Christopher Jones

DRAFT

Achieving distributed consensus with Paxos

Part II Project

Trinity Hall

April 26, 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

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Distributed systems suffer from a number of 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 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 by which to reach 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 [4] that allows for consensus to be reached under the typical unreliable conditions present in a distributed system. The algorithm relies on processes participating in a voting protocol in order 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 [3] system for data centre management also uses Paxos, again to replicate data across machines. The extreme generality of Paxos allows it to be used as an underlying primitive for various distributed systems techniques. State Machine Replication [10] 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 [9] 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 [6], is a variant that reduces the number of message delays between proposing a value. More recently, Flexible Paxos [2] is a variant that relaxes the requirement on agreement between participants in the synod protocol.

There are also a number of similar algorithms that are not based on Paxos. Viewstamped replication [7] is primarily a protocol for implementing state machine replication that was developed independently of Paxos. Raft [8] 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 used most widely in production systems and provides a foundation upon which to use state machine replication; in this case to replicate a key value store application.

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 would be helpful in developing an application with such an emphasis on fault tolerance. OCaml also provides a rich third party eco-system 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 which is compared to that of another popular open source Multi-Paxos implementation.

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 that will be implemented. From this a set of requirements for the project is established, taking into account functionality that is glossed over in theoretical descriptions of the algorithm. Finally the software engineering aspects of the project will be discussed including the choice of tools, build processes and third party libraries.

2.1 Theotrical background

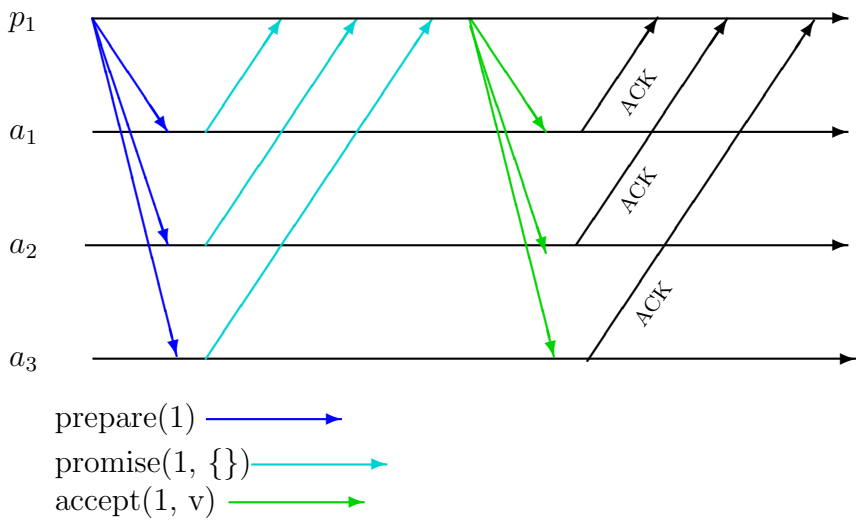
2.1.1 Assumptions of the environment

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 conesnsus system to ensure that consensus is reached under a 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 in 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 assume a system that does not tolerate 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. In consensus algorithms it is assumed that processes can somehow propose values to one participating processes. The goal of distributed consensus is, given a number of processes that can each propose some value v, that one of the proposed values is chosen. This is the *single-decree* case, that is 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. That is, each process will propose a sequence of valus $v_1, v_2, \ldots, v_n$ and the role of the consensus protocol is to have the system choose one such sequence from all those proposed. This multi-decree case allows for the state machine replication technique to be employed to replicate an application across a number of machines in a distributed system.

2.1.3 State machine replication

A desirable goal of distributed computing is to replicate an application across a number of machines so that each *replica* has the same strongly-consistent view of the application's state. This technique is referred to as State machine replication (SMR); it leads to both for increased fault tolerance and higher availability. Multi-decree consensus protocols provide a primitive by which an application (that behaves like a state machine) can be replicated.

Each process participating in the consensus protocol runs the replicated state machine application, with each process starting in the same state. Then by treating the values proposed in the consensus protocol as *commands* to perform a state transition, then by running a consensus protocol each process will receive the same serialized sequence of commands $c_1, c_2, \ldots, c_n$. These commands are treated as commands to perform a state transition and as such each process perform the same sequence of transitions from the same starting state and thus be a replica of the the state machine 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 [5] 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 Paxos algorithm. In reality these roles are often co-located within a single process but it is simply to consider each separately. The prupose of each role and the number of each role required to tolerate f failures is summarised in Table 2.1.

