Mechanised Verification of Paxos-like Consensus Protocols

Anirudh Pillai

February 2, 2018

Thesis advisor: Professor Dr. Ilya Sergey

Anirudh Pillai

Mechanised Verification of Paxos-like Consensus Protocols

Abstract

Distributed systems are hard to reason about.

Acknowledgments

Acknowledgements

Contents

1 Introduction

1.1 The Problem

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.

Aims

1. Learn about distributed system protocols
2. Contribute to open source software

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

1.4 Report Overview

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.

A State Machine M can be defined as $M = \{q_0, Q, I, O, \delta, \gamma\}$ 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 valide outputs δ is the state transition function, $\delta : IxQ- > Q$ γ is the output function, $\gamma : IxQ- > O$

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 is very common and 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 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 serving you '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. In-order 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.

Popular consensus protocols include PageRank used by Google and the Blockchain consensus protocol. George and Ilya verified a subset of the protocol in Coq in their Toychain paper.

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 is used for state machine replication and helps all the replicas achieve consensus on a result.

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.

Safety Invariants:

- A value can only be chosen after it has been proposed.
- Only one value may be chosen.

A processor 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.

Assumptions:

- Processors communicate between each other by exchanging asynchronous messages between each other. Asynchronous here means that messages may take arbitrarily long to be delivered. Messages make be lost or duplicated but a processor will never receive a corrupted message.
- Processors run at an arbitrary speed and may fail or restart.

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 $\{proposal_number, proposal_value\}$. $proposal_number$ is just a natural number associated with a proposal which makes it easy to keep track of all the different proposals.

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

There are a few requirements which are imposed for the algorithm to function correctly. The algorithm ensures that these invariants are maintained.

1. **R1** An acceptor must accept the first proposal it receives.
2. **R2** 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 $proposal_value = v$.

R1 ensures that a value is agreed in the case when only a single value has been proposed by a single proposer. The most basic case is when you only have one acceptor. The acceptor can choose the first proposal that it receives from any of the proposers and then it can ignore the rest. The problem with using this is that if the single acceptor crashes then the algorithm won't make progress.

So you need to have multiple acceptors but this is where the second problem arises. As multiple proposers might propose different values to different acceptors who might end up accepting different values.**R2** ensures that even though the acceptors might accept a proposal from a different proposer, all of the proposals accepted will have the same value. (Not sure about this)

2.2.2 The Algorithm

Simple Paxos algorithm runs in rounds until a consensus is achieved (a successful round has occurred). A successful round of the algorithm has two phases.

1. **Prepare Request** A proposer sends a Proposal (n, v) to each acceptor in a majority subset of all the acceptors. It then expects the following as a response from each of the acceptors.

    - A **Promise Response** which states that the acceptor,from now onwards, won't accept a proposal with $proposalnumber < n$. (If the acceptor has already made a promise to some other proposer to not accept proposals with $proposal_number < m$ where $m > n$ then the acceptor can ignore this Prepare request from this proposer by not responding to it.)
    - If the acceptor has already accepted one or more Proposals, it must respond with the highest numbered Proposal it has accepted that also has $proposal_number < n$.

2. **Accept Request** If a majority of acceptors respond to the the prepare request the proposer can issue an accept request for a Proposal (n, v') where v = (value of response with highest proposal number) or (v if none of the responses contained any proposals)'.

- The proposer then sends this accept request another majority subset of acceptors.
- Any acceptor that receives the accept request responds with an **Accepted Response** if and only if it hasn't already promised not to accept any proposals with $proposal_number < m$ where $m > n$.

2.2.3 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 *distinguishedlearner* 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.

2.3 Disel

2.3.1 Inductive Invariant

2.4 Related Work

3 Requirements and Analysis

3.1 Detailed Problem Statement

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

3.3.1 Requirement 1: Adapted Protocol and State Transition System

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 ofthe 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.

3.3.2 Requirement 2: Inductive Invariant

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.

3.3.3 Requirement 3: Simulation

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.

