Mechanised Verification of Paxos-like Consensus Protocols

Anirudh Pillai

BSc Computer Science

Supervisor: Dr. Ilya Sergey

Submission Date: 30th April 2018

Thesis advisor: Dr. Ilya Sergey Anirudh Pillai

Mechanised Verification of Paxos-like Consensus Protocols

Abstract

Distributed systems have become an integral component of the modern world. Most of these systems power applications with over a million users, thus, the importance of correctly implementing such systems in a way that keeps them up and running and functioning correctly, has never been greater. Despite their widespread use, building correctly functioning distributed systems has remained a notoriously hard challenge.

In this project we use Disel, a framework for *compositional* verification of distributed systems. Recent work has yielded tools that support building verified implementations of the core components of a distributed system, yet, Disel goes beyond them by enabling one to combine the verified implementations of the core components to produce a correct implementation of the entire distributed system. This project aims to use Disel to implement a library of reusable verified distributed components, based on the classical family of fault-tolerant asynchronous Paxos-like consensus protocols, in which a number of participants are supposed to reach an agreement despite the possible failure of a minority of them.

Acknowledgments

I would like to thank my supervisor, Dr Ilya Sergey, for making the time to see me and arranging tutorials to help me understand the material for the project. Without his guidance this project would never have been completed. From undertaking this project and attending the reading groups arranged by him, I have gained valuable knowledge that will help me throughout my career.

Contents

Introduction

1.1 The Problem

Distributed systems are ubiquitous in the modern world yet correctly implementing such systems is still very hard. Furthermore, such systems are composed of various smaller components which in themself are hard to implement correctly. There has been recent work in developing frameworks to verify the implementation of the 'smaller' sub-components but there still remains the problem of checking whether these smaller components, when combined together, achieve the desired goal.

Another fundamental problem in the field of distributed computing is to make a set of process achieve consensus on something, for examples, the occurrence of an event or the decision to take some action. The set of distributed computing protocols which tries to solve this problem are termed as consensus protocols. One of these protocols is the Paxos consensus protocol which has numerous variants according to the constraints imposed on the set of processes.

Disel is a framework for compositional verification of distributed protocols built on top of the Coq proof assistant. It allows one to verify the correctness of the implemented components and also to check whether the verified components function correctly when combined together.

This project involves using Disel to mechanise the proof of the Paxos consensus protocol. We will also be using the code extraction capabilities of Disel to write a client application that uses the verified protocol to make set of distributed processes achieve consensus.

1.2 Aims and Goals

I have highlighted the aims and goals separately. The aims are what I want to achieve out of undertaking this project and the goals are the things that this project tries to achieve.

1.2.1 Aims

1. Learn about distributed computing. I wanted to learn how to reason about and implement distributed programs and the algorithms running on top of them. Working on verifying a popular distributed system protocol will teach me the concepts needed to understand such systems. While actually implementing applications using the protocol will teach me how to apply the concepts to the real world.
2. Contribute to open source software. Disel is an open source framework and working on that will enrich my open source contributions and improve my knowledge of the workflow.
3. Learn about formal verification. This was based partly on my interest in the various courses on logic. Using Coq for verification would enable me to apply my theoretical knowledge and use it for implementing and verifying a popular distributed system protocol.

1.2.2 Goals

1. Read about and understand the classical Paxos-like consensus algorithms.
2. Develop state transition systems for the algorithms and identify the invariants that need to be preserved during the operation of the algorithm.
3. Implement a simulation of the protocols in Python.
4. Formulate the implemented protocols in Disel by using the developed state-transition systems.
5. Mechanise the proofs of the identified protocol invariants in Disel/Coq.
6. Add additional communication channels and prove composite invariants.
7. Provide an abstract specification of the protocol, usable by third-party clients.
8. Mechanise a client application of the protocol verified out of the abstract interface.

1.3 Project Overview

During the course of the project we came up with a workflow for using Disel to mechanise proofs of protocols. This workflow is outlined in detail in section 3.3. The main stages in the project were as follows:

- First stage was to read about and understand the various concepts related to distributed computing and, more importantly, to understand the workings of Paxos-like consensus protocols. Implementing the Python simulation of the algorithm was very helpful in understanding the roles of each of the nodes in the protocol.
- The next stage was to design an adapted protocol that we would prove and to design the state transition diagrams of that protocol. We also designed the client application that would use the protocol and created a Python simulation of it to ensure that the design functioned correctly.
- The final stage was actually mechanising the proofs in Disel showing that the nodes in the client application correctly implemented the protocol while communicating with other nodes and achieving consensus. We first encoded the implementation of the client application and the protocol. Once, the client application functioned correctly, we moved on to writing proofs for it and the encoded protocol.

1.4 Report Overview

In the following chapter we look at various concepts relating to distributed computing and understand the two main components of this project, the Paxos protocol and the Disel framework. Then, with that background information, we look at how we adapted the design of the Paxos protocol and designed state transition systems for it. Later, we move on to actually encoding the protocol in Disel and creating a client application that uses the verified protocol. Finally, we evaluate the results of the project, the experience of using Disel and look at areas for future development.

Background

This chapter lays down all the previous research which the project builds on. Before going over the design decisions on the project we first need to understand this background information and look at related work to see different approaches used to solve the problem.

2.1 Distributed Systems

A distributed system is a model in which processes running on different computers, which are connected together in a network, exchange messages to coordinate their action, often resulting in the user thinking of the entire system as one single unified computer. A computer in the distributed system is also alternatively referred to as a processor or a node in the system. Each node in a distributed system has its own memory.

We will now go over a few concepts of distributed systems which will help us understand the characteristics of the protocols that run on these systems. This will lay down the groundwork for us to understand the Paxos protocol on which this project is based.

2.1.1 Asynchronous Environment

An asynchronous distributed system is one where there are no guarantees about the timing and order in which events occur. The clocks of each of the process in the system can be out of sync and may not be accurate. Therefore, there can be no guarantees about the order in which events occur. Further, messages sent by one process to another can be delayed for an arbitrary period of time.

A protocol running in an asynchronous environment has to account for these conditions in its design and try to achieve its goal without the guarantees of timed events. An asynchronous environment is very common for a real world distributed system but it also makes reasoning about the system harder because of the aforementioned properties.

2.1.2 Fault Tolerance

A fault tolerant distributed system is one which can continue to function correctly despite the failure of some of its components. A 'failure' of a node or 'fault' in a node means any unexpected behaviour from that node, for example, not responding to messages or sending corrupted messages.

Fault tolerance is one of the main reasons for using a distributed system as it increases the chances of your application continuing to functioning correctly and makes it more dependable. As Netflix mention on their blog 'Fault Tolerance is a Requirement, Not a Feature' [1]. With their Netflix API receiving more than 1 billion requests a day, they expect that it is guaranteed that some of the nodes in their distributed system will fail. Using a fault tolerant distributed system they are able to ensure that a small failure in some nodes doesn't hinder the performance of the overall system, hence, enabling them to achieve their uptime metrics.

Fault tolerant distributed system protocols are protocols which achieve their goals despite the failure of some of the nodes of the distributed system they run on. The protocol accounts for the failures and generally specifies the maximum number of failures and the types of failures it can handle before it stops functioning correctly.

2.1.3 State Machine Replication

For a client server model, the easiest way to implement it is to use one single server which handles all the client request. Obviously this isn't the most robust solution as if the single server fails, so does your service. To overcome the problem you use a collection of servers each of which is a replica of the original single server and ensure that each of these 'replicas' fails independently, without effecting the other replicas. This adds more fault tolerance.

State machine replication is method for creating a fault tolerant distributed system by replicating servers and using protocols to coordinate the interactions of these replicated servers with the client. Schneider [2] points out how to use state machine replication to implement fault tolerant services.

A state machine $\mathcal{M}$ can be defined as $\mathcal{M} = \langle q_0, Q, I, O, \delta, \gamma \rangle$ where

- q_0 is the starting state
- Q is the set of all possible states.
- I is set of all valid inputs
- O is the set of all valid outputs
- δ is the state transition function, $\delta : I \times Q \rightarrow Q$
- γ is the output function, $\gamma : I \times Q \rightarrow O$

The state machine begins in the start state and transitions to other states and produces outputs when it receives the inputs. The transition and output are found using the transition and output functions. A deterministic state machine is one whose state transition and output functions are injective, i.e. multiple copies of the machine when given the same input, pass through the same order of states and produce the same output in the same order.

The method of modelling a distributed system protocol as state transition system was established by Lamport [3] and is very common. It is a critical component of this project as we will see soon when we need to encode our protocol in Disel.

State machine replication involves modelling our single server, from the client server model, and using multiple copies (replicas) of the same deterministic state machine and providing all of them with the input from the client. As long as one of the replicas does not crash while resolving the request, we can successfully return a response to the client.

2.1.4 Consensus Protocols

For handling faults in your distributed system you need to have replication. This leads to the problem of making all these replicas agree with each other to keep them consistent. Consensus protocols try to solve this problem.

> Consensus protocols are the family of distributed systems protocols which aim to make a distributed network of processes agree on one result.

These protocols are of interest because of their numerous real world applications. Let us take the example of a distributed database, which is a critical part of almost all large scale real world applications. This distributed database will run over a network of computers and every time you use the database you aren't guaranteed to be served by the same computer.

Suppose you add a file to the database. This action is performed by a node, in the distributed database, that was handling your 'add' request. Later when you want to retrieve the file from the database you might be served by a different node in the distributed database that did not perform the 'add' request. In order for the new node to know that the file exists in the system, you will need to use a consensus protocol which helps all the nodes in the system (which handle user requests) to agree upon the result that the file has been added to the system.

A popular consensus protocol is the blockchain consensus protocol which powers Bitcoin. Pîrlea et al [4] have verified a subset of this protocol in Coq. Other examples of consensus protocols are Raft [5], Stellar Consensus Protocol [6] and Paxos [7], which we look at next.

2.2 Paxos

Having understood the the main concepts behind distributed system protocols, we can now finally get to the protocol at the heart of this project. Paxos is a family of asynchronous, fault tolerant, consensus protocol which achieves consensus in a network of unreliable processes as long as a majority of them don't fail. Paxos was outlined in Lamport's 1998 paper, 'Part Time Parliament' [8].

Paxos is used for state machine replication. Once you have multiple replicas servicing client requests, how do you makes sure that all of these replicas agree on what action to take? The solution is simply to use a consensus protocol like Paxos to make all replicas agree on something. Paxos is heavily used in building software. It has been used at Google to build a fault tolerate database [9] and the Chubby [5] lock service.

Paxos has many variants but the one we will focus on is the one we actually prove in Disel, single decree Paxos, also know as simple Paxos. Simple Paxos is an algorithm that helps a distributed network of processors to achieve consensus. Consensus is achieved when the network of processor agree on a common value.

For simple Paxos, we assume the following assumptions hold about the processors and the environment, in order for the protocol to function correctly.

- Processors communicate between each other by exchanging asynchronous messages between them.

- Processors run at an arbitrary speed and may fail or restart. Handling this relates to the fault tolerant nature of Paxos. Also, we assume that Byzantine faults don't occur. This means that all processors actually work together to try to achieve consensus on a value. There are variants of Paxos which can also handle Byzantine failure but not simple Paxos. (This can be linked to the 'Practical Byzantine Fault Tolerance' paper, by Castro et al [10], which states that any algorithm handling Byzantine faults must have three phases. Simple Paxos only has two phases.)

As for fault tolerance of Paxos, in order to handle a failure of up to f processors, we need to have a minimum $2f+1$ processors participating in the algorithm. This means Paxos functions correctly as long as a majority of the processors in the network do not fail. We will see shortly why just a majority needs to be functioning correctly.

A processor participating in simple Paxos, may have one or more of these three different roles - proposer, acceptor or learner.

