Christopher Jones

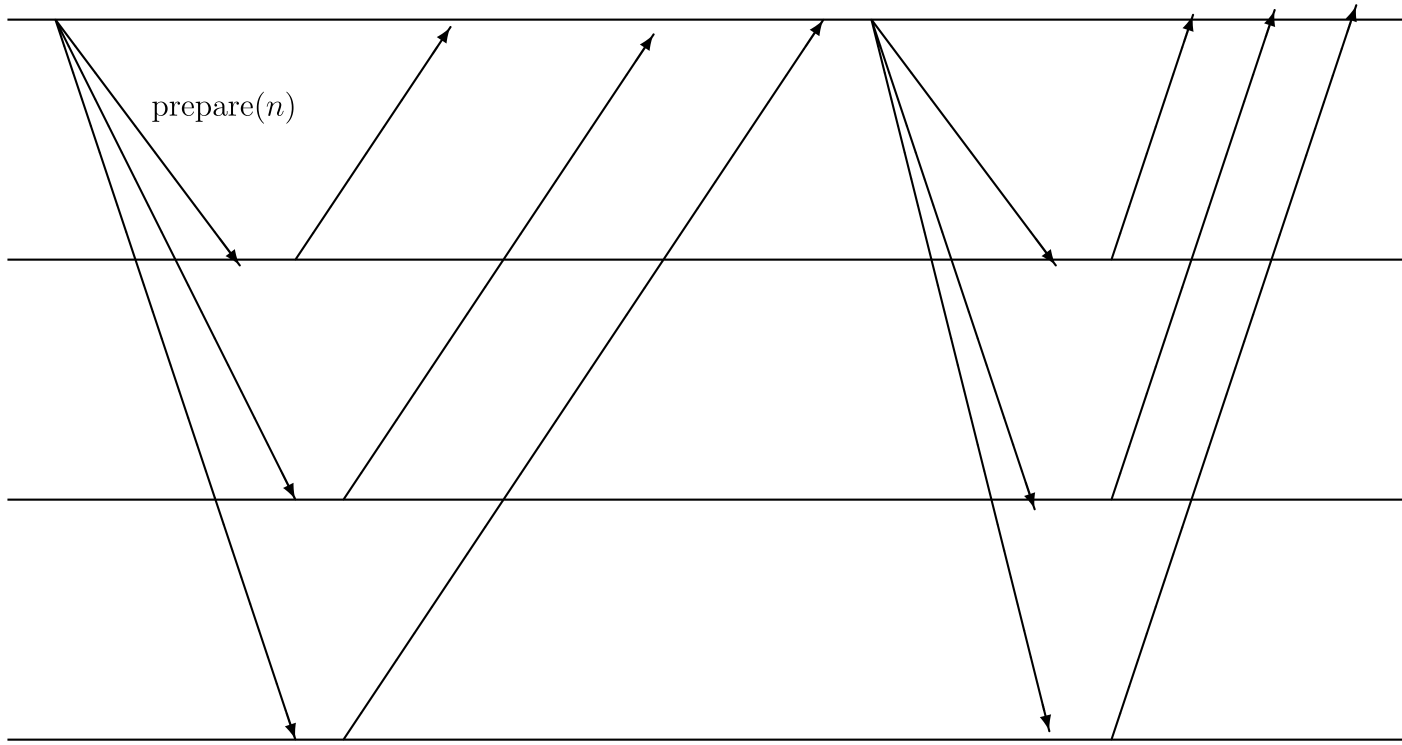
DRAFT

Achieving distributed consensus with Paxos

Part II Project

Trinity Hall

April 12, 2018


prepare(n)

