Analysis of distributed algorithms using CSP

Joshua Warwick

May 2016

Abstract

Distributed algorithms lie in the background of many internet-based services, but what do we know about their correctness? There have been many high profile failures of these systems including just under a year ago, when the New York Stock Exchange halted trading due to an issue with a distributed algorithm. In this project we examine a *consensus* algorithm. Consensus algorithms are where a group of computers are able to agree on some data value, voted for by a majority of them. In addition to this, consensus algorithms need to be resilient to failure - we still want to agree on a value even if only a few computers go offline. The types of failure that can and do occour, and that we analyse, are machine based failures and network based failures. Beacause these algrothms are designed to be resilent to such failures, they can be used to maintain a high level of availblity even in unreliable networks. Some high profile users of consensus algorthms are Apache Zookeeper[1], Chubby[2] and Amazon Web Services. But despite the populatity of these utilities there have been few attempts to apply verifcation techniques to their underlying consenus algrothms [3, 4]. We pick Raft as the focus of this project due to its increased popularity, and the fact some implementations of Raft have contained errors. We explain the details of Raft alongside constructing a CSP model of the protocol. Throughout this construction we explain the modelling choices made, and the impact they have on the validity of the model. We then run refinement checks on the model, verifying key consensus properties about Raft. Then we run the same refinement checks but introduce the different failures modes we have described above into the model. Refinement checking Raft highlights issues with verifying consensus algorithms in general, specifically the difficulty in constructing a model that is verifiable. We show methods of reducing the complexity of the Raft model to reduce the state space of the problem. Finally we discuss the problems of verifying consensus algorithms and how they should be tackled.

Contents

1 Introduction

1.1 Distributed computing

Distributed computing describes a collection of algorithms that use multiple machines to achieve a common goal. By using multiple machines may capture a greater processing power, allowing faster processing of algorithms compared to a single machine. We can also handle much larger problems than any one machine could cope with. By running on machines across different networks, we also allow an algorithm to be highly available, as the chances of simultaneous disruption to all networks is unlikely. These qualities enable the algorithms to be fault-tolerant and powerful, but come at the price of complicating the algorithm.

Distributed algorithms share a close correspondence with concurrent algorithms. The similarity can be seen by comparing the use of *threads* in a concurrent algorithm, to the use of *machines* in a distributed one. They share many types of problem, such as synchronisation and client/server based problems. The main differences come from the fact threads are able to share memory, something that cannot be efficiently replicated amongst physically separated machines.

We know that studying the intricacies of concurrent algorithms is difficult, but the problems faced in distributed computing are harder. Distributed algorithms face most of the same problems as concurrent, with the physical separation of the machines the major source of this extended difficultly. For example, we can no longer guarantee message delivery because the network can (and does) fail. This complexity is the key to why distributed computing becomes a much harder problem than concurrent.

Problems appear when designing these algorithms to be fault tolerant with respect to network and machine failures. Many sutble bugs can appear - traditional examples are deadlock and race conditions. Simple algorithms may appear correct, but after a certain execution path result in an large error. Famous examples of errors in distributed computing are the Bully algorithm [5], AT&T's network crash in 1990 [6], and the major AWS outage in 2012 [7].

Problems such as these arise in distributed computing when something changes from the normal operation. Changes such as message dropping and machine failures are those that tend to cause issues. For example if a machine is waiting upon a message that has been lost, then it will wait forever unless a timeout happens or the message is sent again - an example of deadlock. Whilst it can be relativity easy to design a distributed algorithm under normal operation, when you allow for these failures then the number of edge cases to consider increase substantially. What is worse is that they are often very subtle, occurring in interleavings that may seem improbable to the designer, but are able are reasonable in the real word.

Testing an algorithm for resilience under these failures is much harder than using normal testing methods. The sheer number of failures, and their ordering, creates far too many combinations for anybody to fully test. The most common form of testing is though simulation. This is both time consuming and not guaranteed to pick up errors with the algorithm.

1.2 The consensus problem

We consider a specific instance of a distributed computing problem, consensus. The problem of consensus involves a group of machines called a cluster, that wish to agree upon on a particular value. This value often takes the form of an operation or event that all the machines should perform. Any machine in the cluster should know all of the decisions made by the collective previously. Importantly each machine must have the correct order that the values were agreed upon. This is a crucial property that allows consensus to be used for implementations such as a distributed database, where the order of operations is important. The cluster should be able to make good progress under standard network assumptions and multiple machine failures. These standard network assumptions are that messages may be delayed or not delivered, but never corrupted. This allows for intermittent network connectivity, where a machine can drop in and out of the network and know what happens in the cluster whilst it was away.

Paxos [8] is the most well-known algorthm to solve the consensus problem, and has mnay implementaions based upon its written desciption. Paxos is used in many large scale applications such as Apache Zookeeper, Amazon AWS and Google's Chubby service. Despite these high profile users, Paxos itself is notoriously difficult to understand and implement without errors. Whilst the algorithm is proven correct, the implementations end up varying so much from the Paxos description that they becoming a very different algorthm, and one that is essentually unproven [2]. In fact Paxos's author released a second paper on the algorithm named "Paxos made simple"[9], which only highlights the difficulty of understanding the algorithm.

We should bear in mind that this does not mean Paxos is useless, the difficulty of implementing and understanding is far outweighed by the benefits of a correct implementation. It is the problem of understandability that is crucial to a correct implementation. Given an easier to understand and proven protocol, developers shouldn't have such a hard time implementing. This means developers will avoid introducing any bugs into an algorthm. This is exactly what the designers of the Raft consensus protocol believe, and built Raft based upon understandability from a implementation prospective [10].

1.3 Raft and Project Jepson

Raft [10], the main focus of this project, was developed in 2014 by Diego Ongaro and John Ousterhout. A main objective of the algorithm is its understandability, in order for implementers of the algorithm to create more reliable distributed systems. The authors took to carrying out a study with university students to show their algorithm was much easier to understand when compared with Paxos.

Raft is a *replicated-log* consensus algorithm; this means each machine in the cluster maintains its own ordered log - a simple list of entries that have decided upon by the cluster. Briefly, the algorithm works by electing a single node called the leader. The leader's job is to make sure every other machine's log matches its own. This is Raft's main difference with Paxos. Paxos lets the cluster agree on a entry together, before each machine in the cluster adds the entry to their logs. Raft keeps one special log, the leader's, that copies its own log to the

other machines in the cluster. This approach makes the algorithm much easier to understand. Firstly, the *agree then add* procedure of Paxos is reduced to simply replicating a data structure in Raft.

Machines in Raft do not explicitly agree on a entry together because they all copy the log of the leader. Raft instead introduces the concept of *committed* entries to remain a consensus solving algorithm. The leader keeps the last log entry which has been successfully copied to each machine. If the leader knows a majority of the nodes have copied a certain entry, then this entry, and all the entries preceding it should be regarded as committed. The leader tells the other machines which entry is the last to be committed and only then do the machines regard the entry as being agreed upon.

Like Paxos, Raft is resilient to message dropping, message delay and machine failure. A majority of the machines need to fail before the algorithm stops working. Raft also maintains a number of properties that are key to both its understandability and its correctness:

- Leader Append-Only – This states that a leader does not delete previous entries in its log, only adds new ones.
- Log Matching – This is where, given two machines, if their logs match at one point, then they are identical for all entries before the match.
- Leader Completeness – If a leader contains a committed entries then this entry will be contained in the logs of all later leaders

Despite these properties and its understandabilty claims, Project Jepson [11] - a project which tests consensus algorithms by simulating different types of network failures, has found bugs in some implementations of Raft. The tests Project Jepson carries out are network partition and message-loss based, testing different combinations of the two. This is just one case where lots of subtleties may appear. It is impossible to test every combination of events that could lead to failure. This raises the question if Project Jepson has found these bugs though brute force, what happens if we could perform an exhaustive search for these bugs?

1.4 Automated verification and FDR

When testing an algorithm we cannot consider all possible cases. Automated verification is an approach that allows us to check properties about an algorithm. A model needs to be constructed of such an algorithm, before we can check any properties. When constructing a model there are always abstractions and simplifications made, and it is important to justify how closely the model relates to the original algorithm. A model with too many abstractions will have little bearing on the original, whilst one with too few means we may construct a model we cannot verify. It is important to find a good medium between the two in order to achieve useful results.

Verification cannot check everything though, we can only check given properties called specifications. The model can then be checked against these in an automated way. If the model does not satisfy the specification then the automated tool will tell us the behaviour of the model that caused the failure. Specifications will be normally be properties from the original algorithm. The

idea is that if the model does not satisfy the property, being a weaker form of the algorithm, then the original algorithm also does not satisfy the property.

