Technical Report

UCAM-CL-TR-857
ISSN 1476-2986

Number 857

ARC: Analysis of Raft Consensus

Heidi Howard

July 2014

This technical report is based on a dissertation submitted May 2014 by the author for the degree of Bachelor of Arts (Computer Science Tripos) to the University of Cambridge, Pembroke College.

ISSN 1476-2986

Abstract

The Paxos algorithm, despite being synonymous with distributed consensus for a decade, is famously difficult to reason about and implement due to its non-intuitive approach and underspecification. In response, this project implemented and evaluated a framework for constructing fault-tolerant applications, utilising the recently proposed Raft algorithm for distributed consensus. Constructing a simulation framework for our implementation enabled us to evaluate the protocol on everything from understandability and efficiency to correctness and performance in diverse network environments. We propose a range of optimisations to the protocol and released to the community a testbed for developing further optimisations and investigating optimal protocol parameters for real-world deployments.

Acknowledgements

I have thoroughly enjoyed working on this project and my final year at Cambridge, it wouldn't have been possible without the help and support of the following wonderful people: Anil Madhavapeddy, not only have you been an excellent supervisor but you have been my mentor over the last two years and Diego Ongaro, author of Raft, who has taken his time to support me in reproducing his results and sharing his ideas. I would also like to thank Malte Schwarzkopf and Chris Hadley for providing excellent feedback on my dissertation, which forms the basis of this technical report.

Contents

List of Figures

Chapter 1

Introduction

This technical report describes the implementation, analysis and refinement of the Raft consensus algorithm, for building fault-tolerant applications on a cluster of replicated state machines.

1.1 Motivation

Distributed consensus builds reliable systems from unreliable components, it forms the foundation of many distributed systems and is a fundamental problem in computer science. Its applications range from fault-tolerant databases, lock managers, group membership systems and leader election to terminating reliable broadcast and clock synchronization, not forgetting Greek parliamentary proceedings [14] and generals invading a city [19]. The subject of this report is the implementation of fault-tolerant applications by replicated deterministic state machines and co-ordinating transitions, known as the *replicated state machine approach*.

Leslie Lamport's Paxos protocol [14] is at the heart of distributed consensus in both research and industry. Despite being synonymous with consensus for a decade, it is famously difficult to understand, reason about and implement because of its non-intuitive approach and the lack of universal agreement on the algorithm for multi-Paxos.

Lamport's original description of Paxos [14], though highly cited, is notoriously difficult to understand and therefore has led to much follow up work, explaining the protocol in simpler terms [15, 36] and optimising it for practical systems [18, 17, 25]. Lamport went on to sketch some approaches to Multi-Paxos, based on consecutive runs of Paxos, but its under-specification led to divergent interpretations and implementations.

The search for a more understandable consensus algorithm is not simply about easing the right of passage for computer science undergraduates or aiding software developers but also about bridging the divide between the formal verification and system communities.

Raft [28, 27] by Diego Ongaro and John Ousterhout promises distributed consensus, which provides the same guarantees as Multi-Paxos under the same assumptions, with enhanced understandability from an intuition-focused approach and proof of safety.

On first sight, Raft is the silver bullet to a decade old problem in distributed systems. Since the draft was first released online, it has been popular with developers, leading to a dozen implementations. However at the time of writing, the Raft paper has yet to be published at a peer reviewed venue[1] and therefore there does not exist any formal follow-up literature.

This project will evaluate the claims made by the paper, from its usability to correctness. Our conclusions will be the first published assessment of the protocol and hopefully will either spark a new direction in distributed consensus or demonstrate its shortcomings.

1.2 Related Work

We have seen that distributed consensus is fundamental to today's computer systems. Now we consult with the two schools of thought, the theory of distributed consensus which demonstrates that true consensus is impossible to achieve and the discipline of engineering practical solutions, which aims to achieve the impossible by reshaping our assumptions and guarantees. Figure 1.1 highlights the key developments in this field.

1.2.1 Definition of Consensus

Consensus requires the following conditions to be met:

1. Agreement: all correct nodes[2] arrive at the same value *(the safety property)*;

2. Validity: the value chosen is one that was proposed by a correct node *(the non-triviality property)*;

3. Termination: all correct nodes eventually decide on a value *(the liveness property)*.

Traditional consensus problems require that, in an asynchronous system, agreement and validity must be met regardless of the number of non-Byzantine failures [29]

[1]The paper has now appeared in 2014 USENIX Annual Technical Conference, June 2014.

[2]A *correct node* is a node that is currently running, so either it has not fail-stopped or has restarted after a fail-recover.

1985 FLP Impossibility Result [8].

1988 Oki and Liskov's Viewstamped Repication [26].

1998 Lamport's original Paxos paper "Part-Time Parliment" [8].

2000 Brewer proposes CAP conjecture during Keynote at PODC [2].

2002 Proof of CAP by Gilbert and Lynch [10].

2005 Lamport's Technical Report on Generalized Consensus & Paxos [16].

2013 Raft Consensus Paper first available online [28].

Figure 1.1: Timeline of the key developments in distributed consensus.

and all three must be met given the number of non-Byzantine failures is less than a threshold value. The number of nodes which are required to participate in agreement is known as the *Quorum size*. For multi-Paxos (§1.2.5), Viewstamped Replication (§1.2.6) and Raft (§2.2), to tolerate at most f failures, the quorum size is $f + 1$ for a cluster of $2f + 1$ nodes. The quorum intersection property ensures that for any two quora from the $2f + 1$ nodes there will be an intersection of at least one node. Thus if we operate on any quorum, at least one member of any future quorum will be aware of the previous operation.

1.2.2 CAP theorem: Conjecture, Correctness & Controversy

The CAP conjecture [2] was proposed by Eric Brewer in 2000, it argued that it is necessary to compromise at least one of Consistency, Availability and Partition Tolerance in a distributed system.

CAP was formally proven two years later [10] operating in asynchronous conditions by considering two nodes either side of a network partition. If these two nodes receive two conflicting client requests then they must both accept the requests, thus compromising Consistency or at least one of them must not accept the request, thus compromising Availability.

There has been much confusion over the CAP theorem, as noted by Brewer [1] himself. Brewer argues that the *two of three* formula is misleading as it oversimplifies the complex interactions between the properties. Partitions are often rare and consistency is not a binary concept, but a whole spectrum of models from *Atomicity, Consistency, Isolation and Durability* (ACID) to *Basically Available Soft-state services with*

Eventual-consistency (BASE). The consistency in CAP was intended by Brewer to be single-copy consistency [1].

1.2.3 FLP Impossibility Result

The FLP result [8] proves that it is not possible to guarantee consensus on a single value, if one or more processes is fail-stop faulty, in an asynchronous system with reliable communication. The asynchronism of the system means that we cannot reliably differentiate between a slow process or message and a failed node, even if we assume that communication is reliable.

1.2.4 Two and Three Phase Commit

In two-phase commit a co-ordinator attempts to commit a value in two distinct phases: the *proposal* phase where a co-ordinator contacts all other nodes to propose a value and collect responses, either a commit or abort from each node and the *commit* phase where the co-ordinator tells all other nodes whether to commit or abort the previously proposed value. If a co-ordinator fails during the *commit* phases, then another node could try to recover by asking nodes how they voted in the *proposal* phase. However if an additional node has failed we cannot know who it voted for and we cannot simply abort the transaction as the co-ordinator may have already told the failed node to commit and therefore we must block until the node recovers.

Three-phase commit [32] provides liveness by adding a third *prepare to commit* phase between the *proposal* and *commit* stages. This phase has the co-ordinator send *prepare to commit* messages to all nodes and proceed to the *commit* phase as long as all nodes acknowledge receipt, otherwise it aborts the transaction. If the co-ordinator fails during the *commit* stage, the nodes will timeout waiting for the *commit* phase and commit the transaction themselves, in the knowledge that all nodes must have agreed to commit the transaction for the protocol to reach this stage. If the co-ordinator fails during the *prepare to commit* or *proposal* phases then, safe in the knowledge that no node will have committed the transaction, the nodes will consider the transaction aborted.

Neither 2PC or 3PC allows for asynchronous systems or network partitions.

1.2.5 (Multi-)Paxos

Paxos considers the agreement over a single value, as shown in in Figure 1.2 and Figure 1.3. A node or set of nodes becomes a *Proposer(s)*, each generating a unique sequence number which is greater than any they have previously seen, for example $n + 1 || nodeID$, where n is the first element of the highest sequence number seen so

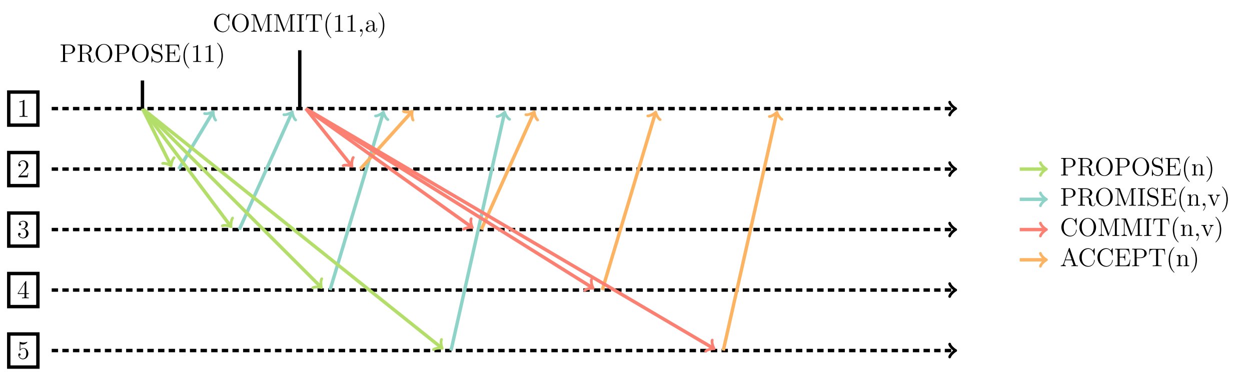


Figure 1.2: Successful Paxos: node 1 is the first proposer and all nodes promise the request as it is the first one they have seen. Node 1 is then free to commit a value of its choosing.

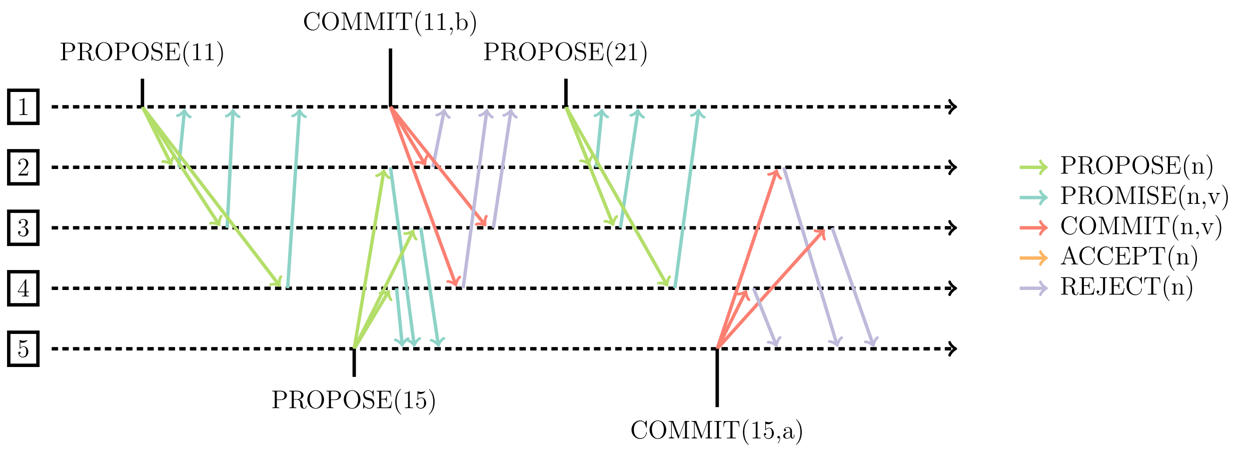


Figure 1.3: Duelling proposers in Paxos: initially node 1 gets the nodes to promise to sequence number 11. Node 5 then gets the nodes to promise to sequence number 15. When the nodes receive the *commit(11,b)* from node 1, they all reply with *reject(15)* since 15 is the last sequence number they have seen. In response, node 1 tries to propose again with sequence number 21 and the nodes promise. When the nodes receive the *commit(15,a)* from node 1, they all reply with *reject(21)* since the last sequence number they have seen is 21. This process could continue indefinitely.

far. The *proposer* broadcasts a *propose* message (including its sequence number) to all other nodes.

On receipt of a *propose* message, each node compares the sequence number to the highest number it has already seen. If this is the highest it has seen, the node replies with a *promise*, including the last value accepted and its sequence number, if any. Otherwise the node responds with a *reject* message and the highest sequence number it has seen. If a strict majority of nodes reply with a *promise* message then the *proposer* chooses the value received with the highest sequence number. If no values are received, then the *proposer* is free to choose a value. The *proposer* then broadcasts a *commit* message (including sequence number and value) to all nodes.

On receipt of a *commit* message, each node compares the sequence number to the highest number it has already seen. If the sequence number is the highest it has seen and if the value is the same as any previously accepted value, then the node replies with an *accept* (as demonstrated in Figure 1.2), otherwise the node responds with a *reject* message (as demonstrated in Figure 1.2) and the highest sequence number it has seen. If a strict majority of nodes reply with an *accept* message, then the value has been committed and the protocol terminates, otherwise the protocol starts again.

Multi-Paxos [15] is the chaining together of a series of instances of Paxos to achieve consensus on a series of values such as a replicated log. Multi-Paxos uses leaders, long-term proposers who are therefore able to omit the *propose* message when they were the last node to commit a value. This series of values is recorded by each node to build fault tolerant applications using the *replicated state machine approach* (detailed in §2.2) and to enable nodes which have fail-recovered to catch up. This is a notoriously complicated protocol to get to grips with, demonstrating the need for new consensus algorithms such as Raft.

1.2.6 Viewstamped Replication

Viewstamped Replication (VR) [26, 20] is the most similar of existing protocols to Raft Consensus, though we will not explain the protocol in detail here due to the considerable overlap with Raft Consensus. It is worth highlighting that VR uses round-robin for leader election instead of the eager algorithm used in Raft and that VR has all nodes actively communicating, allowing for interesting features such as the ability to operate without non-volatile storage, though it can be more difficult to reason about.

Chapter 2

Preparation

2.1 Assumptions

Raft consensus operates under the same set of assumptions as Multi-Paxos. The vast literature on Multi-Paxos details techniques to broaden the application, but here we are only considering classic Multi-Paxos. These assumptions are as follows:

1. Network communication between nodes is unreliable including network delays, partitions, and packet loss, duplication, and re-ordering;
2. Nodes have access to infinite persistent storage that cannot be corrupted and any write to persistent storage will be completed before crashing (known as write-ahead logging);
3. Nodes operate in an asynchronous environment with faulty clocks, no upper bound exists for the delay of messages and nodes may operate at arbitrary speeds;
4. Byzantine failures [29] cannot occur;
5. Nodes are statically configured with a knowledge of all other nodes in the cluster, cluster membership cannot change dynamically;
6. The protocol has use of infinitely large monotonically increasing values;
7. Clients of the application must communicate with the cluster via the current leader, it is the responsibility of the client to determine which one of them is currently leader. The clients are statically configured with knowledge of all nodes; and
8. The state machines running on each node all start in the same state and respond deterministically to client operations.

2.2 Approach

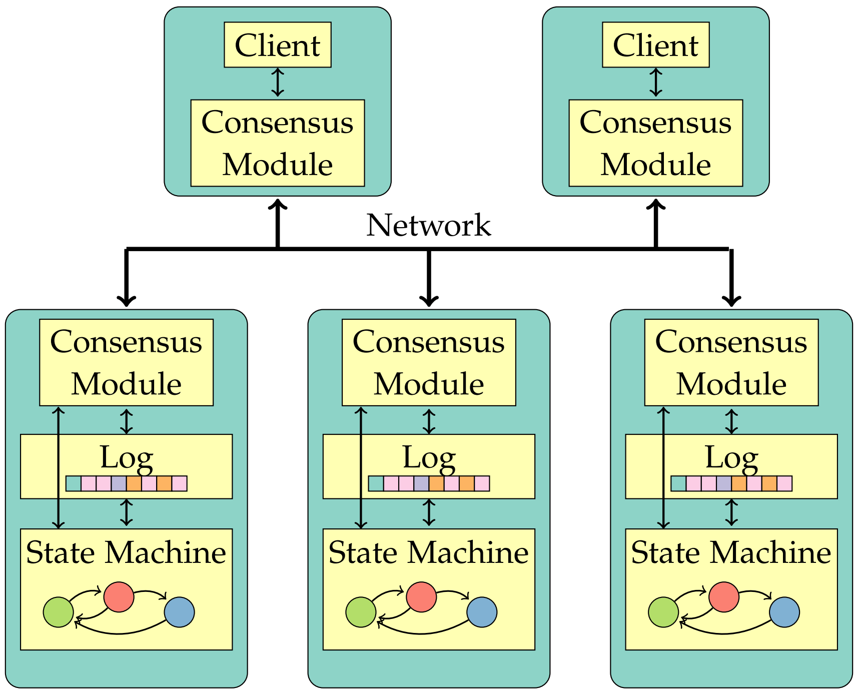


Figure 2.1: Components of the *replicated state machine approach*.

