

Benson

A structured voting extension
for an implementation of the
Raft consensus algorithm



Computer Science Tripos Part II

2014

Log rafts like the one pictured [on the cover] were first developed in 1906 by Simon Benson, a Portland-based timber magnate. Although Benson was not the first timberman to transport his company's logs to market by rafting them together, he was the first to develop an ocean-worthy raft that could dependably transport 'millions of feet' at a time.¹ (Joshua Binus 2004)

My project aims to extend an implementation of the Raft consensus algorithm. One main goal is to enable applications running on top of this implementation to reliably scale to a large number of servers, a bit like Benson's rafts 'dependably' transported large numbers of logs.

¹Photo (Underwood and Underwood 1906) and description reproduced with kind permission from the Oregon Historical Society (<http://www.ohs.org>).

Proforma

Name and College	Leonhard Markert, Emmanuel College
Project Title	Extending Raft with structured voting
Examination	Computer Science Tripos, Part II (June 2014)
Word Count	10,529 words
Project Originator	Leonhard Markert
Project Supervisors	Malte Schwarzkopf and Ionel Gog

Original aims of the project

Extending Rafter,² an implementation of the new Raft consensus algorithm³ written in Erlang, with the ability to form quorums using structured voting schemes. This includes implementing and testing voting structure generators for at least the Majority Protocol and the Grid Protocol. In order to compare the performance of my voting algorithms with those used in the original Rafter code, and to measure how different structured voting schemes scale with the number of nodes, benchmarks need to be run with a Rafter cluster in the Amazon Elastic Compute Cloud.

Work completed

All objectives outlined in the proposal have been met: I have written a structured voting scheme module for Rafter, and implemented voting structure generators for the Majority Protocol and the Grid Protocol. As an extension to the original aims of the project, a voting structure generator for the Tree Quorum Protocol has been implemented. Benchmarks in the Amazon cloud have been run, and the results have been analysed and evaluated.

Special Difficulties

None.

²Rafter is an open source project hosted on GitHub (<http://github.com/andrewjstone/rafter>).

³Ongaro and Ousterhout 2014.

Declaration of Originality

I, Leonhard Markert of Emmanuel College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date 8th June 2014

Acknowledgements

Special thanks to ...

Malte Schwarzkopf and Ionel Gog for agreeing to supervise this project, sharing their experience with Amazon EC2 and Erlang, and for their detailed and helpful feedback on my dissertation drafts.

Christian Storm for introducing me to structured voting and teaching me not to invent my own consensus algorithms.

Andrew Stone for explaining some of the more complex bits of Rafter to me, and for quickly fixing that critical performance bug.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Challenges	2
1.3	Related work	2
2	Preparation	4
2.1	Introduction to distributed systems	4
2.2	Introduction to the Raft consensus algorithm	6
2.3	Introduction to structured voting schemes	8
2.4	Software engineering	10
2.4.1	Requirements analysis	10
2.4.2	Modularisation	10
2.5	Languages, tools and libraries	12
2.5.1	Structured voting extension	12
2.5.2	Benchmarking in the Amazon Elastic Compute Cloud	13
2.6	Development environment and backup	13
2.7	Summary	14
3	Implementation	15
3.1	Voting algorithm and data types	15
3.2	Voting structure and voting state visualiser	17
3.3	Voting structure generators	18
3.3.1	Majority Protocol	19
3.3.2	Grid Protocol	19
3.3.3	Tree Quorum Protocol	21
3.4	Integration with Rafter	22
3.5	Memcached frontend	23
3.6	Failure simulation	25
3.7	Logging	26
3.8	Reporting issues and bugs	27
3.9	Summary	28
4	Evaluation	30
4.1	Overall results	30
4.2	Testing	32

4.3	Benchmarking	32
4.3.1	Helper scripts	32
4.3.2	Amazon EC2 set-up	34
4.3.3	Choice of parameters	35
4.3.4	Results and analysis	36
4.4	Summary	40
5	Conclusions	42
	Bibliography	43
A	Additional resources	45
A.1	Raft safety guarantees	45
A.2	Raft correctness proof: intersection requirement	45
A.3	A note on functional programming	46
A.4	Voting structure and voting state type specifications	47
A.5	QuickCheck testing examples	48
A.5.1	Generator example	48
A.5.2	Property example	48
A.6	Memaslap configuration	49
A.7	Additional graphs	50
B	Grid Protocol probability analysis	52
B.1	C-cover probability	52
B.2	CC-cover probability	52
B.3	Quorum probability	53
C	Original project proposal	55

Chapter 1

Introduction

This dissertation describes how I extended an implementation of the new Raft consensus algorithm with the ability to form quorums using structured voting schemes.

1.1 Motivation

As ever-increasing amounts of data are being handled in commercial settings as well as in research, distributed systems for data processing and storage are becoming increasingly ubiquitous. This development brings about the need to increase fault tolerance in the face of arbitrary network and node failures.

According to Eric Brewer’s famous CAP theorem,¹ a partition tolerant system cannot be both strictly consistent and maximally available.² Many recent distributed data stores sacrifice consistency for availability. Some applications, however, require strong consistency guarantees – data backup and configuration management systems, for example. Consensus protocols like Paxos³ and Raft⁴ guarantee consistency at the cost of decreased availability.

Today’s popular distributed data stores use *unstructured* voting schemes. This means that in order for an operation to proceed, a certain number of processes must agree to it. Majority voting is a widely used instance of an unstructured voting scheme, requiring the consensus of a majority of the processes in the cluster. Some systems, notably Dynamo⁵ and the more recent Cassandra,⁶ allow quorums to be smaller than necessary to guarantee consistency. This gives rise to a less well-defined notion of ‘eventual consistency’, and enables the system administrator to trade decreased consistency for increased availability.

Structured voting schemes, on the other hand, impose a logical structure of some kind onto the cluster. They are strongly consistent by default, and take the identity of each process into account when forming quorums, instead of merely counting them (as unstructured voting schemes do). This allows for much more detailed tuning of

¹Fox and Brewer 1999.

