val send_phase2_message : Pval.t -> Uri.t ->
  (Core.Uuid.t * Ballot.t, string) Result.result Lwt.t
```

Figure 3.5: Types of non-blocking messages

```
val start_new_server :
    ?request_callback:(command -> unit) ->
    ?proposal_callback:(proposal -> unit) ->
    ?response_callback:(command_id * result -> unit) ->
    ?phase1_callback:(Ballot.t ->
      unique_id * Ballot.t * Pval.t list) ->
    ?phase2_callback:(Pval.t -> unique_id * Ballot.t) ->
    string -> int -> Uri.t Lwt.t
```

Figure 3.6: Function to start a new server that presents a number of possible callbacks

3.3 Messages

3.3.1 Interface exposed

The messaging subsystem is concerned with the sending and receipt of messages between processes. It transforms messages from OCaml types into a form suitable for transport over the network, handling packetisation, rentramssion and masking failures.

There are a number of different messages that are necessary to send in Multi–Paxos. In this chapter we focus on the capability of sending these messages rather than their role in the algorithm itself, which is discussed in depth later in the chapter.

Messages are broadly divided into two cateogries: *blocking* and *non-blocking*. Blocking messages are those in which the sender waits on a response, whereas when sending a non-blocking message the process does not wait for a reply. The interface for sending messages is presented in Figure 3.5. The types of non-blocking messages represent the arguments that are associated with each message.

The function `send_non_blocking` takes a value of type non-blocking message and the URI address of the destination. As these messages do not explicity require a reply they immediately return a `unit Lwt.t`. As we can regard a message that is never delivered to a recipient as indefinitely delayed in the network, this type is returned regardless of any errors actually encountered.. It is important to note here that any errors that arise from messages failing to deliver are handled by Multi–Paxos, not the messaging system.

Blocking messages return a type immediately. Hence they are separated into specific functions, `send_phase1_message` and `send_phase2_message`. The arguments represent the phase1a / phase2a messages and the responses represent the phase2a and phase2b messages. This reduces complexity in messaging. Since acceptors send messages only in response to messages from leaders we can treat them as executing remote procedures for leaders. The return type is wrapped in a `Result.result` to express the possibility of an error. These are discarded by leaders when checking for majority quorums.

The message module also exposes a function to start a new server, the type of which is listed in Figure 3.6. The server requires a string and integer representing the IP address and port number on which to listen and a number of *optional functions* that each represents a callback. These functions are called when a corresponding message is received by the server and return the response given by the return type. A process can therefore interoperate with the server by passing functions it wishes to have called whenever a message is received.

3.3.2 Cap'n Proto

Cap'n Proto is used as the underlying system for sending messages. Cap'n Proto requires that one writes a schema file describing a service; this contains data structures and messages that need to be serialised. This schema file, listed in full in Appendix B, contains an interface that describes the service made available to the messaging system. Crucially the schema contains a method for every message, the types mapping to the associated message type described in

```
let service_from_uri uri =
  try Lwt.return_some (Hashtbl.find sturdy_refs uri)
  with Not_found ->
    let client_vat = Capnp_rpc_unix.client_only_vat () in
    let sr = Capnp_rpc_unix.Vat.import_exn client_vat uri in
      Sturdy_ref.connect sr >>= function
        | Ok capability ->
            Hashtbl.add sturdy_refs uri capability;
            Lwt.return_some capability
        | Error _ ->
            Lwt.return_none
```

Figure 3.7: Function that returns a Cap'n Proto capability for the message service of a given URI

Figure 3.5. However, some of the arguments in the Cap'n Proto schema file are stored as `Text` rather than as the types in Figure 3.5, as they are serialised into JSON strings. This provides a degree of extensibility in the way messages are formatted. For example, this helps avoid situations where a new application command would require rewriting and re–compilation of the schema.

A compiler tool is used to generate the OCaml signatures and structures for the message API. The function `start_new_server` returns a service object that has a method for handling receipt of each message. In each of these methods the arguments are deserialised from the Cap'n Proto representation into the message type used by the system and the corresponding callback passed to `start_new_server` called. The result of calling this function is then serialised and returned as a response.

Sending a message requires translating a Cap'n Proto URI[3] that addresses a server into a *capability* which represents a stateful connection to a server. For each server initialized with the `start_new_server` function the supplied IP address and port number are mapped to a URI that is used to address the internal capability. Given a URI one can derive a sturdy reference to the capability and connect via that sturdy reference. Each connection runs over TCP and so rather than have each message setup and teardown a TCP session a cache of connected capabilities is stored by the messaging module that maps URIs to capabilities, listed in Figure 3.7. The function searches the table hashed by URIs first to see if a connection has already been established and returns that if it does. Otherwise if there is a

[3]https://tools.ietf.org/html/rfc3986