An advantage to modeling in CSP is that there is an existing program, FDR [12], that we can use to check models against their specifications. This allows us to automatically verify the model we create. If FDR finds the model does not fit the specification, then it will return an execution path that describes how the modelled algorithm fails. We can then use this path to work out exactly what has caused the failure.

The only drawback to automated verification is the considerations involved when constructing the model. If the model we have created is too complex then will not be able to check any properties as there is not enough computing power. This is something we have to keep a close eye on throughout the construction of our model.

1.5 Contribution

In this project we construct a detailed model of the Raft algorithm in CSP, discussing exactly how each part of the protocol works, and how we have modelled it. First, in section 2, we introduce the knowledge required to understand the construction of the model. This involves introducing the Raft algorithm and the concept of using time in CSP.

Then in section 3 we detail the construction of the CSP model of Raft, explaining the Raft algorithm in full detail and the CSP processes used to model it. We also explain how we have modelled the types of failure that can occur in the real world, and the restrictions we have on doing so.

In section 4 we evaluate the model, discussing the abstractions made in order to both construct it using CSP, and ensure it is verifiable. We also give details of performance enhancements carried out on the model to reduce the compilation time in FDR. These performance improvements become crucial to being able to verify our model. We then discuss how valid the model is, justifying how closely it relates to the description of Raft.

In section 5 we describe the properties we are testing Raft for, and the equivalent CSP specification processes. We then detail the result of model checking these properties, first with the standard Raft model, then with different types of failures introduced.

Finally in section 6, we conclude on the project as a whole and the difficulty involved with creating a model that is both verifiable and true to the algorithm it is modelling. We discuss how this is a problem generally for distributed algorithms, and how to combat this by improving the performance of constructed models.

2 Background

2.1 Communicating Sequential Processes (CSP)

We assume knowledge of CSP as per the Oxford concurrency course [13], based on the book Understaning Concurrent Systems [5], and we introduce any additional knowledge required.

For our model we need to be able to model the passage of time, and use specifications that involve time. Modelling time using CSP can be done in two main ways, *Timed CSP* and *tock-CSP*. The former requires the re-interpretation of all of the operators to include a continuous timing aspect. The later models the passage of time by using an explicit *tock* event. Tock-CSP is chosen both for its simplicity and the fact FDR only supports timed CSP though a translation into Tock-CSP anyway.

Modelling time using Tock-CSP means we have to split time into distinct, but arbitrary lengths of time. We shall not be expressing "I want event A to be offered for 4 seconds", as there is no need to convert between seconds and *tock* events for our use. In essence we measure time with the event *tock* as the unit. Tock events can be used for synchronisation amongst processes, acting a global clock for them.

Here is an example of a timed model of a vending machine. The timing aspect is that if there is no selection of tea or coffee after three units of time, then the coin is eaten by the machine. This style of *timeout* will be the main usage of *tock*-CSP in our model.

$$\begin{aligned} V_0 &= coin \rightarrow V_3 \ \Box \ tock \rightarrow V_0 \\ V_n &= tock \rightarrow V_{n-1} \\ &\quad \Box \ tea \rightarrow tock \rightarrow V_0 \\ &\quad \Box \ coffee \rightarrow tock \rightarrow V_0 \end{aligned}$$

By looking at the traces of the system we are able to see not only the order of events, but how much time has passed between them. This is done by observing when a *tock* event has been performed in the trace. Here is a simple example where can see that it took two *tock* units of time before the coffee was selected.

$$\langle coin, tock, tock, coffee, tock \rangle$$

By introducing *tock* events we can now define specifications that include time. If we consider an environment that offers an event a but refuses the event *tock*, then the event a must happen now. We call this type of event urgent, because if event a does not occur immediately then we delay time. We also have delayable events, an event b is delayable if b is continually offered whilst *tock* events are being repeatedly performed.

We need tocks to happen infinitely often. An equivalent statement is we do not allow an infinite amount of events to happen between two *tock* events. This can be checked with a simple failures-divergences check:

$$\begin{aligned} TOCKS &= tock \rightarrow TOCKS \\ TOCKS &\sqsubseteq_{\text{FD}} P \setminus (\Sigma \setminus \{tock\}) \end{aligned}$$

Here we first hide all the events other than *tock* from P, calling this process P_T. We use a failures divergences check, as a divergence indicates an infinite

number of events happening between two *tock* events. The failures part of the specification checks that the process P does not stop time. If P does stop time then eventually P_T will refuse the *tock* event. Here $TOCKS$ is a simple process that just offers a *tock* event before recursing, representing the passage of time.

The important part is the modelling of timeout events, which plays a part in the Raft algorithm outlined next.

2.2 Raft

We now give a more detailed overview into the workings of the Raft algorithm.

Raft is alternative solution to the consensus problem, with emphasis placed on its understandability. Raft solves a variant of the consensus problem where each machine has its own finite state machine, called *replicated state machine*. Every transition needs to be agreed upon by the machines in the cluster, and they all perform the same transition. The values in the consensus problem are the state machine transitions in Raft. To avoid ambiguity, from now on we refer to physical machines as *nodes*, as to not confuse with a finite state machine.

To go about replicating the state machines, Raft uses a leader system where one node is *elected* by the other nodes to be leader. The leader's job is to then handle the replication of its own state machine to all other nodes. All these other nodes are called *followers*.

The leader must send regular updates to the followers. If this communication stops then the followers know the leader has failed, or is unreachable due to network issues. When this happens the nodes work together to *elect* a new leader.

The procedure for replicating the state machine is carried out by using the logs mentioned in the introduction section. Recall the leader is in charge of copying its log to the other nodes, and a value is *committed* if it has been successfully copied a majority of the followers. Thus only apply a transition to the state machine if it is committed in the log.

Raft's authors split the algorithm description into three key sections in order to enhance its understandability, these sections are described below:

- Leader Election – How the nodes work together to decide upon a new leader. Each leader rules for a *term*, which monotonically increases as more elections occur. A following node has a (bounded random) timeout to tell when a leader has failed, after this hits zero the follower starts a new election.

  Every node stores a value representing which election term it is running in. Each election begins with the candidate node incrementing its term. After this it accumulates votes from the other nodes, needing a majority of the cluster's votes to win the election. If some node has a higher term the candidate's then that node refuses to vote for the candidate. Once the candidate has the required votes it becomes the new leader, and the election procedure ends.

  To combat the effect of a split vote, where two candidates try to win the same election, follower nodes can still timeout between voting for a candidate, becoming leader.

- Log Replication – How the leader replicates the log to the followers. The leader does this by storing, for every follower, the index of the next entry to send to that follower. The leader's match index stores an index for every follower that represents how much of the log has been copied to that follower. The interaction between these two indexes ensures the log is replicated correctly to every follower. When the leader copies an entry to the majority of followers, it is committed and the leader communicates this to the followers. This allows the followers to update their own commit index.

- Safety – This is an extra restriction on the leader election. Imagine a leader fails; as it stands every node is a viable candidate to become the new leader. Recall above that we only need a majority of the nodes to have copied a entry for it to be committed. Now what happens if a node that has not copied this entry becomes leader? There is now no mechanism for the leader to receive this entry that others have committed, leading to an inconsistency. This problem is solved when nodes vote for each other. A follower only votes for a candidate if the candidate's log is at least as up to date as its own. This guarantees all the committed entries are present on the new leader.

Raft nodes communicate asynchronously using *Remote Procedure Calls* (RPCs). An RPC contains different information based upon its task, and a reply to the sender is expected. This reply may simply be a receipt that the RPC has been delivered, but may contain some data in response to the received RPC. Typically a node receiving an RPC will do some processing based upon the information in the message, and includes the result in the reply. Raft uses two types of RPC:

- RequestVote RPC – Used during the leader election, a candidate sends these to request votes. A reply to an RPC says if a vote for the candidate was granted or not. The candidate sends the index and term of the last entry such that the recipient can perform the additional safety check described above.

- AppendEntries RPC – Used during normal operation to implement the log replication. Contains a list of log entries to append. May be empty just to keep up communication with the follower, this type of message is called a *heartbeat*. We also send the index and term of the entry preceding the entries we are sending. The node can then check this is correct before appending the new parts to the log. A reply to this RPC includes a success variable, to indicate if the entries were copied or not. For example, a reply of false would be given when the leader is incorrect about the last index, so the parts of the log are not appended. The leader can then adjust the part of the log it sends to this node in a further RPC.

