Mechanised Verification of Paxos-like Consensus Protocols

Anirudh Pillai

January 18, 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.

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.

Faults 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.

State Machine Replication

In 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, $\delta : IxQ- > Q\,\gamma$ is the output function, $\gamma : IxQ- > O$

The method of modelling a distributed system protocol as state transition system is very common and is a critical component of this project as well.

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

2.4 Related Work

3

Requirements and Analysis

4

Design and Implementation

5

Client Implementation

6

Conclusion and Evaluation

References

A

Interim Report