```
type t = {
  id : replica_id;
  mutable app_state : app_state;
  mutable slot_in : slot_number;
  mutable slot_out : slot_number;
  mutable requests : command list;
  mutable proposals : proposal list;
  mutable decisions : proposal list;
  mutable leaders : Uri.t list;
}
```

Figure 3.8: Types of records representing replica state

cache miss an attempt to connect to the capability is initiated. If this results in a success then the capability cached and then returned. Note here the return type is `Capability.t option Lwt.t` since the connection could result in an error.

3.4 Clients and replicas

3.4.1 Clients

Clients are the simplest processes in the system as they only send commands $(\kappa, \text{cid}, \text{op})$ to replicas in `request(`$\kappa, \text{cid}, \text{op}$`)` messages, where κ is an identifier of the client of type `Uuid.t * Uri.t`. The identifier value is used to uniquely identify the client and the URI is a Cap'n Proto address required for replicas to direct their responses to clients in question.

Each client also maintains a server on a given host name and port number to which `response(`$\text{cid}, \text{result}$`)` messages are sent. This represents the result of command with identifier cid having been applied to the application state, its order with all other commands having been serialised consistently across the system by Multi–Paxos.

3.4.2 Replicas

Following on from clients was the implementation of replicas. The state of a replica is stored as a record with mutable fields, listed in Figure 3.8. Replicas implicitly take the implicit role of learners in order to learn the decided serialisation of client requests. Rather than have the replicas explicity request the result of the synod protocol, replicas forward the requests made by

clients onto the leaders and later receive a response, learning the result implicitly.

Replicas maintain three queues:

- `requests`: A queue of commands received from client requests.
- `proposals`: A queue of commands tagged with a provisional slot number, representing the replica's proposed serialisation of the commands.
- `decisions`: A queue containing the serialisation of commands that has been decided on by the configuration of leaders and acceptors.

These queues are stored by the replica as types `command list` and `proposal list`, as lists in OCaml have a collection of useful helper functions. Further, the type `proposal list` is structurally equivalent to `(slot_number, command) List.Assoc.t`, allowing for proposals to be looked up by their key or inverted and looked up by their command, functionality necessary for managing the flow of proposals through the queues.

In order to track the next slots in which to propose and commit, each replica maintains two counters:

- `slot_in`: Represents the lowest next available slot for which no proposal has yet been allocated; that is $\nexists\ (s, c) \in$ proposals. $s =$ slot_in.
- `slot_out`: Represents the lowest next available slot for which no decision has been committed; that is $\nexists\ (s, c) \in$ decisions. $s =$ slot_out.

Commands and proposals move through this system of queues as shown in Figure 3.9. `request(`c`)` messages sent by clients are entered into the requests queue. Concurrently the replica will attempt to dequeue commands in a producer–consumer relationship, proposing each to `slot_in`, the lowest free slot, and incrementing `slot_in`. Each of these commands is tagged with this slot number and entered into the proposals queue. Once in this queue, it can then be proposed to the synod protocol by broadcasting a `propose(`s`,`c`)` to the set of leaders.

Leaders will at some point in the future return a `decision(`s`,`c'`)` message for each slot s for which a command was proposed. If $c = c'$ for a proposal $(s, c) \in$ proposals then it is removed from proposals and added to decisions. However if $c \neq c'$ then a different command has been decided for the slot than the one proposed by this replica (i.e. the replicas disagreed on their serialisations). In this case (s, c') is committed to decisions and c is returned to requests so that

it can be re–proposed for a different slot number.

Every time that a proposal is committed in the decisions queue, the proposal $(\texttt{slot\_out}, (\kappa, \text{cid}, \text{op}))$ has its operation applied, followed by `slot_out` being incremented. Even though the decisions will be committed in the same sequence by each replica, the order in which they are committed may differ. By sweeping through the decisions queue and executing each we ensure each replica updates the application state in the same order. The application state is updated by applying op to the replica's application state, resulting in an updated state and a result res. A `response(cid,res)` message is sent to the client with identifer κ[4].

Reconfigurations represent a change in the set of leaders and acceptors participating in the synod protocol. Replicas are expected to be able to reconfigure, that is to change the set of leaders with which they share proposals. Reconfigurations are operations that do not alter the application state, instead updating the set of leaders to a new configuration.

A *window* is maintained, so that only a maximum `WINDOW` number of commands are ever being decided upon by the synod protocol at any given time; reconfiguration commands do not take effect until there are no more commands in flight for the previous configuration. Hence all pending proposals will have been decided by the time the reconfiguration has taken place. This requires maintaining the invariant $\texttt{slot\_in} < \texttt{slot\_out} + \texttt{WINDOW}$. Therefore a maximum `WINDOW` number of proposals are being worked on by the synod protocol at once. Figure 3.10 shows an example window undergoing this process.

3.5 Synod protocol

Replicas each propose their own serialisation of commands; varying processing speeds and delays in the network along with crash failures of replicas can cause these serialisations to differ. It is the function of the synod protocol to return a consistent serialisation of the proposals to the replicas. This serialisation must be decided upon in a fault–tolerant manner by the leaders and acceptors. Before we

[4]Recall client identifiers have a URI component so responses can be sent without replicas maintaining a mapping of identifiers to URIs.

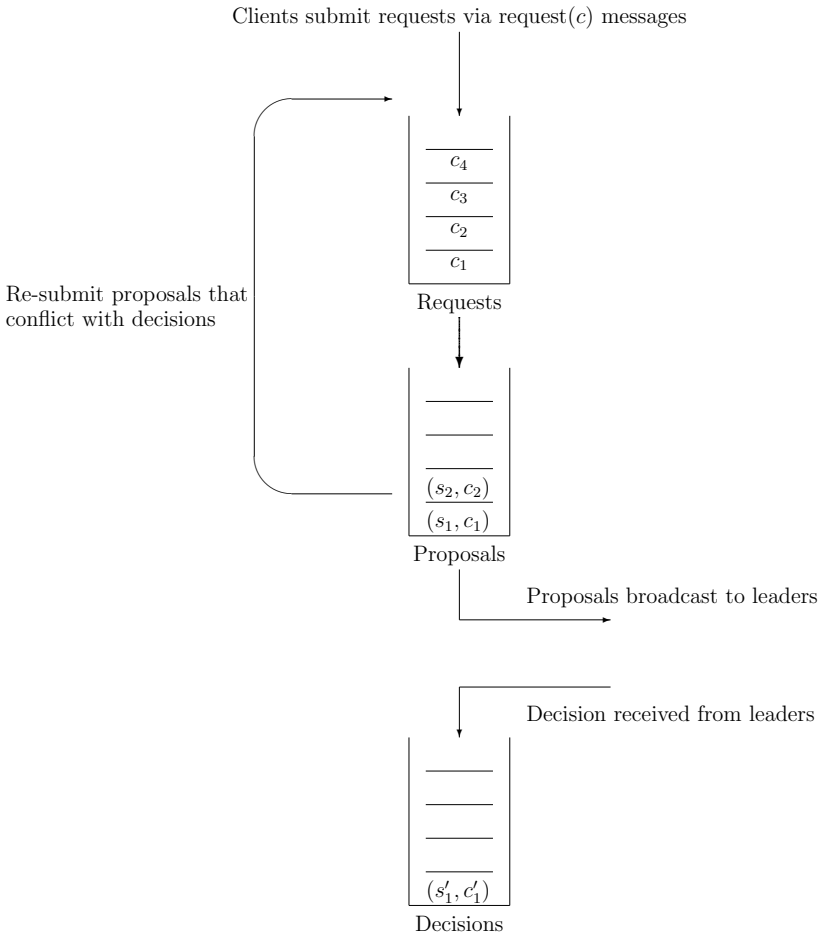

Figure 3.9: Flow of requests and proposals through a replica's queues

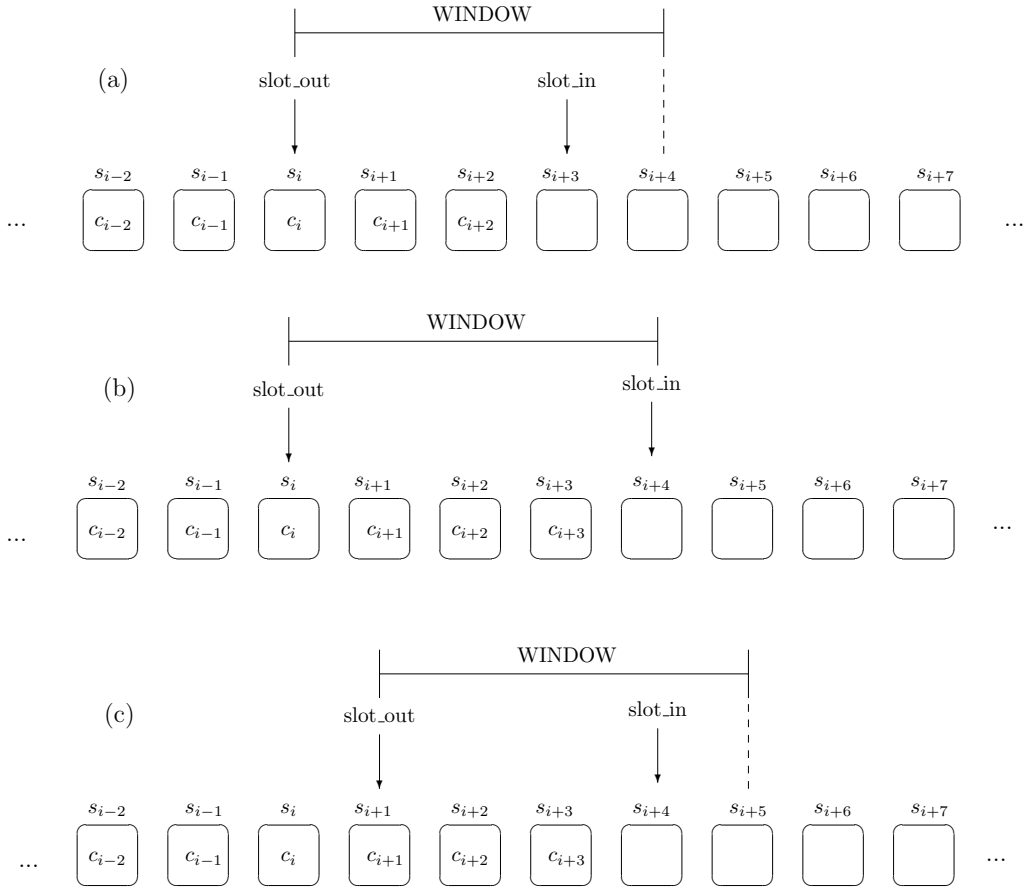

Figure 3.10: (a) shows a window with one available slot into which a command c_{i+3} is proposed, leading to the window shown in (b). In (b) however no further slots are available for proposals since `slot_in` $<$ `slot_out` + `WINDOW`. In (c), a decision is received for c_{i+1} and so the `slot_out` is incremented and now another slot is available for proposals.

```
type t = {
  id : unique_id;
  mutable ballot_num : Ballot.t;
  mutable accepted : Pval.t list
}
```

Figure 3.11: Types of records storing acceptor state

examine the role that leaders play in the next step of the algorithm it is necessary to observe the functionality of acceptors.

3.5.1 Acceptors

Acceptors form a distributed fault tolerant memory of the consensus protocol. As with other processes, an acceptor's state is stored in a record with type given listed in Figure 3.11. Acceptors maintain the following mutable state

- `ballot_num`: The acceptor's most recently *adopted* ballot number, initially equal to $\perp$ so that an acceptor adopts the first ballot it receives.
- `accepted`: The list of pvalues which the acceptor has *accepted*, initially empty.

Acceptors represent passive processes in the algorithm. They change their state and send messages only in direct response to messages from leaders; other than when initialising, all processing occurs upon receiving a phase 1a or phase 2a message from a leader. Given this, an acceptor is just required to maintain its state and provide two callbacks, listed in Figure 3.12 and 3.13.The acceptor maintains a server to receive phase 1a messages and one to receive phase2a messages. Recall from 3.3 that these are blocking messages and so the phase 1b and 2b replies are just the return type of these callbacks.

The callbacks may be executed concurrently by the underyling server. Hence it is necessary to enclose both the callbacks within critical sections to ensure data mutated in one callback isn't used in another. The function `Mutex.critical_section` takes a function that contains the contents of the callbacks and the mutex to lock when entering the critical section. When one function holds the lock any other callbacks will block upon attempting to hold the lock and receive the lock when the function holding the lock exits.

