Christopher Jones

DRAFT

Achieving distributed consensus with Paxos

Part II Project

Trinity Hall

April 19, 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. Transaction systems, distributed databases and leadership elections are all applications that require consensus in order to remain consistent. Consensus algorithms provide a means by which to reach agreement across in a distributed system 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. It is a three-phase commit system that relies on processes participating in the *Synod* voting protocol in order to tolerate 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 and it being chosen. More recently, Flexible Paxos [2] is a variant that relaxes the requirement on same-phase quorums intersecting in order to improve performance.

There are a number of alternative means of reaching consensus. Viewstamped replication [7] is primarily a replication protocol but can be used as a consensus algorithm. Raft [8] is a modern alternative to Paxos that attempts to reduce the complexity of implementing Paxos. Why didn't we use these though?

1.2 Aims

The aim of this project was to produce an implementation of the Multi-Paxos variant of the Paxos algorithm to replicate a toy distributed application. This is the variant that is used most widely in production systems and provides a foundation upon which to use state machine replication to replicate an application.

OCaml will be used as the primary development language. Why OCaml?

Evaluation and simulator explanation.

Chapter 2

Preparation

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

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.

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.

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. A number of optimisations and extensions can be put in place to increase the efficiency of the system. The system here primarily follows PAXOS MADE MODERATELY COMPLEX and introduces different types of nodes. Also discussed here is how to extend the system to use state machine replication. The new roles and their

Role	Purpose	Number required
Client	Send commands to replicas and receive responses	N/A
Replica	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	Request acceptors adopt ballots.	$f + 1$
Acceptor	Fault tolerant distributed memory. Voting protocol.	$2f + 1$

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. Note that clients do not explicitly participate in the protocol and so there is no requirement on any number being live at any given time.

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 and replicas are introduced to provide a means of implementing the replicated state machine.

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 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 applying the associated operation to the application state, maintaining the replicated application.

(Also reconfigurations)

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

Ballots and pvalues

Explain ballots.

- A ballot may map a command to multiple slots.
- A slot may be mapped to multiple ballots.
- Leaders can attempt to secure adoption of multiple ballots concurrently.

Ballot numbers 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.

A *pvalue* is a triple (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. In the single-decree case, we 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

(i.e. 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 $\leq \in \mathcal{B} \times \mathcal{B}$ which satisfies the following two conditions:

$$\forall\,(n,\lambda)\,,(n',\lambda') \in \mathcal{B}.\ (n,\lambda) \leq (n',\lambda') \iff (n \leq n') \vee (n = n' \wedge \lambda \leq \lambda') \quad (2.1)$$

$$\forall\ b \in \mathcal{B}\ .\ \bot \leq b \quad (2.2)$$

Note this implies that we require leader identifiers be equipped with a total order relation as well.

Quorums

Explanation of quorum systems.

Let $\mathcal{Q}$ be the set of all quorums of acceptors, that is $\mathcal{Q} = \mathcal{P}\,(\mathcal{A})$. Quorums are *valid* if they share at least one common member, that is

$$\forall Q_1, Q_2 \in \mathcal{Q}\ .\ Q_1 \cap Q_2 \neq \emptyset$$

Hence we can use a majority quorum by requiring that $|Q_1| = |\mathcal{A}|\ /\ 2$.
Need to really think about these quorum systems.

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.

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_i is the identifier of the ith leader. They do this by broadcasting a `phase1a(`b`)` message in the same format as that below.

Phase 1a Leaders attempt to secure an initial ballot by broadcasting `phase1a(`(n, λ)`)` message.

Phase 1b Acceptors receiving a `phase1a(`(n, λ)`)` compare (n, λ) to the highest ballot number they have adopted thus far. This is initially $\perp$ so acceptors will adopt the first ballot number they receive automatically. If $b \leq (n, \lambda)$, then the acceptor will adopt this new ballot number. In either case, the acceptor will reply with a `phase1b(`b'`, pvals)` message.

Phase 2a ...

Phase 2b ...

Go on to summarise and clear up any other points. Then reference a diagram showing message flow in the case of the synod protocol.