We will frame the challenge of distributed consensus in the context of a replicated state machine (Figure 2.1), drawing a clear distinction between the *state machine* (a fault-tolerant application), the *replicated log* and the *consensus module* (handled by the consensus protocol like Multi-Paxos or Raft).

The clients interact with the replicated state machine via commands. These commands are given to the consensus module, which determines if it is possible to commit the command to the replicated state machine and, if possible, does so. The state machine must be deterministic, so that when commands are committed the state machines remain identical. A fault-tolerant database is an example of one such application. Once a command has been committed, the consensus protocol guarantees that eventually the command will be committed on every live state machine and they will be committed in order. This provides linearisable semantics from the client, defined as each command from the client appearing to execute instantaneously, exactly once, at some point between its invocation and positive response.

This perspective on distributed consensus has been chosen as it mimics real-world applications. Zookeeper [13], the currently most popular open source consensus application and Chubby [4], a fault-tolerant, distributed locking mechanism used by applications such as the Google Filesystem [9] and Bigtable [6], both use the replicated state machine approach [5].

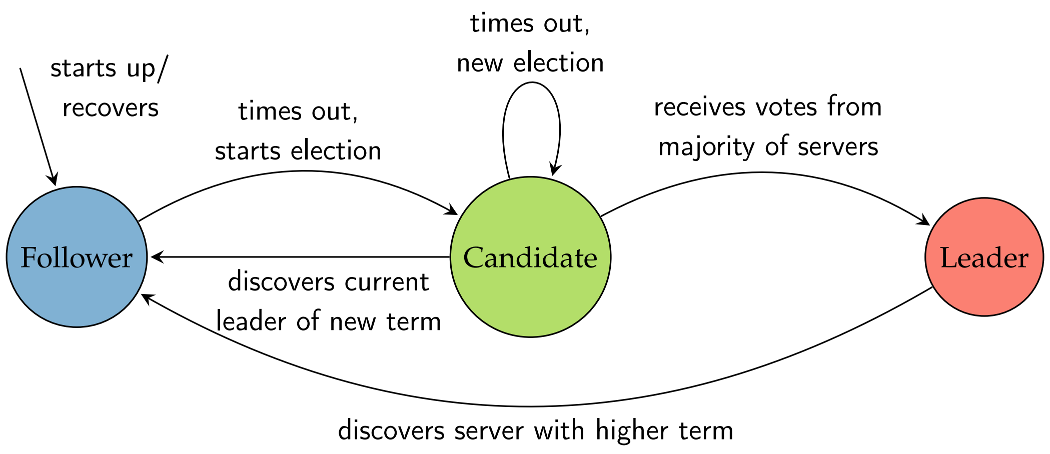


Figure 2.2: State transition model for Raft consensus.

In the pursuit of understandability and in contrast to the approach of Viewstamped Replication, Raft uses strong leadership, which extends the ideas of leader-driven consensus by adding the following conditions:

1. All message passing will be initialised by a leader (or a node attempting to become leader). The protocol specification makes this explicit by modelling communications as RPCs, emphasizing the clear roles of a node as distinctly either active or passive.

2. Clients must contact the leader directly to communicate with the system.

3. For a system to be available it is necessary (but not sufficient) for a leader to have been elected. If the system is in the process of electing a leader, even if all nodes are up, the system is unavailable.

2.3 Raft Protocol

2.3.1 Leader Election

Each node has a consensus module, which is always operating within one of the following modes:

- **Follower:** A passive node, which only responds to RPC's and will not initiate any communications.

- **Candidate:** An active node which is attempting to become a Leader, they initiate *RequestVote* RPC's.

- **Leader:** An active node which is currently leading the cluster, this node handles requests from clients to interact with the replicated state machine and initiates *AppendEntries* RPC's.

Events cause the nodes to transition between these modes, as shown by the nondeterministic finite automaton (NFA) in Figure 2.2. These events are all either temporal, such as the protocol's four main timers *Follower Timeout*, *Candidate Timeout*, *Leader Timeout* and *Client Timeout*. Otherwise they can be spatial, such as receiving *AppendEntries*, *RequestVote* or *ClientCommit* RPC's from other nodes.

Since we cannot assume global clock synchronization, global partial ordering on events is achieved with a monotonically increasing value, known as *term*. Each node stores its perspective of the *term* in persistent storage. A node's term is only updated when it (re-)starts an election or when it learns from another node that its *term* is out of date. All messages include the source node's *term*, which is checked by the receiving node. If the source's *term* is smaller, the response is negative. If the recipient's *term* is smaller, then its *term* is updated before parsing the message, likewise if their *terms* had been the same.

On startup or recovery from a failure, a node becomes a follower and waits to be contacted by the leader, which broadcasts regular empty *AppendEntries* RPC's. A node operating as a follower will continue to do so unless it fails to hear from a current leader or grant a vote to a candidate (details below) within its *Follower Timer*. If this occurs a follower will transition to a candidate.

On becoming a candidate, a node will increment its *term*, vote for itself, start its *Candidate Timer* and send a *RequestVote* RPC to all other nodes. As seen in the NFA (Figure 2.2) for Raft, there are three possible outcomes of this election. Either the candidate will receive at least a strict majority of votes and become leader for that term (Figure 2.3), or it will fail to receive enough votes and restart the election (Figure 2.4), or it will learn that its *term* is out of date and will step down (Figure 2.5). A follower will only vote for one node to be leader in a specific term. Since the vote is stored on non-volatile storage and term increases monotonically, this ensures that at most one node is the leader in a term (detailed in Figure 2.10).

2.3.2 Log Replication

Now that a node has established itself as a leader, it can service requests for the replicated state machines from the clients. Clients contact the leader with commands to be committed to the replicated state machine. On receipt the leader assigns a term and index to the command, that uniquely identifies the command in the nodes' logs. The leader then tries to replicate the command to the logs of a strict majority of nodes and, if successful, the command is committed, applied to the state machine of the leader and the result returned to the client. Figure 2.6 shows an example set of logs,

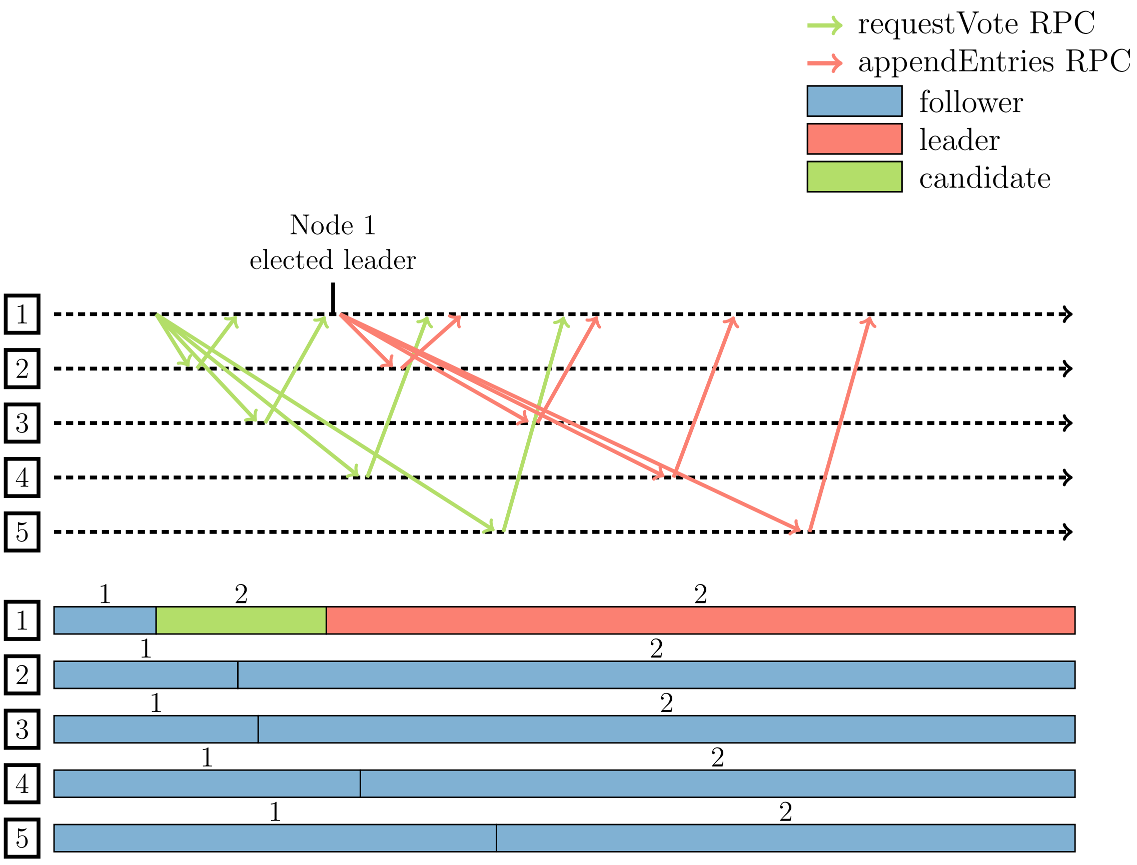


Figure 2.3: Example of a successful election: All nodes start and enter term 1 as *followers*. Node 1 is the first to timeout and step up to a *candidate* in term 2. Node 1 broadcasts *RequestVotes* to all other nodes, who all reply positively, granting Node 1 their vote for term 2. Once node 1 has received 3 votes, it becomes a Leader and broadcasts a heartbeat *AppendEntries* to all other nodes.

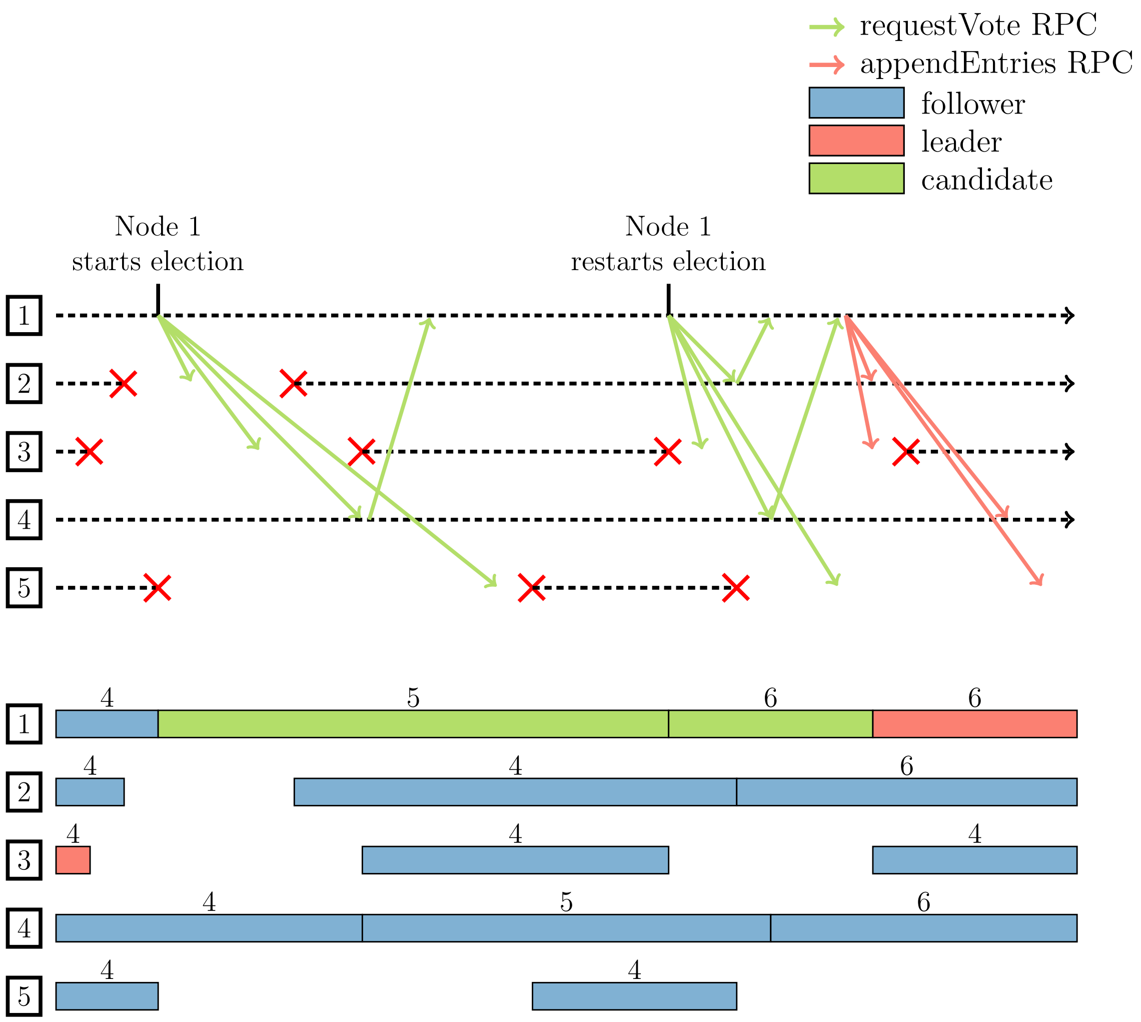


Figure 2.4: Example of an election timing out due to many nodes failing: At the start of this trace, node 3 is leader in term 4 and all nodes are aware of this but the nodes 2, 3 and 5 soon fail. Node 1 is the first node to timeout and start an election in term 5. Node 1 is unable to receive votes from nodes 2, 3 and 5 so eventually times out and restarts the election in term 6. In the meantime, node 2 has recovered and nodes 3 and 5 have recovered and failed again. Node 1 then gets votes from nodes 2 and 4 and wins the election to become leader in term 6.

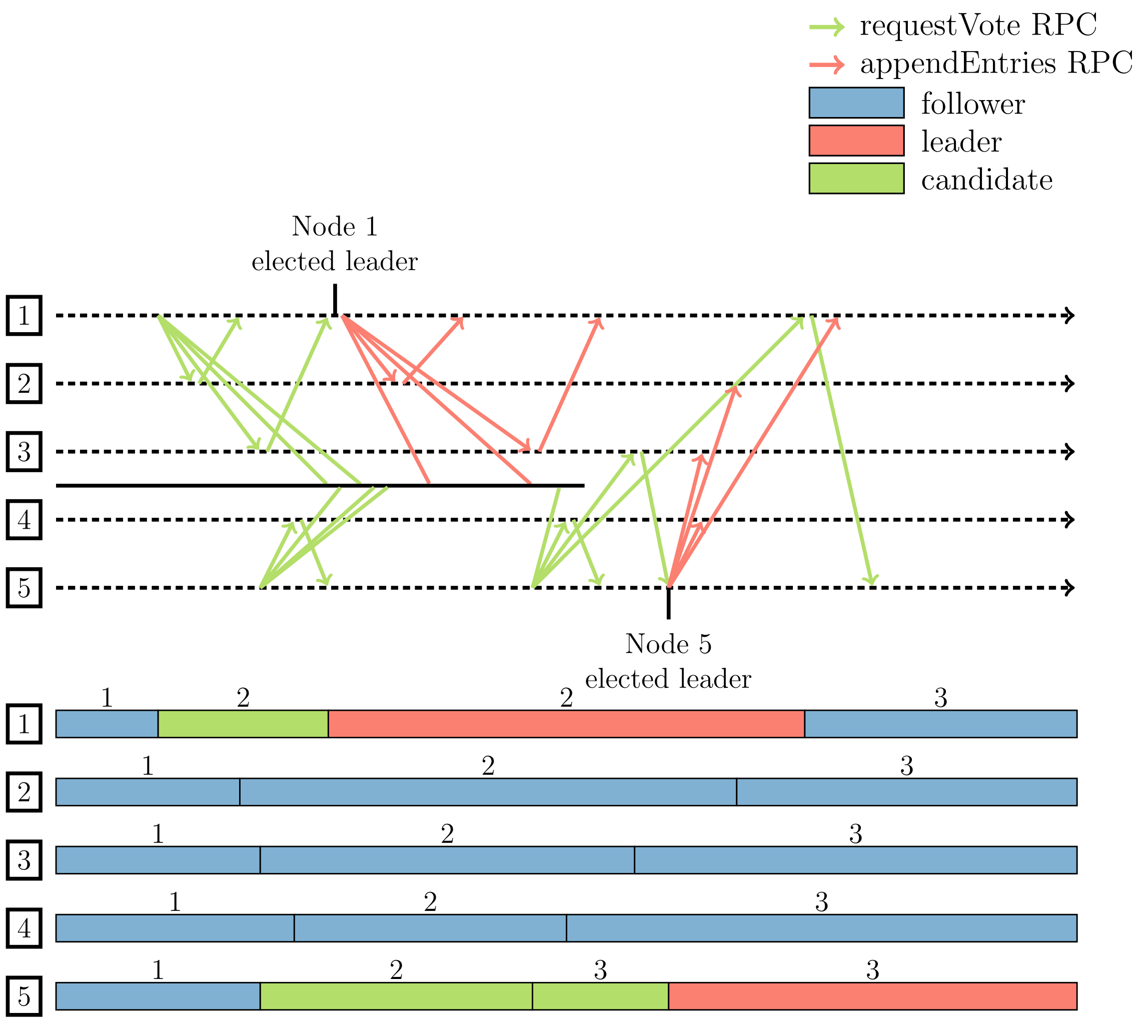


Figure 2.5: Example of an election in the face of a network partition, ultimately resulting in a leader stepping down. This cluster starts up with a partition between nodes 1/2/3 and 4/5. Nodes 1 and 5 each timeout and begin an election for term 2. Node 1 is elected leader by nodes 2 and 3 but node 5 is unable to gain votes from the majority of nodes. After the partition is removed, node 5 times out and restarts the election in term 3. Nodes 1 to 4 grant votes to node 5 as the term is now 3, so node 5 is elected leader in term 3.