- Proposer - A process acting as a proposer listens for client request and proposes a value which the network of processes tries to agree upon.
- Acceptor - Acceptors receive proposed values from the proposers and then respond to them stating whether they are in a position to accept the value or not. For a proposed value to be accepted, a majority of all the existing acceptors have to accept the proposed value.
- Learner - The learner has to be informed when an acceptor accepts a value. The learner can then figure out when consensus has been achieved by monitoring when a majority of acceptors have accepted the same proposal. Once the acceptors agree on a value, the learner may act on the value by, for example, sending a request to the client that informs them about the agreed value.

2.2.1 Choosing a Value

For passing around the value to be chosen from one processor to the other, a processor must send a 'proposal' to the other processor. You can think of a proposal as just a tuple $\langle n, v\rangle$. n is just a natural number associated with a proposal which makes it easy to keep track of all the different proposals and v is the value the value that is being proposed.

A quorum of acceptors is a subset of the set of all acceptors and has a length greater than $N/2$ where N is the length of the set of acceptors. A quorum is just a set denoting a majority of all the available acceptors.

> Consensus is achieved when a proposal is accepted by a majority of acceptors.

The Algorithm

Simple Paxos runs in rounds until consensus is achieved (a successful round occurs when a majority of acceptors accept a proposal). A successful round of the algorithm has two phases, each of which can be subdivided into parts a, b.

- Phase 1a: Prepare Request. A proposer sends a proposal $\langle n, v \rangle$ to each acceptor in any randomly chosen quorum of acceptors. This first message that the proposer sends out is called a prepare request. This phase can be thought of as the proposer trying to 'prepare' the acceptors to 'accept' a value in the future.

- Phase 1b: Promise Response. An acceptor on receiving a prepare request, responds with a promise response, if and only if the acceptor has not already sent a promise response with a proposal containing a proposal number n' where $n' > n$.

  A promise response for proposal $\langle n, v \rangle$ is basically a guarantee (a 'promise') that this acceptor will not respond to any messages with proposals that have a proposal number n' where $n' \leq n$.

  Thus, if an incoming prepare request has proposal number that is not greater than what the acceptor has already promised earlier, then the acceptor can ignore this prepare request by not responding to it. Although, for speeding up the protocol, the acceptor can send out a nack response which tells the proposer to stop trying to achieve consensus with this proposal.

  If the acceptor has not already accepted a proposal, then the body of the promise response can be empty, otherwise, the acceptor must include in it the last proposal that it accepted.

- Phase 2b: Accept Request. If the proposer successfully receives promise responses from a majority of acceptors, then it can send out an accept request. A accept request is a message containing a proposal which tells an acceptor to accept this proposal if it can.

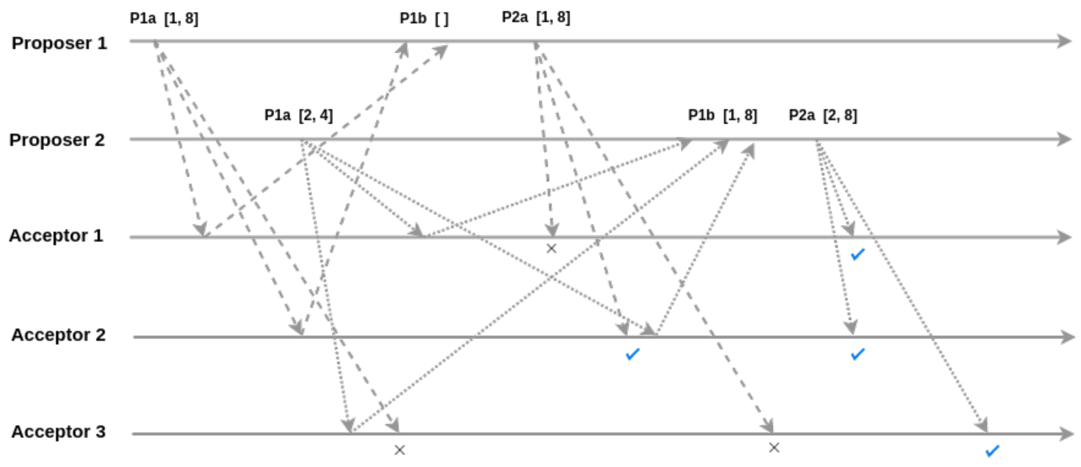


Figure 2.1: A run of Paxos

The proposer creates a new proposal, $\langle n, v' \rangle$ where n is the same as in the proposal which the proposer sent in its prepare request. But, v' is the value from the highest numbered proposal, selected from all the proposals that the proposer receives in the promise responses. If none of the promise responses received by the proposer contain a proposal, the proposer is free to set v' to any value it likes. The proposer then sends this accept request with proposal $\langle n, v' \rangle$ to another `quorum` of acceptors.

- Phase 2b: Accepted Response. Any acceptor that receives the accept request with proposal $\langle n, v \rangle$, accepts the proposal and responds with an accepted response if and only if it hasn't already promised not to respond to any proposals with proposal number n' where $n' >= n$. The accepted response is sent to the learner and contains the proposal that was accepted by the acceptor.

2.2.2 Informing learner

When consensus is achieved, a learner must be informed that a majority of acceptors have agreed on a value. There are various ways to do this.

1. Whenever an acceptor accepts a value, it should send the accepted proposal to all the learners (accepted response). The learners will then know when a majority of acceptors have accepted the same value.
2. We can have a distinguished learner which informs other learners about the chosen value. The acceptors only need to inform this particular learner when they accept a

value. This reduces number of messages sent but the distinguished learner becomes the single point of failure and also requires an additional round of sending messages where the distinguished learner informs other learners that a value has been chosen.

3. We can use a set of distinguished learners. The acceptors inform these distinguished learners who then inform the other learners. This increases reliability but also increases the number of messages exchanged.

2.3 Disel

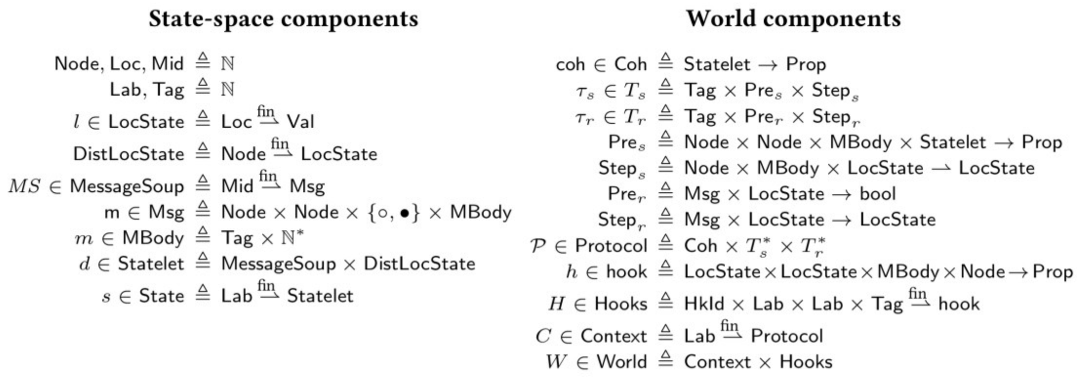
Having learnt the concepts behind distributed systems and the Paxos protocol, we can now look at Disel. Disel [11] is a verification framework, built on top of the Coq theorem prover, that enables one to prove the safety properties of a distributed protocol by breaking down the protocol into its state space invariants and its atomic properties. The unique aspect of Disel is that it also allows one to combine these separate verified protocols to create a complete verified system. Additionally, the code extraction properties of Disel enables one to extract the verified runnable OCaml code for the protocols which can be combined with an shim that supplies the implementation of the various primitive operations like sending or receiving a message.

2.3.1 Protocol Encoding

We will now look at how Disel requires the state space of a protocol to be encoded in it. The figure 2.2 shows the distributed state space and the world components in Disel.

As you can see from 2.2, a statelet is a component in the protocol and consists of the MessageSoup and the DistLocState. A message soup is 'finite partial map from unique message identifiers to messages, each of which carries its sender and recipient node ids, the payload m, which includes a tag, and a boolean indicating whether the message is already received or not yet' [11]. While the local state of a node 'maps each node id into protocol-specific piece of local state, represented as a mapping from locations (isomorphic to natural numbers) to specific values' [11].

Additionally, a protocol P in Disel is defined as a tuple of the Coherence, the set of send transitions and the set of receive transitions. Let us now look at each of these components in detail.



State-space components

$$\begin{aligned}
\mathsf{Node}, \mathsf{Loc}, \mathsf{Mid} &\triangleq \mathbb{N}\\
\mathsf{Lab}, \mathsf{Tag} &\triangleq \mathbb{N}\\
l \in \mathsf{LocState} &\triangleq \mathsf{Loc} \xrightharpoonup{\text{fin}} \mathsf{Val}\\
\mathsf{DistLocState} &\triangleq \mathsf{Node} \xrightharpoonup{\text{fin}} \mathsf{LocState}\\
MS \in \mathsf{MessageSoup} &\triangleq \mathsf{Mid} \xrightharpoonup{\text{fin}} \mathsf{Msg}\\
\mathsf{m} \in \mathsf{Msg} &\triangleq \mathsf{Node} \times \mathsf{Node} \times \{\circ, \bullet\} \times \mathsf{MBody}\\
m \in \mathsf{MBody} &\triangleq \mathsf{Tag} \times \mathbb{N}^*\\
d \in \mathsf{Statelet} &\triangleq \mathsf{MessageSoup} \times \mathsf{DistLocState}\\
s \in \mathsf{State} &\triangleq \mathsf{Lab} \xrightharpoonup{\text{fin}} \mathsf{Statelet}
\end{aligned}$$

World components

$$\begin{aligned}
\mathsf{coh} \in \mathsf{Coh} &\triangleq \mathsf{Statelet} \to \mathsf{Prop}\\
\tau_s \in T_s &\triangleq \mathsf{Tag} \times \mathsf{Pre}_s \times \mathsf{Step}_s\\
\tau_r \in T_r &\triangleq \mathsf{Tag} \times \mathsf{Pre}_r \times \mathsf{Step}_r\\
\mathsf{Pre}_s &\triangleq \mathsf{Node} \times \mathsf{Node} \times \mathsf{MBody} \times \mathsf{Statelet} \to \mathsf{Prop}\\
\mathsf{Step}_s &\triangleq \mathsf{Node} \times \mathsf{MBody} \times \mathsf{LocState} \rightharpoonup \mathsf{LocState}\\
\mathsf{Pre}_r &\triangleq \mathsf{Msg} \times \mathsf{LocState} \to \mathsf{bool}\\
\mathsf{Step}_r &\triangleq \mathsf{Msg} \times \mathsf{LocState} \to \mathsf{LocState}\\
\mathcal{P} \in \mathsf{Protocol} &\triangleq \mathsf{Coh} \times T_s^* \times T_r^*\\
h \in \mathsf{hook} &\triangleq \mathsf{LocState} \times \mathsf{LocState} \times \mathsf{MBody} \times \mathsf{Node} \to \mathsf{Prop}\\
H \in \mathsf{Hooks} &\triangleq \mathsf{HkId} \times \mathsf{Lab} \times \mathsf{Lab} \times \mathsf{Tag} \xrightharpoonup{\text{fin}} \mathsf{hook}\\
C \in \mathsf{Context} &\triangleq \mathsf{Lab} \xrightharpoonup{\text{fin}} \mathsf{Protocol}\\
W \in \mathsf{World} &\triangleq \mathsf{Context} \times \mathsf{Hooks}
\end{aligned}$$

Figure 2.2: Disel's distributed state space and the world components as shown in [11]

A Coherence is predicate, i.e., a function that takes in a statelet and returns a proposition indicating whether the statelet is valid or not. Thus, the coherence allows us to impose constraints on the local state of each node and on the message soup.