Despite the supposed clarity of the Raft algrothm, Project Jepson found a bug in an implementation, called etcd, developed of part of CoreOS [14]. In this case the developers incorrectly assumed transitions in the log on the leader could be applied despite not being committed. This is incorrect as leader may fail before it manages to commit a value, meaning it may be lost on the next leader. This is a subtlety missed by the etcd developers when implementing the algorithm,

opening the question to what other problems may be waiting for developers to trip up on.

3 Raft CSP Model

In this section we explain the Raft protocol in detail via the construction of a CSP model.

3.1 Preliminaries

First we describe the sets that our events will use, and some auxiliary functions on these sets. We also introduce sequence operations that are used to manage the log stored on each node. As FDR requires finite sets for channels to range over, we have to put a cap on the number of terms, nodes, and log indexes in our model.

$maxNodes :: Int$ – The number of nodes in the cluster.
$maxTerm :: Int$ – The maximum term.
$maxIndex :: Int$ – The maximum log index.

$$nodes = \{1 \ldots maxNodes\} \text{ – The set of nodes.}$$
$$terms = \{1 \ldots maxTerm\} \text{ – The set of terms.}$$
$$indexes = \{1 \ldots maxIndex\} \text{ – The set of log indexes.}$$

We also define the following set operations to restrict the set of $terms$. $above_x$ returns the set of terms that are above term x. $below_x$ returns the set of terms below x. not_x returns the set of nodes not including n.

$$above_x = \{y \mid y \leftarrow term, x < y\}$$
$$below_x = \{y \mid y \leftarrow term, x > y\}$$
$$not_n = nodes \setminus \{n\}$$

We use the following sequence operations for modifying the log.

- $!!(seq, i)$ – Given an index i returns the ith element in the sequence.
- $Take(i)$ – Given an index i returns the first i elements of the sequence.
- $\hat{}(seq, \langle x \rangle)$ – A concatenation operator, concatenates $\langle x \rangle$ to the end of seq.
- $decrement(seq, i)$ – Decrements the ith element by one.
- $increment(seq, i)$ – Increments the ith element by one.
- $update(seq, i, x)$ – Updates the value in the ith position to x.

We use two variables to define the length of each type of timeout in a Raft node:

- $electionTimeout :: Int$ – After this number of tocks without an RPC from the leader, a node starts a new election.
- $leaderTimeout :: Int$ – The number of tocks before a leader sends out RPCs.

3.2 Modelling Communication

To represent the communication we use events that indicate that a message is sent between two nodes. The channels for these messages are parametrised by all the variables the RPC or its reply contains. We present the channel definition for each message and the meaning of its parameters.

We begin with the append entries RPC. The key parts to the replication of the log are precIndex and precTerm. These two values are the index and term that precede the entries we are trying to send.

appendEntries

- $.from :: nodes$ – The sender's ID.
- $.to :: nodes$ – The recpient's ID.
- $.term :: terms$ – The sender's term.
- $.precIndex :: indexes$ – Index preceding any log entires sent.
- $.precTerm :: terms$ – Term from then preceding index.
- $.commitIndex :: indexes$ – The senders's commit index.
- $.entry :: boolean$ – The next part of the log to be appended.

Next the reply to an append entries RPC. If success is true then the leader was correct about the entries to send, and the they sent were appended. If success is false then the leader was incorrect, and needs to send the correct ones next time.

appendEntiesReply

- $.from :: nodes$ – The sender's ID.
- $.to :: nodes$ – The recpient's ID.
- $.term :: terms$ – The sender's term.
- $.success :: boolean$ – Was this the correct entry to send?

Now we look at the request vote RPC. The important parts of this RPC are the lastIndex and lastTerm parameters. These are the parameters that allow the extra safety restriction on the leader election process to be applied.

requestVote

- $.from :: nodes$ – The sender's ID.
- $.to :: nodes$ – The recpient's ID.
- $.term :: terms$ – The sender's term.
- $.lastIndex :: indexes$ – The last entry's index in the senders log.
- $.lastTerm :: terms$ – The term from the last entry.

Finally, the request vote reply. If the vote is granted then the sending node has voted for the recipient node.

requestVoteReply

- $.from :: nodes$ – The sender's ID.
- $.to :: nodes$ – The recpient's ID.
- $.term : terms$ – The sender's term.
- $.voteGranted :: boolean$ – Did the sender give its vote?

An RPC call in Raft can be sent by a node and not received by another due to network delay or packet loss. Normally there is a slight delay between an RPC being sent and received. In our model we abstract away this slight delay. We make this design choice to reduce the complexity of our model.

3.3 Modelling a Raft Node

A Raft node can be three possible states at any time: *leader*, *follower* or *candidate*. Hence it makes sense to create three co-recursive processes to represent these states. We use parameters in the process to store the variables in each Raft node state. The details of this are shown below, starting with the follower state:

$Follower($
$n :: nodes,$ – This is the node's ID.
$term :: terms,$ – The largest term the node has seen.
$log :: \langle terms, entry \rangle,$ – The log stored on this node.
$votedFor :: nodes,$ – The node ID that we voted for in the last election.
$timeout :: integer,$ – # of tocks we peform before a new election is started.
$commitIndex :: indexes,$ – The largest log index that is commiteed.
$lastApplied :: indexes)$ – The last index applied to the state machine.

A candidate Raft node stores all the same information as a follower with the addition of keeping track of the number of votes it has accumulated.

$Candidate($
$n :: nodes,$ – This is the node's ID.
$term :: terms,$ – The largest term the node has seen.
$log :: \langle terms, entry \rangle,$ – The log stored on this node.
$votedFor :: nodes,$ – This node's ID.
$timeout :: int,$ – # of tocks before a new election is started.
$commitIndex :: indexes,$ – The largest log index that is committed.
$lastApplied :: indexes,$ – The last index applied to the state machine.
$votes :: int,$ –The number of voted for this node, including this node.
$noReplies :: \{nodes\})$ – Node processes that have not replied to our vote request yet.

A leader Raft node stores even more information than the candidate. The leader process has to also store the next index and match index arrays as sequences, as well as keeping track of nodes that have not replied to our message. This is

so RPCs can be retried.

$Leader($
$n :: nodes,$ – This is the node's ID.
$term :: terms,$ – The largest term the node has seen.
$log :: \langle terms, entry \rangle,$ – The log stored on this node.
$votedFor :: nodes,$ – This node's ID.
$timeout :: int,$ – # of tocks before sending new append entry RPCs.
$commitIndex :: indexes,$ – The largest log index that is committed.
$lastApplied :: indexes,$ – The last index applied to the state machine.
$noReplies :: \{nodes\},$ – Node processes that have not replied to the append entries RPC yet.
$nextIndex :: \langle indexes \rangle,$ – Index of the next entry to send to each node.
$matchIndex :: \langle indexes \rangle)$ – Highest index to have been copied to each node.

3.4 Modelling the Leader Election

In this section we explain the details of the leader election process in Raft, showing how our CSP model either implements or abstracts away each element.

An election is triggered by a node when it believes the current leader has failed, or the current election process is taking too long. The former can be detected by making the leader communicate at regular intervals, even if it has nothing to send. This type of communication is called a *heartbeat*. Each node generates a random number called the *election timeout*, if this amount of time passes then we know the leader has failed. The election timeout is used to decide the limit on the amount of time an election takes.

To simulate a timeout in CSP we use a similar method to the timed vending machine. As mentioned, the process for each node state contains a parameter to keep track of how many tocks may pass before starting an election. Below is the test to see if the follower has timed out.

$$Follower(\dots) = StartElection(\dots) \triangleleft timeout = 0 \triangleright Follower'(\dots)$$

Next is the section of the $Follower'$ process that handles the passage of time, with the only change in the timeout parameter.

$$\begin{aligned} Follower'(\dots) = {} & tock \rightarrow Follower(\dots, timeout - 1, \dots) \\ & \Box\ Follower_A(\dots) \end{aligned}$$

3.4.1 Candidate

A node transitions to a candidate state when it starts an election. To start a new election a node must increment its term and vote for itself, then send request vote RPCs. The follower that starts a new election transitions into the candidate state with the following changes: $term \mapsto term + 1$, $votes \mapsto 1$, $votedFor \mapsto n$, and $noReplies \mapsto nodes \setminus \{n\}$. Hence the resulting process is shown below.

$$StartElection(\dots) = Candidate(n, term + 1, n, \dots, 1, nodes \setminus \{n\})$$

After a candidate is initialised it must send request vote RPCs to win votes. A node may retry sending an RPC if no reply is detected. In Raft these RPCs are sent out simultaneously. We offer the RPC request vote event for every node that has not replied. This has some important implications explained later.