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.

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 for each proposal. 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 the set of acceptors, the proposer replies to each with an `accept(`n'`,`v'`)`, 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 a `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.

Once this process is completed, a majority of the acceptors will have chosen the same proposed value. Learners are required to learn what value was chosen by the majority. A number of different methods can be employed to deliver this information. Acceptors can, on choosing a value to accept, broadcast their decision to the set of learners. An alternative method is to have a distringuished learner (or small subset of) that are sent all decisions which then forward onto the set of learners when they have learned the majority.

Talk about some simple examples with corresponding timing diagrams.

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 [11] 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 SMR. *Replica* processes are introduced to implicitly act as the learners and maintain the replicated state machine. *Client* processes 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.

Diagram showing the communication pattern of nodes in Mutli-Paxos. Contrast with the single-decree case.

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.

Role	**Equivalent**	**Purpose**	**Number required**
Client	N/A	Send commands to replicas and receive responses	N/A
Replica	Learner	Receive requests from clients. Serialize 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.

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 client's sent commands and op is the operation the command should perform. 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 serialize them by converting each command c into a proposal (s, c), where $s \in \mathbb{N}$ is a slot number. The slot number describes ordering of the sequence in which the 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 broadcasts a `propose(`s`,`c`)` message to the set of leaders and awaits a `decision(`s'`,`c'`)` message. The resulting decision may differ in its slot number and so the replica may have to re-propose a command it has proposed for the decided slot. Upon receipt of decisions the replica will apply the associated operation to the application state, maintaining the replicated application.

Diagram showing message flow between clients and replicas to clarify the last points.

Ballots and pvalues

Ballots are the structure over which leaders and acceptors operate when participating in the synod protocol. Ballots provide indirection in that they allow for leaders to secure agreement with acceptors for a number of proposals with different slot numbers or for a given slot to be targeted by a number of proposals associated with 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) (where $r \in \mathbb{N}$ is called a round number and λ is a leader's unique identifier) or $\perp$, a specially designated least ballot number.

Pvalues associate a ballot number with 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. We previously required that each proposer used a disjoint subset of $\mathbb{N}$ for their proposal numbers. We can avoid this requirement as each ballot number encodes the identifier of the leader directly in its ballot number. Hence no two leaders can generate equal ballot numbers.

We require ballot numbers to be totally ordered so that acceptors can compare which ballot number is less than another 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:

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

Note 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 proposals. Instead, it is required that at least a majority of the acceptors have in their memory a decision for a committed proposal. This is so that if any minority of the acceptors crash then there is still going to be at least one acceptor with memory of the committed proposal. 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 *quorum* if $|Q| = f + 1$.

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

The Synod Protocol

The synod protocol is the protocol undertaken by the set of leaders and acceptors in order to decide which command is committed to which slot. 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, so that acceptors do not accept conflicting pvalues. Acceptors *accept* a pvalue by adding it to the set of pvalues, initially empty, that 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') 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 $(0, \lambda_i)$, 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 that below.

Phase 1a Leaders attempt to secure adoption of a ballot b by broadcasting a `phase1a(`b`)` message to the set of 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 the leader 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. If a leader receives a response where $b' \neq b$ then it should restart the protocol with a higher round number.

Phase 2a Leaders proceed to commit a given pvalue (b, s, c) by broadcasting a `phase2a(`b, s, c`)` message to the set of 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 the acceptor 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 noted previously, there are a number of papers that describe the Paxos protocol and its variants. Many of these sources present the algorithm in a different format and nearly all of them present it in a theoretical setting, with little concern for functionality that is generally required in software. Implementation details such as how each of the participating process knows how to address one another are often entirely overlooked. Hence, in developing an implementation of Multi-Paxos, it is necessary to formally recognise the theoretical requirements from the literature as well as identify the additional functionality required to realise the algorithm as a functioning piece of software.

It is therefore necessary to perform a *requirements analysis* of the final system. This was performed by collecting information from the literature on how the algorithm is presented and also considering all of the necessary functionality required to implement such an algorithm. For example, in the Paxos Made Moderately Complex paper processes are described as being able to send messages between one another. This of course presents the requirement for a messaging primitive that operates on top of a best–effort IP network. The final system requirements are listed below.

