Mechanised Verification of Paxos-like Consensus Protocols

Anirudh Pillai

April 4, 2018

Thesis advisor: Professor Dr. Ilya Sergey Anirudh Pillai

Mechanised Verification of Paxos-like Consensus Protocols

Abstract

Distributed systems are notoriously hard to reason about. This project aims to implement a library of reusable verified distributed components, based on the classical family of fault-tolerant asynchronous Paxos-like consensus protocol. The project will use Disel, a framework for compositional verification of distributed protocols built on top of the Coq proof assistant, to verify correctness of the implemented components.

Acknowledgments

Acknowledgements

Contents

1
Introduction

1.1 The Problem

Distributed systems are ubiquitous in the morder world yet they designing correctly functioning 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 implemention of the 'smaller' sub-components but we there still remains the problem of checking whether these smaller components, when combined together, achieve the desired goal.

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 verfied components function correctly when combined together.

Another fundamental problem in the field of distributed computing is to make a set of process achieve consensus on something, for examples, the occurence 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.

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 distrubuted programs and the algorithms running on top of them. Working on verifying a popular distributed protocol, will teach me about all 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 the Disel section of chapter 2. The main stages in the project were as follows:

- First stage was to read about and understand the various concepts related to distibuted 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 and inductive invariant 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 designed functioned correctly.
- The final stage was actually mechanising the proofs in Disel. 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 safety invariants to detect whether it is functioning correctly. 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 and the experience of using the Disel framework and look at areas for future development.

2

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 running 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 distibuted system is also alternatively referred to as a processor or a node in the system. Each node in a distributed systems 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 arbitary period of time.

A protocol running in an asynchronous enviroment 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 eg. not responding to messages, 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'. With their Netflix API receiving more than 1 billion requests a day, they expect that it is guaranteed that some of the components of their distributed system will fail. Using a fault tolerant distributed system they are able to ensure that a small failure in some components 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 components 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 independantly, 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. Fred B.[15] points out how to use state machine replication to implement fault tolerant services.

A State Machine M can be defined as $M = \langle q_0, Q, I, O, \delta, \gamma \rangle$ 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[7] 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 Byzantine Fault Tolerance

Byzantine fault tolerance is the most general form of fault tolerance in which any arbitary form of failure should be defended against. This failure might not just be a node crashing or but could also be some node producing inconsistent output or having corrupted state. A node with a Byzantine fault might also behave 'maliciously', for example, by sending different responses for the same question to different nodes, or it might present itself as failed to some nodes and as functioning to others. It might do this to prevent the distributed algorithm from functioning correctly.

The term comes from the 'Byzantine Generals Problem' by Laport et al[9] in which a set of nodes must decide on a course of action, but at the same time some of the nodes are 'malfunctioning' and give 'conflicting information to different parts of the system'.

2.1.5 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 distibuted 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 everytime 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 the processor that was handling your 'add' request. Later when you want to retrieve the file from the database you might be served by a different computer that did not perform the 'add' request. Inorder for the new computer to know that the file exists in the database, you will need to use a consensus protocol which helps all the computers in the network (which handle user requests) agree upon the result that the file has been added to the database.

A popular consensus protocol is the blockchain consensus protocol which powers Bitcoin. George P. and Ilya S.[13] have verified a subset of this protocol in Coq. Other examples of consensus protocols are Raft[12], Stellar Consensus Protocol[10] and Paxos[14], which we will now look at.

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 unrealiable 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[16] and the Chubby[12] 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 each other.
- 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 buy not simple paxos. (This can be linked to the 'Practical Byzantine Fault Tolerance' paper, by Castro et al[2], 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 upto 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 function 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 calculating when a

majority of acceptors have accepted the same proposal. Once the acceptors agree on a value, the learner may act on the value eg. Send request to client informing 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.

A `quorum` of acceptors is a subset of the set of all acceptors with 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 has occurred, where a majority of acceptors have accepted 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`. As it the proposer tries 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 $n' < n$.

  Thus, if an incoming prepare request has proposal number less 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 sent any promise response before, then the body of the promise response can be empty, otherwise the acceptor must include the last proposal that it promised (before the current one) in the body of the message.

- **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.

  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 $\langle n, v \rangle$ 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 $n' > n$.

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. The learner 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 choose 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.