²The terms consistency, availability and partition tolerance are defined in Section 2.1.

³Lamport 1998.

⁴Ongaro and Ousterhout 2014.

⁵DeCandia et al. 2007.

⁶Lakshman and Malik 2010.

the system’s availability as the structure of the cluster and the properties of individual nodes can guide the selection and configuration of the structured voting scheme.

In this project, I added support for structured voting schemes to an existing implementation of Raft. My aim was to allow more fine-grained availability and scalability trade-offs for data storage systems built on top of this implementation while still providing the same consistency guarantees as the original Raft algorithm.

1.2 Challenges

So far, structured voting schemes have mostly been confined to academic papers and theoretical analysis. I am not aware of any popular modern distributed storage system that uses structured voting schemes to generate quorums. Since the discovery of a unified framework for specifying structured voting schemes by Storm and Theel,⁷ it has become possible to implement a generic voting structure interpreter that works with any voting structure specified in this way. This allows distributed systems to switch between different voting schemes without having to re-implement the code which decides whether or not a given set of nodes constitutes a quorum.

Due to the lack of previous implementations of structured voting scheme algorithms, I had to design my own, based on the high-level description from Storm’s PhD thesis.⁸

In addition, distributed programming in itself is a challenge: any component may fail at the least convenient time; network connections may be unreliable; messages may be dropped, duplicated or reordered. Fortunately, the programming language Erlang⁹ used in this project provides some tools that make writing distributed systems a bit easier.

Lastly, benchmarking a distributed system running in the cloud is difficult. Thorough planning and a high degree of automation is required to make it at all feasible.

1.3 Related work

The primary source of ideas for this project was Christian Storm’s PhD thesis, ‘Specification and analytical evaluation of heterogeneous dynamic quorum-based data replication schemes’ (Storm 2011), which puts unstructured and structured voting schemes into a common framework and analyses them using Petri nets.

Literature on voting schemes goes back much further: Majority voting¹⁰ was introduced in 1979; the Tree Quorum Protocol¹¹ and the Grid Protocol¹² were invented in the early 1990s.

⁷Storm and Theel 2006.

⁸Storm 2011.

⁹<http://www.erlang.org>

¹⁰Thomas 1979.

¹¹Agrawal and El Abbadi 1990, 1992.

¹²Cheung, Ammar and Ahamad 1992; Kumar, Rabinovich and Sinha 1993.

The Raft consensus algorithm, on the other hand, is a very recent addition to distributed systems toolset. Indeed, at the time of this writing, the original Raft paper is still a draft.¹³ It has nonetheless led to a lot of excitement since its first version, and a few dozen implementations in different languages and in various stages of development exist.

During an internship at TNG Technology Consulting,¹⁴ I produced a working prototype of a key-value store based on structured voting schemes using the Node.js JavaScript framework.¹⁵ This was done under the supervision of Christoph Stock and Christian Storm.

¹³Ongaro and Ousterhout 2014.

¹⁴<http://www.tngtech.com>

¹⁵Node.js (<http://nodejs.org>) is a platform for building network applications in JavaScript.

Chapter 2

Preparation

A lot of reading and careful planning was necessary before I could start extending Rafter with structured voting. I first needed to understand Storm and Theel’s unified framework for specifying voting structures, primarily by reading Storm’s PhD thesis.¹ I had to familiarise myself with Rafter and the programming language it is written in, Erlang; and finally, I had to set up a benchmarking environment in Amazon’s Elastic Compute Cloud.

In this chapter, I will first introduce some of the challenges posed by distributed systems in general. I will give an overview of the Raft consensus algorithm, and then describe structured voting schemes, taking the Grid Protocol as an example. I will mention the software engineering techniques that were applied to the project. Finally, a list of the languages, tools and libraries used in this project as well as a short description of my development environment and backup strategy will be given.

2.1 Introduction to distributed systems: CAP, ACID and BASE

Eric Brewer’s famous CAP theorem states that any distributed system can provide at most two out of the three desirable properties consistency (C), availability (A), and partition tolerance (P).² The following definitions are adapted from the proof of the CAP theorem by Gilbert and Lynch:³

Consistency There must exist a total order on all operations such that each operation looks as if it were completed at a single instant. An important consequence of this *linearisable* (or *atomic*) consistency guarantee is that any read operation that begins after a write operation completes must return that value, or the result of a later write operation.⁴

¹Storm 2011.

²Fox and Brewer 1999.

³Gilbert and Lynch 2002.

⁴As Gilbert and Lynch (2002) point out, the term *consistency* is highly overloaded. The reader may observe that the notion of atomic consistency used in this dissertation subsumes what is called atomicity and consistency in the context of ACID (‘atomic, consistent, isolated, durable’) databases.

Availability Every request received by a non-failing node in the system must result in a response.

Partition tolerance The system continues to operate despite arbitrary message loss. This includes network partitions, where all messages sent from nodes in one component of the partition to nodes in another component are lost.

Out of those three, partition tolerance is required in almost all cases. As Hale puts it in his article *You Can't Sacrifice Partition Tolerance*:

For a distributed ... system to *not* require partition tolerance it would have to run on a network which is guaranteed to never drop messages ... and whose nodes are guaranteed to never [fail]. [These] types of systems ... don't exist. (Hale 2010)

This leaves designers of distributed systems with the task of finding the right trade-off between consistency and availability, both of which should be considered as continuous rather than binary properties.⁵ With the rise of commercial databases of unprecedented scale over the last decade, consistency (in its strict form as defined above) is in many cases sacrificed for increased availability.⁶ The authors of 'Dynamo: Amazon's Highly Available Key-value Store' describe this as follows:⁷

For systems prone to server and network failures, availability can be increased by using optimistic replication techniques, where ... concurrent, disconnected work is tolerated. The challenge with this approach is that it can lead to *conflicting changes which must be detected and resolved*. (DeCandia et al. 2007)

The conflict detection and resolution mechanisms required by applications built on top of eventually consistent systems increase their complexity compared to those based on consistent systems, which do not need them. This begs the question: are there techniques that could be used to improve availability without giving up on consistency, allowing for simpler systems? The CAP theorem tells us that we can never build strictly consistent and maximally available systems, but a different trade-off might be possible.