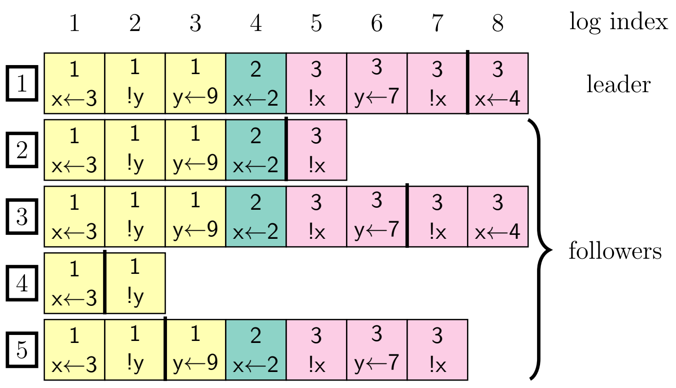


Figure 2.6: Example log states for a collection of 5 nodes. Node 1 is leader and has committed the first 7 entries in the log as they have been replicated on a majority of nodes, in this case nodes 3 and 5. Nodes 2 and 4 may have failed or had their messages lost in the network, therefore their logs are behind. The leader, node 1, is responsible for bringing these nodes up to date.

the state machines is a key values store and the commands are *add* e.g. $x \leftarrow 5$ which associated 5 with key x and *find* e.g. $!y$ which returns the value associated with the key y.

Consider the case where no nodes fail and all communication is reliable. We can safely assume that all nodes are in the same term and all logs are identical. The leader broadcasts *AppendEntries* RPCs, which includes (among other things) the log entry the leader is trying to replicate. Each node adds the entry to its log and replies with *success*. The leader then applies the command to its state machine, updates its *Commit Index* and returns the result to the client. In the next *AppendEntries* message, the leader informs all the other nodes of its updated *Commit Index*, the nodes apply the command to their state machines and update their *Commit Index*. This process repeats many times, as shown in Figure 2.7.

Now, consider the case that some messages have been lost or nodes have failed and recovered, leaving these logs inconsistent. It is the responsibility of the leader to fix this by replicating its log to all other nodes. When a follower receives an *AppendEntries* RPC, it contains the log index and term associated with the previous entry. If this does not match the last entry in the log, then the nodes reply with *unsuccessful* to the leader. The leader is now aware that the log is inconsistent and needs to be fixed. The leader will decrement the previous log index and term for that node, adding entries to the log whilst doing so until the inconsistent node replies with *success* and is therefore up to date, as shown in Figure 2.8.

Each node keeps its log in persistent storage, which includes a history of all commands and their associated terms. Each node also has a *Commit Index* value, which represents the most recent command to be applied to the replicated state machine.

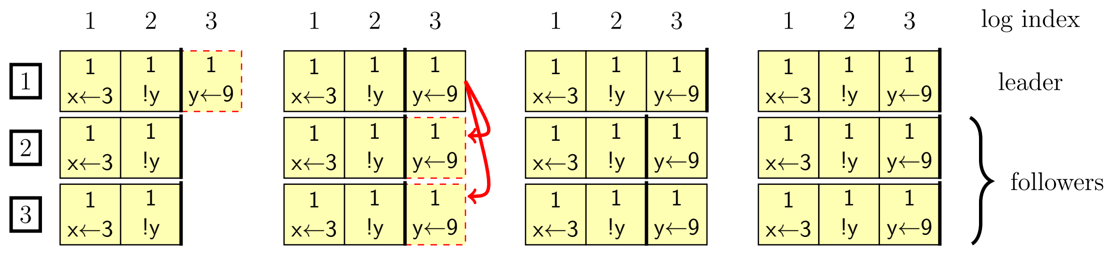


Figure 2.7: Example of node 1 committing $y \leftarrow 9$ in term 1 at index 3. All nodes' logs are identical to start with. From right to left: node 1 receives $y \leftarrow 9$ from the consensus module and appends it to its log. Node 1 dispatches *AppendEntries* to nodes 2 and 3. These are successful so node 1 updates its commit index (thick black line) to 3, applies the command to its state machine and replies to the client. A later *AppendEntries* updates the commit indexes of nodes 2 and 3.

When the *Commit Index* is updated, the node will pass all commands between the new and old *Commit Index* to the state machine.

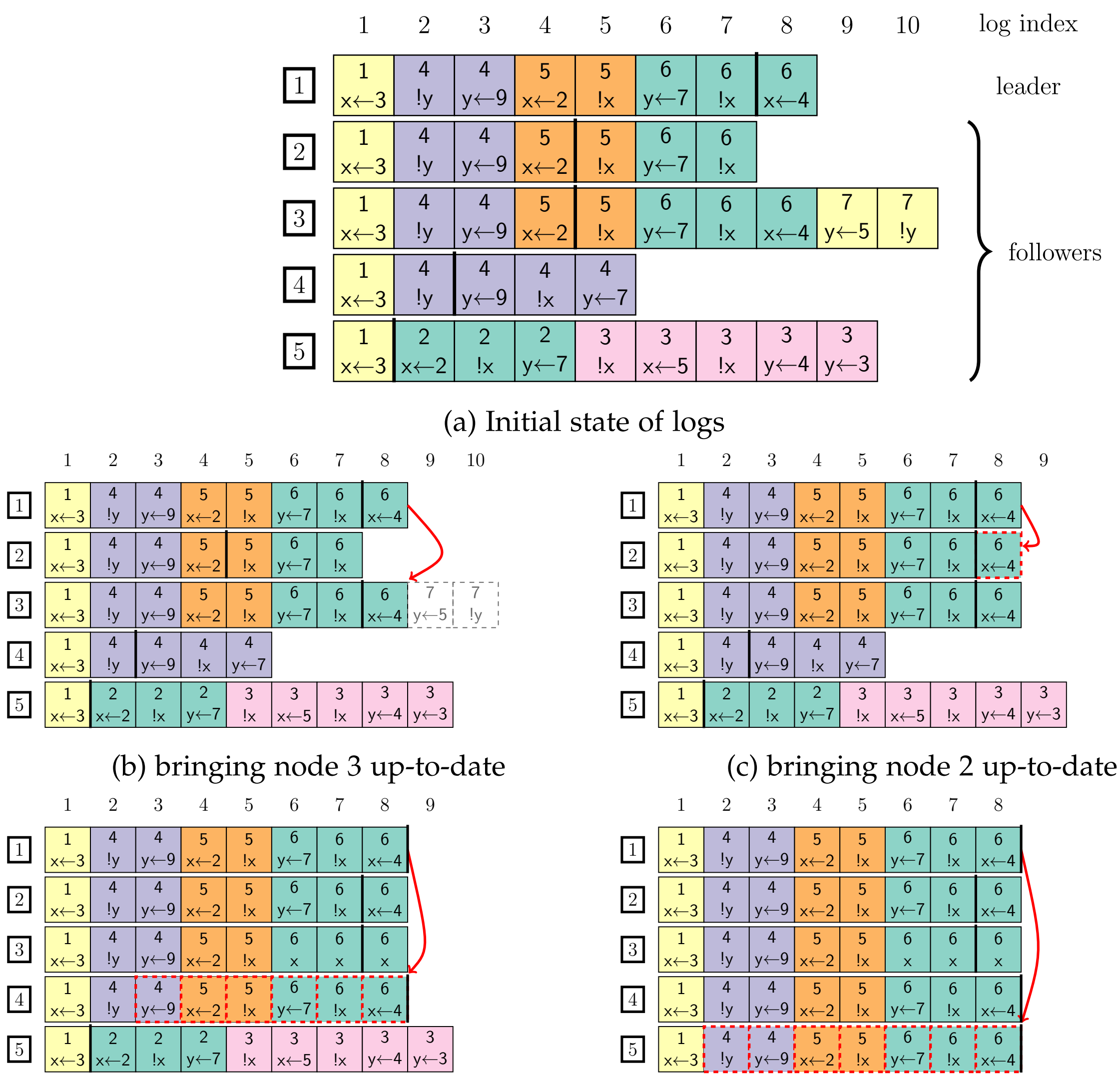
2.3.3 Safety and Extra Conditions

To ensure safety we must augment the above description with the following extra conditions:

Since the leader will duplicate its log to all other logs, this log must include all previous committed entries. To achieve this, Raft imposes further constraints on the protocol detailed so far. First, a node will only grant a vote to another node if its log is at least as up-to-date (defined as either having a last entry with a higher term or, if terms are equal, a longer log).

The leader node is responsible for replicating its log to all other nodes, including committing entries from the current term and from previous terms. Currently, however, it is possible to commit an entry from a previous term and for a node without this entry to be elected leader and overwrite this (as demonstrated in Figure 2.9). Therefore a leader can only commit an entry from a previous term if it has already committed an entry from this current term.

The protocol provides linearisable semantics [12], guaranteeing that a command is committed between the first time a command is dispatched and the first successful response. The protocol doesn't guarantee a particular interleaving of client requests, but it does guarantee that all state machines will see commands in the same order. The protocol assumes that if a client does not receive a response to its request within its *Client Timeout* or the response is negative, it will retry this request until successful. To provide linearisable semantics, we must ensure that each command is applied to each state machine at most once. To address this, each client command has an associated



(a) Initial state of logs

(b) bringing node 3 up-to-date

(c) bringing node 2 up-to-date

(d) bringing node 4 up-to-date

(e) bringing node 5 up-to-date

Figure 2.8: Example of leader, node 1, using *AppendEntries* RPCs to bring the logs of nodes 2-5 upto a consistent state. From top right to bottom left:
(b) node 1 has just become the leader of term 6, it dispatches *AppendEntries* with $< PrevLogIndex, PrevLogTerm > = < 8, 6 >$ and only node 3 is consistent so it removes its extra entries;
(c) node 1 retries *AppendEntries* with $< 7, 6 >$ to nodes 2, 4 and 5, and only node 2 is consistent so appends $< 8, 6 >$;
(d) now node 1 can commit the entry at index 8 and next *AppendEntries* is $< 2, 4 >$ and node 4 removes incorrect entries index 3-5 and appends new entries;
(e) the final *AppendEntries* is $< 1, 1 >$ and node 5 is finally consistent.

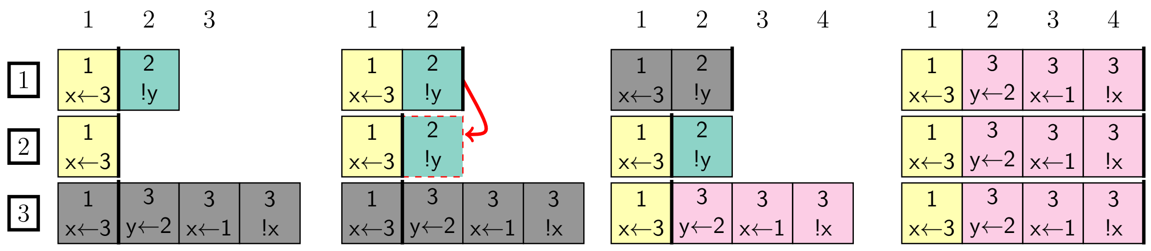


Figure 2.9: Example of Leader Completeness violation, without the extra condition on leadership. From left to right: It is term 4 and node 1 is leader whilst node 3 has failed. The leader, node 1 is replicating its log to node 2 and thus commits $!y$ at index 2 from term 2. Node 1 then fails and node 3 recovers and becomes leader in term 5, node 3 replicates its log to nodes 1 and 2, thus overwritting $!y$ at index 2.

serial number. Each consensus module caches the last serial number processed from each client and response given. If a consensus module is given the same command twice, then the second time it will simply return the cached response. The protocol will never give a false positive, claiming that a command has been committed when it in fact has not but the protocol may give false negatives, claiming that a command has not been committed when in fact it has. To address this, the client semantics specify that each client must retry a command until it has been successfully committed. Each client may have at most one command outstanding at a time and commands must be dispatched in serial number order.

The Raft authors claim that the protocol described provides the guarantees in Figure 2.10 for distributed consensus.

Election Safety: at most one leader can be elected in a given term
Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries.
Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.
Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.
State Machine Safety: if a node has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.

Figure 2.10: Raft's guarantees to provide distributed consensus

2.4 Requirements Analysis

This project aims to evaluate the Raft consensus algorithm as described in the previous section. We believe that in order to evaluate the protocol across a range of network environments, we must not only implement the full Raft consensus protocol but also build a simulation framework around it to allow us to fully explore what the protocol has to offer. Beyond evaluating the performance of the protocol, we were keen to use our simulation traces to catch subtle bugs in our implementation (or even the protocol specification) which may only occur once in 10,000 or even 100,000 traces. Our insight was that we could use our holistic view of the cluster from the simulator, to ensure that all nodes' perspectives of the distributed system were consistent with the protocol. For this reason we designed our own simulation framework instead of opting for traditional event-based simulations like ns3 or OMNeT++ [37].

We divided the development up into the following phases:

1. **Simple event simulation** – Build a simple event-based simulator which takes as an argument a number of nodes, simulates these nodes sending simple messages to each other and responding after random timeouts.

2. **Raft leader election at start-up** – Implement Raft's basic leader election protocol by transmitting the *RequestVote* RPC, terminating the simulation once a leader has been established.

3. **Raft leader election with failures** – Simulate Raft nodes failing and recovering, differentiating clearly between state on volatile and non-volatile storage. Implement the heartbeat functionality of the *AppendEntries* RPC.

4. **Example fault tolerant application** – Build an example application for testing Raft. This is a distributed key-value store.

5. **Raft log replication** – Implement the full *AppendEntries* RPC, testing the consensus protocol using the example application.

6. **Client interaction** – Simulate clients trying to processes their workloads, implement leader discovery and command commit.

7. **Safety model** – Model the behaviour of Raft in SPL and compile to OCaml.

8. **Invariant checking** – Express the protocol's invariants (Figure 2.10) in OCaml and run simulation traces through the checker to ensure no invariants are invalidated.

2.5 Choice of Tools

2.5.1 Programming Languages

Our primary development language was OCaml, which is a multi-paradigm, general purpose language with a focus on safety, which is key for this project. The core protocol implementation is pure, optimising expressiveness in the type system to restrict the behaviour of the protocol to its own safety criteria and liberal use of assertion checks for restricting run-time behaviour. We are not alone in believing OCaml to be a solid choice for distributed systems [34, 23, 30].

Statecall Policy Language (SPL) [22, 21] is a first order imperative language for specifying non-deterministic finite state automata, such as the one used to define node behaviour in Raft (Figure 2.2). We have chosen to use SPL due to its ability to be compiled to either Promela, for model checking in SPIN, or OCaml, to act as a safety monitor at run-time. SPIN is a widely used open source model checker, which takes a distributed algorithm specified in Promela and properties specified in Linear Temporal Logic to generate C code verifying that the algorithm satisfies the properties.

2.5.2 Build Systems

For development we used a wrapper around *ocamlbuild* called *corebuild*, which provides sensible compiler options for production development[1].

2.5.3 Libraries

We used OPAM [35], a source based package manager for OCaml, to manage the libraries for this project and to later release the project. To minimise dependencies, we depend only on Core, an alternative standard library for OCaml, which has been thoroughly tested in industry. Although we developed and tested on Unix machines, we limit the dependence on the *Unix* module to only the *Clock.RealTime* module, for future porting to architectures such as Mirage [23] or Js_of_ocaml[2].