```
let phase1_callback (a : t) (b : Ballot.t) =
  Mutex.critical_section callback_mutex ~f:(fun () ->
    if a.ballot_num < b then
      a.ballot_num <- b;
    (a.id, a.ballot_num, a.accepted))
```

Figure 3.12: Function called when an acceptor receives a phase1a message with associated ballot.

```
let phase2_callback (a : t) (pval : Pval.t) =
  Mutex.critical_section callback_mutex ~f:(fun () ->
    let (b,_,_) = pval in
    if b = a.ballot_num then
       (if not (List.mem a.accepted pval ~equal:Pval.equal) then
         a.accepted <- pval :: a.accepted);
    (a.id, a.ballot_num))
```

Figure 3.13: Function called when an acceptor receives a phase2a message with associated pvalue.

Leaders broadcast `phase1_callback(`b`)` to the acceptors when they seek adoption of ballot number b from a majority quorum. In this case, when an acceptor receives a `phase1_callback(`b`)` message, it will adopt the ballot if it is greater than the last ballot it adopted (and hence inductively it is greater than any ballots adopted previously). The acceptor will then return a triple consisting of its identifier, the ballot it has adopted and the list of accepted pvalues. The identifier of the acceptor is returned so leaders can track which acceptors have replied. Rather than just not reply when it does not adopt b, it responds with the already adopted ballot so that leaders can be *preempted* and try a higher ballot number. The list of accepted pvalues is returned so that if the acceptor has accepted any pvalues from other leaders in the past the leader attempting to secure adoption of b will learn about them from this reply.

Leaders send `phase2_callback(pval)` messages, where `pval` $= (b, s, c)$, when they believe b has been adopted by a majority quorum of acceptors; this is the leaders request to commit the proposal (s, c). When an acceptor receives such a message they may have adopted a higher ballot number or the message may have been delayed. Hence if the acceptor's `ballot_num` $= b$ then the acceptor does accept the pvalue (b, s, c) and adds it to the list of accepted pvalues; otherwise it doesn't. In either case, it returns its identifier and `ballot_num`, so that in case the leader will know whether (b, s, c) was accepted or a higher ballot had been

```
type process_response = Adopted of Ballot.t * Pval.t list
                      | Preempted of Ballot.t
```

Figure 3.14: Types of responses produced by scouts and commanders

adopted by the acceptor.

In this situation each acceptor may have a different list of accepted pvalues. When we note that leaders only decide that a proposal (s, c) is committed after receiving a majority quorum of phase 2 messages then each such proposal must have a corresponding ballot number (b, s, c) accepted a majority of acceptors. This is what gives us the fault tolerant memory; the system can tolerate the crashes of a minority of acceptors f out of $2f + 1$ total participating and still have a consistent memory of the sequence of proposals.

3.5.2 Leaders

Leaders receive from each replica a serialisation of the set of proposals. The leaders are required to between them, in a manner that tolerates the failure of all but one leader, return a single serialisation of the proposals to each of the replicas.

Leaders spawn and manage *scouts* and *commanders* in order to separate attempting to secure adoption of ballots and attempting to secure acceptance of pvalues. Each scout and commander is a sub–process with associated state and execution context. Rather than using the existing messaging subsystem which would produce excessive overhead for managing exchange of data between sub–processes they instead communicate locally.

Each sub–process has a lifetime in which it is spawned, messages acceptors, receives responses, performs some processing and then terminates. A terminating sub–process may return result of its computation to the leader that spawned it. In the case of scouts, this is either a notification of a preemption having occurred or its ballot having been adopted. Commanders may notify the leader on termination of a preemption having occured. The types of these responses are listed in Figure 3.14.

We treat the sub–processes and the leader that spawned them as engaging in a producer–consumer relationship. Figure 3.15 lists code pertaining to the response queue used. Terminating sub–processes produce responses and enqueue them

```
let queue_guard = Lwt_mutex.create ()
let message_queue : process_response Queue.t = Queue.of_list []
let send msg =
  Lwt_mutex.lock queue_guard >|= (fun () ->
  Queue.enqueue message_queue msg) >|= fun () ->
  Lwt_mutex.unlock queue_guard
```

Figure 3.15: Functions and values for manipulating the message queue of process responses

onto `message_queue`, a queue of responses, via the `send : process_response -> unit Lwt.t` function. The leader will periodically consume these responses from the queue, in turn possibly spawning more sub–processes. The queue is protected by a guard mutex `queue_guard` that prevents races from occuring when the leader or any sub–processes attempt to concurrently mutate the state of the queue.

Scouts

As described above, the purpose of a scout sub–process is to take a ballot number and attempt to secure adoption of that ballot number with a majority quorum of acceptors. The scout is spawned by a commander, passed this ballot number and terminates by enqueuing a `Adopted(`b`,pvals)` or `Preempted(`b'`)` response for the leader to consume.

Scouts are described by a signature listed in Figure 3.16. Each scout's state is represented by a mutable record of type `t'`. Calling `spawn` and passing as arguments a leader and a ballot number will initialise a new scout. Since its execution is asynchronous the function returns `()` immediately. Note that the ballot number b over which it seeks adoptions is immutable; even though a leader works over ever increasing ballot numbers, b is fixed for the duration of the a scout's lifetime.

Upon being spawned, a scout broadcasts, in parallel, to the acceptors a `phase1a(`b`)` message. Each of these is bound monadically with a function that returns `()` if an error occured in message delivery (in keeping with the messaging semantics) or the phase 1a return value $(\alpha, b', \texttt{pvalues}')$. Upon receiving each message, the scout checks if it has already obtained a majority quorum of responses (not including the response just received), in which case it discards the response.