2.2.3 Inductive Invariant

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.

The problem with using just an invariant like $x > 2$ is that, you may assume that the invariant holds before the execution of the program, but you still do not have a guarantee that this invariant will hold during and after the execution of the program, i.e. 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.

2.3 Disel

Disel[5] 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 which supplies the implementation of the various primite 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.

'Each global system state s is a finite partial mapping from protocol labels $l \in$ `Lab` to statelets. Each statelet represents a protocol-specific component, consisting of a 'message soup' MS and a per-node local state (`DistLocState`).'

As you can see from the image, a protocol P in Disel is defined as a tuple of the Coherence, the set of send transitions and the set of recieve transitions. Let us now look at each of these components in detail.

A `Coherence` is 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 of 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 reciever 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/reciever 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.

```
(* 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 received 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 reponses to a `prepare_req`. 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`.

```
(* 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))) _).
```

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 futher 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)

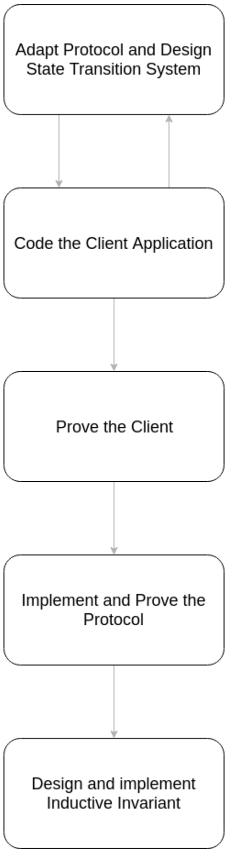
Adapt Protocol and Design State Transition System
Code the Client Application
Prove the Client
Implement and Prove the Protocol
Design and implement Inductive Invariant

Figure 2.1: Disel Workflow

2.3.3 Approach to Mechanising Proofs in Disel

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 'conveniece' parts. For Paxos, this was 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.

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 designed a client where nodes try to acheive consensus on one of the proposals that the proposer is initialised with. This enabled us to see in action the stages of the protocol that lead to consensus being achieved.

While using Disel, we often had 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. 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. Prove the client

The next stage involves 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 fullfill 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.

From this stage onwards, as we move to stages 4 and 5 we end up strengthening the proof of our protocol. Finishing stage 4 completes the proof of the protocol while reaching stage 5 actually adds an inductive invariant that stregthens the proof of the protocol even further.

4. Encode the adapted protocol and the inductive invariant

Having proved that client follows the protocol, the next stage is to actually finish the implementation of the protocol and to prove it. Finishing stage 3, helped us be sure that the state transitions we have are enough for realising the 'core' part of the protocol and that action of the client is proved. Now we need to prove the protocol itself.

Finishing this stage meant that we have finished the proof of the protocol, although we can only be sure that ...

In the proof of Paxos, we were only able to finish still stage 4. We designed the inductive invariant but ran out of time to prove it.

2.4 Related Work

People have attempted to create software for verfying the correctness of distributed systems. IronFleet[4], 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[6] 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[11] is another tool which makes it much easier to find and prove inductive invariants for the system. Other similar tools are PSync[3] and EventML[17].

3 Requirements and Analysis

3.1 Detailed Problem Statement

This project aims to implement a library of reusable verified distributed components, based on the classical family of fault-tolerant asynchronous Paxos-like consensus protocol. The project will use Disel, a framework for compositional verification of distributed protocols built on top of the Coq proof assistant, to verify correctness of the implemented components.

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.

3.3 Analysis

4