When sending an RPC request vote we also include the index and term of the last entry in our log. This is used for the safety check, whereby a candidate can only become leader if its log contains every committed entry.

$$\begin{aligned} Candidate'(\dots) = {} & requestVote!n?to : noReplys!term!logLength!(log!!length) \\ & \rightarrow Candidate(\dots) \\ & \Box\ Candidate_A(\dots) \end{aligned}$$

As well as continually offering the event to send a request vote RPC, we must also offer to receive a reply from a node process. This reply can have the vote granted parameter either *true* or $false$. Note how we are restricting the *term* parameter of the message, this is because we have different behaviour depending on term. For now we only consider offering the request vote RPC events with the term the same as the candidates.

$$\begin{aligned} Candidate_A(\dots) = {} & requestVoteReply?from : Not_n!n!term?True \\ & \rightarrow RunAsCandidate(\dots, (noReplies \setminus from), votes + 1) \\ & \Box\ requestVoteReply?from : Not_n!n!term?False \\ & \rightarrow RunAsCandidate(\dots, (noReplies \setminus from), \dots) \\ & \Box\ Candidate_B(\dots) \end{aligned}$$

The remaining behaviour of the candidate is common to all states so is placed at the end of this section.

3.4.2 Follower

As mentioned, a follower may start a new election by timing out, but we must consider the followers that have not timed out but an election is ongoing. The only way a follower knows an election is happening is when it receives a request vote RPC from a candidate. Crucially, the *term* number for the incoming RPC must be higher than the one stored on the follower for it to be a new election. If the messages *term* is equal to the one the follower stores, then we only consider voting if we have not voted already this term.

To model this we use the *requestVote* events, using the term parameter to produce two disjoint sets of these events. We can then perform different behaviour depending on the term parameter of the message.

$$\begin{aligned} Follower_A(\dots) = {} & requestVote?from : Not_n!n?term' : above_n?cLIndex?cLTerm \\ & \rightarrow DecideVote(\dots, term', \dots, cLIndex, cLterm) \\ & \Box\ requestVote?from : Not_n!n!term?cLIndex?cLTerm \\ & \rightarrow (DecideVote(\dots, term, \dots, cLIndex, cLterm) \\ & \quad \lhd votedFor = null \rhd \\ & \quad Follower(\dots)) \\ & \Box\ Follower_B(\dots) \end{aligned}$$

To consider voting a we check the $lastIndex$ and $lastTerm$ from the incoming message, this is to see if the candidate's log is at least as *up-to-date* as our own. Up-to-date is simply a comparison of the last entry in the candidate's log with the last entry in our own. To deduce which one is more up-to-date this check the following check is carried out: if they are in the same term, then the entry with the higher index is more up-to-date. If they are from different terms then the one with the higher term is more up-to-date.

To model this we use a process, $DecideVote$. The parameters of decide vote are identical to a follower with the addition of $ClastIndex$ and $Clastterm$, to store the term and index of the last entry in the candidate's log. The process's structure is solely based around comparing these values to work out the more up-to-date log, and allow or disallow a vote accordantly.

$$
\begin{aligned}
&DecideVote(\dots) = \\
&\begin{cases}
requestVoteReply!n!from!term!True \\
\quad \rightarrow Follower(\dots), & ClastTerm > log!!logLength \\
\\
requestVoteReply!n!from!term!True, & (ClastTerm = log!!logLength \ \& \\
\quad \rightarrow Follower(\dots) & \quad ClastIndex \geq logLength) \\
\\
requestVoteReply!n!from!term!False, & (ClastTerm = log!!logLength \ \& \\
\quad \rightarrow Follower(\dots), & \quad ClastIndex < logLength) \\
\\
requestVoteReply!n!from!term!False & ClastTerm < log!!logLength \\
\quad \rightarrow Follower(\dots)
\end{cases}
\end{aligned}
$$

Again the remaining behaviour of the follower is placed at the end this section.

3.4.3 Leader

A leader tries to ensure that a new election is not started so it can remain the leader and we do not perform needless elections. The leader must communicate regularly with the followers to prevent a new election form occurring, even if we have no log parts to send.

The leader contains a *noReplies* set, which like the candidate, contains all the nodes that have not replied to an RPC. The leader process combines the idea of timeout and storing who we need to send an RPC to. A leader uses the leader timeout - shorter than the election timeout, which is the minimum interval of communication.

$$
\begin{aligned}
Leader(\dots) &= SendHeartbeats(\dots) \ \triangleleft timeout = 0 \triangleright \ Leader'(\dots) \\
Leader'(\dots) &= tock \rightarrow Follower(\dots, timeout - 1, \dots) \\
&\quad \Box \ Leader_A(\dots)
\end{aligned}
$$

To model the sending of heartbeats, we simply reset the set of nodes that have replied to us. This works because of the way we are using events to send messages, we always offer an append entries event for the nodes contained within *noRelies*. Hence if a node is not in this set it has revived an RPC within the

last leader timout, but we need to re add it so it may receive another during the next.

$$SendHeartbeats(\dots) = Leader(\dots, leaderTimeout, \dots, nodes \setminus \{n\})$$

3.5 Log replication

In this section we explain the details of the log replication part of the the Raft protocol.

The leader node in charge of replicating its log. After a new leader is elected it assumes that every node has replicated everything already. Through sending append entry RPCs and observing the reply, the leader can work out how much of the log is actually replicated, and adjusts which parts of its log to send to each node. We make an abstraction that a leader can only send one entry at a time. This makes the model simpler, and reduces the state space.

The steps for a leader calculating the next log entry to send are outlined here:

- The leader sends an entry, E_i, with the preceding entry E_{i-1}'s term and index as parameters in the RPC.
- A follower receives this message, then checks if the last entry in its log E_{i-1}. It does this by comparing the index and term of the last entry with E_{i-1}'s index and term given by the RPC.
- If they match then the follower appends the new entry and sends an RPC reply saying it was successfully copied.
- If they do not match then the follower rejects the entry, and sends an RPC saying it was not successfully copied. If the follower has an element where E_{i-1} should be, then it deletes it and any items after it in the log (as the leader's log is always the one to replicated).
- The leader then receives this reply, and updates the next index and match index for the follower.
- If the message is a successful copy then the match index is set to the index of E_i, and the next index is incremented by one.
- Otherwise the entry was rejected because the wrong entry was sent. This means the next index for this follower is decremented by one, meaning next time the leader will try to send E_{i-1} to the follower. The index and term in this case will be E_{i-2}'s.

This process repeats until a matching entry is found in the follower, at which point the log can be successfully replicated.

3.5.1 Leader

The Leader node sends append entries to each follower. As described, the leader only sends the part of the log that it believes is not replicated yet, which is given by the next index variable for each follower. Then, depending on the response,

we update this index variable. To model this behaviour we need to keep track of the next index for every follower.

The next index and match index are stored as sequences, where the the next index for node i is at the ith index of the sequence. Using this we are able to construct the part of the leader process that deals with the log replication. Interestingly the calculation in the model is all carried out in a single event. Here we can see that the event offered depends firstly on which node we are sending to, and the log entry to send follows from that. Note how the $precIndex$ is set to the index preceding the entry we are sending. The $precTerm$ is set to the term from the entry at the $precIndex$ position in our own log.

$$
\begin{aligned}
&Leader_A(\dots) = \\
&\quad appendEntries!n?to : noReplies!term!(nextIndex!!to\ -1)\dots \\
&\qquad \dots!(log!!(nextIndex!!to\ -1))!commitIndex!(nextIndex!!to == length) \\
&\quad \rightarrow Leader(\dots) \\
&\ \Box\ Leader_B(\dots)
\end{aligned}
$$

The leader also needs to be able to deal with an append entries reply in the correct way. We first consider the case where the append entries RPC was unsuccessful. Here we simply decrement the next index. If the next index is already at one then we do not decrement but just leave it (as the next index can never be lower than one)

$$
\begin{aligned}
&Leader_B(\dots) = \\
&\qquad appendEntriesReply?from : Not_n!n!term!False \\
&\qquad\quad \rightarrow (Leader(\dots) \\
&\qquad\qquad \triangleleft nextIndex!!from = 1 \triangleright \\
&\qquad\qquad\ Leader(\dots, decrement(nextIndex, from), \dots)) \\
&\qquad \Box\ Leader_C(\dots)
\end{aligned}
$$

Now we consider the case for when an append entries RPC was successful. Here we first update the next index process, before deciding whether or not we have finished replicating the log. We detect this by performing the check $logLength > log!!from\ + 1$, which if true then we still have more to copy. If we have finished then we remove the follower form the $noReplies$ set, as we no longer need to send append entries RPCs to it.