System-like requirements

- The ability to pass the executable file some command line arguments for initializing the process.
- Ability for each node to be nominated to a given role and to start-up in that role (via the command line).

- 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 over the network so processes can send messages, rather than worrying about functionality of network.
- Send messages as required by the Multi-Paxos algorithm, being able 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

- Clients processes can messages with commands to the 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 sequence of the requests to leaders.
- Leaders and acceptors take part in the Synod protocol that 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 to let development commence. The development methods, third party libraries, tools and test strategies that were employed to manage the project and improve productivity are discussed here.

2.3.1 Starting point

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

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 and with as much independence from the rest as possible. To facilitate this and from 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 to be 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 was developed first followed by that of leaders and acceptors in two phases of development.

2.3.3 Libraries

OCaml has a large eco-system 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 any third party libraries via a command line interface.

A library of particular interest to this project is that which provides RPC (remote procedure call) functionality. Cap'n Proto[2] was chosen as it has good support and documentation for its OCaml bindings; separated into Capnp-ocaml[3] for serialization and Capnp-rpc[4] for RPCs itself. The library provides a schema language to describe the required remote procedures and any required structures and a code generator for generating OCaml bindings.

[1] https://opam.ocaml.org
[2] https://capnproto.org/
[3] https://github.com/capnproto/capnp-ocaml
[4] https://github.com/mirage/capnp-rpc

Since this project is in the space of distributed systems it is necessary to consider how concurrent programming would be achieved. Lwt[5] is a monadic concurrency library that leverages the type system to express the promise of a computation, of type `'a Lwt.t`, that returns a value of type `'a`. Monadic functions `return` and `bind` to chain computations together in a safe manner and support for parallel computation, mutexes and Unix bindings. This library was chosen because it has thorough documentation and interoperates with Cap'n Proto.

Whilst OCaml already comes with a standard library, this project uses Core[6] as a popular overlay. Core improves upon the standard library by providing a number of additional useful functions and modules and also in some places improves the type definitions of those present in the default standard library (by introducing named arguments in order to improve readability). Yojson[7] was used for serialization, both in configuration files and in some areas sending data in messages. OUnit[8] was used to manage unit testing for this project and is discussed in 2.3.6.

In the case of each library used their software lisence was made readily available either by OPAM (via the `opam show <pkg-name>` command) or by licensing information made available on a library's Github repository. Each of the libraries used had licenses that made use of the software for this project possible, with YoJson and Uri requiring reproduction of their licensing information distributed with the project. Table 2.3 gives details of each library's individual license.

2.3.4 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 automates the generation of `.merlin` files. Jbuild files are written in a simple S-expression syntax and are composable with one another. Jbuild files also present the ability to include *rules* so that Cap'n Proto OCaml bindings are generated automatically when building the project.

[5]https://github.com/ocsigen/lwt
[6]https://github.com/janestreet/core
[7]https://github.com/mjambon/yojson
[8]http://ounit.forge.ocamlcore.org/

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 ocaml-capnp`

Table 2.3: Software licenses for each of the libraries used in the project.

2.3.5 Tools

Choice of tools is important for being productive in a language. MacVim[9] was used as a text editor. In order to improve productivity Merlin was used to provide code-completion and edit-time type inference features, making it much easier to understand the types of source code and save time reading documentation online. `.merlin` files are auto-generated by Jbuilder that allow for code completion and type inference over files in the project directory itself.

Utop for read-eval-print loop for quick sanity checks and rapid prototyping.

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

In order to simulate the operation and performance of instances of the implementation in a distributed system it was necessary to use a network simulator. Mininet[10] is a network simulator that operates on a proess based virtualisation and allows for arbitrary processes to be simulated as running on hosts on a virtual network. Mininet provides a complete Python API for setting up such simulations and performing experiments. Mininet required use of a guest operating

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

system on VirtualBox[11], virtualization software that allowed Mininet to be used on my own machine.

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 compliment to this; tests focussed on correctly typed arguments to functions yielding appropriate results we expect. The test suites were executed after successful builds of the project.

[11] https://www.virtualbox.org/

Chapter 3

Implementation

This chapter describes the implementation. It begins with a high level overview of the structure of the program and initialisation. This follows with 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 taking 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 ecapsulates all the functionality associated specifically with a given role. Each of those modules will need to send and receive messages of some form; this is abstracted away 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 of the processes requires additional information when they are 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.