Touch on duelling proposals and the impossibility result. Include a duelling proposals timing diagram.

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 [11], processes are described as being able to send messages between one another. This of course presents the requirement that we implement some sort of messaging primitive that operates on top of the unreliable network running typical IP technology. The final system requirements

are listed below.

System-like requirements

- Command line interface
- Select type of process and proceed
- Configuration files and a means of nodes taking a specific config
- Ability to log salient information to disk, adjust granularity of logs, etc.
- Persist data to disk for crash recovery.

Messaging subsystem

- Provide an abstraction over the network so processes can send messages, rather than worrying about functionality of network.
- Send messages as per the requirements of the consensus algorithm
- Provide semantics around failures we would expect in this case.

Consensus algorithm requirements

- Strongly replicated key-value store as application
- Clients send messages with commands for this application and receive responses.
- Replicas.
- Leaders.
- Acceptors.
- Quorums.
- Reconfigurations.

2.2.2 Testing and evaluation requirements

It is crucial that there is a system to check that the algorithm behaves as expected in the environement described in the assumptions.

Talk about using Mininet and the built in scripting ability to provide such a test harness.

Test harness

- ...
- ...
- ...

2.2.3 Extension elements

Talk about FPaxos here (when I've actually built it.)

2.3 Software engineering

2.3.1 Methodology

- Discuss the software development methodology (something along the lines of Agile I imagine)
- Describe the testing strategy. Talk about OUnit for unit testing functions as simple units and then more thorough and complete tests for correct funtionality at the cluster-level - (these can either be Mininet scripts or something else entirely...)

2.3.2 Libraries

- Mention OPAM package manager and how it is used for installing / managing libraries and dependencies
- Discuss the licenses of these packages (professional practice)
- Give particular explanation to Capn'Proto and Lwt as these are the most interesting and pervasive libraries used in the project. Save technical details for implementation.

- Remark briefly about using Core as standard library replacement, Yojson for some serialization, OUnit for unit testing, some lib for command line parsing.

2.3.3 Compiler

JBuilder (Renamed Dune)[1] was used as the compiler for the project. This was chosen as configuration files are presented in a simple S-Expression syntax. It also allowed for Cap'n Proto schema files to be re-compiled into OCaml boilerplate upon each re-build of the project. It also automates the generation of `.merlin` files for auto-completion in Vim (see the section on Tools).

2.3.4 Tools

Choice of tools is important in being productive in a language. Vim was chosen. Merlin was syntax highlighting.
Git was used both for version control and for backups with Github. Additionally, backups were performed by having the project directory located in a Google Drive.

- Vim with Merlin for type inference whilst editing.
- Git for version control, Github (and Google Drive) for backups.
- Mininet (with VirtualBox) for evaluation.

[1] https://github.com/ocaml/dune

Chapter 3

Implementation

3.1 High level structure of program

In this section the high level structure of the program is discussed. See figure blah blah for an overview of the structure of a typical process.

Each of the processes takes the *role* of either a client, replica, leader or acceptor. These roles each have a different function in the system within their *role module*, which contains all the functionality associated specifically with a given role.

Each of the roles shares a number of common middlewares in the program, the main being the messaging subsystem. The purpose of this subsystem is to provide indirection between the way that messages are treated by the role modules and the method by which they are actually sent over the network. The desire is to have messages behave as an atomic primitive with semantics as described in Assumptions section, obscuring all of the protocol-level functionality in sending the messages over an IP network.

The role module is divided up into two separate `Lwt` threads. The response server handles the receipt of messages from other processes and performs any required processing accordingly (this is still role specific), which can result in sending further messages. Any concurrent processing required of the role is performed concurrently in the role specific systems.

Each of the processes requires additional information at start-up, 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 of all roles.