A transition is defined as a tuple consisting of the following:

1. Tag - a unique natural number identifier for the message to be sent in the transition.
2. Precondition - The constraints that are imposed on identity of the sender of the message, identity of the receiver is, the message that is being sent and on the local state of the sender/receiver (depending on whether it is a send transition/receive transition).
3. Step function - Describes how the local state of the sender/receiver changes after making the transition.

You can see in the code example below how the step function and pre condition are encoded for sending the prepare request in Paxos. `PInit`, `PSentPrep` and `PWaitPrepResp` are some states the proposer can be in. The precodition requires the state of proposer p to be either `PInit` or `PSentPrep` and holding things required in the specific state.

```
(* Changes in the Node state triggered upon send *)
Definition step_send (s: StateT) (to : nid) (p: proposal): StateT :=
   let: (e, rs) := s in
   match rs with
   ...
   (* Step function for the sending prepare request *)
```

```
    | PInit p' ⇒
      if acceptors == [:: to] (* if only one acceptor *)
      then (e, PWaitPrepResp [::] p')
      else (e, PSentPrep [:: to] p')
    ...
    | _ ⇒ (e, rs)
    end.

(* Precondition for send prepare request transition *)
Definition send_prepare_req_prec (p: StateT) (m: payload) :=
  (∃ n psal, p = (n, PInit psal)) ∨
  (∃ n tos psal, p = (n, PSentPrep tos psal)).
```

2.3.2 Protocol to Programs

The state transitions that we implement in the protocol encoding phase, are the first step towards creating executable programs using Disel. We can then use the library of *transition wrappers* provided by Disel that allow one to decorate low level send/receive primitives with the transitions that we have defined. These decorated primitives can later on be used to extract code for executable programs. In Chapter 5, we will dive into the details of how we extracted the code for our client application running Paxos.

The `send_action_wrapper` wrapper provided by Disel takes a send transition encoded by us and returns a program that will send a message.

```
Program Definition send_prepare_req psal to :=
  act (@send_action_wrapper W paxos p l (prEq paxos)
     (send_prepare_req_trans proposers acceptors) _ psal to).
```

The `tryrecv_action_wrapper` is similar but the main difference is that in a received transition, we may receive messages from any of the multiple protocols that might be executing at the time. To address this problem, we need to check the tag t returned by the receive wrapper and ensure that this tag belongs to the protocol that was specified in the wrapper. In the code example below, we check that the received message is either a `promise_resp` or a `nack_resp` both of which belong to the `paxos` protocol and are valid responses to a `prepare_req`.

```
(* Non blocking receive *)
```

```
Program Definition tryrecv_prepare_resp := act (@tryrecv_action_wrapper W p
    (* filter *)
    (fun k _ t b ⇒ (k == l) && ((t == promise_resp) || (t == nack_resp))) _).
```

If an incoming message matches the conditions specified, the wrapper returns `Some(from, m)` where *m* is the message and *from* is the sender. Otherwise it returns `None`.

These low level primitives can then be combined together for specifying the roles of each node in the protocol. We can use the `send_prepare_req` to come up with `send_prepare_req_loop` which every proposer performs when it starts up. These functions can then further be combined together to give the entire implementation of a node. `proposer_round` below is the program that each node acting a proposer executes.

```
Program Definition send_prepare_req_loop e (psal: proposal):
  {(pinit: proposal)}, DHT [p, W]
  (fun i ⇒ loc i = st :→ (e, PInit pinit),
   fun r m ⇒ r = tt ∧
           loc m = st :→ (e, PWaitPrepResp [::] pinit)) :=
  Do (ffix (fun (rec : send_prepare_req_loop_spec e) to_send ⇒
           Do (match to_send with
              | to :: tos ⇒ send_prepare_req psal to ;; rec tos
              | [::] ⇒ ret _ _ tt
              end)) acceptors).

Program Definition proposer_round (psal: proposal):
  {(e : nat)}, DHT [p, W]
  (fun i ⇒ loc i = st :→ (e, PInit psal),
   fun res m ⇒ loc m = st :→ (e.+1, PAbort))
  :=
  Do (e <-- read_round;
     send_prepare_req_loop e psal;;
     recv_promises <-- receive_prepare_resp_loop e;
     check <-- check_promises recv_promises;
     if check
     then send_accept_reqs e (choose_highest_numbered_proposal psal recv_promises)
     else send_accept_reqs e [:: 0; 0]).
    (* If check fails then send an acc_req for (0, 0) which will never be
      accepted by any acceptor *)
```

Once this implementation has been finished in Disel we can use Disel's extraction capabilities to extract the OCaml code for executing the program. (Outlined in Chapter 5)

2.3.3 Inductive Invariant

As many other verification tools for distributed protocols, Disel relies on using inductive invariants to prove the correctness of the protocol. In this section, we look at what are invariants and inductive invariants.

A invariant in a program is a property of a program that always holds true, from the start through to the end of execution of the program. An invariant can be for something more specific like a function or even a loop. The only requirement is the property described by the invariant should always hold before, during and after execution of the part of that code. To put it more formally, 'An invariant of a transition system is a property that is always true, in all of the system's reachable states.' [12].

The problem making an assertion like $x > 2$, at a certain point of a program execution, is that you may assume that the it holds at that point in the program execution, but you still do not have a guarantee that it will hold before, during and after the execution of the program, i.e. hold in any state of the program.

Therefore, we need to make our invariants *inductive*. An inductive invariant of a program, is an invariant which if it holds in a particular state s of the program, it is guaranteed to hold in all states of the program reachable from s. Thus, having an inductive invariant that holds in the start state of a program is much more useful, as we can be sure that the invariant will continue to hold throughout and after the execution of the program.

This means that once you establish an inductive invariant from the given invariants of a protocol and a starting state for the protocol in which the invariant holds, in order to show that the protocol maintains the invariant, you only need to prove the inductive invariant maintains the induction property over any possible state transition in the protocol. This is exactly how you use an inductive invariant in Disel. Developing and encoding the inductive invariant for the protocol in Disel is the way to verify it.

2.4 Related Work

People have attempted to create software for verfying the correctness of distributed systems. IronFleet [13], developed at Microsoft Research, can prove not just safety but also liveness properties of distributed systems and has been used to verify a 'complex implementation of a Paxos-based replicated state machine library'. Verdi [14] is another framework built on top of Coq and has been used to develop the 'proof of linearizability of the Raft state machine replication algorithm'. Ivy [15] is another tool which makes it much easier to find and prove inductive invariants for the system. Other similar tools are PSync [16] and EventML [17].

Requirements and Analysis

3.1 Problem Statement

Having looked at the background information on distributed sytems, consensus protocols and Disel, we can formalise the problem statement for the project. The project aims to mechanise the proof of Paxos in Disel.

Mechanisation is a way to use Coq's higher-order logic to build mechanically checked proofs. By mechanising Paxos in Disel we will create a fully verified implementation of the protocol (of the consensus agreement and the code correctness) which can then be used to achieve consensus in a set of nodes running the protocol.

The correctness of Paxos will be shown by, mechanising in Disel, an inductive invariant for it that proves consensus being achieved. We will also implement a client application built on top of the verified protocol by using Disel's code extraction features. Finally, we will mechanise the proofs showing that the implemented client application complies with the encoded protocol.

3.2 Requirements

1. Adapt Paxos for encoding in Disel and devise the state-transition system for this protocol.

2. Develop an inductive invariant for the adapted protocol that ensures the protocol functions correctly by imposing requirements on the global state of the system

3. Implement a simulation of the adapted protocol with the developed state transition system.

4. Mechanise the proof of the adapted protocol in Disel/Coq. Thereby, providing a library of reusable verified distributed components.
5. Mechanise a client application of the protocol verified out of the abstract interface.

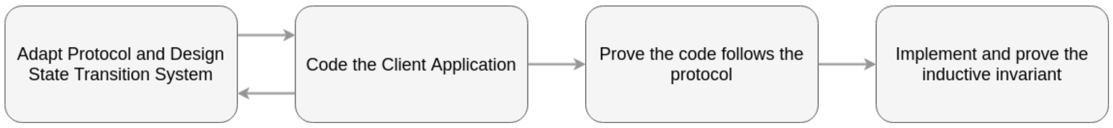

Adapt Protocol and Design State Transition System
Code the Client Application
Prove the code follows the protocol
Implement and prove the inductive invariant

Figure 3.1: Disel Workflow

3.3 Analysis: Approach to Mechanising Proofs in Disel

Having decided on the requirements we came up with a workflow to mechanise the proof in Disel. Finishing all the stages in the workflow will help us to meet all our requirements.

3.3.1 Adapt protocol and design state transition system

By adapting the protocol we try to focus on the 'core' parts of the protocol and to do away with the 'convenience' parts. For Paxos, this is focusing on the part of the protocol that deals with achieving consensus and not looking at the part where the learner is informed of the decision.

We also need to design state transition systems for the nodes that participate in the protocol. This makes it easier for us to encode the protocol in Disel as we already know which send and receive transitions we will need from each state and thus can easily code the transition wrappers. In chapter 4, we look go into the details of how we tackled this stage on our way to mechanising the proof of Paxos.

3.3.2 Code the client application

Secondly, we need to think of a client application that uses the implemented protocol. We need to design and implement a client that will demonstrate the properties of the protocol that we want to prove. In case of Paxos, we will design a client where nodes try to acheive consensus on one of the proposals that the proposer is initialised with. This will enabled us to see in action the stages of the protocol that lead to consensus being achieved.

While using Disel, we will often have to cycle between stage 1 and stage 2. This is because while implementing the client, you realise things like, you are missing one state for a node that is required in order for the protocol to progress. There was a case when we saw this while writing the client for Paxos. We realised that we were missing the `PAbort` state for the proposer, which was necessary to signify when a proposer stops participating in the protocol.

The process of cycling between stages 1 and 2 allow us to solidify our adapted protocol. Doing this at an early stage (before starting the proofs) also has the advantage of us not having to rewrite a lot of code that will also break all the proofs relating to the change.

Additionally, implementing the client also helps us identify unnecessary stages and transitions in the adapted protocol which were not needed to implement the client. Reducing the number of states and transitions vastly reduces the amount of things you will need to prove in the later stages.

3.3.3 Prove the code follows the protocol

The next stage involves encoding and proving the protocol and proving that the code for the client actually follows the adapted protocol. Thus, finishing the proofs in this stage gives us confidence that our adapted protocol can actually be used to fulfil the role which our client performs. In case of Paxos, this helped us realise that our adapted protocol can be used to achieve consensus among the acceptors. Finishing this stage means that we have finished the proof of the client. Yet, at this stage we can only be sure that the code follows the protocol.

Reaching this state also gives us relatively strong correctness properties. This is because, just by ensuring that the client follows the protocol, if we already know that the protocol is correct (as is the case with Paxos because of the proof of its correctness in Lamport's paper [8]), we can know that the client will work correctly. In the mechanising of Paxos we were only able to reach till this stage, we designed the inductive invariant but ran out of time to encode it. As the actual protocol we encoded is the simple Paxos algorithm, it still gives us quite a strong correctness as the simple Paxos algorithm has a proof of safety and has also been used to implement real world systems.

3.3.4 Implement and prove the inductive invariant

An inductive invariant is necessary as it helps verify the entire global system. As without it, the code might follow the protocol, but the protocol might itself be broken. In case of Paxos this could be the case that the nodes follow the protocol but they never achieve consensus.

In Disel, we show that the inductive invariant holds in the initial state of the system. Then we show that the inductive invariant holds with every single state transition in the protocol. This enables us to prove that the inductive invariant holds in the protocol, thus, verifying that the protocol functions correctly.