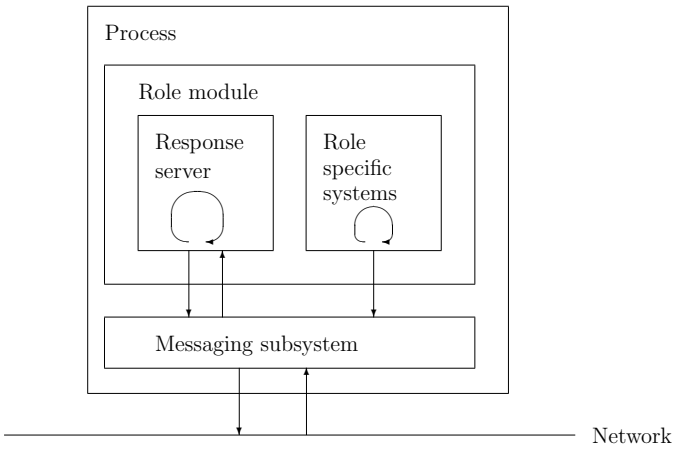


Figure 3.1: High level subsystems of a process

Configuration files are formatted as JSON[1]; JSON was chosen since it is semi-structured data that can be easily handwritten and also parsed using the Yojson library. 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. For example, clients do not need to know the addresses of leaders and acceptors. 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 consistent and maintainable to have a single file used by each of the 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 are used to map directly onto the internal representation.

3.2 Data structures

Before proceeding 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

[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
                   ...
  type result = Success
              | Failure
                ...
  val initial_state : state
  val apply : state -> operation -> state * result
   ...
end
```

Figure 3.2: Signature of the key value store module

process can generate their own unique identifer is used. Universally Unique Identifers[2] (UUIDs) are used in this case 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.

Hence replicas, leaders and acceptor identifiers have the type `Core.Uuid.t`. As we wish to expose clients to a request / response protocol it is necessary to store a map of client identifiers to addresses by each replica. Rather, by modifying client identifiers to be of the form `Core.Uuid.t * Uri.t`, the address actually forms part of the identifier.

3.2.2 Key value store

The key value store is the application that is to be replicated. Each replica process will maintain its own independent copy 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 own semantics described in Table 3.1. The application follows a state machine pattern given that if each replica starts in the initial state and each applies the same sequence of operations in the same order, the resulting state is the same.

[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`.

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.

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.

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 (representing pairs (n, λ)). The type definition is included in Figure blah blah.

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`.

```
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`.

Therefore outside of this module the internal representation of ballot numbers is hidden. This prevents a large number of errors, such as decrementing a round number when they should strictly increase, from arising by rejecting them at compile-time. It is necessary to supply function to work with ballot numbers outside the module.

Hence 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 equality and the ordering of ballots. These work over the structure of ballots, comparing them first for correct types and then checking round number and leader id equality. Also in the Ballot module are functions for serialisation and deserialisation to and from JSON, required by the messaging system for sending ballots.

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 simple OCaml types into a form suitable for transport over an IP network, handling packetisation, rentramssion and masking failures. It also handles the initialisation of servers and the addressing in the system.

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

```
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:(Types.command -> unit) ->
    ?proposal_callback:(Types.proposal -> unit) ->
    ?response_callback:(Types.command_id * Types.result -> unit) ->
    ?phase1_callback:(Ballot.t -> Types.unique_id * Ballot.t * Pval.
       t list) ->
    ?phase2_callback:(Pval.t -> Types.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

rather than their role in the algorithm itself, which is discussed in depth over the rest of this chapter.

Messages are broadly divided into two cateogries: *blocking* and *non-blocking*. Blocking messages are those which expect a direct response to a given message, whereas when sending a non-blocking message the process does not expect an immediate 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 such a message and do not include any information about source and destination addressing, as this is handled transparently by the messaging system.

The function `send_non_blocking` takes a non-blocking message represented by the type described above and a URI that addresses the destination process. As these messages do not explicity require a reply they return a `unit Lwt.t`, that is a promise to perform a computation that terminates with value `()`. As we can

regard a message that is never delivered to a recipient as indefinitely delayed in the network, we return this type regardless of any errors encountered by the underlying subsystem. It is important to note here that any errors that arise from messages failing to deliver are handled by the Multi-Paxos algorithm itself, not the messaging system.

Blocking messages on the other hand return a type immediately. Hence they are separated into their own specific functions, `send_phase1_message` and `send_phase2_message`, each with a return type that represents the arguments in the response. Note that here the arguments represent the phase1a / phase2a messages and the responses represents the phase2a and phase2b messages. We do this because acceptors only respond in response to a message from a leader, so it reduces the functionality required to represent these replies as return types. The error handling here returns a type wrapped in a `Result.result` type which allows for an error to be returned and discarded by the leader when checking whether it has received responses from a quorum.

The message module also exposes a function to start a new server, the type of which is given in 3.6. The server requires a string and integer representing the IP address and port number on which to listen and a number of named 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 Protos requires that one writes a schema file that describes a service; this contains data structures and messages that need to be serialized. This schema file, listed in full in Appendix A, contains an interface that describes the service that we wish to make available to the messaging system. The schema crucially contains a method for every message that can be sent and its associated arguments and return type, with each mapping to the associated message type described in Figure 3.5. Note that some of the arguments in the Cap'n Proto schema file are stored as `Text` types rather than as the types in 3.5 as these are first serialized into JSON using Yojson and then converted to strings. This was to provide a degree of separation in the source code. For example, this helps