2.5.4 Testing

We used OUnit[3] for Unit testing the code base as it is a popular choice with plenty of documentation and support available. The unit tests involve multiple test inputs, focusing particularly on edge cases (e.g. empty lists, 0s, negative numbers, etc.) and

[1]The precise definition of corebuild is available at `github.com/janestreet/core/blob/master/corebuild`

[2]`ocsigen.org/js_of_ocaml`

[3]`ounit.forge.ocamlcore.org`

asserting that outputs are as expected. These tests are run after successful builds to ensure that no regressions have occurred.

The testing framework is composed of building Raft and running a wide range of simulations. These were run in a range of environments using Travis CI, a continuous integration service for open source projects, which runs the test framework against the code for each git commit or pull request.

2.6 Summary

We have outlined the Raft consensus algorithm and assumptions under which it operates. We explained our motivation for building a simulation framework to evaluate the protocol. Finally, we closed with our plans for a safety driven approach in OCaml.

Chapter 3

Implementation

3.1 Simulation Framework

The Event-based Simulation for Raft begins with modelling the activity of a collection of nodes establishing a leader, then terminating, returning the number of time units taken to reach agreement on the leader. This phase assumes reliable communication and always available nodes. From this, we gradually expand functionality and build up a more elaborate model, reducing the underlying assumptions we are making. We calibrate our simulator by reproducing the results from the Raft paper, thus initially evaluating the relationship between the time taken for all nodes to agree upon a leader and the distribution of follower and candidate timeouts.

The simulator supports two modes of operation, *Discrete Event Simulation (DES)* and *Real-time Event Simulation (RES)*, allowing us to tailor the tradeoff between simulation accuracy and time. In the Discrete Event Simulator, each node's local time will run arbitrarily fast. Thus we are making primitive assumptions such as that the notion of time is quantised including assuming all computation time is negligible and if two packets arrive at a node in the same time unit, they may be served in any order. Event-based simulation, particularly DES, will allow us to rapidly iterate over the vast state space of simulation parameters to evaluate performance and detect rarely occurring safety violations.

The basic components of the simulator are:

Events Computation to occur at a specific node, at a specific time which can result in modifications to the state of a node and generate more events, which occur at later times (i.e. checking timeouts) and may occur on another node (i.e. message passing between nodes).

Event List Priority queue of events which are waiting to be simulated, typically implemented with data structures such as a splay tree [33], skip lists, calendar queues [3] or ladder queues.

State Local data for protocol implementation associated with each nodes and client.

Clock Time is a quantised monotonically increasing function where events are instantaneous. The purpose of the clock is solely to provide a partial order on the application events in the Event list.

Non-determinism Random number generation, required both by the protocol for contention resolution and for the simulation framework to model packet delay, loss and node failure.

The main body of the simulator iterates over the events from the event list. Each event is executed by passing it the state currently associated with that node, then the event returns both the new state which is associated with that node and any new events are added to the event list (Figure 3.1).

3.2 Clock

We begin by defining an abstract notion of local time for each node. The purpose of local time for each node is to implement the timeouts traditionally used in asynchronous distributed system to approximately determine if a node is live and to build reliable delivery on unreliable channels. We are able to simulate the variation in local clock speeds by adjusting timeouts accordingly. Every node state and event, has an associated *Time* (defined in Figure 3.2).

In the DES, the events are simulated as soon as they are available, as long as the time ordering is respected. Each node state stores its local time and this is updated to the event time whenever an event is simulated. In RES, each node state stores a function which evaluates to system time and the events wait in the event queue blocking until it is their time for evaluation. A single simulation module, parametrised over the implementation of *Time* (defined in Figure 3.2), implements both DES and RES. This allows us to support both simulation modes with a single simulator, thus making the code easier to maintain and avoiding code duplication. Furthermore, this interface makes it easy to plug in alternative implementations of *Time* which could, for example, add noise to *FakeTime* to simulate different clock speeds.

The module signature of *TIME* contains two abstract time notions, *time* which represents the absolute time such as the Unix time or the time since the simulation was started and *span* which represents an interval of time. The *wait_until* function offers a blocking wait for real time implementations or instantaneously returns *unit* for a DES.

Dynamic erroneous behaviour is handled by *assert* statements. Figure 3.3 is an extract from the *FakeTime* module, which implements time as a monotonically increasing integer, by initialising the integer to 0 and ensuring all operations only increase

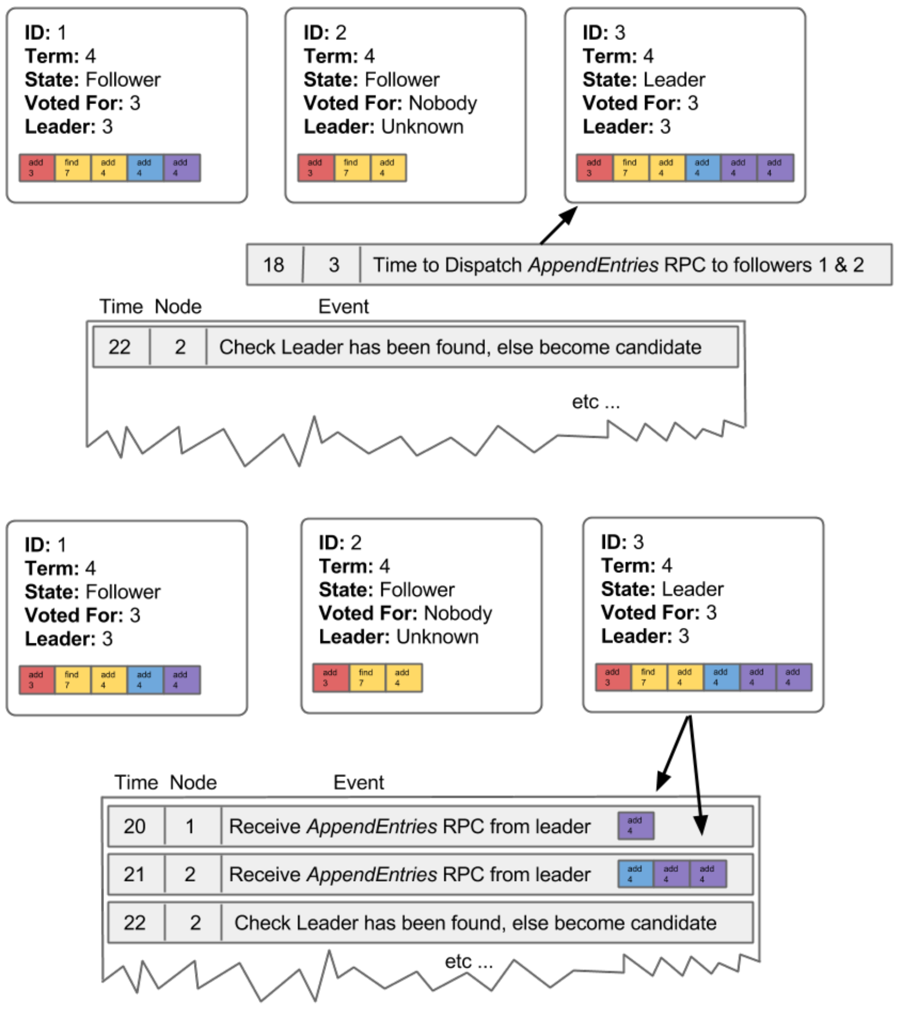

ID: 1
Term: 4
State: Follower
Voted For: 3
Leader: 3
ID: 2
Term: 4
State: Follower
Voted For: Nobody
Leader: Unknown
ID: 3
Term: 4
State: Leader
Voted For: 3
Leader: 3
18 3 Time to Dispatch AppendEntries RPC to followers 1 & 2
Time Node Event
22 2 Check Leader has been found, else become candidate
etc ...
20 1 Receive AppendEntries RPC from leader
21 2 Receive AppendEntries RPC from leader
22 2 Check Leader has been found, else become candidate
etc ...

Figure 3.1: Simplified example of DES, simulating Raft Consensus between 3 nodes. Node 3 is leader in term 4 and is dispatching *AppendEntries* to nodes 1 and 2. A unified event list is used across the nodes to map time and node ID to events.

its value. This output shows how OCaml runtime terminates when assert statements evaluate to false. The application of assertion checking to enforce inductively defined invariants is used systematically throughout our code. This constrains the behaviour of alternative implementations of our signatures, e.g. a poorly designed simulation GUI would not be able to send packets back in time.

In the Raft specification, a *term* is a monotonically increasing value used to order events and messages. A node will only ever need to initialise, increment and compare terms, thus these are the only operations exposed to nodes. As shown in Figure 3.4, each term has one or two phases, the leader election phase and the leadership phase. There will be at most one leader per term and possibly no leaders if consensus is not reached first time. The previous example demonstrated capturing erroneous be-

```
module type TIME =
  sig
    type t
    type span
    val init : unit -> t
    val compare : t -> t -> int
    val add : t -> span -> t
    val diff : t -> t -> span
    val span_of_int : int -> span
    val wait_until : t -> unit
  end
module FakeTime : TIME
module RealTime : TIME
```

Figure 3.2: Shared module signature of *FakeTime* and *RealTime*.

```
# let span_of_int x = assert (x>=0); x;;
val span_of_int : int -> int = <fun>
# span_of_int (-5);;
Exception: "Assert_failure␣//toplevel//:1:20".
```

Figure 3.3: Demonstration of enforcing time monotonicity.

haviour at run time, but we can do even better by catching errors statically. OCaml's interface definitions abstract from implementation details to limit operations to only those explicitly made available. The example in Figure 3.5 demonstrates that even through *term* is implemented as an integer, this detail is not available outside the *Term* module and therefore *Term.t* is statically guaranteed to only ever increase.

3.3 State

The node and client state is only modified by events during their simulation and the state is immutable, providing the ability to record and analyse the sequence of state

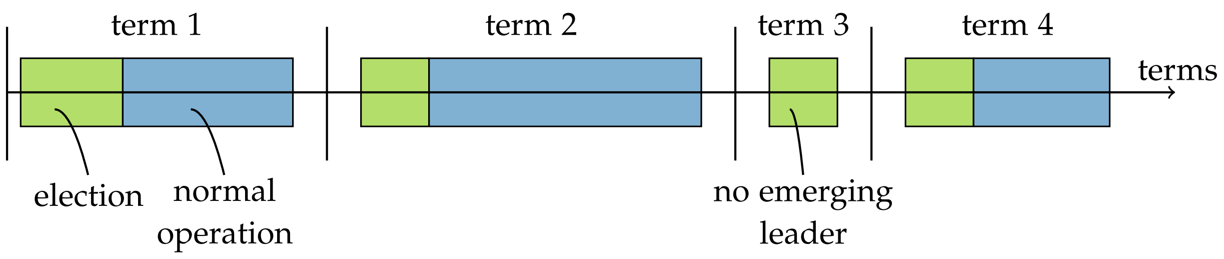


Figure 3.4: Example timeline of *terms*

```
module TERM : sig
  type t
  val succ: t -> t
  val init: unit -> t
  val compare: t -> t -> int
end
```

```
module Term : TERM = struct
  type t = int
    with compare
  let succ = succ
  let init () = 0
end
```

Figure 3.5: Use of type system to enforce monotonicity of *Term*.

changes by maintaining a history of states.

Each event takes a representation of state as its last argument and returns a new representation of state, by applying *statecalls* to the state using a *tick* function, as defined in Figure 3.6. Internally node state is stored as private record, allowing quick read access but restricting modification to statecalls. Statecalls are a variant type denoting a safe set of atomic operations on node state, allowing us, for example, to simulate write-ahead logging. The *tick*, *refresh* and *init* functions are the only methods of generating or modifying node state, either by transformations described by statecalls, simulating failure by clearing volatile state or initialising a fresh state, respectively. This restriction makes many erroneous states simply unrepresentable, thus reducing the state space and making it easier to quantify the valid possible states, as we do in SPL(§3.8).

```
module type STATE =
 functor (Time: TIME) ->
 functor (Index: INDEX) -> sig
  type t
  type statecall =
   | IncrementTerm
   | Vote of Id.t
   | StepDown of Index.t
   | VoteFrom of Id.t
     ....
  val tick: statecall -> t -> t
  val init: Id.t -> Id.t list -> t
  val refresh: t -> t
end
```

Figure 3.6: Functor to limit protocol interaction with state to *statecalls*.

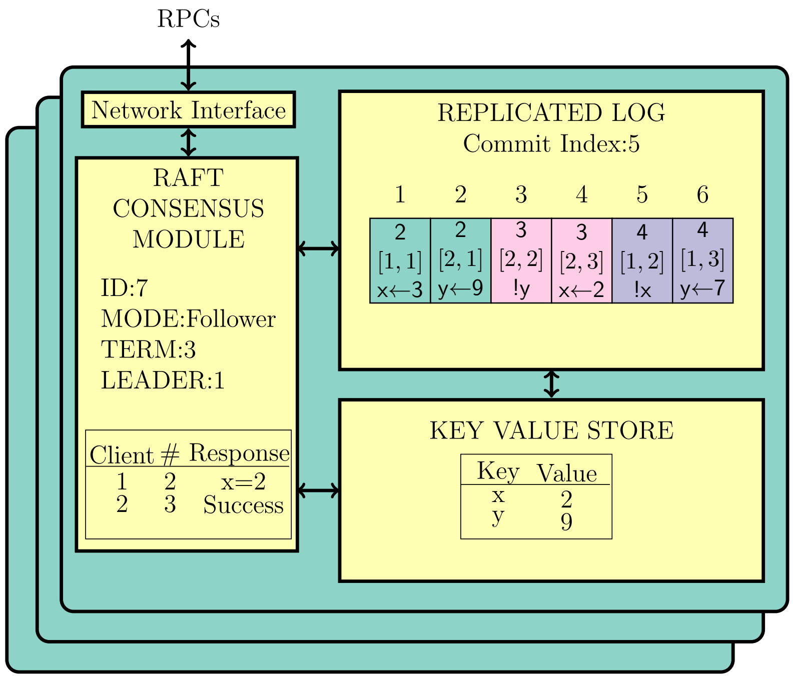


Figure 3.7: Example node running a key value store using Raft consensus.

3.3.1 Node State

Access to the *tick* function (defined in Figure 3.6) for modifying node state is mediated by the *StateHandler* module, which is responsible for ensuring that no state transitions violate the protocol invariants (Figure 2.10) and simulating node failures. The *State-Handler* module compares each updated node state to the states of all other nodes and all past state to ensure that the perspectives from the nodes are compatible and able to form a coherent global picture of the distributed system.

For example when a node wins an election in *term* 7 we can check that no other node has been leader in *term* 7 (checking Election Safety from Figure 2.10), at least a strict majority of nodes have at some time voted for that node in *term* 7 and all committed entries are present in the new leader's log (checking Leader Completeness from Figure 2.10). Or when a follower commits a log entry, we can check that no other node has committed a different command at the same index (checking State Machine Safety from Figure 2.10) and that the command serial number is strictly greater then the last serial number from the corresponding client. To minimise computation, the majority of these checks are implemented inductively, ensuring each monotonic value is greater then the last.

3.3.2 Client State

Like replica state, Client state access is limited to a set of operations, performed atomically by the *tick* function whose access is mediated by the *ClientHandler* module. Clients are responsible for not only generating workloads and committing commands, but also checking the client semantics. This proved vital to the project due to the underspecification of how Raft handles client requests, detailed in §4.3. Locally each client has its own copy of the key-value store which is used to generate expected responses for commands. These can then be checked against the responses from the Raft protocol.

3.4 Events

Our insight is that the Raft protocol can be implemented as a collection of events, performing computation in specified temporal and spatial environments. We define operators (see *opt* in Figure 3.8) as a pure mapping from a node state to a new node state and a list of new events. A simplified pair of mutually recursive type definitions seen in Figure 3.8 represent the relationship between events and operators. Polymorphic over the representation of time, ID's and state, *event* relates the time at which an operator should be simulated, which node should simulate the operator and the operator itself. The variant *RaftEvent* is required as OCaml (by default) does not accept recursive type abbreviations. We extend the cases of the variant to simulate other types of event such as client events, simulation timeouts and simulated failures.

```
type ('time, 'id, 'state) event =
   RaftEvent of ('time * 'id * ('time, 'id, 'state) event)
and ('time, 'id, 'state) opt =
  'state -> 'state * ('time, 'id, 'state) event list
```

Figure 3.8: Mutually recursive type definitions of *t* and *event*.

All events are queued in the event list as partially applied functions, which perform computation when applied to state. If the state was mutable, this could be implemented by withholding a unit argument, or using the lazy evaluation in OCaml. To communicate, nodes stage events by applying all arguments (except state) to a particular RPC operator, passing this to *unicast*, *multicast* or *broadcast* functions in the *Comms* module to produce the associated events. These functions calculate arrival time from the packet delay distribution as well as simulating packet loss or duplication.