Protocol Design and Encoding

This chapter has been split into two parts. First is the 'Modelling' section where look at the design of the adapted protocol and the state transitions for Paxos. We also look at the Python simulator that helped solidify these designs before encoding them in Disel. The latter part of this chapter explains how the adapted protocol was encoded in Disel.

4.1 Modelling

4.1.1 The Adapted Protocol

For mechanising the proof of Paxos in Disel, we followed the approach highlighted in the Disel section of Chapter 3. Applying this approach meant we needed to come up with the state transition system and the inductive invariant for the protocol which we will encode in Disel. Before coming up with these things though, we decided to simplify the actual simple Paxos protocol that we will prove.

The goal of this simplification was to reduce the amount of things that we will need to prove in Disel by focusing on the 'core' parts of the protocol which lead to consensus being achieved. The 'convenience' parts of the protocol, like the sub phase of informing the learner, aren't necessary for consensus being achieved in the global state and can also be proved separately after we have finished proving the 'core' parts of the protocol. Garcia-Perez et al [18] have also showed how the optimizations of a protocol can be verified separately from the 'core' protocol. In their paper, they produce a verified implementation of multi Paxos using their verified implementation of simple Paxos.

In order to adapt the protocol and focus on the 'core' parts, the first thing we did was to do away with the 'Informing the learner' phase of simple Paxos. This is the phase where once an acceptor has accepted a proposal, it then informs a leaner by sending it an accepted request.

We decided against proving this because we can use the inductive invariant to check the global state of the system. The inductive invariant allows to check when a majority of acceptors have accepted a proposal and what proposal each of them has accepted. Thus, we can know from the inductive invariant when consensus has been achieved by checking the values of each of the accepted proposals.

Getting rid of the learning phase also meant that we did not have to implement a learner. The learner is useful in the actual Paxos implementations as it can detect when consensus is achieved and can then inform the client, yet, its presence does not change how consensus is achieved. Removing the learner simplified our state transition diagram for the protocol as we did not have to account for the states of the learner and also the *Phase2b* transition of the accepted request.

Additionally, we decided to remove some optimisations from the protocol. Optimisations are aspects of the protocol that improve its running time or resource consumption by improving upon the 'mundane' way of doing the same thing. Removing optimisations allowed us to simplify our proofs in Disel.

Therefore, we decided that if a proposer fails while trying to achieve consensus with a proposal number (i.e. it receives a nack), then it does not later try to propose again with a higher proposal number. Removing this optimisation meant we did not have to deal with the state of a proposer where it changes the proposal number it was initialised with. Removing this optimisation only effects the liveness of the protocol but not its safety, as if a proposer receives a nack, it is just an indication that it will not be able to achieve consensus with the proposal number that it is currently using. Thus, in order for consensus to be achieved, a proposer which is initialised with a higher proposal number will have to successfully get promises from a quorum of acceptors.

This adapted and simplified protocol allows us to focus on verifying the core logic (the part dealing with achieving consensus) of Paxos. The optimisations and 'convenience' that we removed can be verified separately after the core logic has been verified.

4.1.2 State Transitions

Having adapted the protocol, we then had to create a state transition system for the nodes in our protocol in order to encode it in Disel. For creating the states, we need to look at what the function of each node is in the protocol at a particular moment and what type of data it holds at that time.

A node should only be able to transition from one state to another when it either receives or sends a message. Therefore, the data held by the node in each state should be enough for it to be able to create the message it wants to send or to be able to correctly process the message it receives.

We tried to minimise the number of states and transitions between them, in order to simplify the proof in Disel. This was important because each state transition has to be shown to hold with the invariant so reducing the state transitions, reduces the number of proofs.

We decided that a node can either be initialised as an acceptor or a proposer. The state transitions of each node will depend upon this initial state, so below we will separately look at the state transition systems for the proposer and the acceptor.

The main difference between the state transition for the acceptor and the proposer is that the acceptor sends and receives messages from a single proposer while a proposer has to send and receive messages from all the acceptors.

Proposer

The proposer starts off in the `PInit` state where it is initialised with a `proposal` (a custom defined data type which is tuple of two natural numbers), $\langle p, v\rangle$. The natural number p is the proposal number and v is the value that the proposer tries to achieve consensus on. This means that the first prepare request this proposer sends will include this proposal $\langle p, v\rangle$.

The proposer then moves to the `PSentPrep` state when it starts to send prepare requests to the acceptors. In this state, the proposer still holds the proposal but additionally now also stores a list of natural numbers, `sent_to`. This list stores the natural number identifiers of the acceptors, this proposer has sent requests to. Whenever it sends a prepare request to an acceptor, it adds the identifier of the acceptor to this list. The proposer remains in the `PSentPrep` state and keeps sending prepare requests until the contents of `sent_to` become equal to the global list `acceptors` which holds the identifiers of all the acceptors in the system. This means the proposer stays in this state until it has sent a prepare request to every single acceptor in the system.

Once the proposer has sent the last prepare request, it them transitions to the `PWaitPrepResp` state. In this state the proposer again holds a proposal and another list `promises` which is

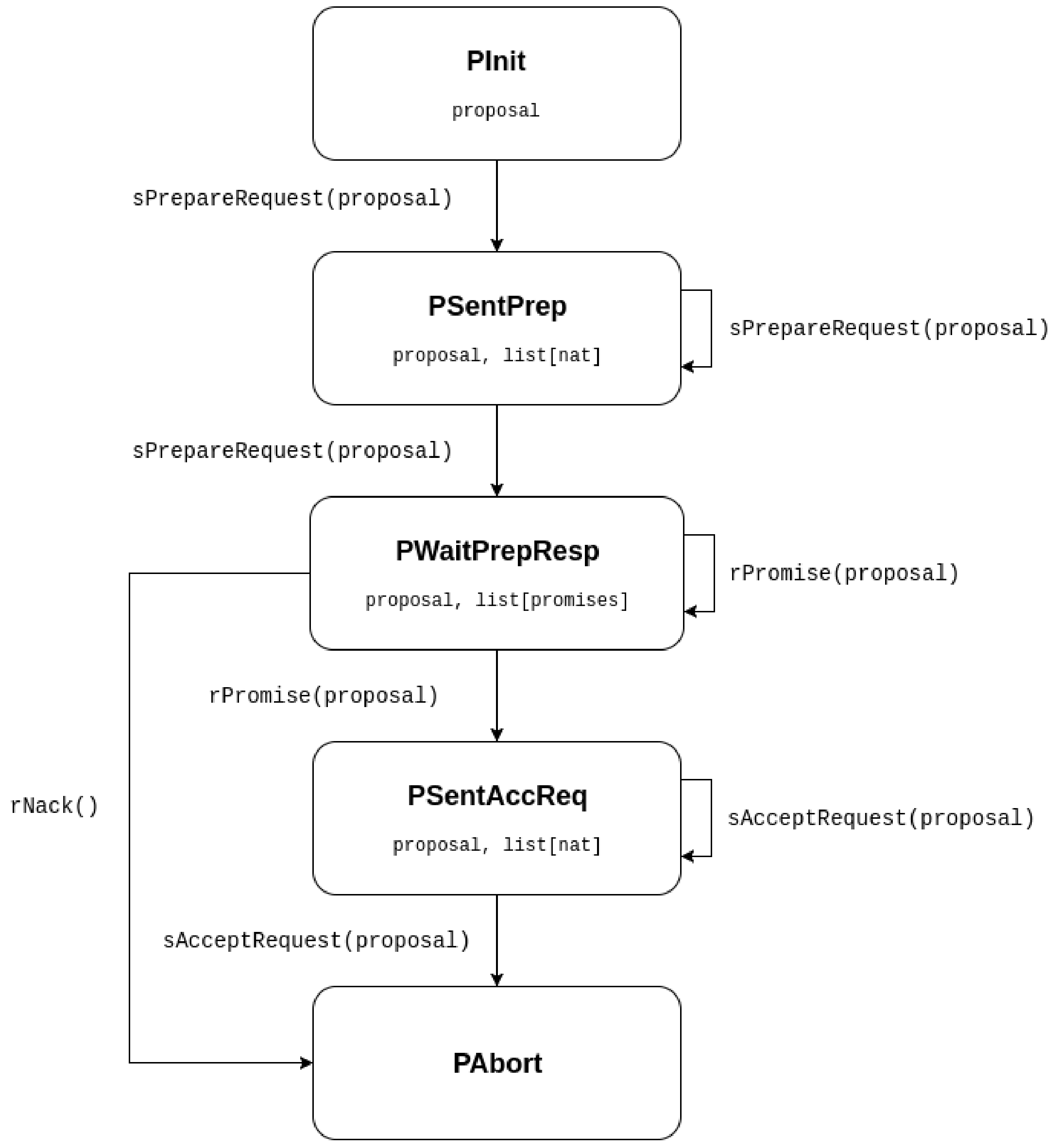


Figure 4.1: Proposer State Transition Diagram

defined as below to be a list of tuples each containing a `nid` (a natural number identifier for a node), a boolean and a `proposal`.

```
Definition promises := seq (nid * bool * proposal).
```

The proposer stays in this state and keeps receiving messages from the acceptor until one of the following two things happen:

1. It receives a nack response from the acceptor. This indicates that the acceptor might already have promised a proposal with a proposal number greater than p. This makes the proposer transition into the `PAbort` state. In this state the proposer basically gives up trying to achieve consensus using the proposal number p that it was initialised with and completely stops sending and receiving messages. Hence, the proposer doesn't need to hold any data in this state.

2. It receives a promise response from every single acceptor. When this happens, the proposer transitions to the `PSentAccReq` state.

When the proposer reaches the `PSentAccReq`, it means it has received a promise from every single acceptor and it can now start sending accept requests to each of the acceptors in the system. In the `PSentAccReq` the proposer again stores a list `sent_to` to keep track of every single acceptor it has already sent the accept request to. It also stores another `proposal` which is has the same proposal number p that the proposer was initialised with but the value v is the the value from the highest numbered proposal it received in a promise response. In the verification section, we will look at how it determines this value by looping over the `promises` list from the `PWaitPrepResp` state. The sending of accept requests works similar to sending prepare requests in the `PSentPrep` state. Finally, when the proposer finishes sending the accept requests to all the acceptors, it transitions to the `PAbort` state where it stops sending and receiving messages.

Acceptor

The acceptor starts off in the `AInit` state. It does not hold any data in this state as it is not sending any messages. It keeps listening for messages and on receiving a prepare request message, it transitions to `APromised` state.

In the `APromised` state, the acceptor holds a proposal. This is the highest numbered proposal that it has received so far in a prepare request message. In this state, on receiving a

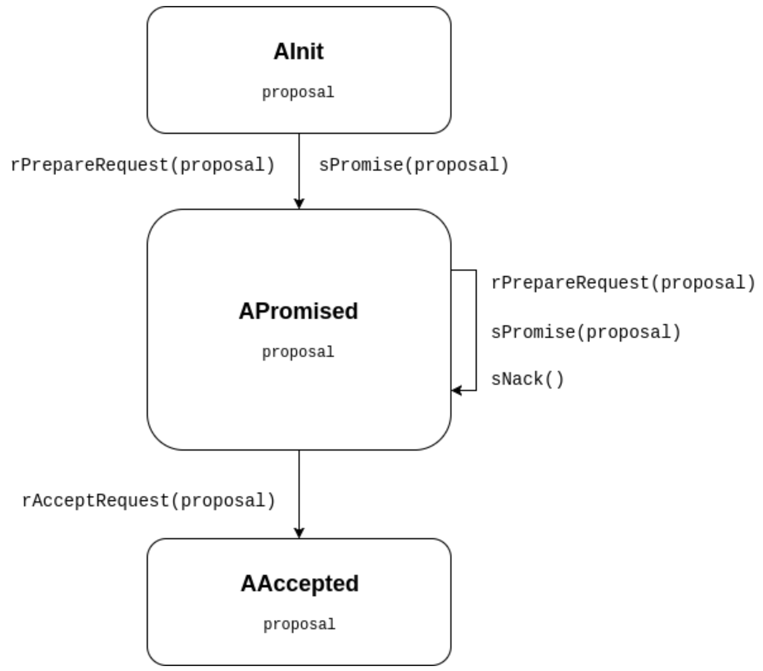


Figure 4.2: Acceptor State Transition Diagram

prepare request, if the proposal number of the proposal in the prepare request is greater than the proposal number of the proposal it currently holds, it updates its current state to hold the new proposal but still remains in the `APromised` state. If the proposal number of the proposal in the prepare request is not greater, the acceptor sends a nack response to the proposer who sent the prepare request and does not update its state.