⁵Brewer 2012.

⁶This has been heralded as a paradigm shift from the ACID to the BASE ('basically available, soft state, eventually consistent') model (Brewer 2000).

⁷Notable other eventually consistent systems include Cassandra (<http://cassandra.apache.org>), Riak (<http://basho.com/riak>) and HBase (<http://hbase.apache.org>).

2.2 Introduction to the Raft consensus algorithm

Although still a draft, the paper describing the new Raft consensus algorithm⁸ has become widely known very quickly; dozens of implementations in various languages and stages of development already exist.⁹

In the following, I will give a short overview of the Raft consensus algorithm. It is meant to introduce the terminology required for the Implementation and Evaluation chapters.

At its core, Raft uses a replicated state machine architecture, implemented using a replicated log (see Figure 2.1): each log contains the same commands in the same order, so each state machine processes the same sequence of commands. Since the state machines are assumed to be deterministic, each computes the same state and the same sequence of outputs.

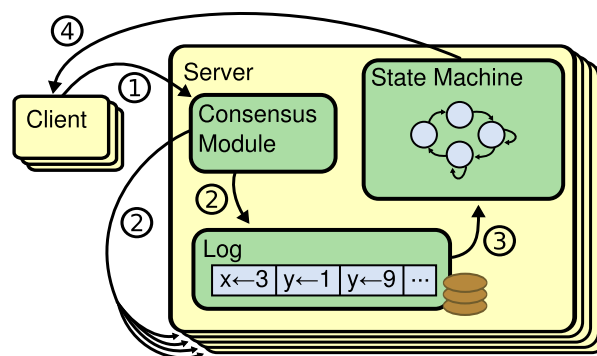


Figure 2.1: Raft’s replicated state machine architecture.

(Ongaro and Ousterhout 2014, p. 2)

Raft implements this replicated log architecture using a collection of servers (a cluster) communicating via remote procedure calls (RPCs), of which there are two: `appendEntries` and `requestVote`. At any given time each server is either a *leader*, *follower*, or *candidate* (see Figure 2.2). In normal operation, there is exactly one leader and all other servers are followers.

This requirement for each server to be in a specific state at any time makes it an *asymmetric* consensus algorithm, as opposed to Paxos, which is *symmetric* since it has no concept of ‘roles’. The designers of Raft made a conscious decision to build an asymmetric voting algorithm since it suits their philosophy of decomposition well:

Having a leader simplifies the management of the replicated log. [...] Given the leader approach, Raft decomposes the consensus problem into three relatively independent subproblems [leader election, log replication and safety.]

(Ongaro and Ousterhout 2014, p. 3)

⁸Ongaro and Ousterhout 2014.

⁹See <http://raftconsensus.github.io/#implementations> for a list of Raft implementations.

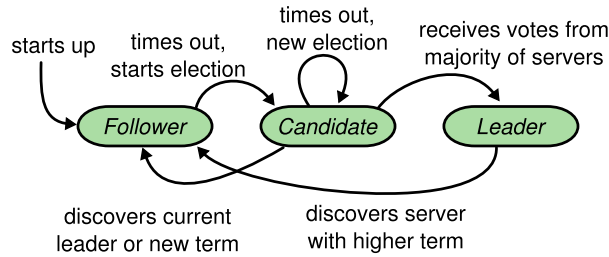


Figure 2.2: Server states in Raft and the transitions between them.

(Ongaro and Ousterhout 2014, p. 5)

Leader election

A server remains in follower state as long as it receives valid heartbeats from a leader or candidate. If any follower receives no communication over a period of time, it assumes that there is no viable leader and starts an election to establish itself as the new leader. To do so, it transitions to *candidate* state and issues `requestVote` RPCs in parallel to all other servers in the cluster. Then one of three things happens: (a) the candidate wins the election, (b) another server establishes itself as leader, or (c) a period of time goes by without a winner, in which case a new election is started.

Each server receiving a `requestVote` RPC will vote for at most one candidate, on a first-come-first-served basis. Once a candidate wins an election, it becomes the new leader and sends heartbeat messages to every other server to establish its authority.

Log replication

Each client request to a leader contains a command to be executed by the replicated state machines. The leader appends the command to its log as a new entry, then issues `appendEntries` RPCs in parallel to all other servers to replicate the entry. When the entry has been safely replicated, the leader applies the entry to its state machine and returns the result of that execution to the client.¹⁰

Safety guarantees

In order to guarantee overall correctness, different components of Raft are designed to make sure certain well-defined safety properties are true at all times (see Appendix A.1 for the complete list.) These properties are shown to hold and then used to prove correctness in the separate correctness proof.¹¹

Since my project involved changing the way in which quorums are constructed, I had to pay special attention to the Election Safety property (stating that there can be at most one leader at any given time) and all the steps in the proof to do with quorums, in order to make sure all the desired properties still hold (see Appendix A.2 for a justification). For a definition of *quorum*, see Section 2.3.

¹⁰For a complete description of what it means for an entry to be ‘safely replicated’, see Ongaro and Ousterhout 2014, subsection 5.3.

¹¹Ongaro and Ousterhout 2013.

2.3 Introduction to structured voting schemes

Voting schemes specify quorums. On an abstract level, they provide a procedure, which, given some knowledge about the entire cluster, decides whether or not a set of processes constitutes a quorum.

A *quorum* is a minimal subset of the cluster that an operation has to obtain Yes votes from in order to be performed. Majority voting, for example, specifies that any set of servers containing at least half the servers in the entire cluster constitutes a quorum.

The Majority Protocol and Read One Write All are examples of unstructured voting schemes: whether or not a set of nodes forms a quorum depends only on its cardinality. In the case of the Majority Protocol, both read and write operations need the consent of a majority of processes; Read One Write All requires all processes to agree on write operations, whereas any one process can serve a read request.