3.5 Event List

Continuing with our theme of protocol safety, our implementation explicitly handles unexpected behaviours with monads and does not use exceptions except for the SPL safety monitor. Figure 3.9 uses the *option* monad[1] which forces the caller to pattern match on the result and explicitly handle the case that the *Eventlist* is empty. Utilising the type system for error handling greatly improves the readability of module interfaces and moves the debugging from runtime to compile time. Thus making it easier to build alternative implementations to test various protocol optimisations and modifications.

```
module type Monad = sig
  type 'a t
  val bind: 'a t -> ('a -> 'b t) -> 'b t
  val return: 'a -> 'a t
end

module Option : Monad = struct
  type 'a t = None | Some of 'a
  let bind value f =
    match value with
      | None -> None
      | Some x -> f x
  let return x = Some x
end
```

Figure 3.9: Monad signature and *Option* monad implementation.

```
module type EVENTLIST = sig
  type item
  type t
  val head: t -> (item * t) Option.t
  val add: item list -> t -> t
  val init: item list -> t
end
```

Figure 3.10: Signature for data structures implementing Event List.

The *Eventlist* defined in Figure 3.10 is the most often used data structure in this project and is ideal for optimisation due to the limited operations that it is required to support. In fact, the only access operation is *head*, which removes the minimal element in the data structure and returns an element and the new data structure, if not empty.

[1]Similar to *maybe* in Haskell.

A naive implementation of EventList would be a sorted list, which clearly provides $O(1)$ for the *head* function, $O(n \log n)$ for *init* from a list of n items and $O(n)$ to *add* a single element in $O(n)$ space. Instead we opted for a splay tree [33], a self-adjusting binary tree which provides amortized $O(\log n)$ for all basic data structure operation before applying optimisations for the DES access patterns [11].

```
module SplayTree (Item: sig type t with compare end) = struct
  type item = Item.t
  ...
end
```

Figure 3.11: *SplayTree* as a functor over *Item*

As detailed in Figure 3.11, *SplayTree* is a functor over implementations of *Item*. *SplayTree* will return a module satisfying the EVENTLIST signature. However this abstraction will mean that it is no longer possible to use *SplayTree* as we have no mechanism of creating things of type *Item.t*. The solution is a type sharing constraint, which will allow us to expose to the compiler that type *item* and *Event.t* are equivalent as shown in Figure 3.12.

```
module EventList =
  SplayTree(Event) : EVENTLIST with type item = Event.t
```

Figure 3.12: Type sharing constraint between *item* and *Event.t*

3.6 Simulation Parameters

The simulator accepts a wide range of parameters in the command line interface or configuration file. These are detailed in Figure B.1 in the appendix.

The statistical distribution fields such as packet delay or follower timer can be given as as either a fixed value, an exponential distribution with mean λ and offset, normal distribution with mean μ and standard deviation σ (and options to discard negative values) or a uniform distribution between minimum and maximum bounds. These parameters are parsed and packaged into a first class module over which the *Simulation* module is parametrised.

3.7 Non-determinism

Non-determinism is essential for simulating networked systems as well as being part of the Raft protocol to avoid pathologically poor performance. The Core standard library provides uniform random floating point values[2], from this we can then generate the required distributions. For example, to simulate the packet delays as a Poisson process we generate the exponentially distributed inter-arrival times as needed, to limit the space requirements. We transform the provided $U \sim U(0,1)$ into $X \sim Exp(\lambda)$ using the following standard result (eq. 3.1) and used the Box-Muller Transform [31] to generate $X \sim N(\mu, \sigma)$ (eq. 3.2 & eq. 3.3).

$$X = \frac{-1}{\lambda} \log U \tag{3.1}$$

$$Z_0 = \sqrt{-2 \ln U_1} \cos 2\pi U_2 \tag{3.2}$$

$$Z_1 = \sqrt{-2 \ln U_1} \sin 2\pi U_2 \tag{3.3}$$

3.8 Correctness

We modelled the operation of a node participating during Raft consensus using the Statecall Policy Language (SPL) (§2.5.1). The simplest of these automata models the transitions between the modes of operations as seen in Figure 2.2. The output from the SPL is shown in Figure 3.14. Once the models for Raft Consensus were developed in SPL, they were compiled to OCaml for execution alongside the simulations. Each node stores a Raft Safety Monitor in its state (detailed in Appendix A) and updates it with *statecalls* such as *StartElection*. The safety monitor terminates execution using a *Bad_statecall* exception if it leaves states defined in the automata like Figure 2.2.

3.9 Summary

We have described our "correct by construction" approach to implementing the Raft algorithm and an event-based simulator. Using the OCaml module system we have drawn abstractions between the specific protocol implementation, its safety and the simulation framework, exposing at each interface only the essential operations. This modular design makes it easy to swap implementations, for example testing an alternative leader election algorithm. We encoded the protocol invariants and safety checks at each *statecall* to ensure that all safety violations are detected in simulation traces. Furthermore, we have harnessed the expressibility of the type system, allowing us to statically guarantee that no possible Raft implementation will invalidate basic invariants, such as *term* monotonically increasing.

[2]*/dev/urandom* is used to generate the seed, if it wasn't available then a value is generated from real time system parameters like time and processID.

```
exception Bad_statecall

type s = [
  |`RestartElection
  |`StartElection
  |`Startup
  |`StepDown_from_Candidate
  |`StepDown_from_Leader
  |`WinElection
  |`Recover
]

type t
val init : unit -> t
val tick : t -> [> s] -> t
```

Figure 3.13: Example OCaml interface to each node's run-time monitor, generated from SPL.

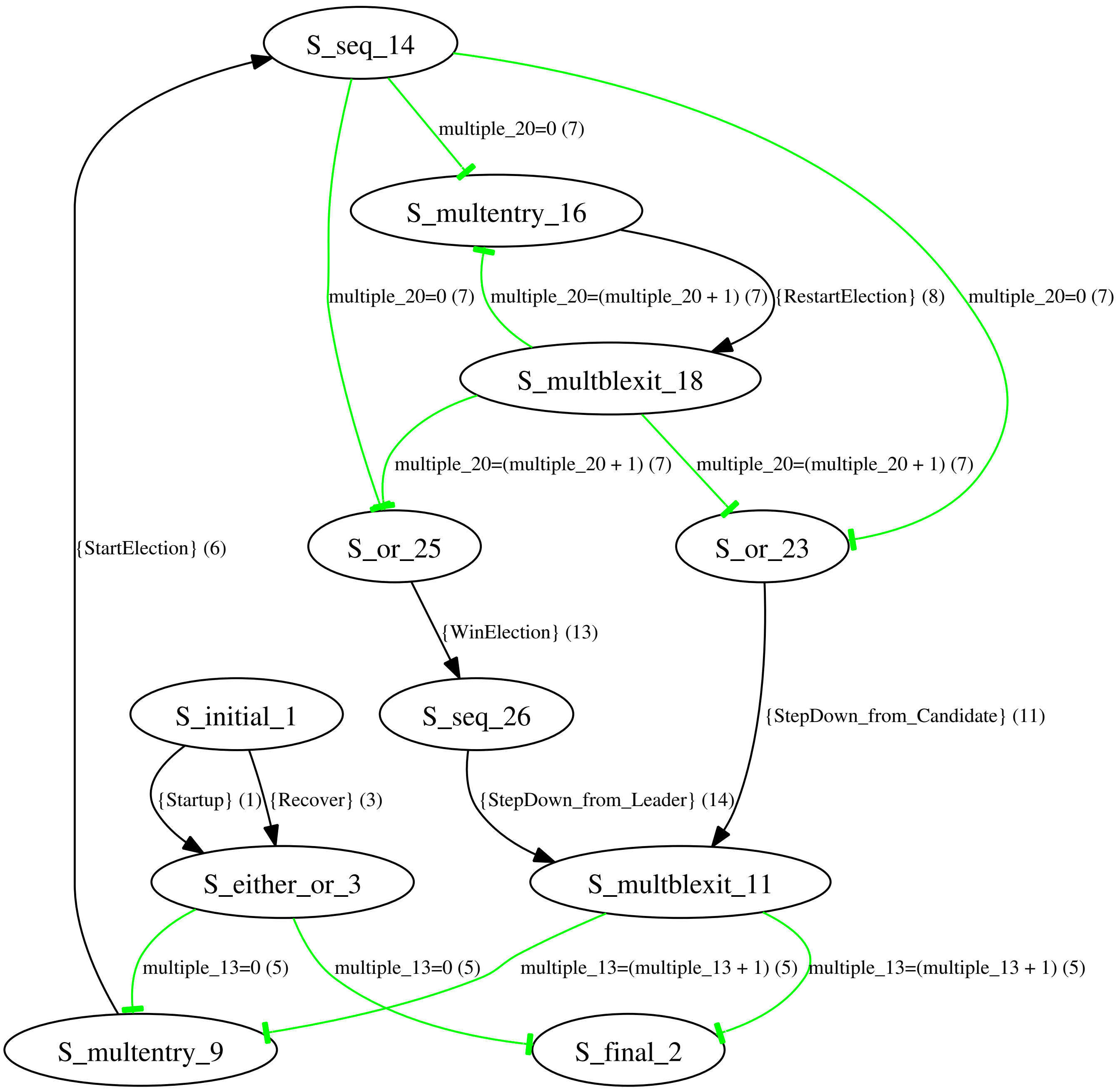


Figure 3.14: Raft's mode of operation transitions modelling in SPL.

Chapter 4

Evaluation

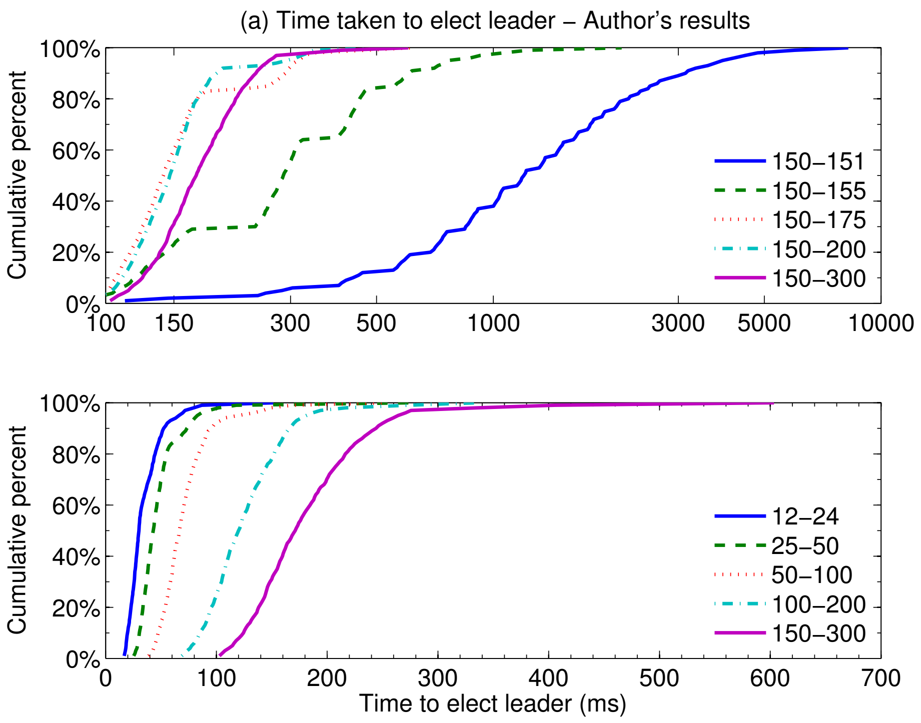
In my experience, all distributed consensus algorithms are either:
1: Paxos,
2: Paxos with extra unnecessary cruft, or
3: broken. **- Mike Burrows**

4.1 Calibration

Reproducibility is vital for research [7], thus our evaluation will begin by attempting to reproduce the Raft authors' leader election results shown in Figure 4.1(a)[1]. This will also allow us to calibrate the following evaluation by determining how to fix simulation parameters which are currently not being investigated. After consulting with the authors, we chose simulation parameters which reflect the original experimental setup using their LogCabin C++ implementation of Raft. Their setup involved five idle machines connected via a 1Gb/s Ethernet switch, with an average broadcast time of 15ms. The time taken to elect a new leader is sufficiently short that we can assume no node fails. Their use of TCP for the RPC implementation means that we need not simulate packet loss, but instead use a long tail for packet delay. The authors aimed to generate the worst case scenario for leader election. They made some nodes ineligible for leadership due to the log consistency requirements and encouraged split votes by broadcasting heartbeat *AppendEntries* RPCs before crashing the leader. We will approximate this behaviour by starting the cluster up from fresh state and attempting to elect a leader. The results of this simulation are shown in Figure 4.1(b).

The first feature we noted is the surprisingly high proportion of the authors' results which fall below the 150ms mark when the minimum follower timeout is 150ms. It is necessary (but not sufficient) for at least one follower to timeout, to elect a leader. Generalising the authors' analysis [27, §10.2.1], we can show that the distribution

[1]Reproduced with the authors' permissions, as shown in Figure 15 in [28] & Figure 10.2 in [27].



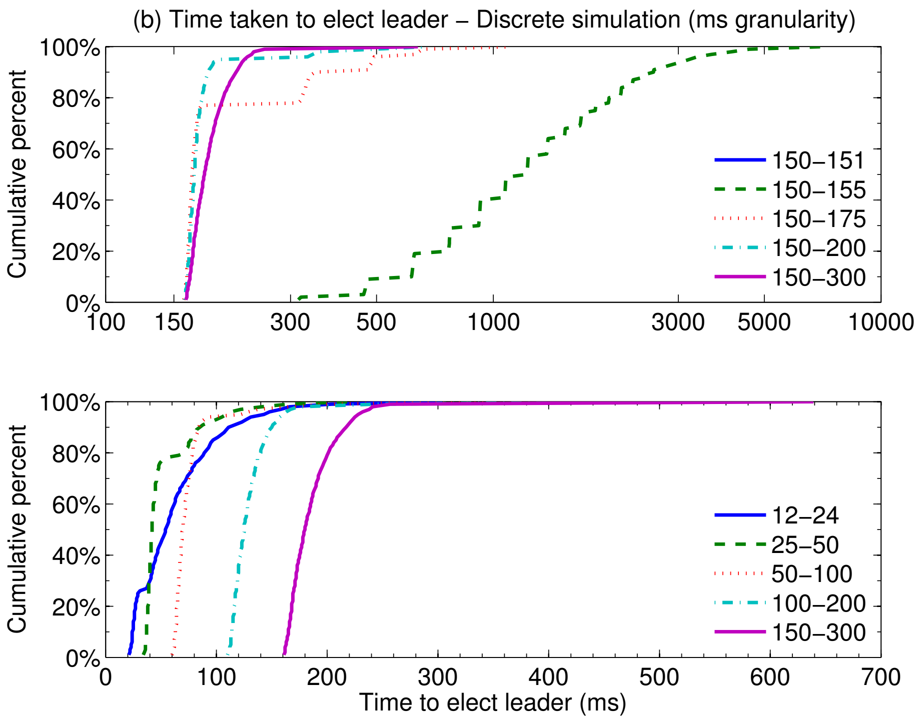


Figure 4.1: Cumulative distribution function of time to elect leader. Each top plot represents the time taken to establish a leader where the follower timeout has varying degrees of non-determinism, with the minimal follower timeout fixed at 150ms. Each bottom plot varies the timeout from T to 2T, for different values of T. The follower timeouts in milliseconds are shown in the legends.

of the time the first of s nodes times out, T_s, where the timeout distribution follows $followerTimer \sim U(N_1, N_2)$. T_s has a cumulative distribution function, $F_s(t)$ and probability density function $f_s(t)$ described below:

$$F_s(t) = \begin{cases} 0 & 0 < t < N_1 \\ 1 - (\frac{N_2 - t}{N_2 - N_1})^s & N_1 < t < N_2 \\ 1 & \text{otherwise} \end{cases} \tag{4.1}$$

$$f_s(t) = \begin{cases} \frac{s(N_2 - t)^{s-1}}{(N_2 - N_1)^s} & N_1 < t < N_2 \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

$$\mathrm{E}[T_s] = N_1 + \frac{N_2 - N_1}{s + 1} \tag{4.3}$$

Figure 4.2 is a plot of the cumulative distribution function (eq. 4.1), this is a baseline for the leader election graphs in Figure 4.1. After discussing our findings with the authors, it arose that they had been crashing the leader uniformly between *AppendEntries* RPC; hence their results are shifted by $\sim U(0, 75)$.