An acceptor in the `APromised` state, on receiving an accept request, if the value of the proposal number of the proposal in the accept request is greater than the proposal number of the proposal that it currently holds, it transitions to the `AAccepted` state where it now holds the new proposal with the greater proposal number. If the proposal number of the new proposal number is not greater, the acceptor remains in the same state.

In the `AAccepted` state, the acceptor stops listening for and responding to messages. This is similar to the `PAbort` state for the proposer.

4.1.3 Simulator

The next step after designing the adapted protocol was to implemented a simulator for it in Python. The simulator was modelled according to Disel in that, each process has send and receive transitions and that each node follows a state transition system. In order to test the protocol, we only used the state transitions we designed in the adapted protocol. The type of messages that each node sends or receives and the data it holds were all taken from the state transition system.

In the code listing below you can see the main body that an acceptor node runs in the simulator. It first checks the type of message and then performs an action depending on its current state. For example, when it receives a `AcceptRequestMessage`, if it is in the `AInit` state then it accepts the proposal, otherwise if it is in the `APromised` state, it first checks the proposal number of the incoming message (proposal) and accepts the proposal only if the number is greater than the number the acceptor has promised already.

```
class Acceptor(Process):
   ...
   def body(self):
      while True:
         msg = self.get_next_msg()
         p_no, p_val = msg.proposal

         if isinstance(msg, PrepareRequestMessage):
            if self.state[0] == "AInit":
               self.state = ("APromised", msg.proposal)
               self.send_promise_resp(p_no)
            elif self.state[0] == "APromised":
               promised_no, _ = self.state[1]
               if p_val < promised_no:
                  self.send_nack_resp(p_no)
               else:
                  self.state = ("APromised", msg.proposal)
                  self.send_promise_resp(p_no)
            else:
               self.send_nack_resp(p_no)
         elif isinstance(msg, AcceptRequestMessage):
            if self.state[0] == "AInit":
               self.state = ("AAccepted", msg.proposal)
            else:
               promised_no, _ = self.state[1]
               if p_val >= promised_no:
                  self.state = ("AAccepted", msg.proposal)
   ...
```

Running the simulator and watching it achieve consensus proved that the design of the adapted protocol functioned correctly. Implementing the simulator before encoding the protocol in Disel, not only helped solidify the understanding of the adapted protocol but also helped catch errors in the protocol design early on. This is because, in Disel, we would first have to encode the protocol then implement the client application and then see whether the it functions correctly. Additionally, the simulator can also be used to test the designed inductive invariant. One can write a function that queries the state of each node and prints out a boolean indicating when the inductive invariant is preserved or raise an error otherwise.

4.2 Encoding the protocol in Disel

Having designed the adapted protocol and tested it on a simulator, the next step was to actually encode it in Disel. The most important part was to encode the state transitions of the protocol. In the encoding, we define an inductive type, `RoleState` that stores all the states a node can be in and the data it can hold.

```
(* States of the nodes *)
Inductive RoleState :=
(* proposer States *)
(* Initialised with a proposal (p_no * p_val) *)
| PInit of proposal
(* Sent prepare message to some acceptors at a current stage *)
(* seq nid holds nodes which were sent the message *)
| PSentPrep of seq nid & proposal
(* Received promises/NACKs from acceptors *)
| PWaitPrepResp of promises & proposal
(* Send AcceptRequest *)
| PSentAccReq of seq nid & proposal
(* Finished executing after sending AccReq or not receiving majority*)
| PAbort
```

```
(* acceptor states *)
| AInit
(* Holds the highest number promised in the proposal *)
| APromised of proposal
(* Holds the highest number proposal accepted *)
| AAccepted of proposal.
```

The `step` functions were defined next. These functions dictate how the state of a node changes on sending or receiving a message. Below is a code listing from the `step_recv` function which handles the change in state on receiving a message. `s` is the current state of the nodes, `mbody` is the message that is received and `mtag` is the tag of the received message. When the node receives a message in the `APromised` state, it first checks whether it is an prepare request or an accept request. It then checks whether the number of the proposal is greater than the one it has already promised. If it is greater, then the node transitions to `APromised`, but holding a new proposal, or to the `AAccepted` state depending on the type of the incoming message.

```
(* Changes in the Node state triggered upon receive *)
Definition step_recv (s : StateT) (from : nid) (mtag : ttag) (mbody : payload):
 StateT :=
 let: (e, rs) := s in
 let: recv_p_no := head 0 mbody in
 let: recv_p_val := last 0 mbody in
 match rs with
 ...
 | APromised p' ⇒
    let: curr_p_no := head 0 p' in
    let: curr_p_val := last 0 p' in
    if mtag == prepare_req
    then if recv_p_no > curr_p_no (* If received higher number *)
        (* Update promised number by storing new proposal *)
        then (e, APromised mbody)
        else (e, APromised p')
    else (* It's an accept request *)
        if recv_p_no > curr_p_no (* If received higher number *)
        then (e, AAccepted mbody) (* Accept the proposal *)
        else (e, APromised p') (* we'll send nack *)
 ...
```

```
  | _ ⇒ s
  end.
```

The complete encoding of the adapted protocol can be found in the `PaxosProtocol.v` file.

Client Application

As outlined in section 3.3, our approach to using Disel involved first encoding the client application and the protocol, and then going on to write the proofs to show that the client follows the protocol. Implementing the client was important for a couple of reasons. Firstly, it ensures that our adapted protocol is adequate for implementing client applications. Having the client application working before writing the proofs showed that the design of the adapted protocol worked and could be used to achieve consensus. Secondly, the client application also helped us test our encoding of the adapted protocol. As we did not encode the inductive invariant, the working of the client application helped us test the properties of the adapted protocol, i.e. observing nodes achieve consensus using the protocol.

In this chapter we look at the implementation of a simple client application that uses the proof of the adapted protocol to create an application where a set of nodes achieve consensus on a value. The chapter first looks at how the client application was designed and encoded in Disel. After which we look into how Disel's `shims` runtime was used to extract the OCaml code for the runnable client application. Then we outline how we extended the client application by writing a wrapper around it. Finally, we look at how the client application was proved in Disel.

5.1 Modelling

The main property that we want to observe from the client is the acceptors achieving consensus on a proposal. This meant we needed to be able to see that a majority of acceptors accepting a protocol and that all of the acceptors in the majority accept the same proposal.

Furthermore, we needed the client application to follow the same state transition system we designed for our adapted protocol. Thus, the correct functioning of the client will give us confidence that our adapted protocol can be proved. This also enables us to catch flaws in our design of the adapted protocol early on and helps us detect things like unnecessary states early on in the process, which makes proving the protocol easier in the later stages.

So our client was simple in that we wanted to initialise two proposers and three acceptors and then see the acceptors achieve consensus. Each proposal that is initialised will only try once to achieve consensus using the proposal number that it is initialised with. If it receives a nack in the process, then it stops and does not retry to achieve consensus with a higher proposal number. As explained in the previous chapter, we choose this 'one shot' process for the proposer in order to focus just on the part of the protocol where consensus is achieved, i.e. where the proposer accumulates enough promises and then sends out an accept request which then may be accepted by each of the acceptors.

5.2 Encoding client in Disel

Having decided on the design of the client, the next step was to produce a runnable implementation of a proposer and an acceptor which each of the nodes in the protocol can run as programs. Here we will only look at the implementation of the proposer, the implementation of the acceptor follows from that and can be found in the `PaxosAcceptor.v` file.

As a proposer starts off in the `PInit` state, the runnable implementation of a proposer needs to take in a proposal as a parameter which it will use to initialise the proposer with. Below is the main function for the proposer. It first sends out the prepare requests and then starts receiving responses from the acceptors. `check_promises` function checks that none of the responses contain a nack request. If no nacks were received then the proposer sends accept requests to all the acceptors by choosing the value from the highest numbered proposal.

```
Program Definition proposer_round (psal: proposal):
  {(e : nat)}, DHT [p, W]
  (fun i ⇒ loc i = st :→ (e, PInit psal),
   fun res m ⇒ loc m = st :→ (e, PAbort))
  :=
  Do (e <-- read_round;
     send_prepare_req_loop e psal;;
     recv_promises <-- receive_prepare_resp_loop e;
     check <-- check_promises recv_promises;
     if check
```

```
    then send_accept_reqs e (choose_highest_numbered_proposal psal recv_promises)
    else send_accept_reqs e [:: 0; 0]).
   (* If check fails then send an acc_req for (0, 0) which will never be
      accepted by any acceptor *)
```

Although, if a nack was received, the proposal still sends accept requests with the proposal $\langle 0, 0 \rangle$. This proposal will never be accepted by any acceptor as its proposal number is not greater than 0. We still need to send these accept requests as both branches of a `if` statement need to have the same type. The distributed Hoare types and the pre and post conditions defined are also show in the code listing. The proposer starts off in the `PInit` state and on finishing the round, ends up in the `PAbort` state.

5.3 Verfying the Client

We will now briefly look at how the proofs were encoded. Below we have part of the code listing for `receive_prepare_req_loop`. In the implementation you can see withing the `@while` loop that it tries to catch an incoming prepare request and return the message if it arrives correctly.

The pre and post conditions state that before executing this loop, the acceptor starts in the `AInit` state and after the execution, it finishes in the `APromised` state while holding some proposal `psal`.

```
Definition r_prepare_req_cond (res : option proposal) := res == None.

(* Invariant relates the argument and the shape of the state *)
Definition r_prepare_req_inv (e : nat) (pinit: proposal): cont (option proposal) :=
  fun res i ⇒
   if res is Some psal
   then loc i = st :→ (e, APromised psal)
   else loc i = st :→ (e, AInit).

(* Loops until it receives a prepare req *)
Program Definition receive_prepare_req_loop (e : nat):
  DHT [a, W]
  (fun i ⇒ loc i = st :→ (e, AInit),
   fun res m ⇒ ∃ psal, (res = Some psal) ∧
      (loc m = st :→ (e, APromised psal)))
  :=
  Do _ (@while a W _ _ r_prepare_req_cond (r_prepare_req_inv e) _
       (fun _ ⇒ Do _ (
         r <-- tryrecv_prepare_req;
         match r with
         | Some (from, tg, body) ⇒ ret _ _ (Some body)
         | None ⇒ ret _ _ None
         end
       )) None).
...
Next Obligation.
  move ⇒ i1/= E1.
  apply: (gh_ex (g:=([::0; 0]))).
  apply: call_rule ⇒ //r i2 [H1]H2 C2.
  rewrite /r_prepare_req_cond/r_prepare_req_inv in H1 H2.
   by case: r H1 H2 ⇒ //p _; ∃ p.
Qed.
```

The simple obligation we have shown in the code listing is to show that given the pre condition holds, show that the execution of the loop will take us to the post condition. In order to prove it, we take an example of a prepare request with the proposal $\langle 0, 0 \rangle$, i.e. we supply a value for the `psal` in the post condition, and execute the statements in the loop. We then use the definition of our loop invariant `r_prepare_req_inv` which states that as long as the incoming prepare request is a valid proposal, you will accept it and move to the `APromised` state. Thus, trivially completing our proof.