By taking into account the identity of nodes (as opposed to just counting them), and superimposing a logical structure on the cluster, we can build *structured* voting schemes. These allow for smaller quorums, as Figure 2.3 shows.

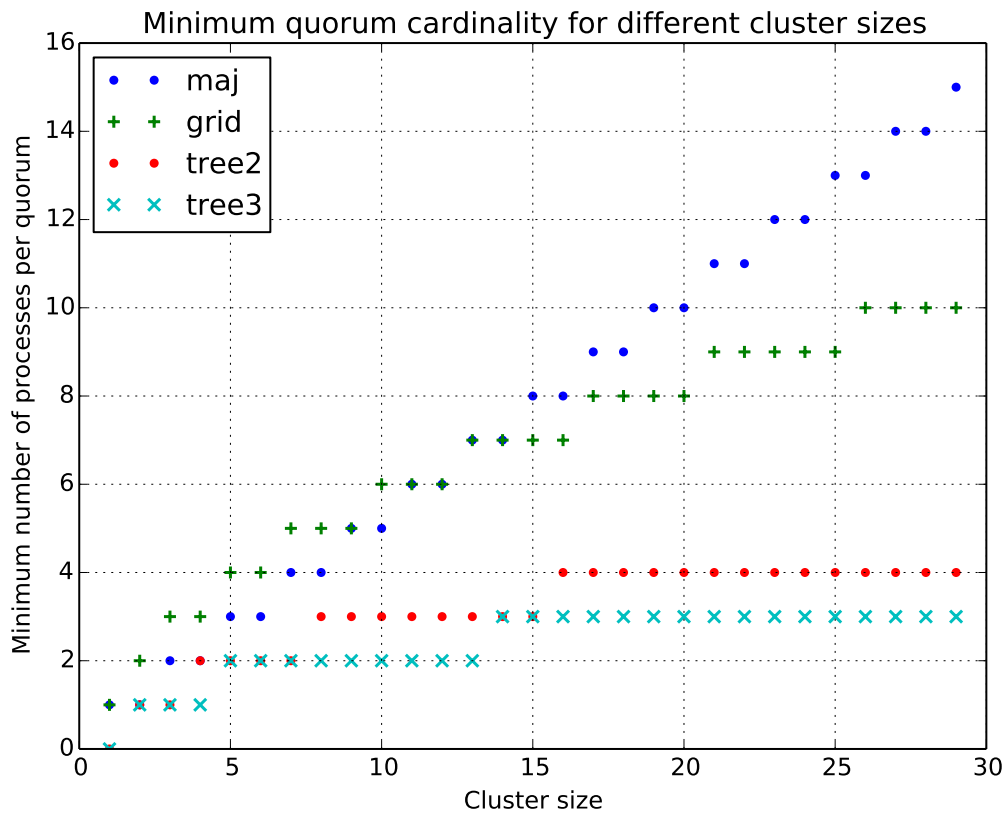


Figure 2.3: Minimum quorum size for the (unstructured) Majority Protocol ('maj'), the (structured) Grid Protocol ('grid') and the (structured) Tree Quorum Protocol ('tree2', 'tree3', for binary tree and ternary trees, respectively)

Quorum systems

A quorum system is defined as a tuple of a read and a write quorum set whose *elements* are called read quorums and write quorums, respectively. These are constructed such that every write quorum intersects with every other write quorum and with every read quorum. Read quorums need not intersect pairwise.¹²

The key realisation that led to this project was that write quorums have precisely the property required for Raft’s Election Safety, namely that any two write quorums intersect. Write quorums were also used to decide whether a write operation is allowed to proceed (since that is what they were originally designed for). From this point on, *quorum* will mean *write quorum* unless specifically noted otherwise.

Structured voting by example

Structured voting schemes impose a logical structure on the set of processes and use structural properties to specify quorum systems. For example, the *Grid Protocol* arranges processes in a logical rectangular grid.¹³ A write quorum consists of all processes from a complete column plus one process from each column.¹⁴

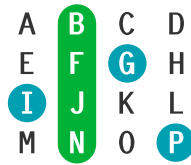


Figure 2.4: A Grid Protocol quorum: The entire second column (B, F, J, N) and one process from each other column (I, G, P) are included.

Specifying voting structures

Storm and Theel present *tree-shaped voting structures* as a universal quorum system representation in the form of semantics-enriched tree graphs. They are produced by voting structure-specific *voting structure generators*.¹⁵

In a tree-shaped voting structure, each node without children (a *physical node*) represents a process. Nodes with children (called *virtual nodes*) represent the structure of the voting scheme and do not correspond to actual processes. The tree is to be interpreted such that votes flow upwards, from the physical nodes (the leaves of the tree) up to the virtual node at its root. Figure 3.1 on Page 16 illustrates this process for a Grid Protocol with four nodes. Examples of voting structures for different structured voting schemes are given in the Implementation chapter.

Note that in general, voting schemes are most naturally represented by directed acyclic graphs. But since from an algorithmic perspective tree structures are easier to operate on, the voting scheme graphs are simply transformed into trees by duplicating nodes as necessary, preserving correctness.

¹²For a formal definition of *quorum*, *quorum set* and *quorum system*, see Storm 2011, section 2.3.

¹³Cheung, Ammar and Ahamad 1992.

¹⁴See Section 3.3.2 for a more detailed description of the Grid Protocol, which also deals with the case where the nodes do not fill the entire grid.

¹⁵Storm and Theel 2006.

When a physical node casts its vote, it is added to the votes its parent virtual node has already received. If the sum of those votes equals or exceeds the parent node's threshold, then the parent node in turn passes its vote to its parent. An annotated example is given in Figure 2.5.