$$
\begin{aligned}
&Leader_C(\dots) = \\
&\quad appendEntriesReply?from : Not(n)!n!term!True \\
&\qquad \rightarrow (Leader(\dots, increment(nextIndex, from), \\
&\qquad\qquad\qquad update(macthIndex, from, nextIndex!!from)) \\
&\qquad\quad \triangleleft logLength > log!!from\ + 1 \triangleright \\
&\qquad\quad\ Leader(\dots, noReplies \setminus from, increment(nextIndex, from), \\
&\qquad\qquad\qquad update(macthIndex, from, nextIndex!!from)) \\
&\quad \Box\ Leader_D(\dots)
\end{aligned}
$$

3.5.2 Follower

A follower node will only accept an entry from the leader if the leader sends the correct preceding entry. To do this we get the entry from our own log at the same index and compare terms. If the terms do not match then we delete the entry in our log and any after it, and reply that it was unsuccessful. Otherwise we check if an entry is being sent, and if so add it to our log. Either way we reply indicating a successful append entries has been sent.

$$
\begin{aligned}
&Follower_C(\dots) = \\
&\quad appendEntries?from : Not(n)!n!term?precIndex?precTerm?LCI?entry \\
&\qquad \rightarrow \begin{cases}
appendEntriesReply!n!from!term!False & precIndex > |log| \\
\quad \rightarrow Follower(\dots) & \\
heartbeatReply!n!from!term!False & precIndex \leq |log| \ \& \\
\quad \rightarrow Follower(\dots)) & precTerm \neq log!!precIndex \\
\\
heartbeatReply!n!from!term!True & precIndex < |log| \ \& \\
\quad \rightarrow Follower(\dots)) & precTerm = log!!precIndex \\
\\
heartbeatReply!n!from!term!True & precIndex = |log| \ \& \\
\quad \rightarrow Follower(\dots log\hat{}\langle term\rangle \dots) & precTerm = log!!precIndex \\
& \ \& \ entry = True \\
\\
heartbeatReply!n!from!term!True & precIndex = |log| \ \& \\
\quad \rightarrow Follower(\dots)) & precTerm = log!!precIndex \\
& \ \& \ entry = False
\end{cases}
\end{aligned}
$$

3.6 Committed Entries

The final piece of Raft is knowing when nodes are able to apply transitions in their log to their state machine. To maintain consensus between the nodes, we are only able to apply transitions after they have been replicated to a majority of the nodes in the network.

Recall the leader keeps a match index for each follower, which keeps track of the log index we have copied to that follower. From this the leader may work out which values can be committed. The leader checks for the highest index that has been replicated across a majority of nodes, it then communicates this value in the append entries it sends to its followers.

This behaviour can be modelled by introducing an auxiliary function that that operates on sequences. This function takes the *matchIndex* sequence and finds the largest index that is on over half the nodes. This is then used by the leader process to determine if the commit index should be updated or not. If the there is a new highest index we then update *commitIndex*, which will get propagated to all nodes though append entries RPCs.

$$ElemsGreaterThan(matchIndex, i) = length(\langle x \mid x \leftarrow matchIndex, x > i \rangle)$$
$$HighestCIndex(matchIndex) = length(\langle x \mid x \leftarrow indexes, ElemsGreaterThan(matchIndex, x) > nodes/2 \rangle)$$

$$\begin{aligned} Leader_D(\dots) = {} & \texttt{let } matchIndex' = HighestCIndex(matchIndex) \texttt{ in} \\ & (matchIndex' > commitIndex) \ \&\ Leader(\dots, matchIndex', \dots) \\ & \Box\ Leader_E(\dots) \end{aligned}$$

3.7 Common Elements

For each states a raft node can be in, we have shown the details of how they work with expected messages. This section deals with receiving RPCs and RPC relies that are unexpected. These are RPCs or replies that are from a node with term higher or lower than our own. We have already considered when followers vote for a candidate which is in a higher term than itself. In fact all nodes respond this way to a request vote RPC from a node with a higher term as this indicates the current node being out of date.

3.7.1 Receiving old RPCs

First we begin by considering messages from nodes with terms below the current node. In this case we simply do nothing, and reply if we need to. This is because any information from a term below is old, possibly a delayed message. Importantly when replying we send an updated term, this means the sending node will update its own term value and be able to become upto date. This response to terms lower than our own is common to all of the states. As the response is identical we use a single process to model all the lower termed events. Here we pass the node process so any state may use this process and go back to being in that state.

$$\begin{aligned} Follower_B(\dots) &= LowerTermMsg(Follower(\dots), n, term) \\ &\Box\ Follower_C(\dots) \\ Candidate_B(\dots) &= LowerTermMsg(Follower(\dots), n, term) \\ &\Box\ Candidate_C(\dots) \\ Leader_F(\dots) &= LowerTermMsg(Leader(\dots), n, term) \\ &\Box\ Leader_G(\dots) \end{aligned}$$

The process itself simply offers all the receive message events from nodes with a lower term, before returning back to the node process given in the parameter. Note we also have to send the node ID and node term, this is because we cannon

access this information from the process also being passed.

$$
\begin{aligned}
&LowerTermMsg(NodeProcess, n, term) = \\
&\quad appendEntries?from : not_n!n?term' : below_{term}?\_?\_?\_?\_ \\
&\qquad \rightarrow appendEntriesReply!n!from!term!False \\
&\qquad\quad \rightarrow NodeProcess \\
&\quad \Box\ requestVote?from : not_n!n?term' : below_{term}?\_?\_ \\
&\qquad \rightarrow requestVoteReply!n!from!term!False \\
&\qquad\quad \rightarrow NodeProcess \\
&\quad \Box\ appendEntriesReply?from : Not_n!n?term' : below_{term}?\_ \\
&\qquad \rightarrow NodeProcess \\
&\quad \Box\ requestVoteReply?from : Not_n!n?term' : below_{term}?\_ \\
&\qquad \rightarrow NodeProcess
\end{aligned}
$$

3.7.2 Receiving newer RPCs

Now we consider receiving an RPC or reply with a term that is greater than our own. Again the behaviour is common with all nodes. For receiving an RPC reply we simply update our term and convert to a follower state. This new follower state has other updated parameters, we have reset the *timeout* and set the *votedFor* to null (as we do not vote for anyone). For RPCs we perform identical behaviour but respond false in our RPC reply. This works in a similar way to the lower termed messages but this time we do not need to keep which state the node was in. Here we always go back to behaving as a follower and need to access the state's parameters. The result is the following process:

$$
\begin{aligned}
&HigherTermMsg(n, term, log, \dots) = \\
&appendEntries?from : not_n!n?term' : aboveterm?\_?\_?\_?\_ \\
&\quad \rightarrow appendEntriesReply!n!from!term!False \\
&\qquad \rightarrow Follower(n, term, log, null, electionTimout, \dots) \\
&\Box\ requestVote?from : not_n!n?term' : above_{term}?\_?\_ \\
&\quad \rightarrow requestVoteReply!n!from!term!False \\
&\qquad \rightarrow Follower(n, term, log, null, electionTimout, \dots) \\
&\Box\ appendEntriesReply?from : Not_n!n?term' : above_{term}?\_ \\
&\quad \rightarrow Follower(n, term, log, null, electionTimout, \dots) \\
&\Box\ requestVoteReply?from : Not_n!n?term' : above_{term}?\_ \\
&\quad \rightarrow Follower(n, term, log, null, electionTimout, \dots)
\end{aligned}
$$

Then we update the state processes to the following:

$$\begin{aligned} Follower_C(\dots) &= HigherTermMsg(n, term, log, \dots \\ &\quad \Box\ Follower_D(\dots) \\ Candidate_C(\dots) &= HigherTermMsg(n, term, log, \dots \\ &\quad \Box\ Candidate_D(\dots) \\ Leader_G(\dots) &= HigherTermMsg(n, term, log, \dots \\ &\quad \Box\ Leader_H(\dots) \end{aligned}$$

It is now the case that every state can receive every type of message, so now we detail how to construct the network process.