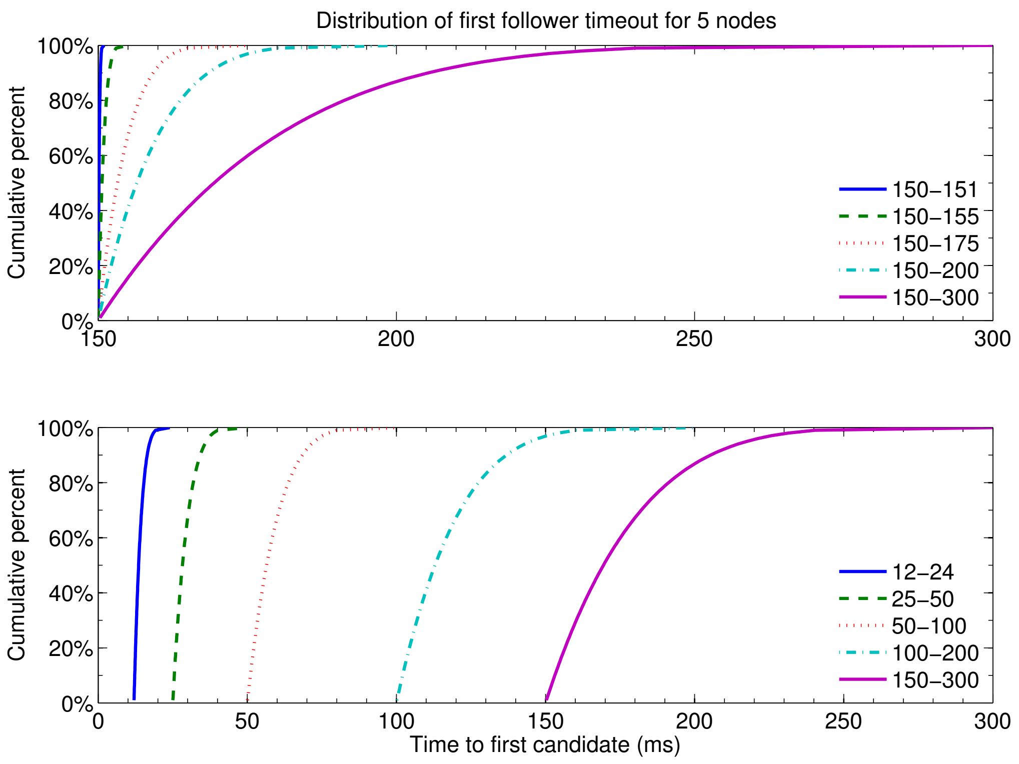


Figure 4.2: Cumulative distribution of the time until first election with varying follower timeouts for five nodes (eq. 4.1).

We observe that our simulation generally took longer to establish a leader than the authors' measurements, particularly in the case of high contention. One likely cause of this is the fact that the authors organised the candidates' logs such that two nodes were ineligible for leadership. These nodes may timeout like any other and dispatch *RequestVotes* to the other nodes, but these nodes will never gain a majority, thus reducing the number of possible leaders from five to three and reducing the likelihood of split votes and multiple rounds of elections. These ineligible candidates may still timeout, causing the term in the cluster to increase. Furthermore, these nodes will reject the last *AppendEntries* triggering the leader to dispatch more until it fails or brings the node's logs up to date. At the time of performing this evaluation, the paper under-specified the experimental setup used, but emphasised that it was a worst-case scenario. This led to our choices to synchronise nodes and have consistent logs to maximise the likelihood of split voting.

After observing that the stepping in the curve of the 150–155ms case was greatly exaggerated in comparison with the authors' results, we were concerned that the quantisation that we used in the DES might have been the cause of this. To test this hypothesis, we repeated the experiment with quanta sizes corresponding to microsecond and nanosecond granularity (see Figure 4.3). We observed no significant difference between the three levels of granularity tested except in the 150–151ms case[2]. Despite this, we chose to proceed with experiments at the nanosecond granularity due to a marginal improvement in results and negligible increase in computation time.

4.2 Leader Election

Optimising leader elections

As demonstrated in Figure 4.1, a suitable follower timeout range is needed for good availability, though the safety of the protocol is independent of timers. If the follower timer is too small, this can lead to unnecessary leader elections, too large and followers will be slow to notice a leader failure. The candidate timer must be sufficiently large to receive all the *RequestVotes* responses, whilst being sufficiently small to not delay restarting elections. Both follower and candidate timers need to be of a wide enough range to avoid split votes. The leader timer must be smaller than the follower timeout. The exact size is a tradeoff between bandwidth usage and speed at which information propagates across the cluster. The conclusion drawn by the protocol's authors is that T=150ms is a good conservative timeout in general when applied to the rules below. The first condition is known as the *timing requirement*, where *broadcastTime* is the average time for a node to send RPCs in parallel to every server in the cluster and receive their responses.

[2]The simulations were terminated after 10s of simulated time.

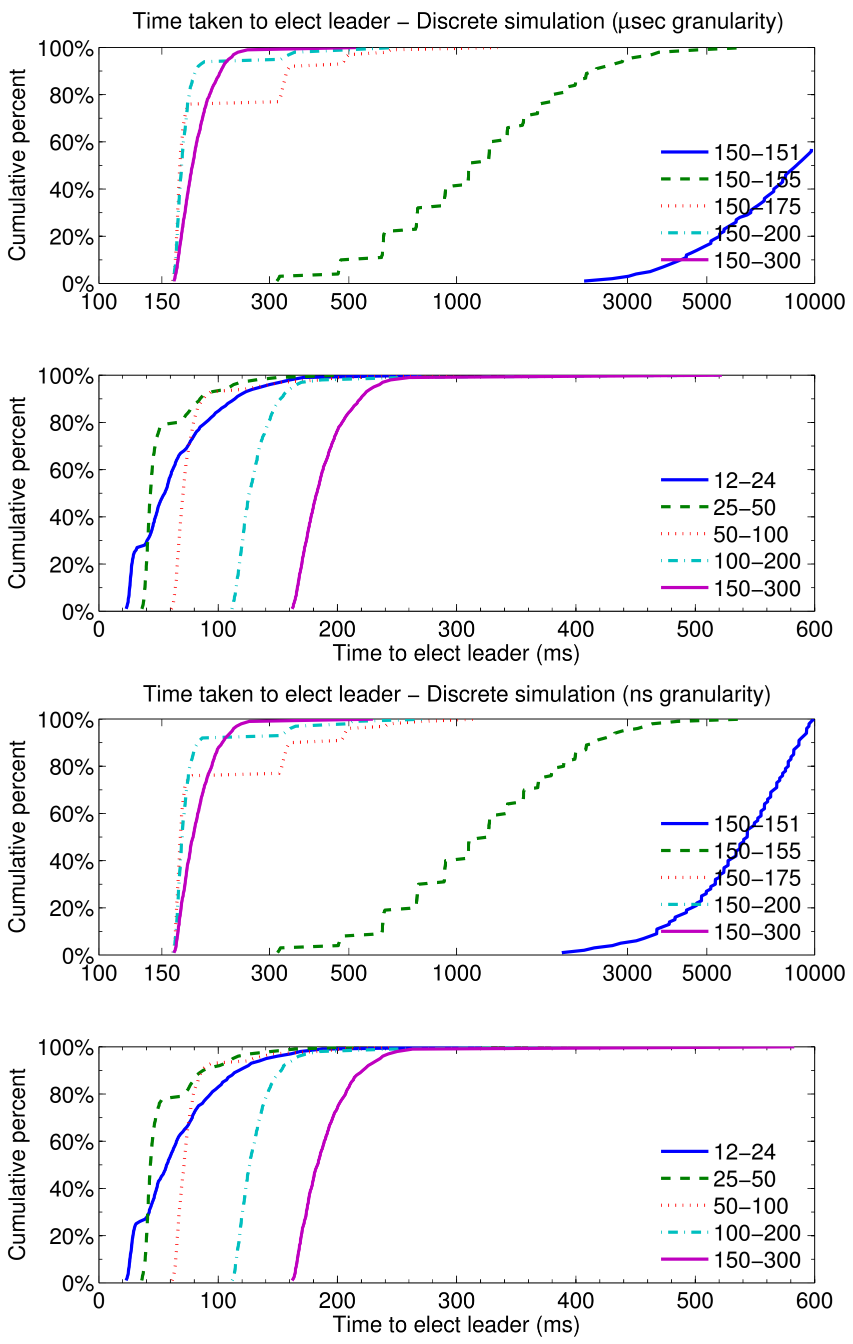


Figure 4.3: Cumulative distribution function (CDF) of time to elect leader. Each top plot represents the time taken to establish a leader where the follower timeout has varying degrees of non-determinism, with the minimal follower timeout fixed at 150ms. Each bottom plot varies the timeout from T to 2T, for different values of T. The follower timeouts in milliseconds are shown in the legends.

$$broadcastTime << candidateTimeout << MTBF$$
$$followerTimeout = candidateTimeout \sim U(T, 2T)$$
$$LeaderTimout = \frac{T}{2}$$

Our hypothesis is that the time taken to elect a leader in a contested environment could be significantly improved by not simply setting the candidate timer to the range of the follower timer. As the authors use the same timer range for candidates and followers, in Figure 4.1 we are waiting a minimum of 150ms (and up to twice that) before restarting an election, despite the fact that, on average, a node receives all of its responses within 15ms. Figure 4.4 (a) instead sets the minimum *candidateTimeout* to $\mu + 2\sigma$. Assuming $broadcastTime \sim N(\mu, \sigma)$, this will be sufficient 95% of the time. We can see that now 99.4% of the time leaders are established within 300ms, compared to 54.3% in Figure 4.1.

Increasing the non-determinism in the candidate timeout clearly reduces the number of split votes, but it does so at the cost of increasing the time taken to terminate an election. Our hypothesis is that we can improve the time taken to elect a leader and make the protocol more resilient to hostile network environments by introducing a binary exponential backoff for candidates who have been rejected by a majority of replicas. Figure 4.4(b) shows the improvement from enabling binary exponential backoff and (c) shows combining the optimisations used in (a) and (b). All optimisations performed considerably better than the orginal implementation, both in terms of time to elect a leader (Figure 4.4) and load on the network (Figure 4.5). Though the binary expontial backoff (b) and combined optimisations (c) took longer to elect a leader than the fixed reduction optimisation, they both further reduced the load on the network (Figure 4.5).

Tolerance to packet loss, delay, reordering & duplication

All tests so far have accounted for packet delay and re-ordering. In Figure 4.4, each packet was delayed by $delay \sim N(7ms, 2ms)$, since each packet delay is computed independently, packet re-ordering is implicit. Our simulator provides options for packet loss and duplication and the protocol remains safe, though performance in the face of packet loss could be greatly improved by simple retransmission where an acknowledgement of receipt is not received (if not already handled by the transport layer protocol).

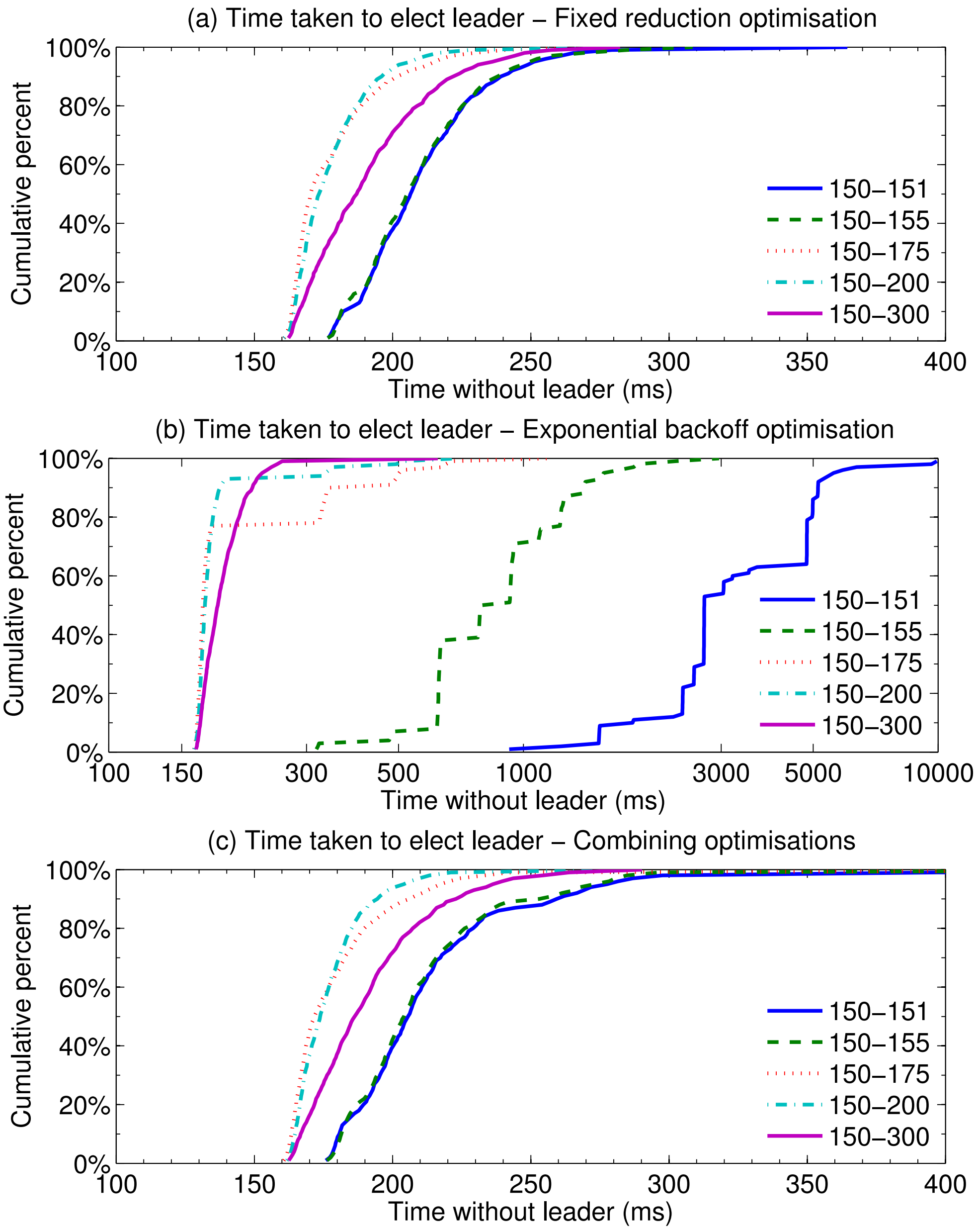


Figure 4.4: Investigating the impact of alternative candidate timeouts, keeping the range of follower timeouts fixed:
(a) sets candidate timeout to $X \sim U(23, 46)$
(b) enables binary exponential backoff for candidate timeout
(c) combines the optimisations from (a) and (b)
The follower timeouts in milliseconds are shown in the legends.

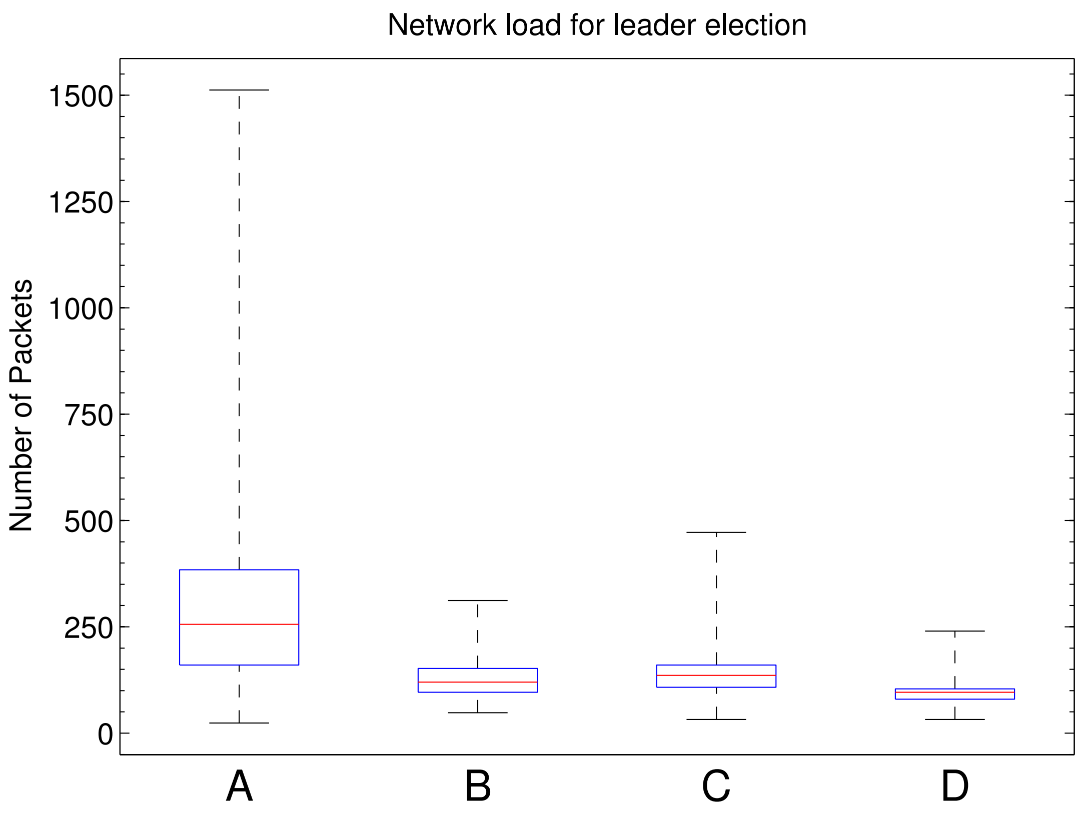


Figure 4.5: Box plot showing effect of candidate timer optimisations on network traffic when electing a leader:
(A) original setup, as seen in Figure 4.1;
(B) fixed reduction optimisation, as seen in Figure 4.4(a);
(C) exponential backoff optimisation, as seen in Figure 4.4(b); and
(D) combining optimisations, as seen in Figure 4.4(c).