5.4 Extraction

Disel programs can be extracted into their corresponding OCaml definitions. The extracted code contains modules that define the various node states and transitions. This extracted code can then be used by a shim to create a client application.

In order for the extraction to work, a runnable program needs to be supplied for each of the nodes participating in the protocol. Additionally, each node needs to be given an initial state that satisfies all the imposed invariants.

The `SimplePaxosApp.v` file uses the runnable implementations of the proposer and acceptor and defines the code to instantiate the nodes. The client implementation instantiates two proposers and three acceptors. Each proposer is instantiated with a unique proposal which is required in the `PInit` state (the state in which each proposer starts off in).

Extracting this code using Disel produces the OCaml code that can be used to run each of the nodes. After extracting the code, a shim file (`PaxosMain.ml`) is written that parses the arguments supplied to it and instantiates a program specified for the given node. Compiling the shim file produces the executable (`PaxosMain.d.byte`) which can be supplied with the right arguments to set up execution of one of the nodes participating in the protocol.

The executable was used in a shell script (`paxos.sh`) to instantiate all the different nodes as different processes. Below are the logs produced by one of the proposers (node 2) on running this script. Nodes 3, 4 and 5 are acceptors while node 1 is the other proposer proposing value 1.

```
initial state is: [0 |→ {dstate = [1 |→ [1 |→ <heap>], 3 |→ [1 |→ <heap>],
   4 |→ [1 |→ <heap>], 5 |→ [1 |→ <heap>]]; dsoup = <>}]
World is [0 |→ <protocol with label 0>]
sending msg in protocol 0 with tag = 0, contents = [2; 2] to 3
World is [0 |→ <protocol with label 0>]
sending msg in protocol 0 with tag = 0, contents = [2; 2] to 4
World is [0 |→ <protocol with label 0>]
sending msg in protocol 0 with tag = 0, contents = [2; 2] to 5
new connection!
done processing new connection from node 3
got msg in protocol 0 with tag = 1, contents = [0; 0] from 3
new connection!
done processing new connection from node 4
got msg in protocol 0 with tag = 1, contents = [0; 0] from 4
new connection!
done processing new connection from node 5
got msg in protocol 0 with tag = 1, contents = [0; 0] from 5
World is [0 |→ <protocol with label 0>]
sending msg in protocol 0 with tag = 3, contents = [2; 2] to 3
World is [0 |→ <protocol with label 0>]
sending msg in protocol 0 with tag = 3, contents = [2; 2] to 4
World is [0 |→ <protocol with label 0>]
sending msg in protocol 0 with tag = 3, contents = [2; 2] to 5
```

The tags identify the messages types. The mapping for the tags is defined in our protocol file (`PaxosProtocol.v`) as follows.

```
Definition prepare_req : nat := 0.
Definition promise_resp : nat := 1.
Definition nack_resp : nat := 2.
Definition accept_req : nat := 3.
```

Thus, the logs show that the proposer first sent out prepare request (`tag = 1`) with the proposal $\langle 2, 2 \rangle$ to the three acceptors and received promise responses back from them (`tag = 1`). After this proposer sent out an accept request (`tag = 3`) to each of the acceptors.

5.5 Extending the extracted code

The previous steps helped setup a simple client application but the logs only show what happened at each individual node. Moreover, the important part is to find out when the entire system has achieved consensus, i.e. a majority of acceptors have accepted the same proposal.

One way of making the client do this would be to go back to the Disel code and write the implementation of a learner and define an accepted request message that will be sent from an acceptor to a learner whenever it accepts a proposal. This learner can then tell us when the entire system has achieved consensus. This method is time consuming because one will have to do all the encoding similar to that for the proposer or acceptor. Although, the final output of this will be a fully verified client where the actions of the learner also adhere to our protocol.

A quicker way to implement the same functionality is to write a wrapper around the extracted code that enables communication with the acceptor. The wrapper can keep checking when each of the acceptors has accepted a proposal and can compare the values in those proposals. Using this the wrapper can 'announce' when the system has achieved consensus. An example of a wrapper, which uses the extracted code, is implemented in the `paxos.py` file. The code listing below shows the output of running this wrapper. The output shows the values accepted by each acceptor and using this information, the wrapper announce that consensus has been achieved on value 2 as a majority of acceptors have accepted it.

```
got msg in protocol 0 with tag = 3, contents = [2; 2] from 2
Acceptor 1 accepted 2
got msg in protocol 0 with tag = 3, contents = [2; 2] from 2
Acceptor 2 accepted 2
```

```
got msg in protocol 0 with tag = 3, contents = [2; 2] from 2
Acceptor 3 accepted 2
========================================

Consesus achieved on value: 2
```

Writing such wrappers helps to extend the functionality of the extracted client application and also shows that the client application can be used in other applications where it provides a verified implementation of Paxos.

Conclusion and Evaluation

6.1 Summary of Achievements

The project achieved all its goals apart from proving the inductive invariant. We delivered a partial mechanised proof of Paxos in Disel which included a client application that was shown to adhere to the encoded protocol. However, the complete correctness of the protocol itself is missing because we ran out of time to fully mechanise the proof of the inductive invariant that verifies the protocol. Having followed the Disel workflow we can match the deliverables to the stages in the workflow.

1. The Adapted Protocol
   We designed the state transition systems for Paxos in order to help encode it in Disel. The state transition systems were also tested on the Python simulator and then encoded in Disel.

2. Client Application
   We also created a client application that uses the encoded protocol. The code for the client was extracted using Disel and we successfully wrote wrappers around it as well.

3. Proof of the Client and the Protocol
   We also encoded the adapted protocol in Disel and completed the majority of the proofs for it. There are still a few proofs remaining in the client and protocol which we look at in the next section.

6.1.1 Line counts for Proofs

File	Description	Specs	Proofs
PaxosProtocol.v	Defines the state transitions and step functions	278	87
PaxosProposer.v	Implemetation of a Proposer	155	96
PaxosAcceptor.v	Implemetation of an Acceptor	176	137
SimplePaxosApp.v	Initialises the client application with the proposers and acceptors	73	11

6.2 Critical Evaluation of the project

We first look at each of the proof files to evaluate which proofs were finished. Then we look at the things that were missing from a fully complete mechanisation of Paxos and come up with a percentage of how far we got to the goal.

6.2.1 Summary of Proofs

PaxosProtocol.v

1. `RoleState`
   Defines the states that the nodes can be in.
   Fully implemented.

2. `step_send`
   Defines all the node transitions on sending a message.
   Fully implemented.

3. `step_recv`
   Defines all the node transitions on sending a message.
   Fully implemented.

PaxosAcceptor.v

1. `read_state`
   Returns the current state of the node.
   Fully Proven.

2. `receive_prepare_req_loop`
   Receives and handles an incoming proposal from a prepare request.
   4 of 4 obligations fully proven to show that loop invariant holds, and that given the pre condition, we reach the post condition after execution of the loop.

3. `resp_to_prepare_req`
   Takes in the incoming proposal and value of current promised number and sends a promise or a nack response.
   1 of 1 obligation remaining to be proved that involves showing the pre and post conditions hold. The problem is to show that the node from which the prepare request was received is a proposer.

4. `receive_acc_req_loop`
   Handles an incoming accept request and updates the current state accordingly
   4 of 4 obligations fully proven to show that loop invariant, the pre and post conditions hold.

5. `acceptor_round`
   Combines all the loops to give the implementation of an acceptor.
   1 of 1 obligation remaining to be proved that involves proving the pre and post condition hold. The difficulty is in proving the post condition which states that an acceptor can be in either `AInit`, `APromised` or `AAccepted` after the round is finished.

PaxosProposer.v

1. `read_round`
   Returns the current round of the given node.
   Fully Proven.

2. `send_prepare_req_loop`
   Sends prepare requests to all the acceptors.
   1 of 2 obligations remaining to be proved. The proven obligation shows that we can reach the post condition from the pre condition. The remaining obligation involves showing both the pre condition and post condition hold. The problem in proving this obligation is showing that the node sending the prepare request is a proposer.

3. `receive_prepare_resp_loop`
   Handles the incoming promises or nacks and updates the state accordingly.
   1 of 4 obligations remaining. The proved obligations show how to reach the post condition from the pre condition. The remaining obligation deals with showing that both

the pre and post conditions hold. The difficulty with the remaining obligation is to show that the invariant of the loop holds irrespective of whether a promise or a nack was received.

4. `send_accept_req_loop`
   Sends an accept request to the entire set of acceptors.
   1 of 1 obligation remaining to be proved that deals with proving that the pre and post conditions hold. The problem in this is showing that the node sending the prepare requests is a proposer.

5. `proposer_round`
   Combines all the loops to give the implementation of a proposer.
   1 of 1 obligation remaining to be proved that involves showing the pre and post conditions hold. The difficulty was in showing how the proposer reaches the `PAbort` state after finishing execution.

6.2.2 Missing Pieces in the Mechanisation

There were a few things missing for a complete mechanised proof of Paxos. Also, there were also a few places in the project where things were not proved mechanically.

1. Inductive Invariant
   We did not mechanise the proof of the designed inductive invariant in Disel. This will enable us to reach the final stage of the previously mentioned Disel workflow, thus, strengthening the proof by completely verifying the adapted protocol.

2. Learner in Client Application
   We were successfully able to interface with the extracted code for the client application but writing a wrapper around it. The wrapper basically performed the role of the learner in Paxos and announced when the acceptors had achieved consensus. Using the wrapper meant that client application was not fully verified. In order to achieve that, one would have to design and add the state transition system for the learner in the adapted protocol and encode it in Disel.

6.2.3 Conclusion

We were only able to reach the third stage of the Disel workflow by verifying the client application and showing that it follows the adapted protocol and ran out of time to prove the inductive invariant. While this stage does not show that the protocol itself is completely verified, we can still think of it as relatively strong correctness as the protocol is based on the simple Paxos protocol. We have show that the client follows the protocol and if one trusts the protocol, then we have an implementation that has been proven correct with respect to it (modulo the missing obligations discussed above). Additionally, simple Paxos already has a proof of correctness provided by Lamport [8]. The way to prove the complete correctness of the protocol will be to encode the inductive invariant of Paxos in Disel, which we leave to future work.

Furthermore, although the learner in the client application was not verified, writing the wrapper enabled us to show that it was possible to interface with the code extracted from Disel. This showed that the verified code generated by Disel can be used in other larger application as the verified implementation of a protocol.

The designed adapted protocol and state transition system focused on the 'core' protocol because as previously mentioned, the proof of the 'core' protocol can be used to verify the optimisations. The adapted protocol and state transition system were tested on the Python simulator and the client based on it was also verified. Iterating between the proof of the client and design of the protocol, helped solidify the design and also make it minimal by removing unnecessary state transitions.

Thus, having looked at everything we can say we completed about 75% of the mechanisation of Paxos in Disel. This is because the invariant is a major chunk of the proof and there are also a few obligations, as pointed out in section 6.2.1 that still need to be completed.

6.3 Summary of Experience with Mechanised Verification in Disel

Disel was the critical component of this project and for most purposes, it stacked up very well to meet all the requirements. The experience of encoding in Disel felt very natural. This was because focusing on the 'core' logic of the protocol and then representing it as state transition diagram is quite intuitive. Which then makes encoding it as a `step` function in Disel very easy.

Building the client application showed that one can successfully extract the OCaml code for the verified protocol. Moreover, writing wrappers around the code showed that extracted code can be used as a verified library in other larger applications.