Items (1) and (2) are consumed via comand line arguments when the program is run. (3) requires the use of a configuration file, the relative path to which is

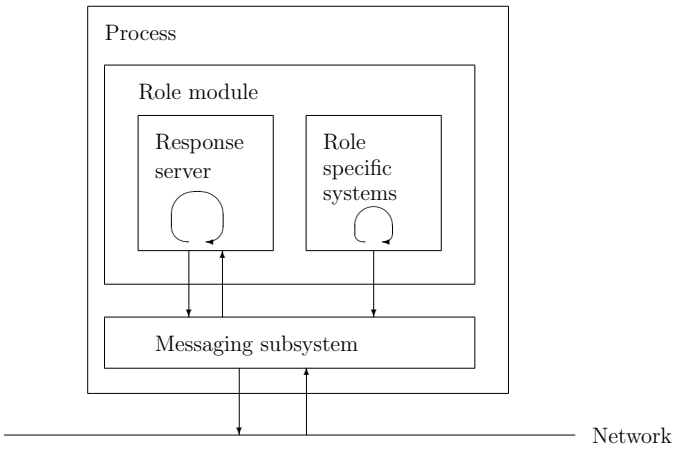

Figure 3.1: High level subsystems of a process

provided as a command line argument. The configuration file is formatted in JSON and provides, for each role, a list of all of the addresses of each process. An example configuration file is included in Reference the appendix. Figure blah blah shows a flow diagram describing how initialisation proceeds

(Addresses are provided as an IP address (V4 or V6) and a port number)

Flow diagram of init.

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 at start-up a method where any process can generate their own unique identifer is used. Universally Unique Identifers[1] (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.

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

```
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

Command	Arguments	Semantics
Nop		No operation, No change to state
Create	`(K,V)`	Add new `(K,V)` pair to the state If key already present, then 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.

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. Hence each replica process will maintain its own independent copy of the application state. The application itself needs not maintain any synchronisation logic, it need only behave as a state machine. The state is represented by an association list (commonly called a dictionary) 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

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

Figure 3.3: Types of ballot numbers.

the same sequence of operations in the same order, 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, which represents the updated state held by the replica and the resulting information returned to the replica (the client does not receive a whole copy of the new state in its reply). The signature of this key value store is shown in figure blah blah.

3.2.3 Ballots

The definition of ballot numbers from the Preparation chapter lends itself to the types of ballot numbers being represented 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 blah blah shows the interface exposed by the Ballot module. Note that the concrete type of ballot numbers isn't exposed, only the abstract type `t`. Therefore outside of this module the 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. However, this requires a number of functions be supplied so that ballot numbers can be manipulated outside this 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

```
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`.

```
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

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 the packetisation, rentramssions and other semantics. 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 and have been touched on in the Preparation chapter. 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 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`, explain lwt unit. As we can regard a message that is never delivered to a recipient (perhaps due to a broken link in the network or a crashed process) 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 possibly returned. ...

3.3.2 Cap'n Proto

Cap'n Proto is used as the underlying system for sending messages. Cap'n Proto support is provided in OCaml in two separate libraries; `capnp-ocaml` is used to compile OCaml bindings for Cap'n Proto schema files and `capnp-rpc` provides OCaml bindings to Cap'n Proto RPC mechanisms.

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 for transport. 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. It contains a definitions for a structure that represents commands as their representation is fixed for the Multi Paxos protocol. Explain why we serialize some stuff with JSON. The schema crucially contains a method for every message that can be sent and its associated arguments and return type, with each mapping naturally to the associated message type described in Figure 3.5.

A compiler tool is then used to generate OCaml code that contains signatures and structures for the interface described in the schema. This is automated in the build process by including a *rule* in the JBuild file that runs the Cap'n Proto code generator on the schema when compiling the project. Describe how serialisation and deserialisation are performed.

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.

Diversion into Cap'n Proto URIs. Talk about how we cache them so to the above system there is no caching involved. Talk about addressing and the conversion we need. Then talk about a hash table that is used to cache the mapping from addresses to services.

Within Cap'n Proto sturdy refs are identified by Cap'n Proto URIs[2]. Each process maintains a server that is addressed by .... The function that computes this is listed in Figure 3.6

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

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

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

```
let send_request message uri =
  (* Get the service for the given URI *)
  service_from_uri uri >>= function
  | None -> Lwt.return_unit
  | Some service ->
    match message with
    | ClientRequestMessage cmd ->
      client_request_rpc service cmd;
    | DecisionMessage p ->
      decision_rpc service p;
    | ProposalMessage p ->
      proposal_rpc service p;
    | ClientResponseMessage (cid, result) ->
      client_response_rpc service cid result
```

Figure 3.7: Types of non-blocking messages