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, state transitions and inductive invariant 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 2 by first writing and testing the code for the protocol before writing the proofs. 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 prove in Disel by actually 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[1] 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 'Learning the Chosen Value' 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.

Doing away with the learning phase also meant that we did not have to implement a learner in our protocol as the role of the learner (detecting that consensus has been achieved) is performed by the inductive invariant. The learner is useful in actual paxos implementations as it can detect when consensus is achieved and can the inform the client, 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 optimisation from the protocol. Optimisations are aspects of the protocol that improve its running time or resource consumption but improving upon the 'mundane' way of doing the same thing. Removing optimisations allowed us to simplify our proofs in Disel.

Firstly, we decided that a proposer, if it fails while trying to achieve consensus with a proposal number (i.e. it receives a nack), it then does not retry 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.

Additionally we removed the minor optimisation of choosing a majority for sending requests from the proposer. The proposer instead of choosing a quorum of acceptors for sending a message, chooses the entire set of acceptors. This does not alter the protocol as the entire set of acceptors is a valid quorum of acceptors.

This adapated and simplified protocol allows us to focus on verifying the core logic (the part dealing with achieving consensus) of paxos. The optimisations and 'conveniece' 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 intial 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 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 identi-

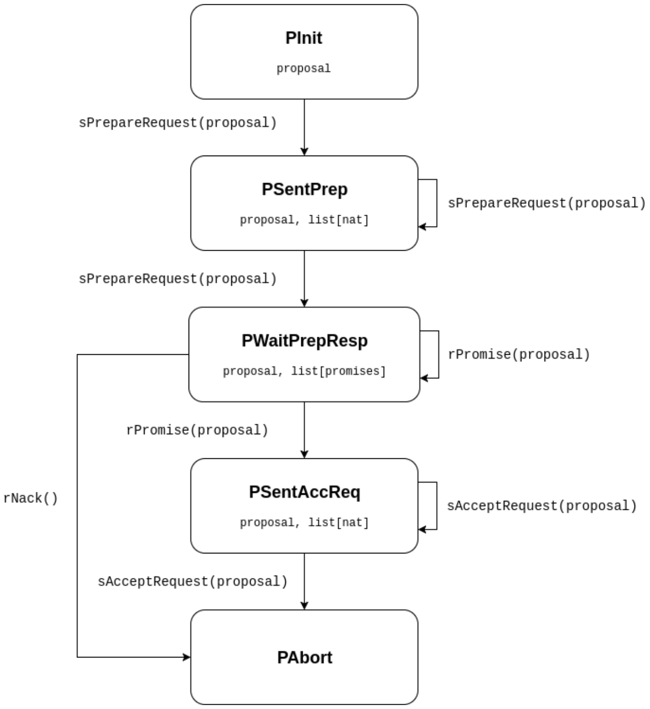

Figure 4.1: Proposer State Transition Diagram

fiers 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 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 states 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 leads to the proposer to 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.

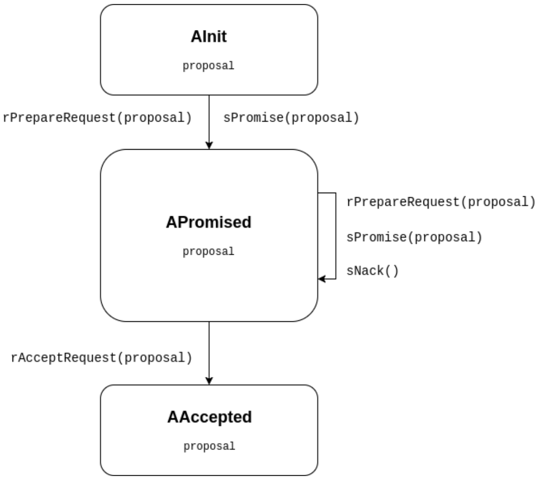

Figure 4.2: Acceptor State Transition Diagram

Acceptor

The Acceptor starts off in the `AInit` state. It doesn't 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 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.

In the `APromised`, on receiving an accept request, if and only if the value of the proposal number 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 Inductive Invariant

A critical part of proving our protocol in Disel was desiging an inductive invariant for our adapted protocol. The inductive invariant helps ensure the correctness of our adapted protocol enabling us to 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.

Inductive invariant means a property which when it holds for a state s, it will hold for any state s' reachable from s.

The crux of Paxos' correctness lies in the prepare phase where, before sending the accept request, the Proposer must first set the value of the proposal, that it wants to propose, to be the value of the highest numbered proposal it receives as a promise. This ensures that when consensus has been achieved on a value 'v', further rounds of the protocol also ensure that consensus will only be achieved on 'v'.

We established two invariants **I1** and **I2** which together form an inductive invariant for our protocol that also proves its safety.

- **I1** simply tries to say that there can only be one unique value associated with a particular proposal number for any proposal that has been accepted.
- **I2** states that once consensus has been achieved on a value v, every higher number proposal accepted by an acceptor also has the value v.

The mathematical representations for the invariants is given by.

- **I1** - $\forall a_i, a_j \in A, \langle p_i, v_i \rangle \in a_i.accepted \langle p_j, v_j \rangle \in toa_j.accepted \rightarrow v_i = v_j.$
- **I2** - $\forall \langle p_i, v_i \rangle, \forall a_j \in A, \exists \langle p_j, v_j \rangle \in a_j.accepted, p_j > p_i \rightarrow v_i = v_j$