There were numerous instances where, while doing the proofs in Disel ended up revealing flaws in the implementation. One of these situations was when we found a mistake in our proposer implementation within `send_accept_reqs`. Initially, to check whether we received a promise from every single acceptor, we had the condition, `map fst' recv_promises == acceptors`. The problem with this was that it required the promises to be received in the same order from the acceptors with which the acceptors were ordered in the `acceptors` set. We cannot impose such ordering on the messages received, so we had to replace the condition with `perm_eq (map fst' recv_promises) acceptors` that correctly checks whether a promise had been received from every acceptor by checking that the set of acceptors who responded (`map fst' recv_promises`) is a valid permutation of the set of acceptors.

Furthermore, we also found a bug in our acceptor implementation. We came across it when we were not able to prove the post condition of `acceptor_round` as the acceptor was never reaching the `AAccepted` state. The reason for this was because we when an acceptor receives an `accept_request` we checked if the incoming proposal had a proposal number greater than the one currently held by the proposal, and only then move to the AAccepted state. The fact that we were reading the state of the acceptor after receiving the incoming message meant that, by then the acceptor had already changed its state accordingly when it received the message, and so the proposal number held by it was always equal to the incoming message's proposal. This meant that the acceptor would not send a promise response.

There were also instances where doing the proofs required the pre or post conditions of the transition to be strengthened by case analysing on the the message being sent or received. One example, in the post condition of `resp_to_prepare_req`, is as follows. Initially, the post condition stated that after responding to a `prepare_req`, the acceptor would go to the `APromised` state either holding the proposal it already had or the new proposal it received. The strengthened post condition looks at the received message and checks if the incoming proposal has proposal number greater that the one it has already promised, only then does it transition to the `APromised` state holding the new incoming proposal.

```
(* Initial weaker post condition *)
fun (_ : seq nat) m ⇒ loc m = st :→ (e, APromised p_current)
   ∨loc m = st :→ (e, APromised p_incoming))

(* Final stronger post condition *)
fun (_ : seq nat) m ⇒
```

```
    if head 0 p_incoming > head 0 p_current
    then loc m = st :→ (e, APromised p_incoming)
    else loc m = st :→ (e, APromised p_current))
```

Catching these bugs shows how useful it is to do the proofs in Disel as they would not have been easily noticed otherwise. There obviously is the drawback that implementation time is increased as one needs to prove every single transition. Furthermore, when the inductive invariant is added, all the transitions need to be show to adhere to it, thus, requiring more time to implement the proofs.

Furthermore, the current alternatives to Disel, as pointed out in Section 2.4, do not allow for proving invariants separately from verifying the code for compliance with a protocol. Using Disel enabled us to implement the client application and prove that it follows the encoded protocol, even without having to mechanise the proof of the inductive invariant.

6.4 Future Work

First steps would definitely be to finish mechanising the remaining things pointed out in section 6.2. This will help us reach the final stage in the Disel workflow. After that, it would be possible to try proving the optimisations in Paxos based on the proof of the 'core' protocol. It would also then be possible to use 'Hooks' [11] in Disel to compose the proof with proofs of other verified components.

There are also a few areas for future work in Disel. Finding the inductive invariant for the protocol is a much harder task than designing the state transition system. The Disel paper [11] mentioned the prospect of combining Disel with Ivy [15], which would offer assistance in finding the inductive invariant. Another thing, also mentioned in the Disel paper, is the absence of proving the liveness properties in Disel. Hence, the proof of Paxos which we implemented also does not account for liveness whereas tools like IronFleet [13] enable one to also prove liveness. Finally, another aspect of Paxos which we did not look at was its fault tolerance properties. There could be ways to encode it in as a part of the protocol but maybe it might also be possible for Disel to provide an abstraction which can be used to prove the fault tolerance properties of any arbitary protocol.

Bibliography

[1] "Fault tolerance in a high volume, distributed system," https://medium.com/netflix-techblog/fault-tolerance-in-a-high-volume-distributed-system-91ab4faae74a, [Online; accessed 04-April-2018].

[2] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[3] L. Lamport, "The implementation of reliable distributed multiprocess systems," *Computer Networks*, vol. 2, pp. 95–114, 1978.

[4] G. Pirlea and I. Sergey, "Mechanising blockchain consensus," *Proceedings of 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'18)*, 2018.

[5] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," *Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference*, 2014.

[6] D. Mazieres, "The stellar consensus protocol: A federated model for internet-level consensus," https://www.stellar.org/papers/stellar-consensus-protocol.pdf, 2016, [Online; accessed 04-April-2018].

[7] R. V. Renesse and D. Altinbuken, "Paxos made moderately complex," *ACM Computing Surveys*, vol. 47, no. 3, 2015.

[8] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, pp. 133–169, 1998.

[9] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," *OSDI '06 Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 335–350, 2007.

[10] M. Castro and B. Liskov, "Practical byzantine fault tolerance," *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.

[11] I. Sergey, J. R. Wilcox, and Z. Tatlock, "Programming and proving with distributed protocols," *ACM on Programming Languages*, vol. 2, no. POPL, 2018.

[12] A. Chlipala, "Formal reasoning about programs," http://adam.chlipala.net/frap/frap_book.pdf, [Online; accessed 04-April-2018].

[13] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, "Ironfleet: Proving practical distributed systems correct," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, 2018.

[14] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson, "Verdi: a framework for implementing and formally verifying distributed systems," *PLDI. ACM*, pp. 357–368, 2015.

[15] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, "Ivy: safety verification by interactive generalization," *PLDI. ACM*, pp. 614–630, 2016.

[16] C. Dragoi, T. A. Henzinger, and D. Zufferey, "Psync: a partially synchronous language for fault-tolerant distributed algorithms," *POPL. ACM*, pp. 400–415, 2016.

[17] V. Rahli, D. Guaspari, M. Bickford, and R. L. Constable, "Formal specification, verification, and implementation of fault-tolerant systems using eventml," *AVOCS. EASST*, pp. 400–415, 2015.

[18] A. Garcıa-Perez, A. Gotsman, Y. Meshman, and I. Sergey, "Paxos consensus, deconstructed and abstracted," http://ilyasergey.net/papers/paxos-deconstructed-esop18-extended.pdf, 2018, [Online; accessed 03-April-2018].

Project Plan

A.1 Aims

To learn about the family of Paxos-like consensus protocols and then to formulate a proof of their correctness by implementing a library of reusable verified distributed components using Disel. Disel is a framework for compositional verification of distributed protocols, built on top of Coq proof assistant, to verify correctness of the implemented components. Hence, this project will help me learn about the workings of distributed protocols and how to reason about their correctness.

A.2 Objectives

1. Read about and understand the classical Paxos-like consensus algorithms. Develop state transition systems for the algorithms and identify the invariants that need to be preserved during the operation of the algorithm.
2. Implement a simulation of the protocols in Python.
3. Formulate the implemented protocols in Disel by using the developed state-transition systems.
4. Mechanise the proofs of the identified protocol invariants in Disel/Coq.
5. Add additional communication channels and prove composite invariants.
6. Provide an abstract specification of the protocol, usable by third-party clients.
7. Mechanise a client application of the protocol verified out of the abstract interface.

A.3 Expected Deliverables

- Descriptions of the state-transition system and invariants of the Paxos-like Consensus protocols.
- Python implementation of simulations of the Paxos-like Consensus protocols.
- Proofs of correctness of the implemented protocols in Disel. Thereby, providing a library of reusable verified distributed components written in Disel/Coq.

A.4 Work Plan

- Project start to end October (4 weeks).
  - Read about and understand the workings of Paxos-like Consensus protocols.
  - Implement the protocols in Python.
- November start to mid-December (6 weeks).
  - Create state transition diagrams and identify invariants of the protocols.
  - Formulate the protocols in Disel.
  - Prove correctness of the identified protocol invariants in Disel/Coq.
- Mid-December (8 weeks) to mid February.
  - Iterate on the Disel proof to make it more concise.
  - Formulate the abstract interface.
  - Verify a simple client application.
- January start to March end (12 weeks).
  - Submit Interim Report (due in late January).
  - Work on Final Report.

Interim Report

B.1 Current Status

Having studied the Paxos protocol in detail, we adapted the protocol for implementing it in Disel. We decided to focus on single decree paxos and to do away with the learner for the first version of the proof in order to prove the part of the protocol where consensus is achieved.

We developed the state transition system for the nodes in the protocol. In Paxos each node can have different roles but we had to split up each role into different states depending on the current data held by the node and the current function of the node in the protocol. We decided on the states each node could be in and how and when it transitions between them. This helped us come up with precondition and postcondition for the state of each node when it transitions on receiving or sending a message. We tried to minimise the number of transitions and the data held in each node's state in order to simplify the proof in Disel.

We also had to come up with an inductive invariant for the protocol such that if the inductive invariant holds in some state then in holds in every state reachable from that state. The inductive invariant was critical as it helped ensure that the protocol functions correctly by imposing requirements on the global state of the system. For proving the correctness of paxos we found that our invariant had to capture when consensus is achieved on a value and also that once consensus is achieved on a particular value, further rounds of the protocol don't change this value. We then also came up with a proof for how this inductive invariant holds in our adapted protocol.

I have also implemented a simulator for Paxos in Python which is modeled according to how Disel works. The simulator is based on a state-transition system like Disel and uses separate processes to simulate different nodes in the distributed system. In the simulator, I implemented our adapted Paxos algorithm, with the state transitions we had decided to use with

Disel. The working of the simulator gave us confidence that our state transition system for Paxos will work correctly in Disel.

After studying the Disel paper and looking at similar examples, I implemented the core of the adapted protocol in Disel. I also implemented a client application in Disel. The pre and post conditions from the state transition system helped me to implement the client application in such a way to adhere with the main protocol. Using the extraction feature in Disel and the shims runtime, I successfully extracted a working program of the client application in OCaml.

B.2 Remaining Work

Although, I have implemented the protocol in Disel, I still need to prove that the 'coherence' (the constraints imposed on the local state of each node and the messages exchanged) holds in the protocol. I will need to learn Coq and SSReflect in more detail to be able to finish the proofs and to understand the proofs of other protocols in Disel.

For the client implementation, I still need to prove how the implemented node roles satisfy the pre and post conditions, imposed on the state of the node, by the protocol.

I also need to mechanise the proof of our inductive invariant in Disel and prove how the pre and post conditions of each node hold with respect to the inductive invariant when the node undergoes a state transition on receiving or sending a message.

I aim to finish mechanising the proofs by the end of term (23 March) and also to get most of the project report finished by them, especially the initial sections.

Disel Proof Code Listing

The complete source code can be found at https://github.com/anirudhpillai/disel. The encoding for Paxos is in `disel/Examples/Paxos`. Here we are only showing the code for the wrapper around the client application and the main shim file for it.