Contents

List of Figures

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

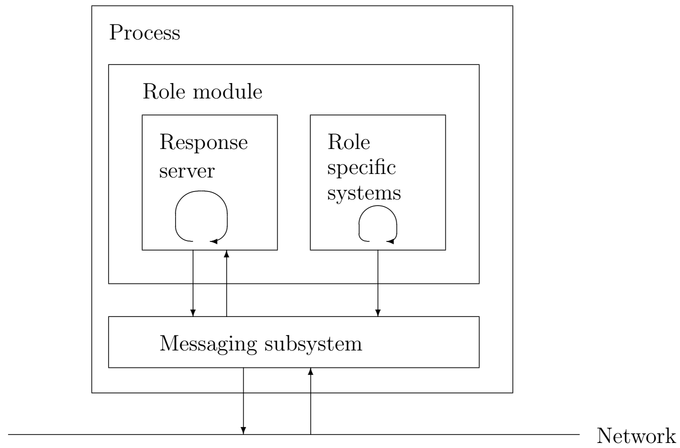


Figure 3.1: High level subsystems of a process

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_request : 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

Start with a high level overview and goals of the messaging system.

Now talk about the different messages we can send. In this case show the datatypes. Draw distinctions between the blocking and non-blocking nature of the phase1/2 and other messages. Perhaps at this point show the interface we wish to expose to the above subsystem. This would include send_non_blocking, send_blocking messages. This essentially should provide the semantics we desire.

Talk about the interface exposed on behalf of the server

```
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: Types of non-blocking messages

```
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

Now discuss Cap'n Proto. A quick overview of how it works. Discuss the schema file and how it corresponds to the messages we want to send. Talk about how compiling this produces ML interfaces and signatures we can use. Probably skim over the boilerplate code, just describe at a high-level how serialization works.

On the topic of serialization, talk about how things are serialized by their own respective methods. This can probably be quite a quick overview.

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.

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.

```
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

```
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 3.9: Types of non-blocking messages

Put a snippet here...?

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. 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:

- A queue of commands received in request messages from clients.
- 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.
- A queue of decisions. This represents the serialization of commands that has been decided on by the synod protocol. The sequence of decided commands should be the same for each replicas.

The formulation of these data structures introduces the requirement for some program invariants, such as ...

```
let new_replica host port leader_uris =
  let replica = initialize leader_uris in
    start_server replica host port >>=
    fun uri -> print_uri uri >>=
    fun () -> propose_lwt replica
```

Figure 3.10: Snippet of startup of replica

```
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 3.11: Types of records representing replica state. Note use of mutable keywords allowing for the records to be implicity modified withouth having to store and deal with record types.

- Leading on from clients, explain how messages can be re-ordered / lost / delayed from clients and how broadcasts from replicas allow progress in the event of replica failure.

- Describe the two-fold operation of replicas. They receive and perform de-duplication of broadcast client messages. They then go on to propose to commit commands to given slots. They also handle when leaders reject their proposals so they can be re-proposed later. They also maintain the application state (in this case the replicated key-value store.)

- Interesting points to discuss include looking at a snippet of the de-duplication function and discussing it. Also looking at how mutexes are used to hold locks on the sets of commands, proposals, decisions etc.

- Include a diagram of the three different sets of commands, proposals and decisions and how each moves from one set to another in different sets of circumstances.

3.5 Synod protocol

3.5.1 Quorums

- 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.

3.5.2 Acceptors

- Discuss the operation of acceptors - how they form the fault tolerant memory of Paxos.

- Discuss the functions (along with snippets) by which acceptors adopt and accept ballots.

3.5.3 Leaders

- Describe how the operation of leaders is split into two sub-processes - scouts and commanders.

- Operational description of how scouts and commanders communicate concurrently. Diagram displaying how there is a message queue that these sub-processes use to send "virtual" messages.
- Snippets of the signatures and structs used to construct these sub-processes.

Scouts

- Describe how scouts secure ballots with acceptors. Relate this to operation of acceptors above.
- State how scouts enqueue virtual adopted messages after securing adoption from quorum of acceptors.
- Describe how pre-emption can occur when a commander is waiting for adoption.

Commanders

- Explain how commanders attempt to commit their set of proposals.
- Give detail on the pmax and "arrow" function as used in the paper.
- Describe how pre-emption can occur when a commander is waiting for acceptance.

3.6 Summary

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