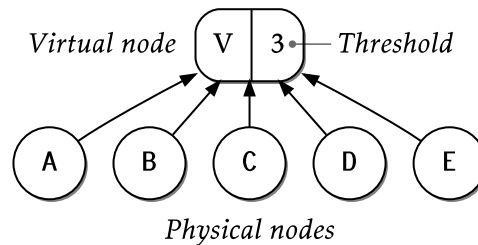


Figure 2.5: A tree-structured voting scheme for a Majority Protocol managing five processes. Three processes are enough to form a quorum, so the threshold of the root node is three.

2.4 Software engineering

An analysis of the requirements made the dependencies between the different components of the project explicit (see Figure 2.6 on Page 12). This helped me draw up a development schedule, which is outlined in my proposal.

The main software engineering technique I used to increase my chances of success were *modularisation* and *testing* as explained below.

2.4.1 Requirements analysis

The requirement for this project to be deemed a success was a demonstration of the extended version of Rafter running some application on a cluster in the Amazon Elastic Compute Cloud, using my structured voting modules to find quorums, and performing similarly to the original version of Rafter by Andrew Stone. Table 2.1 gives a more detailed breakdown of the objectives, and Figure 2.6 shows the dependencies between the different components of the project.

I chose to build a minimalistic Memcached server as the application to run on top of the extended Rafter implementation (see Section 3.5 for more details on Memcached).

2.4.2 Modularisation

This technique permeated all levels of the project. The code itself was written in a functional style, where each function typically spans between one and ten lines. This increased understandability by forcing me to split larger functions into smaller ones, each performing a small and well-defined task.

<i>Objective</i>	<i>Importance</i>	<i>Difficulty</i>
Voting structure and state type specification	●●●	●●
Core voting algorithm	●●●	●●●
Voting structure visualiser	●●	●●
Majority Protocol voting structure generator	●●●	●
Grid Protocol voting structure generator	●●	●●●
Tree Quorum Protocol voting structure generator	●	●●●
Rafter integration	●●●	●●●
Memcached frontend	●●●	●●
Amazon EC2 benchmark set-up	●●●	●●
Benchmarking and analysis of the results	●●●	●●

Table 2.1: Objectives of the project. ●●● high importance or hard; ●● medium importance or difficulty; ● low importance or easy.

All development took place using Git, a popular version control system.¹⁶ I used Git’s branching facility to keep independent pieces of code separate during their development. Whenever code from different branches needed to be combined, I merged the required feature branches into a temporary branch. For convenience, all documentation (including the proposal, the progress presentation and this very dissertation) was kept under its own branch in the same repository.

<i>Branch name</i>	<i>Description</i>
structured-voting	Voting algorithms and voting structure generators
memcached	Memcached frontend
time-to-consensus	Time-to-consensus logging
failure-command	Follower failure simulation
ec2-integration	Amazon EC2 instance administration scripts

Table 2.2: Feature branches used in the project repository.

The voting algorithm, the voting structure visualiser and the generators for the Majority Protocol and the Grid Protocol were initially written and tested independently from Rafter. I integrated it with Rafter only once I was sufficiently confident in the correctness of the code. Thanks to the modularity of Rafter and the clear separation of concerns in my code, the integration process went through without major unexpected difficulties.

¹⁶Git (<http://git-scm.com>) is an open source distributed version control system emphasising speed, originally developed by Linux Torvalds to simplify the development of the Linux kernel. Git supports very cheap and fast branching, a feature I used extensively in this project.

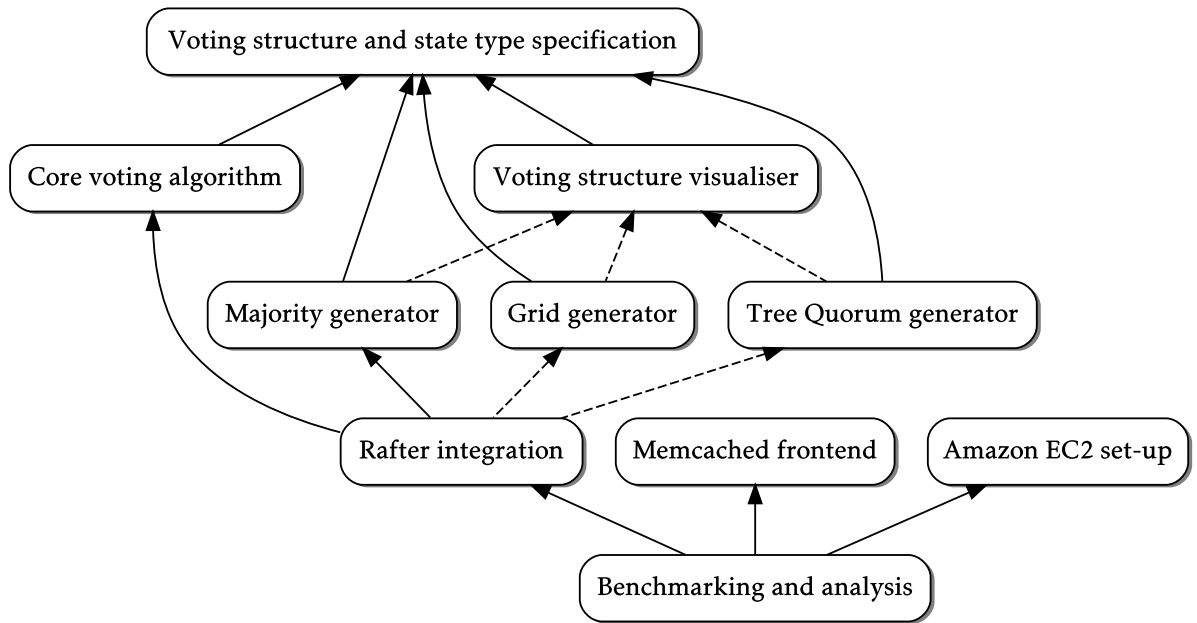


Figure 2.6: Dependency graph of the project’s main components. Arrows point from dependent components to their dependencies. Dashed arrows indicate optional dependencies.

2.5 Languages, tools and libraries

In the inception phase of the project, I was faced with the decision of whether to use an existing implementation of Raft or write one from scratch. Since I was primarily interested in implementing structured voting algorithms and many implementations of Raft already exist, I decided to adapt an existing implementation.