```
module type SCOUT = sig
  type t' = {
    b : Ballot.t;
    acceptor_uris : Uri.t list;
    receive_lock : Lwt_mutex.t;
    mutable pvalues : Pval.t list;
    mutable quorum : (Uri.t, unique_id) Quorum.t;
    mutable terminated : bool
  }
  val spawn : t -> Ballot.t -> unit
end
```

Figure 3.16: Signature of scouts

If $b = b'$ then the scout adds `pvalues'` to the set `pvalues` already stored and adds α to the quorum of responses it has received. If the scout has now obtained a majority quorum then the scout terminates, adding a `Adopted(`b`,pvals)` message to the queue for a leader to consume. If not, the scout continues to wait for further responses from the acceptors.

If $b \neq b'$ then an acceptor has adopted a higher ballot number from another leader and so the scout terminates with a `Preempted(`b'`)` message so that the leader is notified that this ballot has been abandoned.

Commanders

Commanders are spawned and passed a pvalue (b, s, c). A commander spawned with this pvalue operates under the assumption phase 1 has concluded for the ballot b and so the commander will attempt to secure acceptance of a pvalue with ballot number b.

It begins by broadcasting a `phase2a(`b`,`s`,`c`)` message to the acceptors and waits for a response from each. It behaves in a similar pattern to a scout: If the commander already has a majority quorum of responses it discards the response. If not, then the commander examines the (α, b') return type.

If $b = b'$ then the commander adds α to the quorum. If, having added α the commander has secured a majority quorum then the proposal (s, c) has successfully been committed. The commander broadcasts to all replicas a `decision(`s`,`c`)`; otherwise the commander continues to wait for further responses.

If $b \neq b'$ then, just as we had with scouts, the commander will terminate with a `Preempted(`b'`)` so the leader can abandon this ballot.

Leaders

Having described the operation of scouts and commanders, it is now necessary to discuss when leaders spawn sub–processes and how they process their responses. Leaders maintain the following important state:

- `ballot_num`: initially $(0, \lambda)$ for leader with identifier λ. This is the ballot over which the leader currently attempts to secure adoption and acceptance of a corresponding pvalue.
- `active`: a boolean value that describes whether the leader is currently in *active* or *passive* mode.
- `proposals`: the list of proposals the leader has received from replicas.

A passive leader has spawned a scout and is waiting for phase 1 of the synod protocol to conclude. The leader initialises in passive mode when it attempts to perform phase 1 with the ballot $(0, \lambda)$. Conceptually we can treat leaders as having receiving three different kinds of messages: proposals from replicas, adoption messages from scouts and preemption messages from scouts or commanders (although of course the implementation of how these are processed differs).

Leaders receive `Propose(`s, c`)` messages from replicas and, if they're not already present, add them to their list of proposals. If the leader is active then it will spawn a commander to attempt to commit this proposal, since a quorum of acceptors will have adopted the leader's current ballot. If the leader is not active then it simply waits until an adoption message is received, in which case it will then spawn commanders for all stored proposals.

Preemption messages `Preempted(`b`)` occur when a sub–process receives from an acceptor a ballot number b higher than `ballot_num`. In this case the commander moves into passive mode, increments the round number of its ballot and spawns a new scout to attempt to secure adoption with this higher ballot number.

An `Adopted(`b`,pvals)` message indicates that phase 1 of the protocol has concluded. The leader will in this situation be in passive mode and so enter active mode, beginning phase 2 of the protocol. From `pvals` the leader for each slot number the command with the highest corresponding ballot and adds it to their

list of `proposals`, removing from `proposals` any that have the same slot number. For each of these proposals the leader spawns a commander sub–process for each pvalue, executing phase 2 of the protocol and seeking acceptance for each.

3.6 Summary

This chapter described the implementation details of the project; beginning with a high level overview of the architecture and then delving into the message system, each role a process can take and their ability to communicate. Chapter 4 goes on to describe the evaluation of the software that took place.

Chapter 4

Evaluation

With an implementation of Multi–Paxos having been developed it is necessary to evaluate it, both in terms of its ability to perform under the assumptions of the environment in which it operates and also to characterise its performance in a number of typical cases.

4.1 Experimental setup

This section describes the framework upon which simulations were setup to allow for evaluation to take place.

4.1.1 Experimental measurements

Performance evaluation requires the measurement and subsequent analysis of experimental data. There are two key metrics required for evaluating the performance of networked systems – *latency* and *throughput*.

Latency is the time taken elapsed between a client sending a request to the system and receiving a response. It is measured by a client broadcasting a command with a `Nop` operation to the set of replicas and starting a timer. The timer is stopped when the response with the corresponding command identifier is received and the elapsed time is calculated and output to a trace log file.