C.1 scripts/paxos.py

Python wrapper using the extracted executable (`PaxosMain.d.byte`) created using Disel.

```
#!/usr/bin/env python3

import subprocess
import re
import time


proposer1 = "(./PaxosMain.d.byte -me 1 -mode proposer 1 127.0.0.1 8000 2 127.0.0.1 8001 3 127.0.0.1 8002 4
       127.0.0.1 8003 5 127.0.0.1 8004 &) > proposer1.log 2>&1"
proposer2 = "(./PaxosMain.d.byte -me 2 -mode proposer 1 127.0.0.1 8000 2 127.0.0.1 8001 3 127.0.0.1 8002 4
       127.0.0.1 8003 5 127.0.0.1 8004 &) > proposer2.log 2>&1"
acceptor1 = "(./PaxosMain.d.byte -me 3 -mode acceptor 1 127.0.0.1 8000 2 127.0.0.1 8001 3 127.0.0.1 8002 4
       127.0.0.1 8003 5 127.0.0.1 8004 &) > acceptor1.log 2>&1"
acceptor2 = "(./PaxosMain.d.byte -me 4 -mode acceptor 1 127.0.0.1 8000 2 127.0.0.1 8001 3 127.0.0.1 8002 4
       127.0.0.1 8003 5 127.0.0.1 8004 &) > acceptor2.log 2>&1"
acceptor3 = "(./PaxosMain.d.byte -me 5 -mode acceptor 1 127.0.0.1 8000 2 127.0.0.1 8001 3 127.0.0.1 8002 4
       127.0.0.1 8003 5 127.0.0.1 8004 &) > acceptor3.log 2>&1"

a1 = subprocess.Popen(acceptor1, shell=True)
a2 = subprocess.Popen(acceptor2, shell=True)
a3 = subprocess.Popen(acceptor3, shell=True)
p1 = subprocess.Popen(proposer1, shell=True)
p2 = subprocess.Popen(proposer2, shell=True)

consensus_achieved = True
consensus_value = None

exit_codes = [p.wait() for p in [a1, a2, a3]]
print(exit_codes)

# wait for processes to write to files
time.sleep(2)

for i in range(1, 4):
    line = subprocess.check_output(['tail', '-1', "acceptor{0}.log".format(i)])
    line = line.decode("utf-8")
    print(line.rstrip())

    pattern = r"got msg in protocol (\d) with tag = (\d), contents = \[(\d); (\d)\]"
    match = re.match(pattern, line)

    if not match or (consensus_value and consensus_value != match.group(4)):
        print("Error in acceptor %d" % i)
        consensus_achieved = False
        break
    else:
        print("Acceptor %d accepted %s" % (i, match.group(4)))
        consensus_value = match.group(4)

print("=" * 40)

if consensus_achieved:
    print("\nConsesus achieved on value: %s" % consensus_value)
else:
    print("\nConsesus not achieved")
```

C.2 shims/PaxosMain.ml

```
open Datatypes

open Util
open Shim
open Unix

type mode = Proposer | Acceptor

let mode : mode option ref = ref None
let server_name : Datatypes.nat option ref = ref None
let me : Datatypes.nat option ref = ref None
let nodes : (Datatypes.nat * (string * int)) list ref = ref []

let usage msg =
  print_endline msg;
  Printf.printf "%s usage:\n" Sys.argv.(0);
  Printf.printf " %s [OPTIONS] <CLUSTER>\n" (Array.get Sys.argv 0);
  print_endline "where:";
  print_endline " CLUSTER is a list of triples of ID IP_ADDR PORT,";
  print_endline " giving all the nodes in the system";
  print_newline ();
  print_endline "Options are as follows:";
  print_endline " -me <NAME> the identity of this node (required)";
  print_endline " -mode <MODE> whether this node is the proposer or acceptor (required)";
  print_endline " -proposer <NAME> the identity of the proposer (required if mode=client)";
  exit 1


let rec parse_args = function
  | [] → ()
  | "-mode" :: "acceptor" :: args →
     begin
       mode := Some Acceptor;
       parse_args args
     end
  | "-mode" :: "proposer" :: args →
     begin
       mode := Some Proposer;
       parse_args args
     end
  | "-me" :: name :: args →
    begin
      me := Some (nat_of_string name);
      parse_args args
    end
  | "-proposer" :: name :: args →
    begin
      server_name := Some (nat_of_string name);
      parse_args args
    end
  | name :: ip :: port :: args → begin
     nodes := (nat_of_string name, (ip, int_of_string port)) :: !nodes;
     parse_args args
   end
  | arg :: args →
    usage ("Unknown argument " ^ arg)


let main () =
  parse_args (List.tl (Array.to_list Sys.argv));
  match !mode, !me with
  | Some mode, Some me →
    begin
      Shim.setup { nodes = !nodes; me = me; st = SimplePaxosApp.init_state };
      match mode with
      | Acceptor →
        begin match int_of_nat me with
        | 3 →
          begin
            try
              SimplePaxosApp.a_runner1 ()
            with _ → print_endline "acceptor 1 exiting."
          end
        | 4 →
          begin
            try
              SimplePaxosApp.a_runner2 ()
            with _ → print_endline "acceptor 2 exiting."
          end
        | 5 →
          begin
            try
              SimplePaxosApp.a_runner3 ()
            with _ → print_endline "acceptor 3 exiting."
          end
        | n → usage ("unknown acceptor name " ^ string_of_int n)
        end
      | Proposer →
        begin match int_of_nat me with
        | 1 →
          begin
            try
              SimplePaxosApp.p_runner1 ()
            with _ → print_endline "A acceptor closed its connection, proposer 1 exiting."
          end
        | 2 →
          begin
            try
              SimplePaxosApp.p_runner2 ()
            with _ → print_endline "A acceptor closed its connection, proposer 2 exiting."
          end
        | n → usage ("unknown proposer name " ^ string_of_int n)
        end
    end
  | _, _ → usage "-mode and -me must be given"

let _ = main ()
```

Simulator Code Listing

The complete source code for the simulator can be found at https://github.com/anirudhpillai/paxos.

D.1 paxos.py

```python
#!/usr/bin/env python3

import os
import signal
import sys
import time

from paxos.proposer import Proposer
from paxos.acceptor import Acceptor

NACCEPTORS = 3
NPROPOSERS = 2


class Env:
    """
    Sets up the environment and runs the protocol.
    Initialises each of the nodes in the protocol.

    Attributes:
        :procs holds all the executing processes
    """

    def __init__(self):
        self.procs = {}

    def send_msg(self, dst, msg):
        if dst in self.procs:
            self.procs[dst].deliver(msg)

    def add_proc(self, proc):
        self.procs[proc.id] = proc
        proc.start()

    def remove_proc(self, pid):
        del self.procs[pid]

    def run(self):
        proposers = range(1, NPROPOSERS + 1)
        acceptors = range(NPROPOSERS, NPROPOSERS + NACCEPTORS)

        for i in acceptors:
            pid = "Acceptor %d" % i
            Acceptor(self, i)

        for i in proposers:
            pid = "Proposer %d" % i
            Proposer(self, acceptors, i, i)

    def terminate_handler(self, signal, frame):
        self._graceful_exit()

    def _graceful_exit(self, exit_code=0):
        sys.stdout.flush()
        sys.stderr.flush()
        os._exit(exit_code)


def main():
    e = Env()
    e.run()
    signal.signal(signal.SIGINT, e.terminate_handler)
    signal.signal(signal.SIGTERM, e.terminate_handler)
    signal.pause()


main()
```

D.2 paxos/process.py

```python
import multiprocessing
import threading


class Process(threading.Thread):
    """
    Attributes:
        :state current state of the Acceptor
        :id id of the Acceptor
        :env environment this Acceptor is running in
    """

    def __init__(self, env, id):
        super(Process, self).__init__()
        self.inbox = multiprocessing.Manager().Queue()
        self.env = env
        self.id = id

    def run(self):
        try:
            self.body()
            self.env.remove_proc(self.id)
        except EOFError:
            print("Exiting...")

    def get_next_msg(self):
        return self.inbox.get()

    def send_msg(self, dst, msg):
        self.env.send_msg(dst, msg)

    def deliver(self, msg):
        self.inbox.put(msg)
```

D.3 paxos/message.py

```python
"""
This file defines the Message super class and all the other
types of messages used in the adapted Simple Paxos protocol.
"""


class Message:
    """
    Message super class which all the differnt message types inherit.

    Attributes:
        :src sender of the message
        :proposal proposal contained in the message
    """

    def __init__(self, src, proposal):
        self.src = src
        self.proposal = proposal

    def __str__(self):
        return str(self.__dict__)


class PrepareRequestMessage(Message):
    def __init__(self, src, proposal):
        Message.__init__(self, src, proposal)


class AcceptRequestMessage(Message):
    def __init__(self, src, proposal):
        Message.__init__(self, src, proposal)


class PromiseResponseMessage(Message):
    def __init__(self, src, proposal):
        Message.__init__(self, src, proposal)


class NackResponseMessage(Message):
    def __init__(self, src):
        Message.__init__(self, src, (-1, -1))
```

D.4 paxos/proposer.py

```python
from .process import Process
from .message import AcceptRequestMessage, PrepareRequestMessage, \
   PromiseResponseMessage, NackResponseMessage


class Proposer(Process):
   """
   Implementation of the Proposer in the adapted Simple Paxos protocol.

   Attributes:
      :state current state of the Proposer
      :acceptors set of acceptors in the current environment
      :env environment this Proposer is running in
   """

   def __init__(self, env, acceptors, p_no, p_val):
      Process.__init__(self, env, p_no)
      self.state = ("PInit", p_no, p_val)
      self.acceptors = acceptors
      self.env = env
      self.env.add_proc(self)

   def send_prepare_req(self):
      _, p_no, p_val = self.state
      for acceptor in self.acceptors:
         self.send_msg(
            acceptor,
            PrepareRequestMessage(p_no, (p_no, p_val))
         )

   def send_accept_req(self):
      _, p_no, p_val = self.state
      for acceptor in self.acceptors:
         self.send_msg(
            acceptor,
            AcceptRequestMessage(p_no, (p_no, p_val))
         )

   def body(self):
      _, p_no, p_val = self.state

      self.send_prepare_req()
      self.state = ("PWaitPrepResp", [], p_no, p_val)

      while True:
         msg = self.get_next_msg()
         if isinstance(msg, PromiseResponseMessage):
            if self.state[0] == "PWaitPrepResp":
               src = msg.src
               recv_p_no, recv_p_val = msg.proposal
               recv_promises = self.state[1]

               if src not in map(lambda x: x[0], recv_promises):
                  recv_promises.append((src, recv_p_no, recv_p_val))
                  if sorted(
                     map(lambda x: x[0], recv_promises)
                  ) == sorted(self.acceptors):
                     recv_promises.sort(key=lambda x: x[1])
                     highest_numbered_value = recv_promises[0][2]
                     self.state = (
                        "PSentAccReq", p_no, highest_numbered_value
                     )
                     self.send_accept_req()
                     print("Exiting proposer", p_no)
                     break
                  else:
                     self.state = (
                        "PWaitPrepResp", recv_promises, p_no, p_val
                     )
         elif isinstance(msg, NackResponseMessage):
            print("Exiting proposer", p_no)
            break
```

D.5 paxos/acceptor.py

```python
from .process import Process
from .message import AcceptRequestMessage, PrepareRequestMessage, \
   PromiseResponseMessage, NackResponseMessage


class Acceptor(Process):
   """
   Implementation of the Acceptor in the adapted Simple Paxos protocol.

   Attributes:
      :state current state of the Acceptor
      :id id of the Acceptor
      :env environment this Acceptor is running in
   """

   def __init__(self, env, id):
      Process.__init__(self, env, id)
      self.state = ("AInit",)
      self.env = env
      self.env.add_proc(self)
      self.id = id

   def send_promise_resp(self, to):
      p_no, p_val = self.state[1]
      self.send_msg(to, PromiseResponseMessage(self.id, (p_no, p_val)))

   def send_nack_resp(self, to):
      self.send_msg(to, NackResponseMessage(self.id))

   def body(self):
      while True:
         msg = self.get_next_msg()
         p_no, p_val = msg.proposal

         if isinstance(msg, PrepareRequestMessage):
            if self.state[0] == "AInit":
               self.state = ("APromised", msg.proposal)
               self.send_promise_resp(p_no)
            elif self.state[0] == "APromised":
               promised_no, _ = self.state[1]
               if p_val < promised_no:
                  self.send_nack_resp(p_no)
               else:
                  self.state = ("APromised", msg.proposal)
                  self.send_promise_resp(p_no)
            else:
               self.send_nack_resp(p_no)
         elif isinstance(msg, AcceptRequestMessage):
            if self.state[0] == "AInit":
               self.state = ("AAccepted", msg.proposal)
            else:
               promised_no, _ = self.state[1]
               if p_val >= promised_no:
                  self.state = ("AAccepted", msg.proposal)
                  print("Node %d has state %s" % (self.id, self.state))
```