Resilience in the face of diverse network environments

Our analysis so far has been limited to considering a cluster of nodes directly connected via an idle network, since packet loss, delay and failure are all assumed to be independent of the nodes and links involved. This topology is shown in Figure 4.6(a).

Suppose we instead deployed Raft on a network such as the one shown in Figure 4.6(c), where the replicated state machine is a file synchronization application working across a user's devices such as four mobile devices and a cloud VM. The mobile devices will often be disconnected from the majority of other devices, thus running election after election without being able to gain enough votes. When these devices rejoin the network, they will force a new election as their term will be much higher than the leader. This leads to frequently disconnected devices becoming the leader when reattached to the network, thus causing regular unavailability of the system when they leave again.

In this setup it would be preferable for the cloud VM to remain leader, unless several of the other devices often share networking that is disconnected from the internet. We note that if this limitation is a problem for a specific application, it is likely that the Raft's model, of providing strong consistency at the cost of availability, may be unsuitable for that application. In our example topology for a file synchronization application over a user's devices across the edge network and cloud, a more suitable consistency model is likely to be BASE.

It is also interesting to consider the impact of a single node being behind an asymmetric partition such as a NAT box or firewall (Figure 4.6(b)). This node will repeatedly re-run elections as it will not be able to receive incoming messages, each *RequestVotes* will have a higher term than the last, forcing the leader to step down. This will prevent the cluster from being able to make progress.

A further challenge for the protocol is modern heterogeneous networks as shown in Figure 4.6(d), the high levels of clustering (e.g. at the rack, switch and datacenter levels) may exacerbate the problem of split votes and network partitions.

4.3 Log Replication and Client Semantics

During the evaluation of the protocol, we noted that the client commit latency of the system can be improved by employing the following modifications and optimisations.

Client request cache

The Raft authors argue that linearisable semantics for client commands to the state machine are achieved by assigning all commands a sequential serial number when committed to the cluster by a client. The state machine then caches the last serial

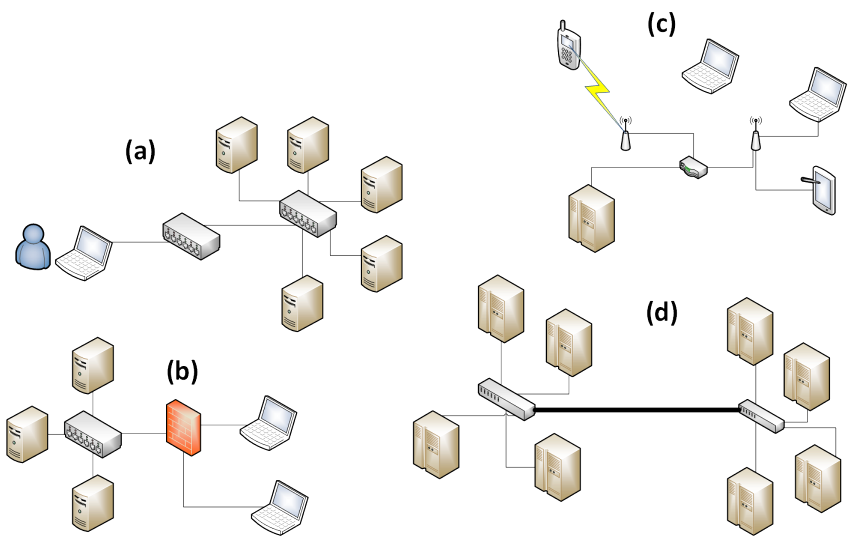


Figure 4.6: Example diverse network topologies: (a) the original deployment; (b) the asymmetric link; (c) the disconnected edge nodes; and (d) the clustered network.

number for each client along with its corresponding response. This is referred to as the *client request cache*. Thus when a client retransmits a request, if the serial number of the committed command is found in the cache, the command is not reapplied to the machine and the cached response is returned. This assumes that commands are committed to the state machine in order of serial number; thus each client is limited to at most one outstanding request.

Our first approach, implied by the paper at the time, was to handle this completely at the state machine level, i.e. a command's serial number and the state machine cache are abstracted from the consensus module so all client requests are added to the log regardless of serial number. First, we observed logs being populated with repeated sequential commands. This reduces performance, but is safe as the command will only by applied to each state machine once. The greater challenge was the question of whether the state machine should handle commands whose serial numbers are neither the same as the last serial number or the successor. Commands with lower serial numbers can arrive from client retransmission or network duplication and network delay. Our state machine simply ignores these commands since we know that the client cannot possible be waiting for a response. Commands with a higher serial number (but not the successor) can only occur where a client has given up on committing a command and tried to commit the next command. We deal with this by requiring clients to continue retrying commands until successful. The potential for large numbers of extraneous entries from this approach leads us to optimise this by exposing serial numbers to the consensus module, which can now use a *fast discard* or *fast response* in response to a client's request.

Inspired by the approach taken by Viewstamped Replication [26], we moved the *client request cache* to the consensus module and abstracted it from the state machine. For each incoming request to the leader, the leader looks up the client ID in the *client request cache*:

- If the requested serial number is less then the last serial number, we can ignore the command, as the client must have already received a response.
- If the requested serial number is the same as the last serial number, we reply with the cached response if present.
- If the requested serial number is successor of the last serial number, the command can be added to the log and the last serial number updated. Later, once the command is committed to the state machine, the response can be cached alongside the serial number.

But how can it be safe to only store the last serial number and response, when entries can removed from the log? If any node has ever seen a serial number greater than any others last serial number, then the cached response will never be needed as the client must have received the response from that committed entry. This positioning of the client request cache introduced a livelock bug, which we later address (§4.3).

Client commit latency

As expected, the vast majority of client commands were committed with a latency of approximately $BroadcastTime + RTT$, where RTT refers to the round trip time between the client and the leader. A small minority of commands took considerably longer, most commonly representing leader failure and subsequent new leader election. A significant overhead was observed by clients waiting for responses from failed nodes during leader discovery. Unlike the other RPC's used by Raft, the client commit timer, which is used by clients to retransmit requests to nodes, is much higher than the RTT because the client must wait whilst the leader replicates the client command to the nodes in the cluster. Introducing a *ClientCommit* leader acknowledgement would allow us to split this into two distinct timers, the first which is just above RTT handles retransmitting normal *ClientCommit* requests, whilst the second is used after the client has received a leader acknowledgement and is much higher to allow the node time to replicate the request.

Leader discovery protocol

The authors recommend contacting nodes randomly to discover which node is the leader and try another node at random if the leader appears to have failed. We chose to instead contact nodes using round robin to reduce the system's non-determinism and thus make it easier to validate the simulation traces. We recommend retrying the leader before moving back to trying other nodes. This is because if the client has not heard from the leader it could simply be due to network delay or loss. Otherwise, even if the leader has failed, it may have recovered since. If it has then it is more likely than other nodes to win an election due to the extra condition on elections (§2.3.3). Our recommendations for leader discovery approach are:

- if the target application is highly latency sensitive, we suggest an approach such as broadcasting *ClientCommit* to all nodes or having a leader broadcast to clients when they come into power.
- if the target application is highly traffic sensitive, we suggest clients back off when nodes do not know which node is the leader as it is likely that there is no leader.

Commit-blocking livelock

We believe we have observed permanent livelock in our simulation trace caused by the interaction between the extra condition on commitment (detailed in §2.3.3) and the placement of the client request cache (detailed previously). The situation described in Figure 4.7 is one example of this livelock. Figure 4.7 shows this livelock when there is only one client, but we believe this can also occur when there are multiple such clients. We recommend that if a client request is blocked by the extra condition on commitment, the leader should create a no-op entry in its log and replicate this across the cluster, as shown in Figure 4.7. Later, when optimising the handling of read only commands, we implemented the authors' suggestion of appending a no-op at the start of each leadership term which is also sufficient to fix this problem.

4.4 Testing

We have extensively tested this project throughout the implementation, making particularly effective use of static typing. Where erroneous behaviour cannot be detected at compile time, we utilised unit testing specific modules with OUnit, assertion checking throughout the code, automated checking for each simulation trace and checking each node's activity against a safety monitor in SPL. All of these were checked in a range of environments using Travis CI, a continuous integration service for open source projects, which runs the test framework against the code for each git commit or pull request.

Assertions are used liberally throughout the code base. This means that safety checks are constantly being made as the code runs and so any runtime errors which

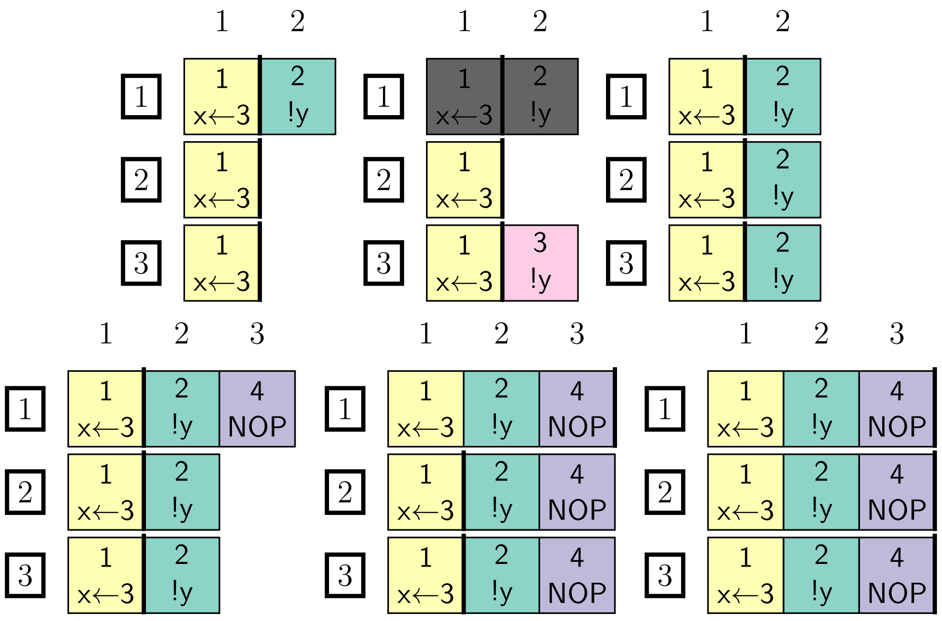
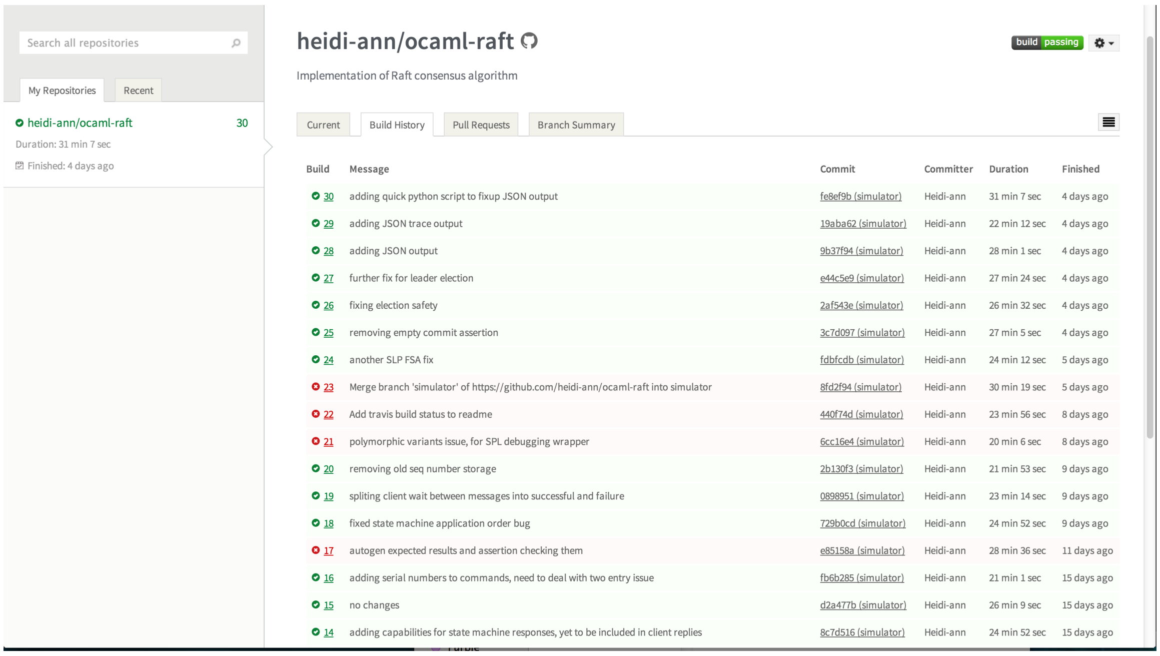


Figure 4.7: Example of livelock and our proposed solution. Top row from left to right: It is term 2 and node 1 as leader is trying to commit !y at index 2. Node 1 failed before replicating !y to another node and node 3 is now trying to commit !y in term 3. Node 3 failed before replicating !y to another node and node 1 is now leader in term 4. Node 3 recovers and node 1 replicates !y to it. No node will be able to commit !y until they commit an entry from their own term but the client will not send new entries until !y is committed. The client retries !y but since it is already in the logs, it is ignored by the nodes and the cluster will not be able to make progress. Bottom row: node 1 creates a no-op and replicates it across the nodes, so it can now commit !y.



Implementation of Raft consensus algorithm

Build	Message	Commit	Committer	Duration	Finished
✓ 30	adding quick python script to fixup JSON output	fe8ef9b (simulator)	Heidi-ann	31 min 7 sec	4 days ago
✓ 29	adding JSON trace output	19aba62 (simulator)	Heidi-ann	22 min 12 sec	4 days ago
✓ 28	adding JSON output	9b37f94 (simulator)	Heidi-ann	28 min 1 sec	4 days ago
✓ 27	further fix for leader election	e44c5e9 (simulator)	Heidi-ann	27 min 24 sec	4 days ago
✓ 26	fixing election safety	2af543e (simulator)	Heidi-ann	26 min 32 sec	4 days ago
✓ 25	removing empty commit assertion	3c7d097 (simulator)	Heidi-ann	27 min 5 sec	4 days ago
✓ 24	another SLP FSA fix	fdbfcdb (simulator)	Heidi-ann	24 min 12 sec	5 days ago
✗ 23	Merge branch 'simulator' of https://github.com/heidi-ann/ocaml-raft into simulator	8fd2f94 (simulator)	Heidi-ann	30 min 19 sec	5 days ago
✗ 22	Add travis build status to readme	440f74d (simulator)	Heidi-ann	23 min 56 sec	8 days ago
✗ 21	polymorphic variants issue, for SPL debugging wrapper	6cc16e4 (simulator)	Heidi-ann	20 min 6 sec	8 days ago
✓ 20	removing old seq number storage	2b130f3 (simulator)	Heidi-ann	21 min 53 sec	9 days ago
✓ 19	spliting client wait between messages into successful and failure	0898951 (simulator)	Heidi-ann	23 min 14 sec	9 days ago
✓ 18	fixed state machine application order bug	729b0cd (simulator)	Heidi-ann	24 min 52 sec	9 days ago
✗ 17	autogen expected results and assertion checking them	e85158a (simulator)	Heidi-ann	28 min 36 sec	11 days ago
✓ 16	adding serial numbers to commands, need to deal with two entry issue	fb6b285 (simulator)	Heidi-ann	21 min 1 sec	15 days ago
✓ 15	no changes	d2a477b (simulator)	Heidi-ann	26 min 9 sec	15 days ago
✓ 14	adding capabilities for state machine responses, yet to be included in client replies	8c7d516 (simulator)	Heidi-ann	24 min 52 sec	15 days ago

Figure 4.8: Screenshot of Travis CI web GUI for testing git commit history.

may have slipped through can be caught near the source rather than propagating through the code. This made locating the source of errors much easier and also helps in assessing the safety of the protocol as any invalid behaviour will be checked for and caught rather than create erroneous results.