I1 is preserved because if there are n proposers, they are initialised with a unique proposal numbers and throughout the running of our adapted protocol, the proposer always uses this unique proposal number for any value that it proposes. Hence, two different proposers never propose a proposal with the same proposal number. Additionally, each proposer only sends one round of accept requests with the same proposal. So as each proposer proposes only one value with a unique proposal number, we can deduce that each accepted proposal will have a unique value associated with a particular proposal number.

Once consensus has been achieved on a value, further runs of the algorithm don't change the value on which consensus has been achieved. We need to show that once consensus has been achieved on a proposal with value v then every other proposal, with a higher proposal number, on which consensus is achieved will also have proposal value set to v.

In order for consensus to be achieved on a new proposal, the new proposal first needs to be accepted by an acceptor.

$\Rightarrow$ If consensus has been achieved on a proposal $\langle p_1, v_1 \rangle$ then every other proposal $\langle p_2, v_2 \rangle$ accepted by any acceptor, where $p_2 > p_1$, has $v_2 = v_1$.

Further, acceptors can only accept a proposal which has been proposed by a proposer. So we can reduce the requirement as follows.

$\Rightarrow$ If consensus has been achieved on a proposal $\langle p_1, v_1 \rangle$ then every accept request $\langle p_2, v_2 \rangle$ sent by the proposer with $p_2 > p_1$, has $v_2 = v_1$.

In order to prove the above, let's assume that consensus has been achieved on a proposal$\langle p_1, v_2 \rangle$.

(4.1)

After that lets say that the systems achieves consensus on $\langle p_2, v_2 \rangle$ where $p_2 > p_1$ and there does not exist p_x such that consensus has been achieved on a proposal with proposal number p_x where $p_1 < p_x < p_2$.

So from our assumption (4.1), there must be a majority of acceptors such that they have accepted the proposal $\langle p_2, v_2 \rangle$. So we need to show that in $v_2 = v_1$. This is ensured in Paxos because of Phase 1 where the proposer must first get promises from a majority.

So any majority the proposer for p_2, v_2 gets in Phase 1, will have at least one acceptor a which has accepted $\langle p, v \rangle$. Paxos also ensures that before sending the accept request for p_2, v_2, the proposer must select the value of the highest numbered proposals which it receives in its promises.

So the acceptor a will send $\langle p_1, v_1 \rangle$ in its promise message to the proposer. As $\langle p_2, v_2 \rangle$ is the only proposal number which has proposal number greater than p, the proposer must set $v_2 = v_1$ in its accept request message $\langle p_2, v_2 \rangle$ as v_1 is the value of the highest numbered proposal that it receives as a promise response. Thus, meeting our above requirement.

4.1.4 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 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 adpated 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 *)
(* Storing a proposal and not just a nat as this makes it easier to catch the payload
later on in the transitions *)
| 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.

5
Client Application

As outlined in chapter 2, our approach to using Disel involved first coding the protocol and the client application, and then going on write the proofs. 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.

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 to 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 protocol.

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 intialised 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 on 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 initilise 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 recieved, 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`.

5.3 Extraction

Disel programs can be extracted into their corresponding OCaml definitions. The extracted code contains modules to 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. Additionnaly, each node needs to be given an initial state that satisifed 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 proposol 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 `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.

5.4 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 has 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* 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 wrapper which uses the extracted code is implemented in the `paxos.py` file. 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.

5.5 Verifying client in Disel