```
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

avoid situations where a new command for the replicated application would require rewriting of boilerplate code for messaging and recompilation of the schema.

A compiler tool is used to generate OCaml code that contains signatures and structures for the interface described in the schema. Describe serialisation and deserialisation.

So far we have only use the serialization functionality provided by Cap'n Proto, providing a means of marshalling our messages into a suitable data exchange format. This serialization will be used by the RPC protocol implemented by Cap'n Proto. This is achieved by *functorising* the generated API with a suitable RPC library (in this case `canp_rpc_lwt`). Explain what then needs to be implemented and how to implement it.

Talk about Vats and Capabilities. Talk about how Cap'n Proto works under the hood.

Within Cap'n Proto sturdy refs are identified by Cap'n Proto URIs[3]. 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]https://tools.ietf.org/html/rfc3986

```
Capability.call_for_unit t method_id request >|= function
      | Ok () -> ()
      | Error e -> Hashtbl.clear sturdy_refs
```

Figure 3.8: Dealing with errors as a result of Cap'n Proto message delivery failure

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 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.

3.3.3 Masking errors

Next go on to talk about how we use Error types in the rpc responses to catch killed nodes and simply. Talk about how we need to refresh the cache in this case. Talk a bit more about the types used in phase1/phase2 messages as these may return Errors to the above subsystem.