3.8 Putting It All Together

To construct the system we first have to specify what we want our preliminary variables to be. We can change these values to construct a slightly different system to verify later.

$$\begin{aligned} maxNodes &= 3 \\ maxTerm &= 4 \\ maxIndex &= 5 \end{aligned}$$

We now define the alphabet for a single node, these are all the RPC messages and replies that it can both send and receive:

$$\begin{aligned} SendAlpha(n) &= \{\!|appendEntries.n, appendEntriesReply.n, \\ &\qquad requestVote.n, requestVoteReply.n|\!\} \\ ReciveAlpha(n) &= \{\!|appendEntries.m.n, appendEntriesReply.m.n, \\ &\qquad requestVote.m.n, requestVoteReply.m.n \mid m \leftarrow nodes|\!\} \\ TotalAlpha(n) &= SendAplha(n) \cup ReciveAlpha(n) \end{aligned}$$

Next we create a process that controls the initial state of each process in the network. Again the choices of parameters here can be adjusted.

$$RunNode(n) = \begin{cases} Leader(1, 1, \langle\rangle, 1, 1, 0, 0, nodes \setminus n, \\ \quad \langle 1 \mid x \leftarrow nodes\rangle, \langle 0 \mid x \leftarrow nodes\rangle) & n = 1 \\ \\ Follower(n, 0, \langle\rangle, 1, 2, 0, 0) & otherwise \end{cases}$$

Finally we are now able to construct the network process that represents the total cluster running.

$$Network = \|_{i=1}^{maxNodes} (RunNode(i), TotalAlpha(i))$$

The result of constructing the network like this has a substantial impact on the way messages are sent and received in the model. By synchronising over the RPC events for the messages we introduce synchronous communication between

processes. For a node process a to send a RPC message to node process b, both a and b need to be offering the same event representing such a message. This means process a is unable to send an RPC if b is not ready to receive it, as there is no channel buffering in CSP. To avoid any unwanted blocking of messages, we have made sure every process is able to receive every message. This has an undesirable impact on the performance, which we will discuss later.

3.9 Modelling Failures

Now we have constructed the model for Raft, we turn our attention to how we can model failures in the system. We introduce both machine based failures which run dynamically, and communication based failures we make statically. We begin by explaining the communication based failures.

3.9.1 Communication

As mentioned above, the mechanism for sending messages is synchronous. This means we can introduce a new process which controls the communication of node process. The new process is simply offers only the events we want to restrict the node to be able send. If we want node a to only communicate through node b we can introduce the following process:

$$\begin{aligned} MessageBlock(a,b) = {} & appendEntries!a!b?\_?\_?\_?\_?\_ \rightarrow MessageBlock(a,b) \\ & \Box\ appendEntries!b!a?\_?\_?\_?\_?\_ \rightarrow MessageBlock(a,b) \\ & \vdots \\ & \Box\ requestVoteReply!b!a?\_?\_ \rightarrow MessageBlock(a,b) \end{aligned}$$

All that remains is to put the node process a in parallel with $MessageBlock$:

$$RestictedNode = RunNode(a) \underset{TotalAlpha(a)}{\parallel} MessageBlock(a,b)$$

The downside of implementing communications failure like this is we have to set it statically before running a refinement check on it. We have to do this because a running a verification check with this modelled dynamically would consider all possible subsets of nodes communicating. This would make our model unverifiable. As we explain later, this has an impact on the quality of the verification we can carry out.

3.9.2 Machines

Fortunately, it is possible to model machine failures dynamically without much change to the model. To do this we simply introduce a failed node process, which is able to receive messages, but does nothing with them and does not reply. Perhaps surprisingly this enough to model a failed node. We add the restriction that a node cannot fail and revive between two *tock* events, this is necessary to prevent divergence in the model (a node could fail and revive continually). We add two new channels, $fail.n$ and $revive.n$ to represent the failure and revive of node process n. In Raft a failed machine keeps its log and

term value from when it failed, hence the resulting failed node process is the following:

$$
\begin{aligned}
&FailedNode(n, log, term, votedFor) = \\
&\quad appendEntries?\_!n?\_?\_?\_?\_?\_ \\
&\qquad \rightarrow FailedNode(n, log, term, votedFor) \\
&\quad \Box\ appendEntriesReply?\_!n?\_?\_ \\
&\qquad \rightarrow FailedNode(n, log, term, votedFor) \\
&\quad \vdots \\
&\quad \Box\ tock \rightarrow FailedNode'(n, log, term, votedFor)
\end{aligned}
$$

$$
\begin{aligned}
&FailedNode'(n, log, term, votedFor) = \\
&\quad appendEntries?\_!n?\_?\_?\_?\_?\_ \\
&\qquad \rightarrow FailedNode'(n, log, term, votedFor) \\
&\quad \vdots \\
&\quad \Box\ tock \rightarrow FailedNode'(n, log, term, votedFor) \\
&\quad \Box\ revive.n \rightarrow Follower(n, term, log, votedFor, electionTimeout, 0, 0)
\end{aligned}
$$

All that remains is to adjust the node processes such that they can choice to transition to a failed node state. We only show the leader process for brevity, but $Follower_D$ and $Candiadte_D$ are defined identically.

$$
\begin{aligned}
&Leader_H(n, term, log, votedFor, \dots) = \\
&\quad fail.n \rightarrow FailedNode(n, log, term, votedFor)
\end{aligned}
$$

We name the network in which machines can fail as $Network_{failures}$.

4 Model evaluation

In this section we evaluate the model we have constructed. The main issue we have faced is the state complexity of the model; It has been difficult to develop good abstractions and design choices in order to make the model verifiable. Here we detail the abstractions we have made, and their implications on the model. We have been careful to not abstract away too many details, so we are still able to verify important properties.

4.1 Abstractions

Here we detail all of the abstractions we have made to the model. All of these abstractions have been made either because it is needed for CSP, or because they they make the model more verifiable.

Bounded Terms and Logs We have abstracted away the fact terms and indexes should both be unbounded. We are forced to this as we cannot possibly verify a model with any unbounded variable (infinite state space). This does not directly affect the model much, rather the usefulness of verifying properties. It may be the case that a failure happens when the index is n and we only every verify properties upto $n - 1$.

Cluster membership Similarly we have abstracted away from allowing new nodes to be added or removed from the cluster itself. This causes similar issues to bounding variables, we have may not have enough nodes to cause a failure. An example of this is shown in Understanding Concurrent Systems [5] with the Bully algorithm. In the authors model, the algorithm is verified with three nodes, but fails with four.

Message broadcasting Another abstraction is the way we send messages. In the Raft description they are sent out in parallel, however this would introduce a significant explosion in the state space of the model. Instead we allow any messages to be sent in any linearisation. The impact this has on the state space of the model is negated by implications of abstracting it way. If we pick an order to send the messages, then we have too much influence over the model we are verifying. A subtle error is much more likely to happen in a strange ordering, compared to us specifying an order.

Randomised timeout The random bounded timeout is also something we have abstracted, instead replacing with static variables $electionTimeout$ and $leaderTimeout$. This does not introduce intimidate problems, it does mean our model is susceptible to not being able to elect a leader. This is not a huge problem as this is a well known result of the consensus problem, an implication of the FLP impossiblity result [15].

Node processing time We abstract away any processing time between receiving an RPC and replying. To implement this we would still have to offer all message events whilst doing the processing, otherwise we would be blocking

the sender. This is because of the communication properties mentioned in the modelling failures section.

Unique RPCs We also abstract away some RPC call features one would normally expect a good implementation to have. In an implementation we would expect each RPC call to have an ID such that we know when we get a reply, we know what RPC this is in relation to. This does not cause problems in our model, and the implementation of these RPCs is an implementation detail, not something that Raft authors have mentioned are required.

Applying entries Another abstraction is not actually applying a committed entry to the state machine. It is enough to verify that a committed entry is in the same position in each nodes log, as this is the order that the transitions will be applied to the state machine.

Transitions in the log We have also abstracted away from storing transitions in the log entries. Adding the transitions would mean introducing another datatype. Even a simple boolean value would be enough to make the model unverifiable. If we consider the possible arrangements of a log where we have pairs of terms and transitions, we get a factor of $2^{maxIndex}$ times the arrangements of just the terms. This is a significant factor, and will be explained verification section.

4.2 Performance Improvements

As the complexity of our model is a concern, we make some changes to the way we store data to make it more FDR friendly. The way we do this is by abstracting out the log and leader indexes to external processes and interacting with them though channels. We then hide the events these channels use to make them internal. This also cleans up the code slightly as can be seen in the appendix.

The reason we do this is because FDR runs refinement checks in two stages:

- Compilation – In this stage FDR is constructing a transition system that represents both the specification and the model.
- Evaluation – In this stage FDR runs a breadth first search across the transition system, attempting to find a counterexample to the specification. If non is found then the specification passes.

We tackle the problems in the compilation stage, as FDR is not as optimised compared to the evaluation stage. The compilation only runs one thread of the CPU compared to the evaluation stage's ability to run multi-threaded, and on multiple cores. Because of this we reduce the amount of state stored in a process (the parameters) - this reduces the amount of processes that FDR has to compile. The worst performing parameters are sequences, as they are equivalent to having n^m different processes, where n is the size of sequence datatype, and m is the sequence length. This in itself is not the problem, the issue is we have sequences replicated across all the nodes in the cluster. For this reason we abstract away the log and leader indexes into a single, separate, process definition.