Throughput is the rate at which the system services requests and hence measured in requests per unit of time. Clients submit requests at a fixed rate (e.g. 10 requests per second) and the number of responses returned per second is measured by clients. This gives a throughput for a given arrival rate and allows for a *peak throughput* measurement to be made. That is, a certain arrival rate of requests will maximise the output throughput which represents the maximal rate at which the system services requests.

4.1.2 Mininet

Mininet is the network simulator that was used for the evaluation of the implementation, running a virtual machine with a Linux guest operating system with 2GB of memory. Mininet ships with a Python API that allows for scripts to run network simulations. Mininet allows for arbitrary Linux processes to be run on any of the virtual hosts. This involved installing an OCaml runtime to execute bytecode compiled on the development machine. The simulation runs Python scripts that performs simulations that proceeds as follows

1. Setup up network topology, populating it with hosts and switches and their interconnects (and the performance parameters)
2. For a given number of each role participating, a JSON configuration file is generated that lists each host IP address and port number
3. Multi–Paxos instances are started via the OCaml runtime on each virtual host with their parameters set (e.g. whether they are clients)
4. The simulation starts and proceeds to execute for a given amount of time
5. The simulation is terminated. Configuration files are deleted and the virtual network is torn down and each process is terminated.

Log files are saved by each process on a shared filesystem with the host machine upon which they are analysed by another Python script which produces plots using matplotlib[1].

Unless otherwise stated each experiment used 1Gbps, 20ms delay links with a 0% packet loss probability[2] and switches with queue sizes of 10^7 packets. These characteristics were chosen to help prevent properties of the network from dominating measurements of system performance. The topology of the networks used is discussed in 4.1.4.

4.1.3 Duelling leaders and the impossibility result

When performing initial experiments on the system it became apparent that systems that have more than one leader would regularly livelock when processing requests from clients. Inspection of log files led to me finding out this was a

[1] https://matplotlib.org/

[2] As Cap'n Proto uses TCP the developed system does not suffer failures from packet loss, rather it will simply introduce further delays caused by retransmissions

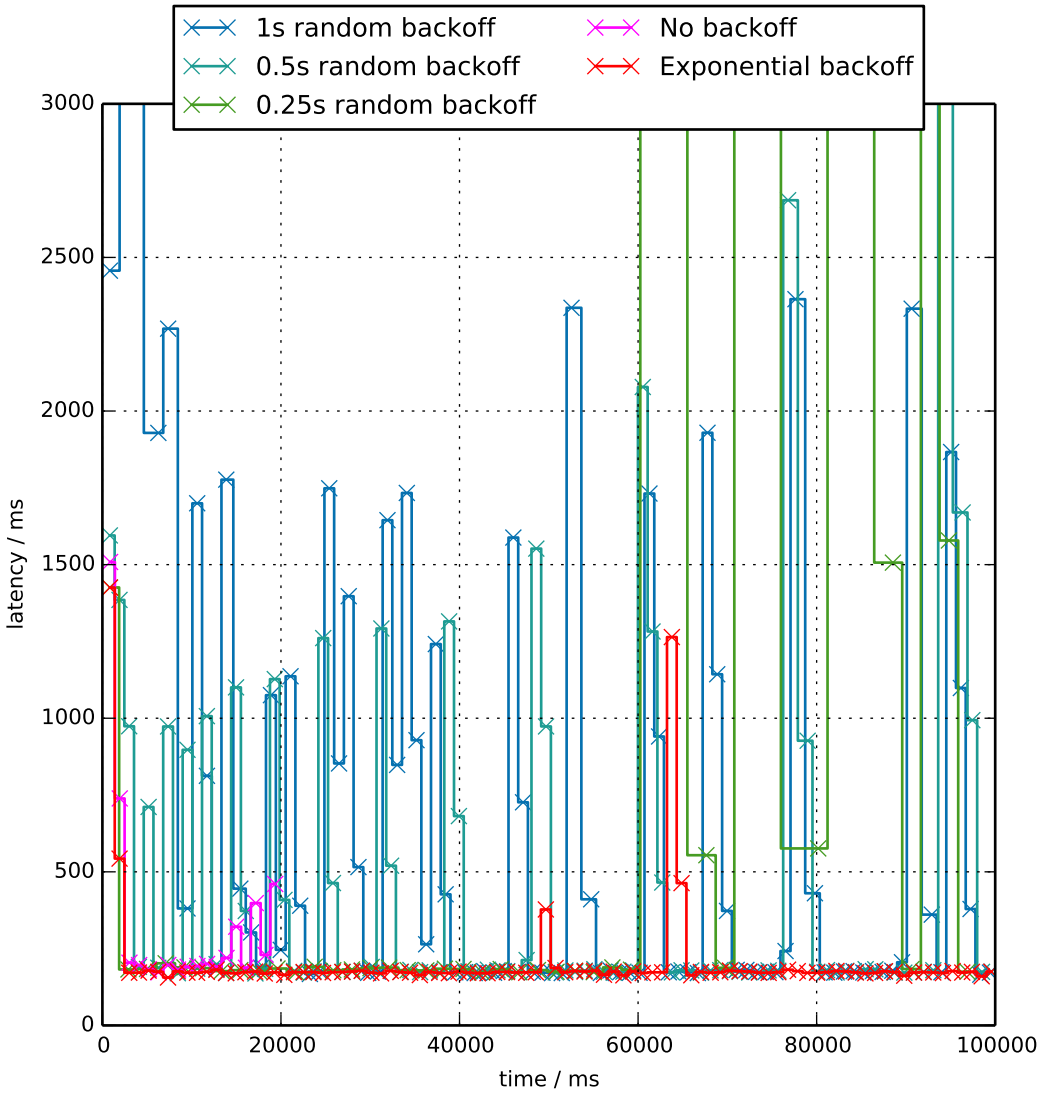

Figure 4.1: Graph of backoffs

well–known case of *duelling leaders*. This occurs when leaders will attempt to gain adoption of their ballot but are preempted by another leader. The leader will then attempt to gain adoption of another ballot before the other leader's ballot is accepted, causing the other leader to attempt adoption of a higher ballot. The leaders will continually increment their ballots and seek adoption without any given ballot being accept, livelocking the system and leaving it unable to make progress.

There is a well known result called the FLP Impossibility result, outlined in Fischer et al's paper [2] that proves that consensus is never guaranteed in an asynchronous environment that permits crash failures. The problem of duelling leaders is an instance of this problem and so there is no guarantee that consensus can be reached in an asychronous environment.

However, there are methods to vastly reduce the chance that leaders will duel. One simple method is to have a *random backoff* so that each leader will pause their operation for a random amount of time in some range. This increases the chance that a leader will have its ballot accepted before being preempted by another leader. A slightly more sophisticated method is to employ *exponential backoff*. In this case each leader will employ a random backoff within some range and upon being preempted this range is doubled. The backoff time is reduced to an initial value when an adopted message is received by a leader. This reduces the amount of idle waiting that may occur when using a random backoff and no duelling is occurring.

Figure 4.1 shows a number of traces with different backoff strategies. With no backoff, the system halts after 18 requests are processed and makes no further progress. Three traces of random backoffs with different maximum wait times were collected. With greater maximum wait times there are a number of greater sporadic delays in the system as these leaders often have to wait idly even when duelling is not occuring. Making the random timeout too small, as in the 0.25s case, reduces the number of delays but increases the chance leaders will duel for some time before one waits long enough for the other to make progress, resulting in very large delays. The exponential backoff (starting with a value of 0.25s) performs best, with very little delay in the general case and ocassional delays as backoff windows increase in size when leaders do duel.

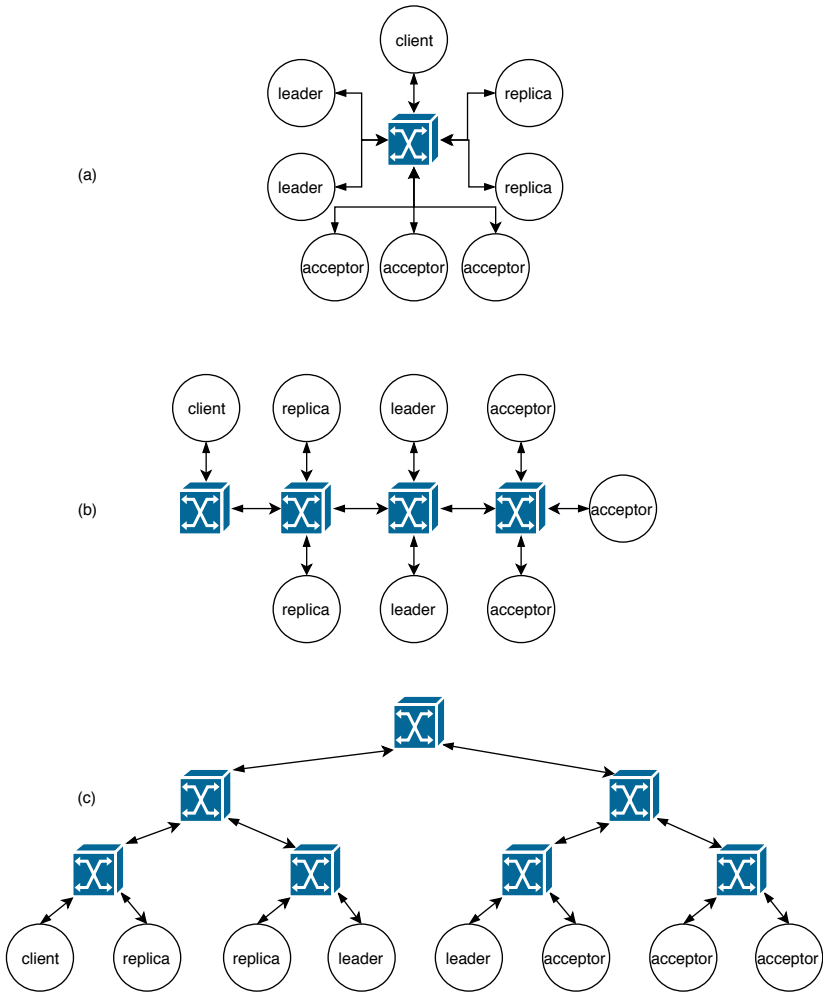

Figure 4.2: Diagrams of different network topologies investigated. Topology (a) represents a star topology, (b) represents a linear topology and (c) a tree topology.

4.1.4 Topologies