3.3.4 Requirement 4: Proof

3.3.5 Requirement 5: Client Application

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.

4 Design and Implementation

4.1 Modelling

4.1.1 The Protocol

4.1.2 State Transitions

Having adapted the protocol, we then had to create a state transition system for the nodes in your 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 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 `PWait-PrepResp` 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:

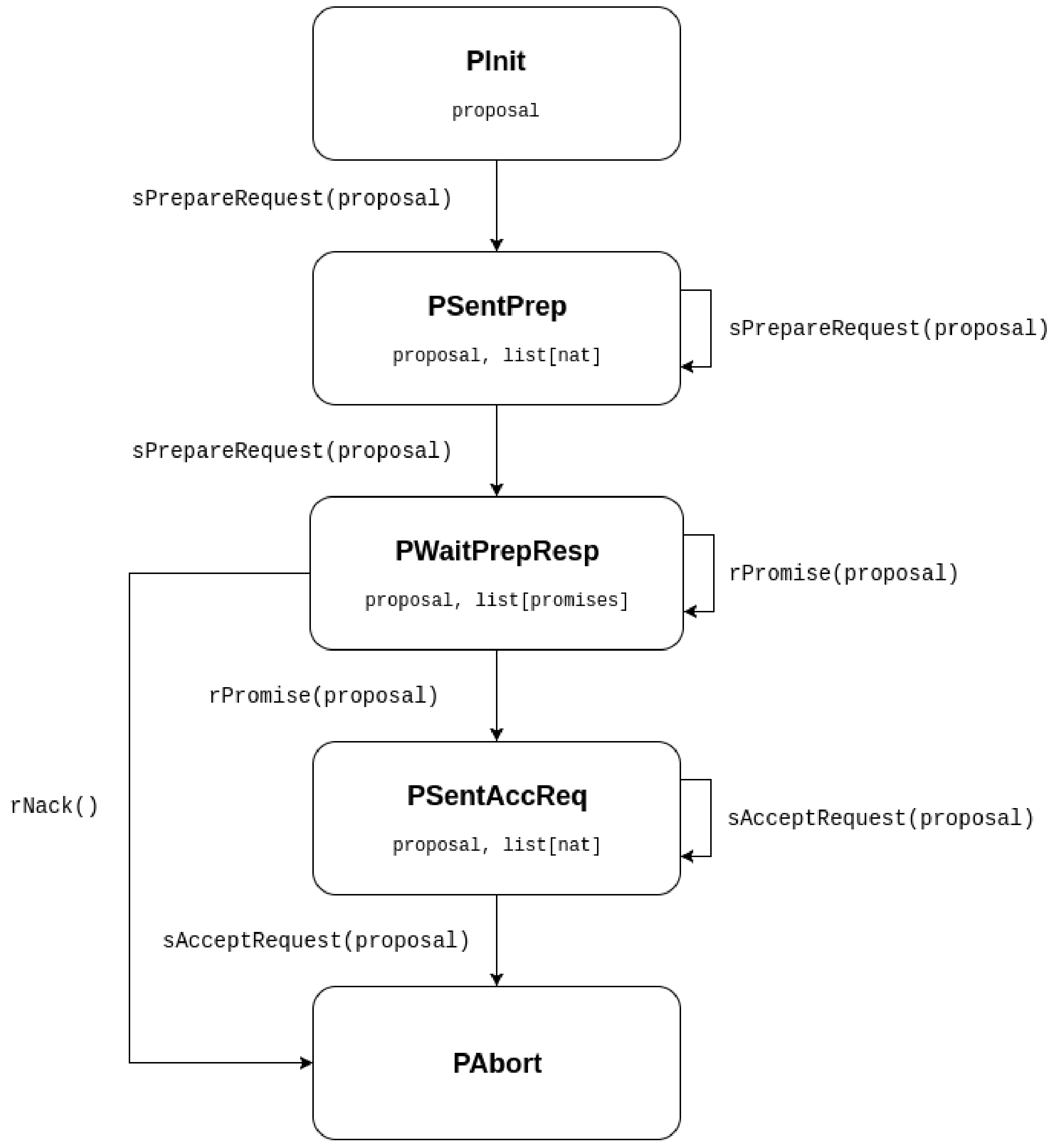

Figure 4.1: Proposer State Transition Diagram

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.