3.4 Clients and replicas

3.4.1 Clients

Clients are the simplest processes in the system as they need 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 serialized consistently across the system by Multi-Paxos.

```
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 3.9: Types of records representing replica state

3.4.2 Replicas

Having discussed the functionality of clients it is necessary now to talk about the implementation of replicas. The state of a replica is stored as a record with mutable fields as presented in Figure 3.9. More introduction.

Replicas are required to implicitly take the role of learners in order to learn the decisions made on how to serialize the sequence of client requests. Rather than have the replicas explicity request the result of the synod protocol from the leaders or acceptors, acceptors forward the requests made by clients onto them and receive a response, implicitly learning the result.

Replicas maintain three queues:

- `requests:` A queue of commands received in request messages from clients.
- `proposals:` A queue of proposals. These are commands that have been tagged with a provisional slot number. This queue forms the replica's own attempt at serializing the commands. However, these are not committed in this order until the configuration of leaders and acceptors has decided each command for that slot.
- `decisions:` A queue of decisions. This represents the serialization of commands that has been decided on by the configuration of leaders and acceptors; it represents the consensus reached on the sequence of committed commands. It is in this order that the commands are applied to the replicated application state.

These queues are stored by the replica as types `command list` and `proposal list`. This is because list types in OCaml have a large library of helper functions

in the Core library. Further, the type `proposal list` is structurally equivalent to `(slot_number, command) List.Assoc.t`, that is an association list of commands keyed by their slot number. This allows 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 submitted; 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 3.10. `request(`c`)` messages sent by clients arrive at the replica and are entered into the requests queue. Concurrently the replica will attempt to dequeue commands in 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 configuration participating in 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 by a replica. 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 serializations and the other has won out). 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 (`slot_out`, $(\kappa, \text{cid}, \text{op})$) has its command executed, 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 the application state is updated in the same order by each replica. The application state is updated by applying op to the replica's application state, resulting in an updated state and

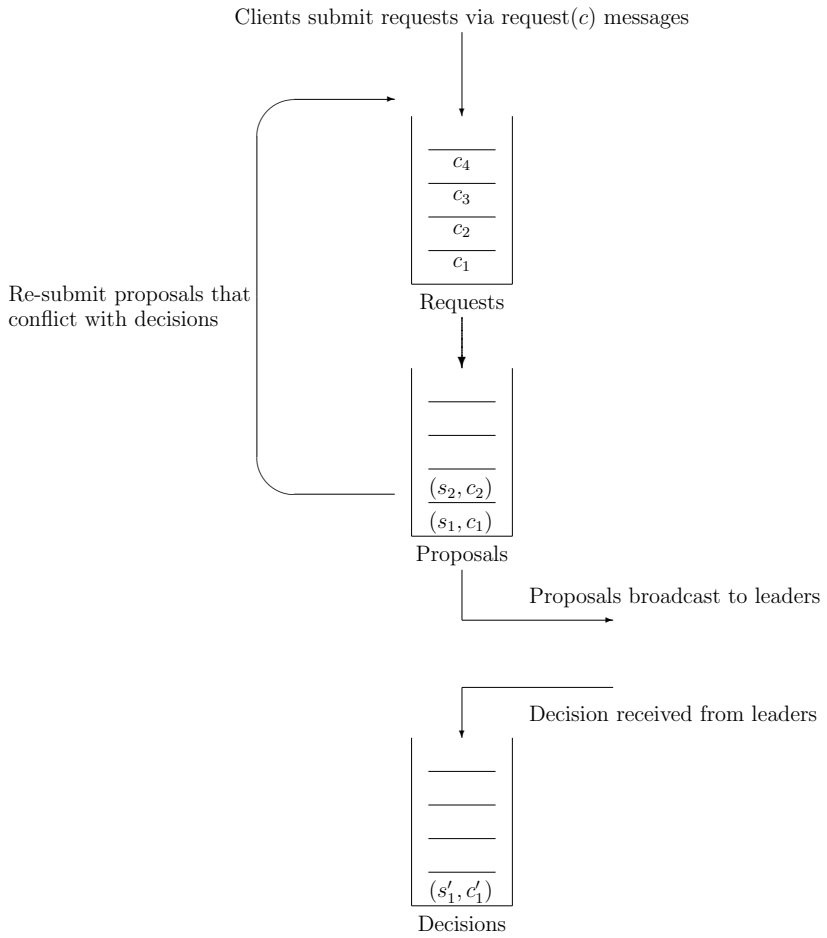


Figure 3.10: Flow of requests and proposals through a replica's queues

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 therefore expected to be able to reconfigure, that is to change the set of leaders with which they share their proposals. Reconfigurations are commands sent like any other, except that when executed by the replica (having been decided a slot in the same fashion any other command would), rather than being applied to the application state they update the set of leaders to the new configuration.

A window of commands is maintained so that only a maximum number of

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

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

Figure 3.11: Types of records storing acceptor state

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. We therefore maintain a `WINDOW` value that represents the number of slots to which a value can be proposed before a given reconfiguration command takes effect. Equivalently this involves maintaining the invariant `slot_in` $<$ `slot_out` $+$ `WINDOW`. Hence the queue of proposals has a maximum capacity of `WINDOW` and implies that a maximum `WINDOW` proposals are in flight (that is, being serialized by the synod protocol) at once. This therefore means that given a reconfiguration doesn't take effect for `WINDOW` slots that after any reconfiguration is triggered all pending proposals will have been decided by the time the configuration has taken place.

Include an example execution diagram of this sliding window.

3.5 Synod protocol

Replicas each propose their own serialization of client commands; varying processing speeds and delays in the network along with crash failures of replicas can cause these serializations to differ. In this case, it is the function of the Synod protocol to produce a single serialization of those proposals that is returned to all the replicas. This serialization must be decided upon in a distributed manner by the configuration, that is the set of leaders and acceptors. Before we 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 what is known as the disitributed fault tolerant memory of the consensus protocol. As with the other processes, the internal state is stored in a record with type given in Figure 3.11. Acceptors maintain the following mutable state

```
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.

- `ballot_num`: The ballot number that has been most recently *adopted* by the acceptor. Initially, `ballot_num` is equal to $\perp$ so that an acceptor adopts the first ballot it receives (following from $\perp$ being the least ballot).
- `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, all an acceptor is required to do is maintain its state and provide two callbacks, listed in Figure 3.12 and 3.13, when starting a server; one 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.