Having written a distributed key-value store in Node.js during an internship, I decided to look for an implementation that used a programming language with built-in support for distributed programming that I had heard of, but never used before: Erlang. Out of the five Raft implementations written in Erlang, Andrew Stone’s *Rafter* seemed to be the most mature.¹⁷

2.5.1 Structured voting extension

The choice of Erlang and Rafter then determined my primary programming toolkit for the structured voting extension:

Erlang A dynamically typed functional programming language originally designed to run telephone switches, Erlang features a run-time system with built-in support for concurrency, distribution and fault tolerance. (<http://www.erlang.org>)

Open Telephony Platform Commonly referred to as ‘the OTP’, the Open Telephony

¹⁷Rafter is an open source project hosted on GitHub (<http://github.com/andrewjstone/rafter>).

Platform provides a set of Erlang libraries and design principles providing middleware to develop Erlang applications. (<http://www.erlang.org/doc>)

QuickCheck A commercial testing framework for Erlang by Quviq SE. Rafter's unit and integration tests use it, and I adapted and amended them to cover my code as well. (<http://www.quviq.com>)

Dialyzer The 'Discrepancy Analyzer for Erlang programs' is used to statically type-check Erlang programs. (<http://www.erlang.org/doc/man/dialyzer.html>)

2.5.2 Benchmarking in the Amazon Elastic Compute Cloud

In order to test and benchmark my code in a realistic environment, it needed to be run on a cluster of machines (as opposed to on a single computer) as this is how systems like Rafter are deployed in the real world. It was decided early on that I would run the project in the Amazon Elastic Compute Cloud (Amazon EC2). This allowed me to execute my benchmarks on up to twenty instances within the same data centre.

Orchestrating a cluster of Amazon EC2 instances and running benchmarks required its own set of tools.

Fabric is a Python library and command-line tool for streamlining the use of SSH for application deployment and systems administration tasks. Administrative functions like starting an Erlang node on an instance or compiling Rafter were defined as *tasks* within the Fabric framework, which could then be executed on arbitrary EC2 instances with the help of the 'awsfabrictasks' plugin. (<http://fabfile.org>, <https://awsfabrictasks.readthedocs.org>)

Boto is a Python interface to Amazon Web Services, which includes the Elastic Compute Cloud. Awsfabrictasks is built on top of Boto, but I occasionally needed to use Boto's lower-level functionality directly. (<http://docs.pythonboto.org>)

Memaslap was used to generate a test load on the cluster. It is part of libMemcached, an open source client library and toolset for the Memcached server. It is highly configurable, but still in an early phase of development, requiring me to patch its Makefile in order to get it to compile. (<http://libmemcached.org>)

2.6 Development environment and backup

The entire project was developed on my private laptop running Arch Linux, using the Vim text editor.¹⁸ I used version R16B03 of the Erlang interpreter and compiler. Apart from a few temporary files and compilation artefacts, every file I created was checked into some branch of the project's Git repository (see Table 2.2 for a list of the main long-running branches created.)

¹⁸Vim is an extensible open source text editor for programmers. <http://www.vim.org>

I used a public GitHub repository at <https://github.com/curiousleo/rafter> as a backup and a way to share files with my project supervisors; I pushed my changes to this repository after every significant change, but at least every few hours while I was working on the project. I took daily backups of my home directory, which contained the project repository, to an external hard drive.

2.7 Summary

In this chapter, I gave a brief introduction to distributed systems, precisely defining the terms consistency, availability and partition tolerance as they are understood in the context of the CAP theorem. This was followed by an overview of the Raft consensus algorithm, where I explained the state machine approach to building distributed systems. Structured voting schemes were introduced by way of an example, namely the Grid Protocol. Tree-shaped voting structures were explained, and a labelled example was given for the Majority Protocol.

I listed this project's objectives, analysed the dependencies between its components, and elaborated on the role of modularisation and testing. Lastly, I talked about the programming languages and tools I used in this project and the development environment and backup strategy employed.

Chapter 3

Implementation

This chapter describes the implementation of the structured voting extension for Rafter and the benchmarking set-up.

The implementation phase of this project took place in two stages: At first, I concentrated on writing the structured voting code, which I subsequently integrated into Rafter.

In the second stage, I set up the benchmarking environment. This included adding logging facilities to Rafter, but also writing Fabric tasks to automate the actual benchmarking process in the Amazon Elastic Compute Cloud.

3.1 Voting algorithm and data types

I started the implementation of the project by defining the data structures used to represent voting structures and voting states. These are passed around between different components, and designing them well was therefore a high priority.

The data structure representing a *voting structure* has two components:

- The voting structure itself. It is a tree whose nodes are annotated with the number of votes and, in the case of virtual nodes, the threshold (see Figure 2.5 for an annotated example).¹
- Some way of locating a physical node in that tree, given its identifier. This was done using a dictionary mapping identifiers to sets of paths.² Each path is a list of integers; starting at the root and descending into the child specified by the head of the list, the path leads to the physical node which represents that identifier in the tree.

As an example, the identifier C in Figure 3.1 has paths [0, 0, 1] and [1, 0, 1] associated with it: C can be reached by going left (0), left (0), right (1) from the root, or via right (1), left (0), right (1).

¹The voting structure diagrams shown in this dissertation do not include the number of votes for each node since in all the voting schemes discussed, every node has exactly one vote.

²As noted in the Introduction, voting schemes are by their nature directed acyclic graphs. In order to transform them into trees, we must occasionally duplicate nodes. This is why identifiers are mapped to *sets* of paths.