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

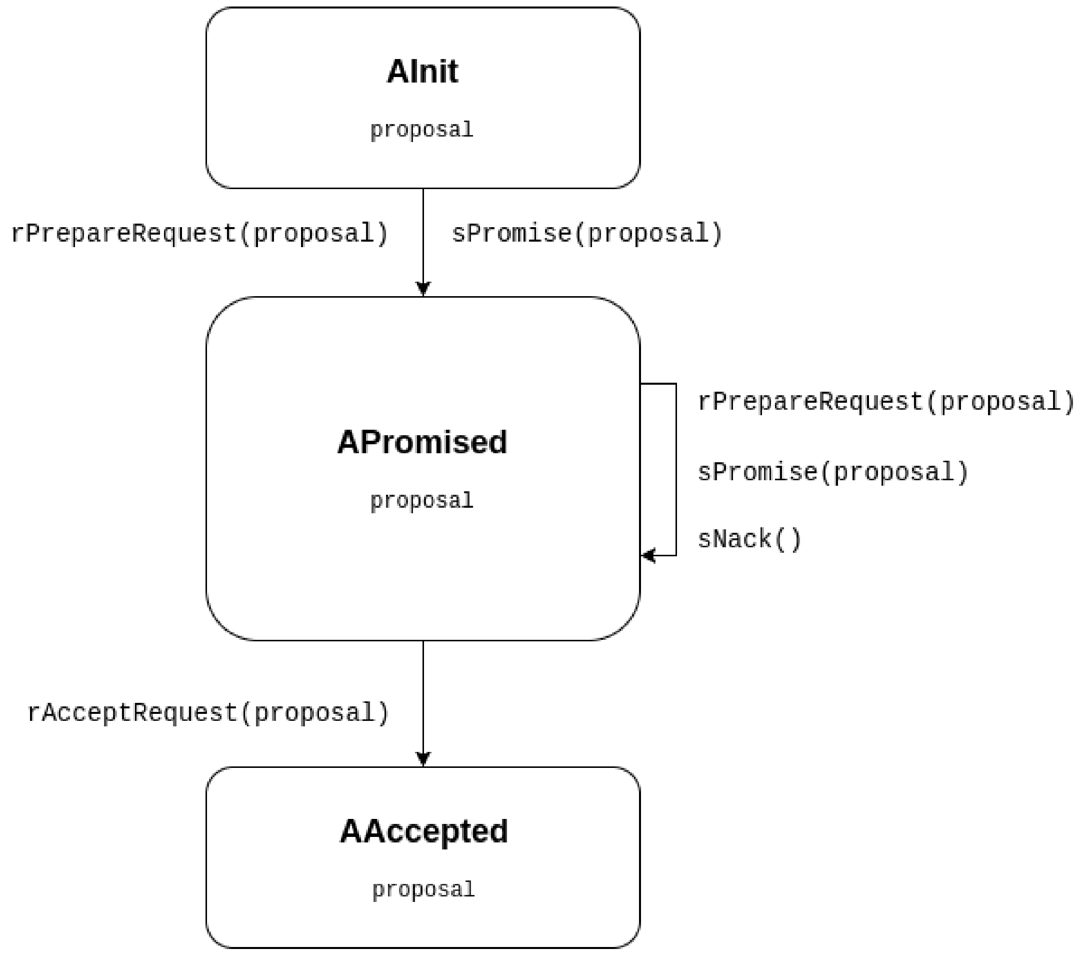

Figure 4.2: Acceptor State Transition Diagram

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.

4.2 Verification

5
Client Application

6 Conclusion and Evaluation

6.1 Summary of Achievements

6.2 Critical Evaluation

6.3 Future Work

6.4 Final Thoughts

References

A

Interim Report