An important consideration when evaluating the system on a simulated network is to ensure that the results collected do not reflect the characteristics of the network itself rather than reflecting the characteristics of the system itself. For example we do not wish for delays in queues at switches to dominate the measurements of latency for the system itself. An initial experiment was performed to observe the steady state behaviour of the system with three different network topologies, shown in Figure 4.2. A client submit requests to the replicas every 1.0s and the latency of these requests is measured. Topology (a) represents a star topology wherein each host is connected to a single switch. Topology (b) represents a linear network wherein each process with the same role

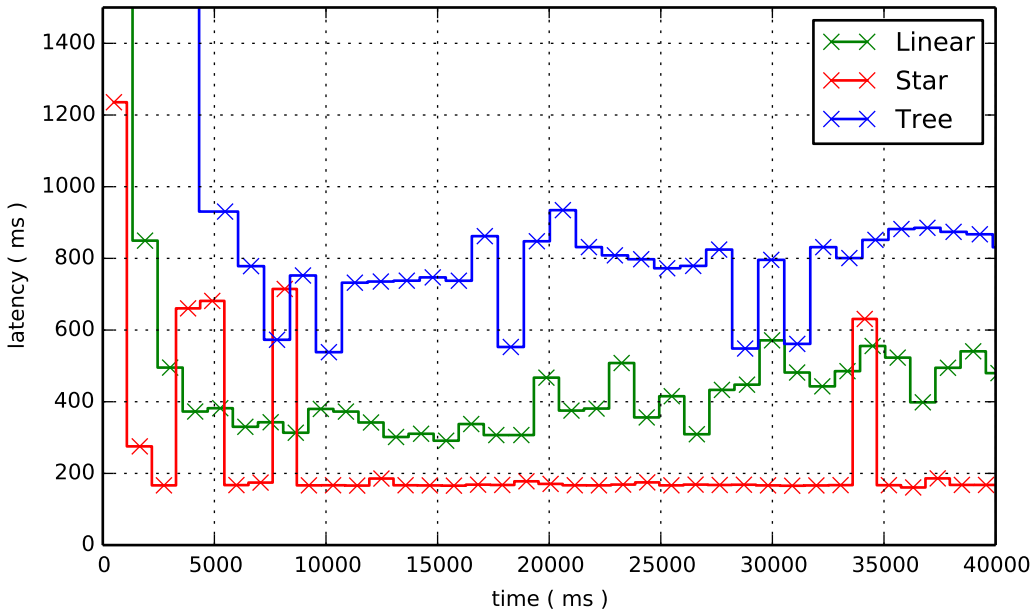

Figure 4.3: Graph of latencies over time with different network topologies.

is connected to a single switch and these are each connected together. Topology (c) represents a tree topology with switches arranged in a binary tree structure.

Figure 4.3 shows the traces produced when measuring the steady state latency of each topology. No unexpected behaviour occurs in any of the three schemes. The main difference to note is that the average latency of each trace varies between each topology. This is because the number of hops from each sender and receiver participating in the system varies. For example, every pair of hosts that need to communicate is two hops away in the star topology, 3 hops away in the linear topology and a variable number of hops away in the tree topology. There is no unexpected queuing behaviour resulting in very large delays for any of the given topologies.

4.2 Performance evaluation

The performance evaluation explores two key areas. The first is how the latency and throughput of the system change as the number of processes participating in the protocol grows. The second is how these performance metrics change in the

face of failures.

4.2.1 System size

It is important to characterise the behaviour of the system as the number of processes is varied. For these experiments the number of each role is fixed in a system with 1 client and $f = 1$; that is 2 replicas, 2 leaders, 3 acceptors. Then for each experiment the number of each process was increased and the latency and throughput observed.

Method

For each system size we wish to characterise the average latency and throughput in the steady state. For this experiment the system was arranged in a star topology, with the network parameters described in . When collecting latency measurements, clients submit requests every second for 60 seconds. The first 5 seconds of measurements are discarded since they exhibit non–steady–state behaviour. The experiment produces a collection of latencies $t_1, t_2, t_3, \ldots t_N$. The average latency for this trace is the sample mean of these, that is

$$\bar{t} = \frac{1}{N} \sum_{n=1}^{N} t_n$$

In order to construct a confidence interval, the simulation was repeated $M = 5$ times, producing a collection of average latencies $\bar{t}_1, \bar{t}_2, \bar{t}_3, \ldots \bar{t}_M$. The sample mean of these is computed to give T, the latency averaged over each run of the simulation

$$\bar{T} = \frac{1}{M} \sum_{m=1}^{M} \bar{t}_m$$

and sample variance

$$\sigma^2 = \frac{1}{M-1} \sum_{m=1}^{M} \left(\bar{t}_m - \bar{T}\right)^2$$

This allows us to construct a 95% confidence interval for each experiment, given by

$$\left[\bar{T} - \frac{1.96\ \sigma}{\sqrt{M}},\ \bar{T} + \frac{1.96\ \sigma}{\sqrt{M}}\right]$$

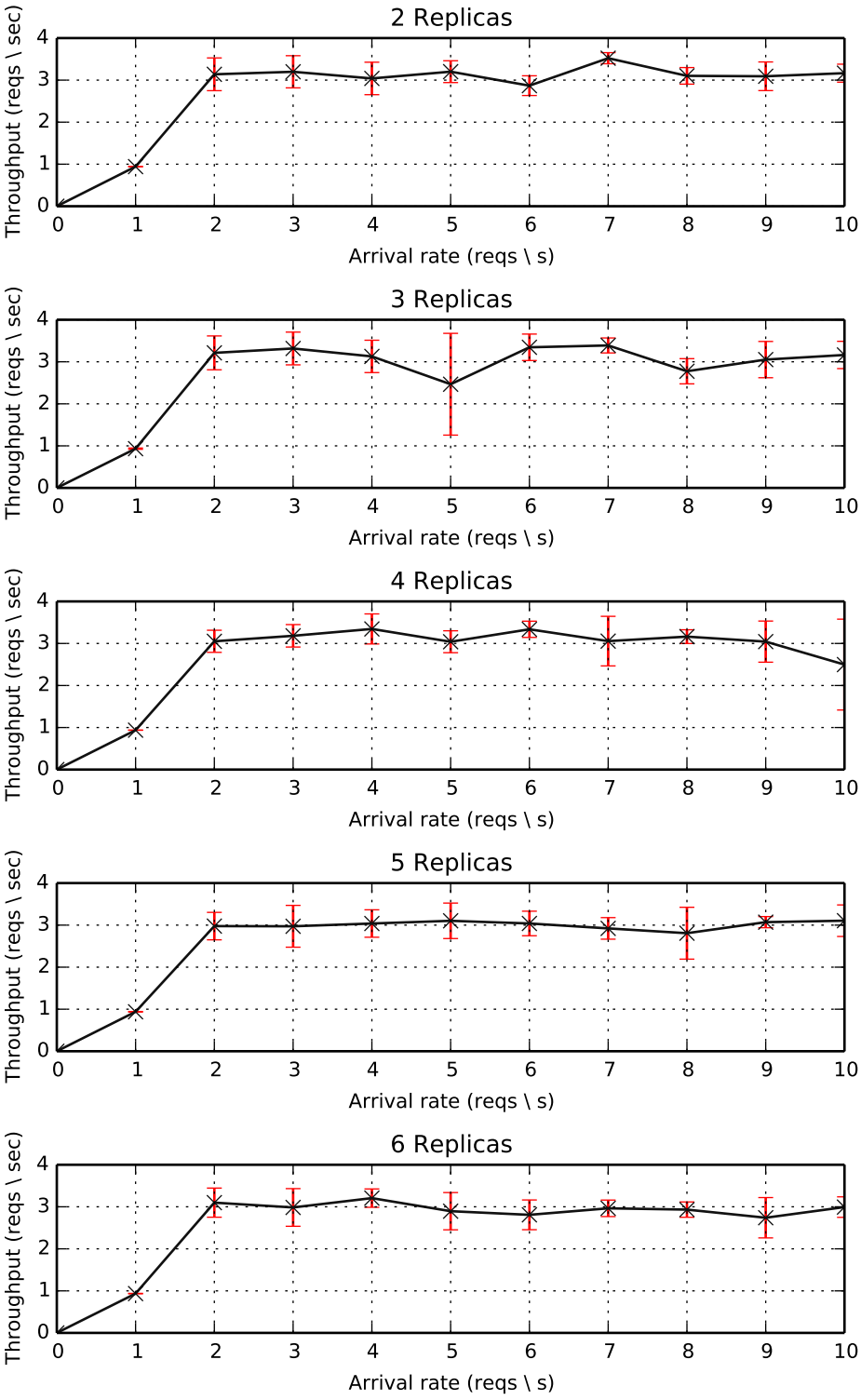

Figure 4.4: Graphs displaying how throughput varies with arrival rate of requests for a number systems with different replica sizes.

The *peak average throughput* was determined for every system size. For each system size, the throughput of the system was measured as the rate of requests from the client, called the arrival rate, was varied. For each arrival rate, the average throughput was computed by dividing the number of responses by the time interval over which they were received.

The results of such an experiment produce a set of graphs like those in Figure 4.4, which contains how these throughputs vary as the number of replicas is increased. Increasing the sending rate of the client causes the throughput to increase; this is because when the system is under–utilised the faster requests are sent the faster they will be processed. This occurs up to a certain sending rate, wherein the system becomes fully utilised and increasing the sending rate does not increase the throughput but instead causes it to oscillate near the limit or drop slightly as the system becomes overloaded.

Results

TODO: Do more runs of the throughput simulations to construct confidence intervals.

TODO: Finish analysing acceptor throughputs.

The results of the experiment are displayed in Figure 4.5. In each case there is no drastic change in latency or throughput of the system as the number of each role is varied.

In the general case most replicas will propose the same serialisation of commands to the leaders which are then de–duplicated. Hence increasing the number of replicas does not introduce additional overhead except to increase the number of messages in the network. This does not incur a performance penalty when the network is not congested. The number of leaders exhibits decreasing latency as the system is increased to 4 leaders and then increases again. This unusual behaviour may be attribute to random variations in the simulations rather than the behaviour of the system; as the confidence intervals of each experiment overlap. This is also reflected in the throughput as the system exhibits unusually high throughput for the case of 4 leaders.

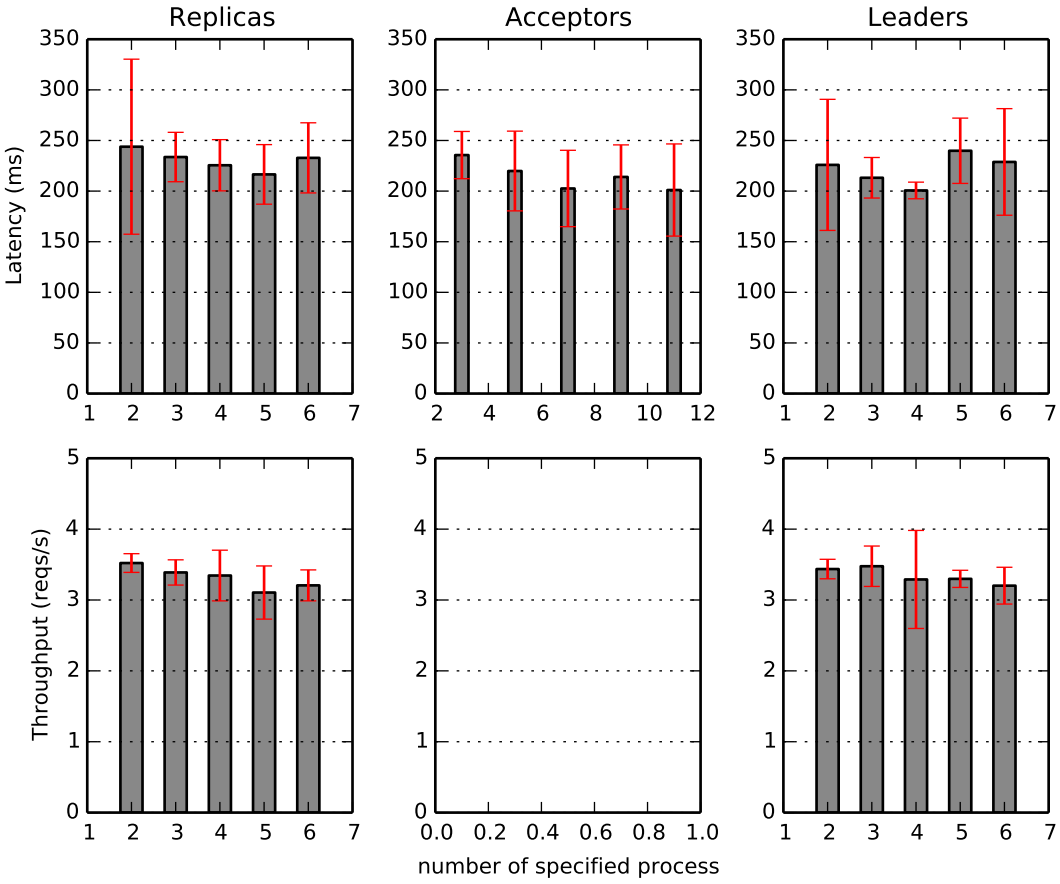