```
(* Step 1: Receive prepare req *)

(* Ending condition *)
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. by apply: with_spec x. Defined.
Next Obligation. by move:H; rewrite /r_prepare_req_inv (rely_loc' _ H0). Qed.
Next Obligation.
  apply:ghC⇒i1 psal[/eqP⟶{H}/=E1]C1; apply: step.
  apply: act_rule⇒i2/=R1; split; first by case: (rely_coh R1).
  case⇒[[[from e']d i3 i4]|i3 i4]; last first.
  - case⇒S/=[]?; case; last by case⇒?[?][?][?][?][?][].
   case⇒_ _ Z; subst i3⇒R2; apply: ret_rule⇒i5 R4/=.
   by rewrite (rely_loc' _ R4) (rely_loc' _ R2)(rely_loc' _ R1).
  case⇒Sf[]C2[]⇒[|[l'][mid][tms][from'][rt][pf][][E]Hin E2 Hw/=]; first by case.
  case/andP⇒/eqP Z G⟶[]Z1 Z2 Z3 R2; subst l' from' e' d.
  move: rt pf (coh_s (w:=W) l (s:=i2) C2) Hin R2 E2 Hw G E; rewrite prEq/=.
  move⇒rt pf Cj' Hin R E2 Hw G E.
  have D: rt = receive_prepare_req_trans _ _.
  - move: Hin G; by do! [case⇒/=; first by move⇒⟶].
  subst rt⇒{G}.
  have P1: valid (dstate (getS i2))
   by apply: (@cohVl _ coh); case: (Cj')⇒P1 P2 P3 P4; split⇒//=; done.
  have P2: valid i2 by apply: (cohS (proj2 (rely_coh R1))).
  have P3: l ∈ dom i2 by rewrite -(cohD (proj2 (rely_coh R1)))/ddom gen_domPt inE/=.

  apply: ret_rule⇒//i5 R4.
  - rewrite /r_prepare_req_inv; rewrite (rely_loc' _ R4) (rely_loc' _ R) locE//=.
   rewrite /PaxosProtocol.r_step /=.
   rewrite -(rely_loc' _ R1) in E1.
   rewrite (getStK _ E1).
   by rewrite /step_recv /mkLocal.
Qed.
```

```
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.
```

6
Conclusion and Evaluation

6.1 Summary of Achievements

6.2 Critical Evaluation of the project

6.3 Critical Evaluation of Disel

6.4 Future Work

6.5 Final Thoughts

References

[1] Alvaro Garcıa-Perez, Alexey Gotsman, Y. M. & Sergey, I. (2018). Paxos consensus, deconstructed and abstracted. http://ilyasergey.net/papers/paxos-deconstructed-esop18-extended.pdf. [Online; accessed 03-April-2018].

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

[3] Cezara Dragoi, T. A. H. & Zufferey, D. (2016). Psync: a partially synchronous language for fault-tolerant distributed algorithms. *POPL. ACM*, (pp. 400–415).

[4] Chris Hawblitzel, Jon Howell, M. K. J. R. L. B. P. M. L. R. S. S. B. Z. (2018). Ironfleet: Proving practical distributed systems correct. *Proceedings of the ACM on Programming Languages*, 2(POPL).

[5] Ilya Sergey, James R. Wilcox, Z. T. (2018). Programming and proving with distributed protocols. *ACM on Programming Languages*, 2(POPL).

[6] James R. Wilcox, Doug Woos, P. P. Z. T. X. W. M. D. E. & Anderson, T. E. (2015). Verdi: a framework for implementing and formally verifying distributed systems. *PLDI. ACM*, (pp. 357–368).

[7] Lamport, L. (1978). The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2, 95–114.

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

[9] Leslie Lamport, R. S. & Pease, M. (1982). The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3), 382–401.

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

[11] Oded Padon, Kenneth L. McMillan, A. P. M. S. & Shoham, S. (2016). Ivy: safety verification by interactive generalization. *PLDI. ACM*, (pp. 614–630).

[12] Ongaro, D. & Ousterhout, J. (2014). In search of an understandable consensus algorithm. *Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference*.

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

[14] Renesse, R. V. & Altinbuken, D. (2015). Paxos made moderately complex. *ACM Computing Surveys*, 47(3).

[15] Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4), 299–319.

[16] Tushar Chandra, Robert Griesemer, J. R. (2007). Paxos made live - an engineering perspective. *PODC '07: 26th ACM Symposium on Principles of Distributed Computing*.

[17] Vincent Rahli, David Guaspari, M. B. & Constable, R. L. (2015). Formal specification, verification, and implementation of fault-tolerant systems using eventml. *AVOCS. EASST*, (pp. 400–415).

A
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.

B

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.