```
let send_phase1_message (b : Ballot.t) uri =
  service_from_uri uri >>= function
  | None ->
    Lwt.return_error "Error sending phase 1 message"
  | Some service ->
    phase1_rpc service b >>= fun response ->
    Lwt.return_ok response
```

Figure 3.8: 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 3.9: Types of non-blocking messages

case. Talk a bit more about the types used in phase1/phase2 messages as these may return Errors to the above subsystem.

Put a snippet here...?

3.3.3 Server

Talk about the interface exposed on behalf of the server.

3.4 Clients and replicas - better title?

3.4.1 Clients

- Talk about how clients operate outside the system
- How they send messages to replicas.
- Include .mli interface for replicas, the record (with mutable fields) for storing replicas.

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 3.10. 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

```
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 3.10: Types of records representing replica state

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` can be considered isomorphic to `(slot_number * command) List.Assoc.t`, that is an association list (commonly known as a dictionary) of commands keyed by their slot number. This allows for proposals to be looked up by their key and 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 3.11. `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 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 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 a result res. A `response(cid,res)` message is sent to the client with identifer κ[3].

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

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

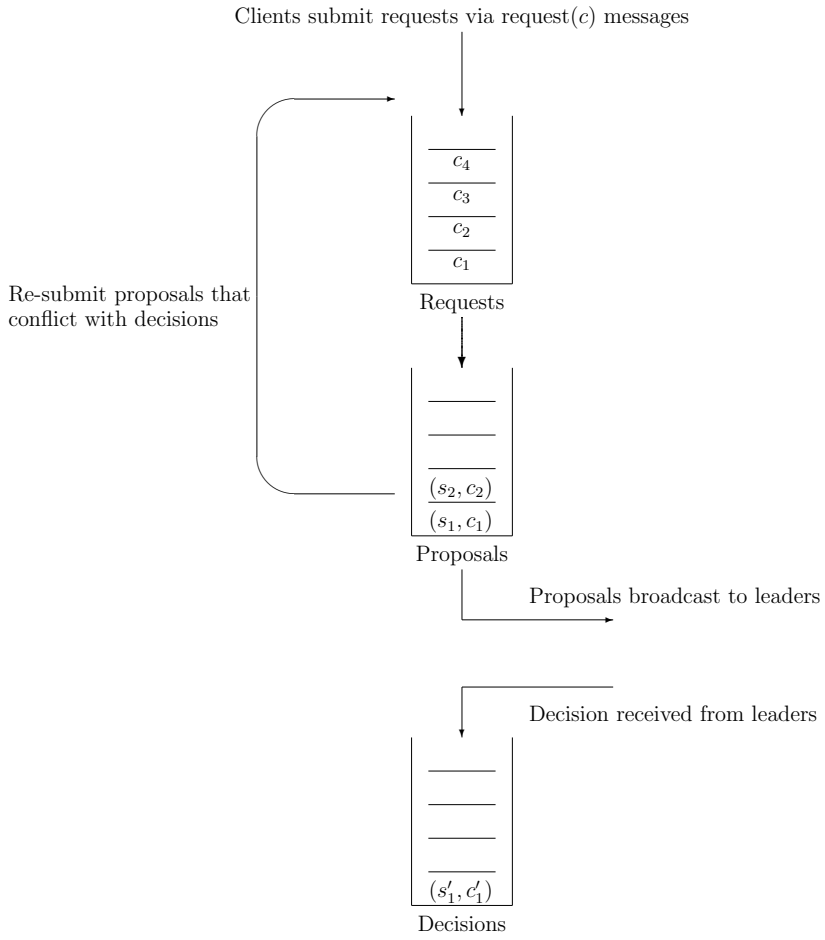

Figure 3.11: Flow of requests and proposals through a replica's queues

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. Explain therefore the need for a window.

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 replicas.

3.5.1 Quorums

...We need to think about what I'm going to explain and show here...