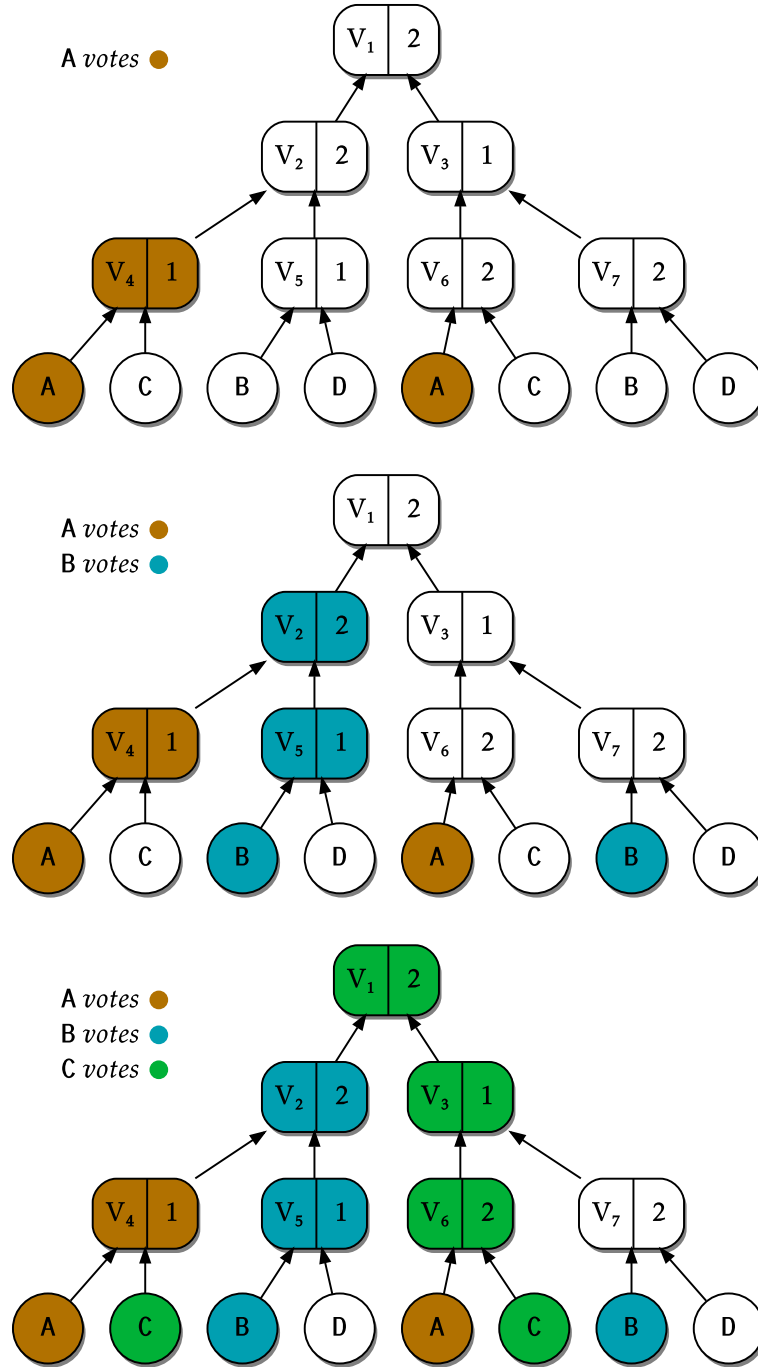


Figure 3.1: A vote using the Grid Protocol. Each node has one vote. A votes first (maroon). V_4 has a threshold of one, so it votes as well. V_2 and V_6 both have threshold two, so their votes are pending. Then B votes (blue), triggering a vote from V_5 . Now both V_4 and V_5 have voted, so V_2 's threshold of two is reached and it votes, too. At this stage, V_1 and V_7 each require one more vote. Lastly, C votes (green). V_4 has already voted, so nothing happens in this branch. V_6 receives a second vote, so it passes its vote to V_3 , immediately triggering it to vote to the root node V_1 , where the threshold is now reached, ending the vote.

Another data type is used in the structured voting code to represent the *state of a vote*. Both are implemented as record types in Erlang, and are called `vstruct` and `vstate`, respectively; their internal structure is very similar (see Listing A.3 for the full definition). `vstate` additionally contains information about if and how a physical node has voted (`true`, `false` or `pending`, corresponding to *Yes vote received*, *No vote received*, and *not yet voted*, respectively), and a summary of the children's votes for each virtual node (represented as integer fields `yes_nodes` and `no_nodes`). They are used as follows:

1. A *voting structure generator* creates a `vstruct`.
2. This voting structure is used to initialise a new voting state `vstate`.
3. The voting state is updated by the voting algorithm whenever new votes arrive until the vote succeeds or fails.

The core voting algorithm as illustrated in Figure 3.1, in combination with the `vstate` data structure described above, lends itself well to a recursive implementation: After looking up the paths in the identifier–path dictionary, the function recurses into the tree along the path and updates the vote at the leaf. After the recursive call, it updates the `yes_votes` and `no_votes` fields of the virtual nodes along the path, ascending to the root. The resulting `vstate` is then returned.

A vote is finished when it either succeeds or fails. It succeeds when the root node of the tree-shaped voting structure votes `Yes`; it fails when the root votes `No`. The voting algorithm I implemented uses a small optimisation to fail early, that is, to vote `No` as soon as a quorum for `Yes` cannot be reached. This works as follows: when a child node votes `No`, its parent node checks whether the threshold is greater than the maximum number of `Yes` votes that can still be cast:

$$\text{threshold} > \overset{?}{\text{possible Yes votes}} = \text{number of children} - \text{No votes received}$$

If that is the case, then a quorum cannot be formed and the node votes `No`.

It follows from the description of the algorithm above that the computational complexity of updating the voting state with one more vote is in the worst case proportional to the height of the tree-shaped voting structure. The voting structure height as a function of the number of nodes is given in the individual voting structure generator subsections below.

3.2 Voting structure and voting state visualiser

Once the Erlang representation of voting structures and voting states was fixed, I wrote a visualiser module for them. This allowed me to visually debug my voting algorithm and the voting structure generators.