4.2.1 The Log

Rather than modelling the log variables explicitly, a log may modelled by a process. A log process offers events that represent the operations previously performed on the log. These events are shown below:

- $logGetTerm.index.term$ – Represents the $!!(log, index)$ operation.
- $logLength.length$ – Represents the length of the log.
- $logAppend.term$ – Represents the $(log, \langle term \rangle)$ operation
- $logRemoveFrom.index$ – Represents the $Take(log, index)$ operation

We now detail the process that models a working log. Firstly, if the log is empty then we can only perform the operation indicating how long the log is, or append a new entry. Hence we only offer both these events. In the case of an appending event happening, we add that entry to sequence in the parameter of the process.

$$\begin{aligned} Log(log, 0) = {} & logLength!0 \rightarrow Log(log, 0) \\ & \Box\ logAppend?entry \rightarrow Log(log^\frown\langle entry\rangle, 1) \end{aligned}$$

Now we consider the case when the log is not empty, here we offer all the events for the operations we can perform based upon what the log currently contains. The main purpose of the log process is to maintain the sequence as if it was a log stored on a node. In fact we manipulate the log in exactly the same way a node process does:

$$\begin{aligned} Log(log, length) = {} & logGetTerm?index : \{1..length\}!(log!!index) \\ & \quad \rightarrow Log(log, length)) \\ & \Box\ logLength!length \rightarrow Log(log, length) \\ & \Box\ logAppend?incTerm \\ & \quad \rightarrow Log(\langle incTerm\rangle^\frown log, length + 1) \\ & \Box\ logRemoveFrom?index : \{1..length\} \\ & \quad \rightarrow Log(Take(log, index - 1), index - 1) \end{aligned}$$

Now in a node process we need to use the events to access the log, rather than accessing it directly. We show this using $Leader_A$ as an example. Note how we are using the $logGetTerm$ and $logLength$ events to get the information we normally would have obtained straight from the log.

$$\begin{aligned} & Leader_A(\dots) = \\ & \quad logGetTerm!(nextIndex!!to\ -1)?prevTerm \\ & \qquad \rightarrow logLength?length \\ & \qquad\quad \rightarrow appendEntries!n!to!term!(nextIndex!!to\ -1) \\ & \qquad\qquad\qquad !prevTerm!commitIndex!(nextIndex <= length) \\ & \qquad\qquad \rightarrow Leader(\dots) \\ & \quad \Box\ Leader_B(\dots) \end{aligned}$$

As each node has its own log we need to put each node process in parallel with a log process. We first define the alphabet of the log process before adjusting $RunNode$.

$$LogAlpha = \{|logGetTerm, logLength, logAppend, logRemoveFrom|\}$$

$$RunNode_{Log}(n) = RunNode(n) \underset{LogAlpha}{\|} Log(\langle\rangle, 0)$$

This completes the change.

4.2.2 Leader Indexes

We also store the next and match indexes as sequences in the leader process, so we abstract these away as well. Here we use a different method, creating one process that can store a single value and run multiple copies of this one process to store multiple values. For the indexes themselves we only need getter and setter events:

$nextIndexGet$

$.follower$ – The follower's index we want to get

$.index$ – The index stored for the follower

$nextIndexSet$

$.follower$ – The follower's index we want to set

$.newIndex$ – The updated index for the follower

We also model the match index for every follower in the same way.

$matchIndexGet$

$.follower$ – The follower's index we want to get

$.index$ – The index stored for the follower

$matchIndexSet$

$.follower$ – The follower's index we want to set

$.newIndex$ – The updated index for the follower

As mentioned its only a single process that represents the store of one value, here are the processes for both the next and match indexes:

$$nIndexStore(follower, index) =$$
$$nextIndexGet!follower!index \rightarrow nIndexStore(follower, index)$$
$$\Box\ nextIndexSet!follower?newIndex \rightarrow nIndexStore(follower, newIndex)$$

$$mIndexStore(follower, index) =$$
$$matchIndexGet!follower!index \rightarrow mIndexStore(follower, index)$$
$$\Box\ matchIndexSet!follower?newIndex \rightarrow mIndexStore(follower, newIndex)$$

We now put them in the replicated parallel operator to construct each whole index. Note we have also initialised the indexes to 0 in this process.

$$nIndex = ||_{i=1}^{maxNodes} (nIndexStore(i, 0), \{|nextIndexSet.i, nextIndexGet.i|\})$$
$$mIndex = ||_{i=1}^{maxNodes} (mIndexStore(i, 0), \{|matchIndexSet.i, matchIndexGet.i|\})$$

We now show how its used in practice, again we use the $Leader_A$ process as an example:

$$\begin{aligned}
&Leader_A(\dots) = \\
&\quad nextIndexGet?to : noReplys?nextIndex \\
&\qquad \rightarrow logGetTerm!nextIndex - 1?prevTerm \\
&\qquad\quad \rightarrow logLength?length \\
&\qquad\qquad \rightarrow appendEntries!n!to!term!nextIndex - 1!prevTerm \\
&\qquad\qquad\qquad\qquad !commitIndex!(nextIndex <= length) \\
&\qquad\qquad\quad \rightarrow Leader(\dots) \\
&\quad \Box\ Leader_B(\dots)
\end{aligned}$$

This helps the compilation time differently to the log. Here we have abstracted the sequence away completely meaning its simple to compile the single process both indexes. We could apply the same method to the single log process to gain an even better compile time, however our work is enough for our means.

4.3 Validation of Model

Finally in this section we discuss the validity of the model we have constructed. A simple yet effective method of validating is comparing the Raft description to the behaviour of our model. This is carried out by simulating networks in the model and testing their execution paths to ensure they follow the Raft algorithm's written description. Also by carrying out the property checks we want to verify on the algorithm, it is clear when the model breaks because of implementation rather than Raft's design.

By following the exact description of the Raft algorithm, and being clear about any abstractions and their implications we can reasonably sure our model is representative of the Raft protocol. Using the probe tool in FDR we are able to gain an important insight into exactly how the nodes are behaving in the model.

During the construction of the model we performed simple refinement checks to making sure the model was behaving as we expected. The checks we carried out were important when new parts of the model were added, if suddenly one of the checks failed then we knew that we had introduced some unwanted behaviour or restricted behaviour required. These checks were:

- Processes are deterministic.
- Node processes offer all receive message events.
- Processes do not stop time.

One example where this came in useful is when we accidentally made the RPC calls synchronous, so caused a sending node process to be blocked while it was waiting for a RPC reply. This was picked up by the fact a node process must be able to receive RPC messages at any point.

5 Specifications

In this section we discuss the properties we test on Raft and their CSP specifications we use to verify the model. We also discuses the problem of state space when carrying out these refinement checks and how small changes to our model can have significant impact on the number of states.

5.1 Safety Properties of Raft

In this section we show the the properties of Raft we will be model checking.

To verify properties we introduce signal events into the model such that we can specify the properties in terms of these CSP signal events. First we use a signal event so a leader node can indicate it thinks it is the leader, and the term for which it thinks this.

$$Leader_H(\dots) = signalLeader!n!term \rightarrow Leader(\dots)$$

Next we add another signal event into each node process that signals when a log entry is committed. This event consists of an index and term from an entry that a node considers to be committed.

$$\begin{aligned} Leader_H(\dots) =& signalCommit?index : \{1..commitIndex\}!(log!!index) \\ & \rightarrow Leader(\dots) \end{aligned}$$

The signal events are what we use in our specifications

5.1.1 Specification One

Firstly we consider the main property of the leader election, the fact there should ever be only one leader per term. This property is important as if there are two leaders in the same term then there is potential for different log entries to be committed easily.

$$\begin{aligned} &Spec1(leader, maxTerm) = STOP \\ &Spec1(leader, term) = signalLeader!leader!term \rightarrow Spec1(leader, term) \\ &\quad \Box\ signalLeader?leader?term' : above(term) \rightarrow Spec1(leader, term') \end{aligned}$$

Passing this specification means Raft never elects two leaders for the same term. This is important for fault tolerance as having the possibility of getting two leaders at the same time causes many subtle considerations. By eliminating this from occurring, Raft doesn't have to have a mechanism for dealing with such a situation, which makes Raft simpler and easier to understand.