Both the phase 1 and phase 2 callback functions can be executed concurrently as each callback is executed when the underlying server receives a message. 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. Hence 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 the criticial section.

Leaders broadcast `phase1_callback(`b`)` to the set of acceptors when they seek an adoption of ballot number b form a majority quorum of acceptors. 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 previous ballots it has ever adopted). The acceptor will then return a triple consisting of its identifier, the ballot it has adopted (this will either be the ballot it has just received or the ballot it previously 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 abandon this adoption attempt 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 even 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 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, we 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.

```
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

3.5.2 Leaders

Leaders receive over time from each replica their serialization of the set of proposals which may differ. The leaders are required to, between them and in a manner that tolerates the failure of all but one leader, return a serialization of the proposals such that each replica receives the same serialization.

Leaders spawn and manage *scouts* and *commanders* in order to separate the concerns of attempting to secure adoption of ballots and attempting to secure acceptance of pvalues. Each scout and commander is a sub-process with an associated state and execution context that is spawned by a given leader. Rather than using the existing messaing subsystem which would produce excessive overhead for managing exchange of data between these threads we instead have them communicate locally in a producer consumer relationship.

In essence each sub-process has a lifetime in which it is spawned, messages acceptors, receives responses, performs some processing and then terminates. The terminating sub-process may have to pass back to the leader the result of its computation. In the case of scouts, this is either a notification of a preemption having occurred or its ballot having been adopted (along with the pvalues of all the previously accepted ballots). 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. Terminating sub-processes produce these responses and enqueue them onto `message_queue`, a queue of responses, via the `send : process_response -> unit Lwt.t` function. The leader periodically will consume these responses from the queue, in turn possibly spawning more sub-processes that repeat the cycle. The queue is protected by a guard mutex `queue_guard` that protects from races from occuring when the leader and one or more sub-processes attempt to mutate the state of the queue at once.

```
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: Types of responses produced by scouts and commanders

```
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
  }

  val spawn : t -> Ballot.t -> unit
end
```

Figure 3.16: Signature of scouts

Scouts

As described above, the purpose of a scout sub-process is to take a given ballot number and attempt to secure adoption of that ballot number with a majority quorum of acceptors. The ballot 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 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 start a initialize a new scout and start its asynchronous execution. 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 given scout's lifetime.

Upon being spawned, a scout broadcasts (in parallel) to the set of acceptors a `phase1a(`b`)` message. Each of these is bound monadically with a function

that returns unit if an error occured in message delivery (in keeping with our 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.

If $b = b'$ then the scout adds `pvalues'` to the `pvalues` it already has 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 this pvalue.

It begins by broadcasting a `phase2a(`b`,`s`,`c`)` message to the set of 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 this 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. Hence 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 the 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.

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).

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 $(0, \lambda)$. The leader becomes active upon phase 1 of the protocol concluding and consuming a `Adopted(`b`,pvals)` response from the message queue. The leader in this case spawns a commander for every proposal ... that haven't been committed to a given slot. Leaders revert back to passive mode when they are preempted and have to repeat phase 1 of the ballot. Asynchronously leaders receive proposoals from replicas and will add them to their list of proposals, assuming it hasn't already been proposed to a given slot. 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.

Chapter 4

Evaluation

Chapter 5

Conclusion

...

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

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

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

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

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

[7] 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.

[8] 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.

[9] 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.

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

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

Appendix A

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

  # -------- CHANGE TYPES OF THOSE BELOW
     ---------------------------------

  phase1 @4 (ballotNumber :Text) -> (result :Text);
  # Method phase1 is a message sent from leader to an acceptor
  # As an acceptor responds to each phase1 message with a reply
  # based deterministically on the request we implement in a simple
  # request / response format as above.
  #

  phase2 @5 (pvalue :Text) -> (result :Text);
  # ...
  # ...
  # ...

  # Wrt arguments of the last two messages we have a more
     experimental approach:
  #   - Instead of providing each argument as a Capnp type, instead
  #     each argument will be provided as JSON text. This is because
  #     the format of these messages will be optimised later (state
  #     reduction can be performed on the pvalues we are required to
  #     send) so no point in writing lots of serialization code when
  #     it will change in the future anyway
}
```

Appendix B

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[2]; 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].

TODO: Put the proposal's bibliography at the end of the document

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