Figure 4.5: Results of latency and throughput as system size was varied for each role separately. For each graph the roles that did not vary were fixed in a system with $f = 1$. Error bars show 95% confidence intervals.

4.2.2 Failure traces

This experiment observes the behaviour of the system when failure of a process is triggered. Each simulation ran for 120 seconds with a client submitting commands at a rate of 1 per second and a system size with $f = 1$. After 60 seconds of simulated time has elapsed a process failure is triggered; this involves the killing of a specific process on a virtual host. The simulation continues to run for another 60 seconds so the behaviour after the failure can be observed. Simulations were performed separately for killing a replica, leader and acceptor.

The traces produced are shown in Figure 4.6. In each case the system begins as

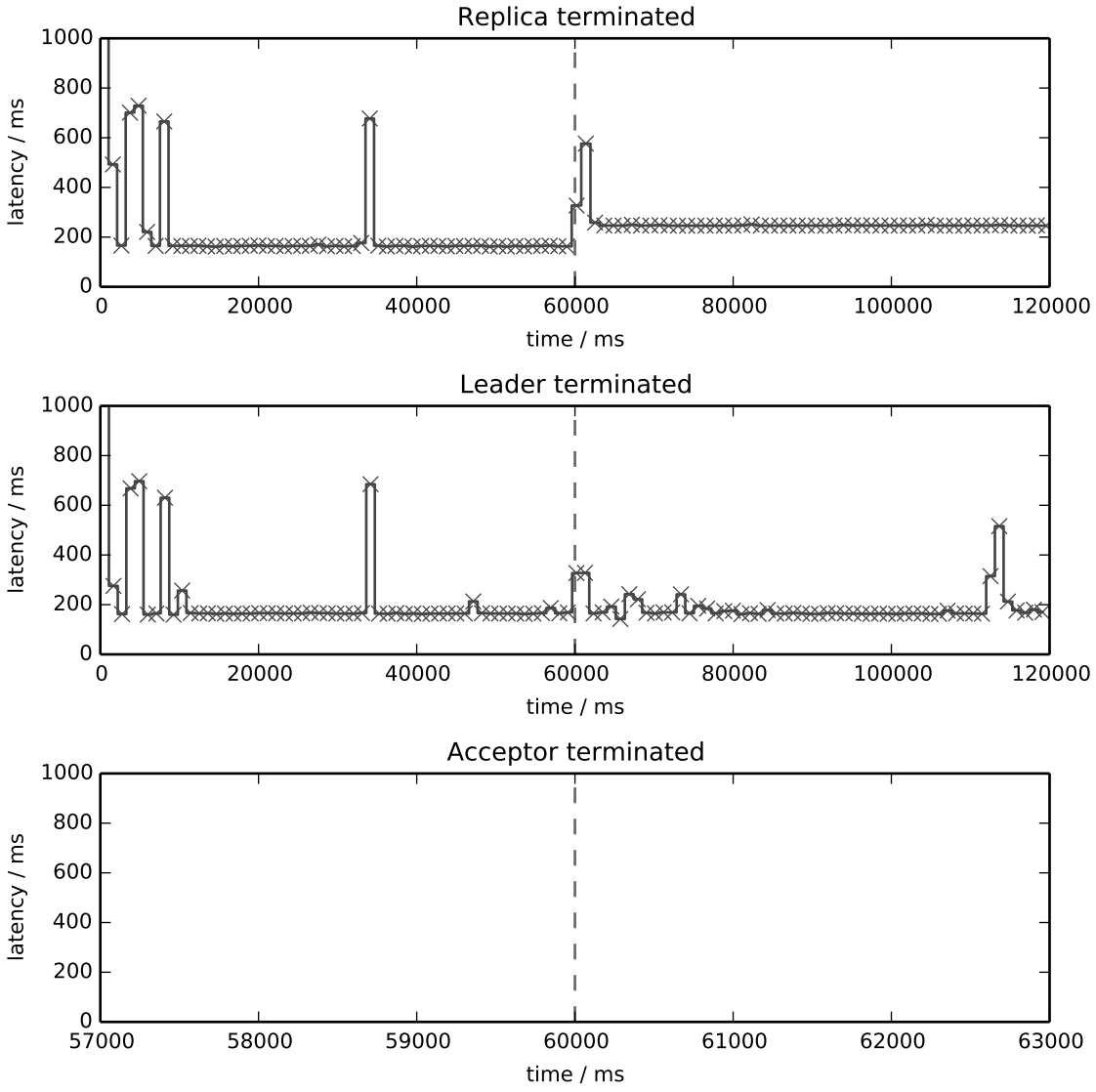

Figure 4.6: System size

usual with very high latencies before settling into steady state behaviour. In the case of a replica process failing, there is a transient increase in latency before settling down to a latency higher than before the crash. The leader process failing causes an increase in latency, smaller than that of the replica, and causes small transient delays briefly thereafter. Unlike the replica failure the latency returns to that of before the crash.

TODO: Have a look over the acceptor code because it doesn't seem to be behaving as it should. Repeat experiment after.

Chapter 5

Conclusion

The aim of this project was to produce an implementation, in OCaml, of the Multi–Paxos consensus algorithm. This chapter concludes with a consideration of the successes of the project, any limitations that have been discovered, a discussion of possible future work that could be undertaken and closes with any final remarks.

5.1 Successes

The implementation of this project was a success in that an implementation of Multi–Paxos was developed as described in the Requirements section in Chapter 2. A strongly consistent key value store was replicated across a number of replica processes with the consistenty maintained by having their commands serialised by leaders and acceptors. The system tolerated failures as described by the assumptions laid out in 2.1.1.

5.2 Limitations

A limitation encountered over the course of the project was the use of Mininet as a network simulator. Problems arose with linking to external libraries meant the program developed could only be interpreted as OCaml bytecode rather than compiled native code. This led to delays in evaluation and ... . Furthermore, the size of simulations that could be performed was constrained by the resources available to the virtual machine running Mininet. In the future it would be beneficial to run larger simulations with a less constrained network simulator.

Another limitation encountered in the evaluation was in making a performance comparison with LibPaxos. Inexperience with the C language and lack of documentation made it difficult to construct a comparable implementation of the application on top of LibPaxos. Because of this it was not possible to make the desired performance comparsion.

5.3 Future work

The large amount of literature and ongoing research into consensus algorithms presents lots of opportunities for future work on the project. One such opportunity for future work is to modify the project to that of Flexible Paxos and measure the associated performance gains against the original.

Other opportunities could arise from *open sourcing* the project and providing extensions to make it attractive to the wider OCaml community, for example providing compatibility with Mirage OS [8] or creating a GUI interface for visualising log files. A number of optimisations suggested in the literature could be implemented, including introducing *read–only* commands that do not need to be decided on by the synod protocol and reducing the amount of state retained by replicas over time.

5.4 Final remarks

This project has been both insightful and rewarding, providing me with experience designing and implementating a distributed system and exploring the challenges faced when doing so.

Bibliography

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

[2] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[3] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. *CoRR*, abs/1608.06696, 2016.

[4] Michael Isard. Autopilot: Automatic data center management. Technical report, April 2007.

[5] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[6] Leslie Lamport. Paxos made simple. pages 51–58, December 2001.

[7] Leslie Lamport. Fast paxos. *Distributed Computing*, 19:79–103, October 2006.

[8] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *SIGPLAN Not.*, 48(4):461–472, March 2013.

[9] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[10] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[11] Luís Rodrigues and Michel Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication. *IEEE Trans. on Knowl. and Data Eng.*, 15(5):1206–1217, September 2003.

[12] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[13] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, February 2015.

Appendix A

Third party software licenses

Library	**License**	**Information availabble**
Lwt	LGPL with OpenSSL linking exception	`opam show lwt`
Core	Apache-2.0	`opam show core`
YoJson	Specific license file	`https://github.com/mjambon/yojson/blob/master/LICENSE.md`
OUnit	Specific license file	`https://github.com/mjambon/yojson/blob/master/LICENSE.md`
Ocaml-capnp Capnp-rpc	Apache	`opam show ocaml-capnp` `opam show capnp-rpc`
Uri	ISC	`opam show uri`

Appendix B

Cap'n Proto schema file