5.1.2 Specification Two

Next we introduce a specification for the log replication. Here we represent the property that committed entries should in the same index in all nodes. We check this by checking the term of the entry regarded as committed. This process keeps a log of the first committed term for each index it has seen, it

then only allows the signalling that of this first term at that particular index.

$$
\begin{aligned}
&Spec2 = signalCommit?index?term \rightarrow Spec2(\langle term \rangle) \\
&Spec2(clog, maxIndex) = STOP \\
&Spec2(cLog, length) = signalCommit?index : 1..length!(cLog!!index) \\
&\qquad\qquad \rightarrow Spec2(cLog, length) \\
&\qquad\qquad \Box\ signalCommit?indexlength + 1..maxIndex?term \\
&\qquad\qquad \rightarrow Spec2(cLog\hat{}term)
\end{aligned}
$$

This specification is important as it is checks the main property about a consensus algorithm. This check ensures that the state machines across the nodes in Raft are consistent with each other. To satifsy this property means Raft is working as expected. A counterexample means there is an execution path that can lead to an inconsistency, and the Raft algorithm will need to be revised.

With the specification above it would be good to be able to check transitions rather than the term it was added. This is a slightly stronger property about the log. However due to the abstractions we have made we cannot do this.

5.1.3 Split Brain

A split brain is where a network is severed into two halves. The two halves then continue to run amongst themselves. When running separately it is possible for algorithms to violate safety rules about the cluster as a whole. To model this in CSP we can use the $MessageBlock$ process to block communication. We set $maxNodes$ to 4 and and construct the split brain network. In this network nodes 1 and 2 can communicate, and so can 3 and 4, but they cannot communicate between the pairs.

$$
\begin{aligned}
SplitBrain = &\ MessageBlock(1,3) \parallel Messageblock(1,4) \\
&\parallel Messageblock(2,3) \parallel Messageblock(2,4)
\end{aligned}
$$

$$
Network_{split} = Network_{failures} \underset{TotalAlpha}{\parallel} SplitBrain
$$

5.2 Verifying the System

Now we have the model and specifications, all that remains is run the refinement checks in FDR. In this section we give the details of how the maximum size of variables in the model affects the ability to verify properties. We shall see that adding a single node causes the state space to increase by a factor of 10.

We perform the refinement check on all of the networks we have constructed. We use traces-refinement as the properties we are verifying are safety properties. all events other than signal events hidden from the network. The refinement checks look like the following:

$$
\begin{aligned}
&Spec1(0,0) \sqsubseteq_{\mathrm{T}} (Network \setminus totalAlpha) \\
&Spec1(0,0) \sqsubseteq_{\mathrm{T}} (Network_{failures} \setminus totalAlpha) \\
&Spec1(0,0) \sqsubseteq_{\mathrm{T}} (Network_{split} \setminus totalAlpha)
\end{aligned}
$$

All of these refinement checks pass, but they only terminate for small values of $maxTerm$, $maxNode$ and $maxIndex$. Above these small values the problem the complication space is to large.

The second refinement checks follow the same pattern as above:

$$Spec2 \sqsubseteq_{\mathrm{T}} (Network \setminus totalAlpha)$$
$$Spec2 \sqsubseteq_{\mathrm{T}} (Network_{failures} \setminus totalAlpha)$$
$$Spec2 \sqsubseteq_{\mathrm{T}} (Network_{split} \setminus totalAlpha)$$

We find All of these specifications also pass, again only for small values of the given parameters. We discuss the implications of this later, but for now Raft has lived up to its safety claims. This is a good result for Raft, however much the author would have liked to have found a counterexample!

5.2.1 Analysis and Evaluation of Verification

There are two factors at play when using FDR to verify properties. One is the complexity of the processes, which affects the compile time. We have alleviated a major part of this problem by abstracting away the log and leader indexes into their own processes. Despite the log being used in the same way, we reduce the compile time by a factor of 3. This is because there were three unique processes with a log before, the three states. Now there is just one unique process, hence the saving.

The secondary problem is the size of the state space when we evaluate the equivalent transition system in FDR. We found that a major source of the state space explosion was in the values given to $leaderTimout$ and $electionTimeout$. We found that a single increment in the $electionTimeout$ resulted in approx. 7 times more states required to check the property and $leaderTimeout$ approx. 4 times. This was tested using 3 nodes and 3 terms. To combat the effect of state space it is enough to reduce these down such that $leaderTimer$ is 1 and $electionTimer$ is 2. This has a minimal effect on the model as there is not a substantive difference between a timeout happening sooner or later. What does matter is the relation between the two timeouts - obviously if $leaderTimer$ is longer than $electionTimeout$ then followers will timeout very often.

One main sticking point is the fact we have only verified for small values of $maxTerm$ and $maxIndex$, more verification tests need to be carried out on larger values. Without the abstraction of the log and indexes or the timeout reduction this model would be unverifiable, meaning the abstractions we have made are just enough to be able to verify this model.

Despite these problems with FDR we have successfully show raft satisfies two key properties it claims to, for up to 5 nodes, 4 terms and 4 indexes. We have demonstrated that Raft is safe even under a split brain scenario. The reason Raft is safe under the split brain is the fact a candidate needs a majority of the node's votes to become leader, in network with four nodes it still requires 3 votes (including its own).

6 Conclusions

This project has used Raft as a case study for verifying distributed algorithms in CSP, and has shown the difficulties faced when modelling a distributed algorithm. The main problem is the size of the state space produced by a model of a distributed algorithm. This stems from the fact nodes have to use messages or RPCs to communicate with each other, and need to store state information. Whilst we have shown techniques that can be used to reduce the space of latter - by abstracting some state into common processes, there is no such clear workaround for the former. The more messages and the more information transmitted the worse the problem becomes.

One abstraction made to face state-space that the author would have liked to avoided is the log entries not containing a transition. This abstraction means we are unable to verify more properties about the log replication. In the authors opinion this is the best improvement that could be made to the project.

Another important consideration is the fact we are forced put a bound on the size of variables, and the number of nodes in the cluster. If the cluster fails to satisfy a property when a variable is set above k and we only test up to $k-1$, we will not be able to find this counterexample. It may be the case an inductive argument can be made for the size of the variables, in which case we would know the minimum value we need to set them to.

However, despite these difficulties, we have successfully demonstrated how to construct a model for a distributed algorithm in CSP. We have shown the model is a good representation of Raft, and verified properties about the model. We have also highlighted specific abstractions made when developing the model and how these have affected the usefulness of the model. In effect we have constructed a model that is both verifiable and meaningful; and have shown that Raft's leader election and log replication assertions are correct in our model.

6.1 Further Work

Given more time for this project the author would like to explore the less well depicted features of Raft such as cluster membership changes, log compaction and client interaction. Further to this it would be interesting to model a specific implementation of Raft such as CoreOS [14] or Consul [16].

Going even further, it would be worth comparing a few distributed algorithm models and their state space. It would be beneficial to the field to pinpoint the exact causes of the state space explosion with respect to CSP models.

Acknowledgements

I would like to thank Thomas Gibson-Robinson for suggesting this project back in June last year, and guiding me though what has been a very interesting and rewarding subject matter. Particular thanks must be given for proof-reading many drafts of this project.

References

[1] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, p. 11.

[2] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 335–350.

[3] I. Konnov, H. Veith, and J. Widder, "Who is afraid of model checking distributed algorithms?" 2012.

[4] G. Delzanno, M. Tatarek, and R. Traverso, "Model checking paxos in spin," in *ArXiv*, 2014.

[5] A. Roscoe, *Understanding Concurrent Systems*, 1st ed. New York, NY, USA: Springer-Verlag New York, Inc., 2010.

[6] D. Burke, "All circuits are busy now: The 1990 at&t long distance network collapse," nov 1995.

[7] A. W. Services. Summary of the october 22, 2012 aws service event in the us-east region. [Online]. Available: https://aws.amazon.com/message/680342/

[8] L. Lamport, "The Part-time Parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.

[9] ——, "Paxos Made Simple," *SIGACT News*, vol. 32, no. 4, pp. 51–58, Dec. 2001.

[10] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[11] K. Kingsbury. Project jepson. [Online]. Available: https://aphyr.com/tags/Jepsen

[12] T. G.-R. P. A. A. B. A. Roscoe, "FDR3 — A Modern Refinement Checker for CSP," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, E. brahm and K. Havelund, Eds., vol. 8413, 2014, pp. 187–201.

[13] T. Gibson-Robinson, "Concurrency." [Online]. Available: https://www.cs.ox.ac.uk/teaching/materials15-16/concurrency/

[14] I. CoreOS. Coreos, inc. [Online]. Available: https://www.coreos.com

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[16] Hashicorp. Consul. [Online]. Available: https://github.com/hashicorp/consul

Appendix