- Describe how the quorum system was implemented. Show the .mli interface as a snippet and describe how this can be used to achieve majority quorums.
- Show briefly how the majority checking function was implemented.

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

Figure 3.12: Types of records storing replica state

3.5.2 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 3.12. Acceptors maintain the following mutable state

- `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.

Explain what acceptance and adoption is, or reference that we mentioned it in section 2.

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 3.13 and 3.14, when starting a server; one to receive phase 1a messages and one to receive phase2a messages. Recall from the messaging section 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

```
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 3.13: 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 3.14: Function called when an acceptor receives a phase2a message with associated pvalue.

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

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

Figure 3.15: Types of responses produced by scouts and commanders

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.

3.5.3 Leaders

Introduction to leaders... Leaders receive over time from each replica their serialization of the set of proposals, which of course may differ. The leaders are required to, between them, 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

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

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 associated pvalues). Commanders may notify the leader on termination of a preemption having occured. The types of these responses are listed in Figure 3.15.

We treat the sub-processes and the leader that spawned them as engaging in a producer consumer relationship. 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[4] `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. The code for all of this is described in Figure 3.16.

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 3.17. 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

[4]Note here that we use a Lwt mutex rather than a Core mutex in order to match the types say more

```
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 3.17: Signature of scouts

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 ...
- `active`: a boolean value that represents the mode in which the leader operates at a given time.
- `proposals`: the list of proposals the leader has received from replicas.

The flag `active` describes whether the leader is currently in *active* or *passive* mode.
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).

3.6 Summary

Chapter 4

Evaluation

A brief introduction to the evaluation here. We discuss the two-fold evaluation strategy: tests of the system as a whole to check that it works against our assumptions and the evaluation of its performance both in the steady state and in the face of failures.

4.1 Experimental setup

Talk about how for our evaluation we require the simulation of a network that admits the failures described in the preparation chapter. Hence Mininet.

4.1.1 Mininet

Introduction to Mininet. What it is. Why we're using it. Explain it runs on VirtualBox.

How Mininet works. Allows us to write Python scripts to simulate. Process-based virtualisation. How we need to install OCaml runtime etc on it. Explain how this was a problem and so we run OCaml as bytecode.

Perhaps a diagram here showing the workflow of how we use Mininet.

4.1.2 Experimental measurements

Talk about the need in our evaluation to measure two quantities - latency and throughput.

Define latency and throughput in the context of our simulation.

Explain how these were implemented and integrated into the program via command line arguments etc.

Might be a point-scorer to includle a simple graph of latency and throughput just to show what sort of measurements we get in any case.

4.2 Simulation tests

Define here what we mean by simulation tests: tests that ensure the expected behaviour of the system as a whole is undertaken.

Explain this fits into our definitions of what a test harness is and how we're going to use that to perform evaluations. Explain this also requires modification of the program to ensure we can crash / delay acceptors suitably so as to simulate the sort of failure modes we require.

Perhaps describe a system test and then also reference a table describing each test in the appendix.

4.3 Performance evaluation

Introduction to the performance evaluation.

4.3.1 Steady state behaviour

Describe what we mean by steady state behaviour.

Describe the network architecture; describe the no of clients, the messages they send, how many they send; rate of sending. Explain we want to avoid measuring delays in links and delays in dropped packets in queues etc.

Graphs of latency and throughput then for this steady state case. Compare this with Libpaxos.

4.3.2 System size

Describe how we want to measure commit latency and throughput as the number of different nodes in the system is varied. Achieve this by varying number of nodes and measuring latency and throughput.

Explain the confidence interval calculations.

Should end up with two bar charts for latency and throughput as a function of cluster size. Plot LibPaxos values for each as well

Compare performance to LibPaxos.

4.3.3 Failure traces

Describe how we need to assess the system in the case of failures.

Talk about computation of means, EWMA, etc.

Replica failure traces. Include plain failures, failures including restarts and cascading failure until we get no further latency and throughput. Include LibPaxos traces over the time. Should result in six graphs if we're lucky.

Compare performance to LibPaxos.

4.4 Summary

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].

Need to work out how to put a second bibliography for the proposal

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