As an additional, high level form of testing the calibrations run against the reference implementation results from the Raft paper (§4.1) were helpful in determining that the behaviour exhibited by this implementation was consistent with how the protocol is expected to behave.

4.5 Summary

The complexity of the project far exceeded our expectations. Nevertheless we met all of the success criteria of the project including one of our optional extensions by using our work to propose a range of optimisations to the protocol. The project has been a great success: we have implemented the Raft consensus protocol and used our simulation framework to ensure that protocol operates in asynchronous environments with unreliable communication. Our key value store provides strong consistency from the linearisable client semantics and a simple protocol wrapper allows our implementation to be run as a real-world implementation. Our implementation is able to make progress whilst the majority of nodes are available, with the patch described in §4.3.

Chapter 5

Conclusion

During the course of this project, a strong community has built up around Raft with over 20 different implementations in languages ranging from C++, Erlang and Java to Clojure, Ruby and Python. The most popular implementation, go-raft, is the foundation of the service discovery and locking services used in CoreOS[1] as well as being deployed in InfluxDB[2] and Sky[3]. Distributed consensus is never going to be simple and Raft's goal of being an understandable consensus algorithm aims to address a clear need in the community.

In a well-understood network environment, the protocol behaves admirably; provided that suitable protocol parameters such as follower timeouts and leadership discovery methods have been chosen. As demonstrated in the evaluation, our simulator is a good approximation to Raft's behaviour and is a useful tool for anyone planning to deploy Raft to rapidly evaluate a range of protocol configurations on their specific network environment. Despite this, further work is required on the protocol before it will be able to tolerate modern internet-scale environments, as demonstrated in our evaluation. In particular, a cluster can be rendered useless by a frequently disconnected node or a node with an asymmetric partition (such as behind a NAT).

5.1 Recommendations

We would recommend the following modifications/implementation choices to future implementations of the Raft Consensus protocol and subsequent deployments:

- **Separation of follower and candidate timeouts** – We demonstrated (§4.2) significant performance improvements from the separation of follower and candidate timers, allowing us to set them more appropriately for their purposes. We also

[1]coreos.com
[2]influxdb.org
[3]skydb.io

demonstrated that applying a binary exponential backoff to the candidate timer also provides significant performance benefits.

- ***ClientCommit* leader acknowledgement** – We addressed the high cost of node failure on the client leader discovery protocol, by introducing a *ClientCommit* leader acknowledgement. This would allow the *Client commit* timer to be reduced to slightly above RTT and when the client receives the new leader acknowledgement it can backoff, setting a much higher timer to give the leader time to replicate the command (detailed in §4.3).

- **Leader discovery protocol** – Depending on intended tradeoff between the client latency and traffic: query nodes in a random or round robin fashion, possibly asking multiple nodes in parallel or broadcasting queries to all nodes (detailed in §4.3).

- **Diverse network topologies** – As we have discussed, Raft consensus can experience severe availability difficulties in some network topologies. The authors are considering how to address these and it remains a goal for future work.

- **Client semantics** – It is our understanding, that the use of serial numbers restricts each client to at most one outstanding request. Otherwise a later client command could be added to the leader's log before an earlier one. If the client wishes to have multiple outstanding requests, they can submit the requests as if from multiple clients. The protocol will no longer guarantee that these commands will be executed sequentially, but all state machines will commit the commands in the same order (detailed in §4.3).

- **Read commands** – Read commands[4] need not be replicated across all nodes. It's sufficient to execute them only on the leader, assuming that the leader has committed an entry from its term and recently dispatched a successful *AppendEntries* to a majority of nodes.

- **Commit blocking** – We demonstrated a liveness bug (§4.3) from the interaction of the client request cache and the extra condition. We went on to propose a solution by allowing leaders to add no-ops to their logs.

- **Client instability** – To tolerate client failure and subsequent recovery, clients need to store their current serial number and outstanding command on non-volatile storage or take a new client ID. All clients must have distinct ID's.

[4]This is referring to any class of commands whose execution is guaranteed to not modify the state machine.

5.2 Future Work

In the future we would like to extend the project to include:

- **Full Javascript interface** – Combining our simulation trace visualiser with js_of_ocaml and an interface for specifying simulation parameters including loss and partitions on a link-by-link basis and failure rates on a per node basis.

- **Solutions to liveness challenges of diverse network topologies** – evaluate a range of possible approaches and determine if the best is sufficient to make Raft consensus suitable for building distributed systems between users' devices on the edge network [30].

- **Comparison to Viewstamped Replication (VR)** – Utilise the OCaml module system to implement VR within the same simulation framework and compare performance.

- **Log compaction and membership changes** – Implement and evaluate Raft's proposed solutions for log compaction and membership changes, drawing comparisons to the approach taken by existing consensus algorithms.

- **Byzantine fault tolerant Raft** – Taking inspiration from existing literature on Byzantine tolerant Multi-Paxos [24], design and implement the first Byzantine tolerant Raft.

Bibliography

[1] Eric Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[2] Eric A Brewer. Towards robust distributed systems. In *Principles of Distributed Computing*, page 7, 2000.

[3] Randy Brown. Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10), 1988.

[4] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, 2006.

[5] Tushar Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live-an engineering perspective (2006 invited talk). In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, volume 7, 2007.

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[7] Bryan Clark, Todd Deshane, Eli Dow, Stephen Evanchik, Matthew Finlayson, Jason Herne, and Jeanna Neefe Matthews. Xen and the art of repeated research. In *Proceedings of the USENIX Annual Technical Conference*, pages 47–47, 2004.

[8] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[10] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[11] Rick Siow Mong Goh and Ian Li-Jin Thng. Dsplay: An efficient dynamic priority queue structure for discrete event simulation. In *Proceedings of the SimTecT Simulation Technology and Training Conference*, 2004.

[12] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[13] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[14] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[15] Leslie Lamport. Paxos made simple. *ACM SIGACT News 32.4*, pages 18–25vi, 2001.

[16] Leslie Lamport. Generalized consensus and paxos. *Microsoft Research, Tech. Rep. MSR-TR-2005-33*, 2005.

[17] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[18] Leslie Lamport and Mike Massa. Cheap paxos. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 307–314, 2004.

[19] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[20] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical report, MIT technical report MIT-CSAIL-TR-2012-021, 2012.

[21] Anil Madhavapeddy. *Creating high-performance statically type-safe network applications*. PhD thesis, University of Cambridge, 2006.

[22] Anil Madhavapeddy. Combining static model checking with dynamic enforcement using the statecall policy language. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, pages 446–465, 2009.

[23] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the eighteenth international conference on architectural support for programming languages and operating systems*, pages 461–472. ACM, 2013.

[24] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine paxos. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 402–411, 2004.

[25] David Mazieres. Paxos made practical. `http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf`.

[26] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17, 1988.

[27] Diego Ongaro. Consensus: Bridging theory and practice. PhD thesis, Stanford University, 2014 (work in progress). `http://ramcloud.stanford.edu/~ongaro/thesis.pdf`.

[28] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference*, 2014.

[29] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[30] Charalampos Rotsos, Heidi Howard, David Sheets, Richard Mortier, Anil Madhavapeddy, Amir Chaudhry, and Jon Crowcroft. Lost in the edge: Finding your way with Signposts. In *Proceedings of the third USENIX Workshop on Free and Open Communications on the Internet*, 2013.

[31] David W Scott. Box–muller transformation. *Wiley Interdisciplinary Reviews: Computational Statistics*, 3(2):177–179, 2011.

[32] Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, (3):219–228, 1983.

[33] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[34] Romain Slootmaekers and Nicolas Trangez. Arakoon: A distributed consistent key-value store. *SIGPLAN OCaml Users and Developers Workshop*, 2012.

[35] Frederic Tuong, Fabrice Le Fessant, and Thomas Gazagnaire. OPAM: an OCaml package manager. *SIGPLAN OCaml Users and Developers Workshop*, 2012.

[36] Robbert Van Renesse. Paxos made moderately complex. `http://www.cs.cornell.edu/courses/cs7412/2011sp/paxos.pdf`, 2011.

[37] András Varga et al. The OMNeT++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference*, volume 9, page 185, 2001.

Appendix A

Raft Specification for Simulation

The following Raft specification has been updated and extended from the original Raft specification [28] for the purpose of simulating the protocol and testing a range of implementation choices.

A.1 State

Persistent State (for all modes of operation)

- Current Term – Monotonically increasing value, initialised at start up and incremented over time, used to order messages.
- Voted For – The node which has been voted for in the current term, if any. This value is cleared each time the current term is updated and once it has been set it cannot be changed, thus each node will only hand out one vote per term.
- Log – The replicated log, each entry includes the client's command, command serial number, log index and term.
- Last Log Index & Term – Index and term associated with most recent entry in log.
- State Machine – The state machine which is being replicated[1].
- Commit Index – The index of the last command to be passed to the replicated state machine, initialised at startup and monotonically increasing[1].
- Client Serial Numbers and responses – For each client, the serial number of the last command that has been committed to the state machine. Each is initialised at start-up and monotonically increasing. Associated with each serial number is a cached copy of the response from the state machine.

- ID – Unique reference to this particular node.
- All Nodes – References (and network addresses) to all other nodes in the cluster.

Volatile State (for all modes of operation)

- Mode – Current mode of operation (as shown in Figure 2.2).
- Timers – Implementation of various timers, dependent on current mode, including packet re-transmissions and follower, candidate and leader timeouts.
- Time – Function from unit to time, specific implementation depends on if a simulation mode like discrete is being used thus it returns the current simulated time or realtime which uses the unit to operate on system local time.
- Leader – The current leader (if known) to redirect client to.
- Safety Monitor – SPL automata to catch run-time behaviour leading to divergence from safe protocol NFA.

Volatile State (for candidates)

- Votes Granted – List of nodes who have successfully granted this candidate a vote, initialised to empty when election is started/restarted and only appended to. A pre-appending check ensures that a node isn't already present in list ensuring that votes are only counted once per replica, if network duplicates packets, and asserts that the node's ID is not present in the Votes Not Granted list.
- Votes Not Granted – List of nodes who have not granted this candidate a vote, initialised to empty when election is started/restarted and only appended to. A pre-appending check ensures that a node isn't already present in list, ensuring that votes are only counted once per node, if network duplicates packets, and asserts that the node's ID is not present in the Votes Granted list.
- Backoff Election Number – Number of elections this Candidate has run which have received a rejection from a majority of candidates.

Volatile State (for leaders)

- Next Index – For each node, the leader maintains an overestimate of how consistent the node's log is with its own. This value is initialised to one more then the leader last index and denotes which log entries will be included in the next *AppendEntries* RPC.
- Match Index – For each node, the leader maintains a safe underestimate of how consistent the node's log is with its own. This value is initialised on becoming leader and is monotonically increasing.

- Outstanding Client Requests – All requests that have been received and the client is awaiting responses for (at most one per client).

State for each Client

- Workload – The set of commands and associated serial numbers, that the client will attempt to commit.
- Expected Results – Expected outcome of each item in the workload to check responses, automatically generated by applying workload to local copy of state machine.
- Time – Function from unit to time, specific implementation depends on if a simulation mode like discrete is being used, thus it returns the current simulated time, or realtime which uses the unit to operate on system local time.
- All Nodes – References (and network addresses) to all nodes in the cluster.
- Leader – Latest node to be selected as leader, if known, or details of how to choose next node to query.
- Outstanding request – The client's outstanding commit request, if applicable (at most one).
- Timer – Implementation of timer, to re-transmit packet when nodes fail or packets are lost.

A.2 RPCs

A.2.1 Append Entries

Arguments: Term, LeaderID, PrevLogIndex, PrevLogTerm, Entries, LeaderCommitIndex

Results: Success, Term

Method Body:

1. Term Handling: If packet Term is strictly less then node's Term then the leader needs neutralising so respond with false and current term. If packet Term is strictly greater then node's Term then the node needs to update its term and step down if its current mode is Candidate or Leader, then proceed. If packet Term is equal to node's Term then proceed
2. Update Follower Timer: Node resets its follower timer as it's now heard from a valid Leader so we don't want the node to timeout

3. Leader Discovery: Node updates its leader to LeaderID for redirecting a client

4. Test Log Consistency: Use the packet's PrevLogIndex and PrevLogTerm to determine which of the following consistency states the node's log is:

    (a) Perfect Consistency – The node's Last Log Index and Term are the same as PrevLogIndex and PrevLogTerm, so logs are identical (under the log matching guarantee)

    (b) Consistent with surplus entries – An entry with PrevLogIndex and PrevLogTerm is present in the node's log, remove all extra log entries and then logs are identical

    (c) Inconsistent – An entry with PrevLogIndex is present in the node's log but its associated term is different

    (d) Incomplete – No entry with PrevLogIndex is present in the node's log as the node's log doesn't have enough entries

    If the log consistency test outcome was inconsistent or incomplete, reply with false and wait for the leader to retry, otherwise (perfectly consistent or consistent with surplus), the node's log is now consistent and ready for new entries

5. Append New Entries: If Entries is non-empty then append the entries to the log and update Last Log index and Term, otherwise, if Entries is empty, this was a heartbeat message so no action needs to be taken

6. Commit Indices: If LeaderCommitIndex is greater than node's current Commit Index and less than Last Log index then update Commit Index and apply the log entries between the old Commit Index and new Commit Index.

A.2.2 Client Commit

Arguments: ClientID, SerialNumber, Class, Command
Results: Successful, LeaderID, Results, ClientID, SerialNumber
Method Body:

1. Check client request cache: Search client request cache for the ClientID, if corresponding SerialNumber is greater than request SerialNumber then discard. If serialNumbers are equal, then reply to client with cached response. Otherwise proceed

2. Leader check – If node is not currently the leader, it responds to the client with Unsuccessful and the ID of the current leader, if known

3. Read Commit – If the Class of command is read and the leader has committed an entry from its current term, then the leader can apply the command to the state machine and reply to client

4. Write Commit – Save the client request, add the new SerialNumber to the client request cache and append entry to log. Broadcast AppendEntries RPCs to all nodes with the new update

5. Application to state machine – When the leader match Index for at least a strict majority of nodes is at least as high as the command entry, then the leader can update its commit Index and apply the command to the state machine

6. Response – The response from the state machine can now be cached in the client request cache, and dispatched to the client

A.2.3 Request Vote

Arguments: Term, CandidateID, LastLogIndex, LastLogTerm
Results: VoteGranted, Term
Method Body:

1. Term Handling: If packet Term is strictly less then node's Term then the candidate needs neutralising so respond with false and current term. If packet Term is strictly greater then node's Term then the node needs to update its term and step down if its current mode is Candidate or Leader, then proceed. If packet Term is equal to node's Term then proceed

2. Vote Granting: Grant vote to this candidate if:

   (a) Mode criteria: node's current state isn't Leader
   (b) Single Vote per Term: node hasn't yet handed out its vote for this term (or it was to this candidate)
   (c) Safe Condition of Leader Elections: candidate's log is at least as complete as follower's log, hence grant if either the candidate's last term is greater than node's last term or if the last terms are equal, the candidate's last index is greater or equal to the follower's last index

3. Update Follower Timer: If vote has been granted, node resets its follower timer as it's now heard from a valid candidate so we don't want the replica to timeout

[1]The state machine and commit index, need not be persistent. If not persistent then on restart initialise the state machine and commit Index, then when commit index is updated the state machine will be brought upto date

Appendix B

Simulation Parameters

Name	Description
Follower Timeout	Distribution of the Follower Timeout
Candidate Timeout	Distribution of the Candidate Timeout
Backoff	Enable the binary exponential backoff for Candidate Timeouts
Leader Timeout	Distribution of Leader Timeout
Client Timeout	Time client waits for response from node
Workload Size	Number of commands for the client to commit
Unsuccessful Commit Delay	Time client waits after unsuccessful commit to try again
Successful Commit Delay	Time client waits after successful commit to try next command
Nodes	Number of nodes in the cluster
Mode	Discrete or Realtime Simulation
Node Failures	Distribution of node failure[1]
Node Recovery	Distribution of time until recovery for nodes
Packet Delay	Distribution of packet delay
Packet Loss	Probability of any packet being lost
Packet Duplication	Probability of any packet being duplicated by the network
History	Store a complete history of the all state changes to nodes
Conservative	Aggregate requests and dispatch only on epoch
Debug	Enable debugging Output
JSON	Enable JSON Output
Iteration	Number of simulations to run
Termination criteria	Terminate simulation once a leader has been elected or all client commands have been committed or time has exceeded a given value

Figure B.1: Table detailing the available simulation parameters