```
@0x9207445e65eea38d;

interface Message {
  # Interface for RPC messaging system

  struct Command {
    clientId @0 :Data;
    # Id of the client that issued command

    commandId @1 :UInt16;
    # Id of the command being issued

    operation @2 :Text;
   # Encodes the operation that will be applied to state of application
    # The type of this is temporary for now

    clientUri @3 :Text;
  }
  # Structure represents the command sent in RPC

  clientRequest @0 (command :Command) -> ();
 # Method clientRequest is a message sent from the client to a replica
 # The client issues a command and response is returned in another message

  decision @1 (slot_number :UInt16, command :Command) -> ();
  # Replicas receive decision messages sent by a leader
  # Consists of a command and a slot number
 # Slot number is the place slot in which the command has been decided
  # to occupy by the synod protocol

  sendProposal @2 (slot_number :UInt16, command :Command) -> ();
```

```
  # Method sendProposal is a message sent from a replica to a leader.
  # Proposals consists of a command and a slot for which that command
  # is proposed.

  clientResponse @3 (commandId :UInt16, result :Text) -> ();
  # Method clientResponse is a message sent from replica to a client
  # Returns the id of the command and result of issuing it

  phase1 @4 (ballotNumber :Text) -> (result :Text);
  # Method phase1 is a message sent from leader to an acceptor
  # Used to secure adoption of ballot given in argument

  phase2 @5 (pvalue :Text) -> (result :Text);
  # Method phase2 is a message sent from leader to an acceptor
  # Used to secure acceptance of pvalue given in argument
}
```

Appendix C

Projct Proposal

Computer Science Tripos - Part II - Project Proposal

Achieving Distributed Consensus with Paxos

Christopher Jones, Trinity Hall
Originator: Christopher Jones
20th October 2017

Project Supervisor: Dr Richard Mortier

Director of Studies: Prof Simon Moore

Project Overseers: Dr Markus Kuhn & Prof Peter Sewell

Introduction

Paxos is a widely used algorithm that allows consensus to be reached in the context of failure-prone distributed systems, by having a number of processes agree upon a proposed value. A variant of this algorithm, Multi-Paxos, allows for a sequence of values to be agreed upon by electing a leader at the start.

This project wil consist of the implementation of Multi-Paxos in OCaml. To demonstrate an application of distributed consensus, a strongly consistent replicated key-value store will be implemented using this Multi-Paxos implementation. A message passing system, leveraging a RPC library, will be developed that allows nodes running the application to communicate. On top of this messaging functionality, the Multi-Paxos algorithm itself will be implemented.

The performance of this implementation will be evaluated with respect to LibPaxos[1], an existing Paxos library that will be used to compare performance to the implementation developed for this project. Testing and evaluation will take place on an emulated network providing a stable and adjustable test environment.

Work to be done

The project breaks down into the following main sections:-

[1] http://libpaxos.sourceforge.net

1. Development of a messaging system using an RPC library that allows networked processes to communicate. This will provide the underlying system for message-passing that will be used to implement Multi-Paxos.

2. The development of a test harness that simulates failures, such as processes crashing, restarting and stalling. It will be necessary to ensure these actions can be triggered at specific points in the execution of Multi-Paxos. A network emulator will be used to test possible failure modes such as dropped packets or broken links.

3. The main body of work for the project will consist of the implementation of the core aglorithm. Being distributed in nature, the algorithm consists of sending messages, using the system developed prior, between networked processes. The ability to select random quorums of processes will also need to be included.

4. Evaluation of the algorithm will take place on an emulated network, with varying topologies and a number of failure modes. Tests will be performed to ensure consensus is reached under given assumptions about the network and processes. Performance will be measured in terms of latency and throughput as the number of participating nodes is varied and compared against LibPaxos running under similar test conditions.

Starting point

A large amount of literature is available on Paxos that will be used as a specification of the algorithm that will be developed in this project.

I'm starting the project with no prior knowledge of OCaml and its environment/tools, only some knowledge of Standard ML. OCaml libraries such as Core[2], Async[3] and Cap'n Proto[4] (for RPCs) may be used.

LibPaxos, an existing open source implementation, will be used as a benchmark against which to compare performance in terms of latency and throughput.

[2]https://github.com/janestreet/core
[3]https://github.com/janestreet/async
[4]https://capnproto.org

Success criteria

A clear success criterion for this project is that the application that runs Multi-Paxos, the replicated key-value store, is in fact strongly consistent across all replicas.

Paxos operates under a set of assumptions about the network, processes on the network and their respective failure modes. These are assumptions such as packet loss, packet re-ordering and processes that can crash, stall and restart. A success criterion of this project is that given the set of assumptions the implementation of Multi-Paxos achieves consensus and makes progress.

Paxos should be able to make progress if F processes in a network of $2F + 1$ processes fail. This is a key criterion laid out in descriptions of Paxos and as such will be used as a judgement for success - numerous tests to check progress will be conducted, given the simulated failure of up to F processes at a number of points of execution.

The performance of the implementation and the existing LibPaxos library will be compared in terms of latency and throughput as the number of processes on the network is varied. All tests will be performed on an emulated network and will provide a means by which to judge the performance of this implementation against one already used in applications. If the perfomance of the implementation in terms of these metrics is within 30% (an achievable but still desirable performance when compared to a popular library) of that of LibPaxos it will be deemed successful in terms of performance.

Possible extensions

A desirable extension to Mulit-Paxos is Flexible Paxos[3]; a variant of the algorithm that weakens the requirement that quorums in each stage need intersect. This could be evaluated against LibFPaxos[5], a prototypal extension of LibPaxos.

Another possible extension is to implement an interactive application on top of the replicated key-value store, such as a concurrent editor.

[5]https://github.com/fpaxos/fpaxos-lib

Work Plan

1. **Michaelmas weeks 3-4** Gain familiarity with OCaml. Prepare build automation, package management, continuous integration and version control. Research Core, Async and Cap'n Proto. *Deadline: 01/11/2017*

2. **Michaelmas weeks 5** Gain familiarity with Mininet, write scripts to produce different network topologies and collect example data. Run a test networked OCaml application on Mininet. Thoroughly research Multi-Paxos algorithm and research possible evaluation strategies. *Deadline: 8/11/2017*

3. **Michaelmas weeks 6** Begin implementation of the project. Develop the key-value store. Integrate the RPC library to allow for processes to communicate. Pass unit tests that confirm this messaging system behaves as expected on Mininet. *Deadline: 15/11/2017*

4. **Michaelmas week 7-8** Define each of the roles nodes play in the algorithm. Begin the implementation of the core algorithm, starting with the leadership election phase. Begin implementing ability to select random quorums, generate unique proposal numbers, prepare and promise requests. *Deadline: 29/11/2017*

5. **Michaelmas vacation** Continue with implementation of aglorithm, finishing phase one. Next complete phase two - implement accept requests / responses. Pass tests to ensure expected functionality. Prepare network environments for evaluation and collect preliminary data. *Deadline: 10/01/2018*

6. **Lent weeks 0-2** Write progress report. Prepare presentation. Continue with evaluation of project by running LibPaxos on Mininet under the same conditions. Collect data on LibPaxos that will be compared to this implementation. *Deadline: 31/01/2018*

7. **Lent weeks 3-4** Finish up any remaining experiments required for evaluation. Calculate confidence intervals of data. Prepare plots for presentation in dissertation. *Deadline: 14/02/2018*

8. **Lent weeks 5-6** If time permits, begin an extension of FPaxos and start testing that under the same conditions. Otherwise, continue any evaluation still outstanding. *Deadline: 28/02/2018*

9. **Lent weeks 7-8** Finish up implementing and evaluating possible extension(s) to the project. Start writing dissertation. *Deadline: 14/03/2018*

10. **Easter vacation** Continue writing dissertation. Complete a draft before the end of the vacation. *Deadline: 18/04/2018*

11. **Easter weeks 0-2** Complete final changes to dissertation. *Deadline: 09/05/2018*

Resource declaration

I will use my own machine (2014 Macbook Air, 1.4 GHz Intel Core i5, 4GB RAM, 128GB SSD) for software development, connected to the University network in order to access online resources. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. Should my machine fail I will use the MCS facilities.

Git will be used with Github for version control and regular backups. The development directory will reside in a Google Drive for further backup.

Mininet[6], an open source network emulator, will be used for testing and evaluation. In order to run Mininet on my system, I will use VirtualBox[7].

[6] http://mininet.org
[7] https://www.virtualbox.org/