The visualiser module generates output in the DOT graph description language. A DOT node is written out for each node in the tree, and edges connect parents and their

children. The dot program, which is part of the Graphviz package from AT&T Labs Research, was then used to create PDF or SVG files for display.³

Many of the tree diagrams in this dissertation were created using this visualisation tool and then edited and annotated using Inkscape.⁴

3.3 Voting structure generators

In this section, I will describe how the voting structure generators for the three voting protocols I implemented work. These generators output tree-shaped voting structures, which are universal in expressing quorum systems.⁵ The challenge in building them is to translate the protocol specifications into tree-shaped voting structures.

To simplify this translation, I will now introduce two common patterns that will be used in the description of the Grid Protocol and the Tree Quorum Protocol: *any*- and *all*-nodes.⁶

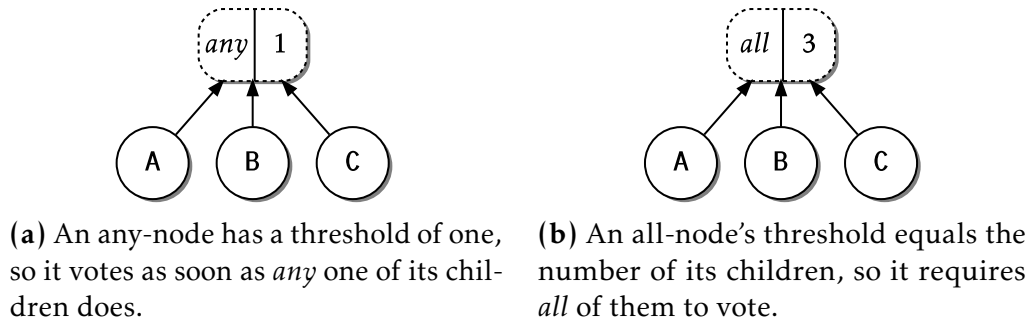


Figure 3.2: Common patterns in tree-shaped voting structures.

Any-nodes have a threshold of one. This means that they vote YES as soon as any one of their children does. In logical terms, any-nodes express a logical *or*: In Figure 3.2a, the root votes if A *or* B *or* C vote. More formally, if for a process p , $p = \text{true}$ denotes that p votes YES,

$$\text{any-node} = \bigvee_{p \in \text{Children}} p$$

By contrast, all-nodes have a threshold equal to the number of children they have, so they vote YES when all their children have voted YES. In this sense, all-nodes are the tree-shaped voting structure equivalent of the logical *and*: The root in Figure 3.2b votes if and only if A *and* B *and* C vote. Taking the same mapping between truth assignments and voting as above,

³Graphviz (short for Graph Visualisation Software): <http://graphviz.org>

⁴Inkscape is an open source vector graphics editor. <http://inkscape.org>

⁵Theel 1993.

⁶Note that the terms any-node and all-node are not used in structured voting literature, but I found them conceptually helpful when designing or reading voting structures.

$$\text{all-node} = \bigwedge_{p \in \text{Children}} p$$

In the following, n refers to the total number of processes.

3.3.1 Majority Protocol

The Majority Protocol (also known as Majority Consensus Voting⁷) requires quorums to contain any $\lceil (n+1)/2 \rceil$ processes. This makes it an unstructured voting protocol. However, unstructured voting protocols can be regarded as a subclass of structured voting protocols, since they fit into the same framework of voting structure generators.

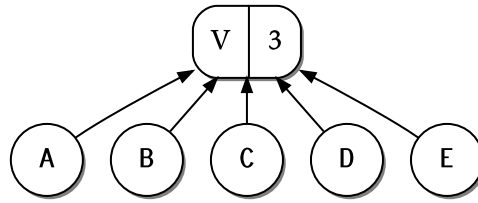


Figure 3.3: Voting structure for the Majority Protocol with five processes and a threshold of three.

This voting structure has a rigid format, and therefore lends itself to a straightforward implementation: It consists of n physical nodes, all sharing the same virtual parent node which also acts as the root of the tree-shaped voting structure. Each physical node has one vote, and the threshold of the root node is $\lceil (n+1)/2 \rceil$, so the vote is decided as soon as this many physical nodes vote YES.

An example for a Majority Protocol voting structure with five processes and a corresponding threshold of three is given in Figure 3.3.

3.3.2 Grid Protocol

The Grid Protocol was introduced in the Preparation chapter as an example of a structured voting scheme (see Section 2.3). It logically arranges processes in a rectangular grid;⁸ a quorum consists of one process from each column plus all processes from a complete column.

Assuming for now that our nodes fit into a rectangular $r \times c$ grid with r rows and c columns⁹ so that the number of nodes is $n = rc$, we can translate the requirements of this specification into a voting structure as follows:

⁷Thomas 1979.

⁸Cheung, Ammar and Ahamad 1992.

⁹The Grid Protocol does not specify where any node should be placed in the grid. This allows the designer of the distributed system to use a placement policy that takes into account properties of the underlying physical network – for instance by making each rack a column in the grid.

‘one process from each column’ This means that for each column, there must be one process which has voted Yes. The literature calls this the C-cover, for *column cover*. In logical terms, it can be expressed as $\bigwedge_c \bigvee_{p \in c} p$, where c ranges over the columns.

Using the analogy between logic and tree-shaped voting structures introduced at the beginning of this section, we can translate this formula into a voting substructure as follows: for each column c , there is an any-node whose children are the processes in that column (this models $\bigvee_{p \in c} p$). The root of the substructure is an all-node, which has the any-nodes as its children, giving $\bigwedge_c \bigvee_{p \in c} p$ altogether. This structure is the ‘C-cover subroot’ in Figure 3.4.

‘all processes from a complete column’ Translating this requirement (called CC-cover for *complete column cover*) into logic, we have $\bigvee_c \bigwedge_{p \in c} p$.

We transform this expression into a tree-shaped voting structure as follows: for each column c , we add an all-node whose children are the processes in that column (giving $\bigwedge_{p \in c} p$). These all-nodes are given an any-node as their parent, modelling $\bigvee_c \bigwedge_{p \in c} p$. This node is called ‘CC-cover subroot’ in Figure 3.4.

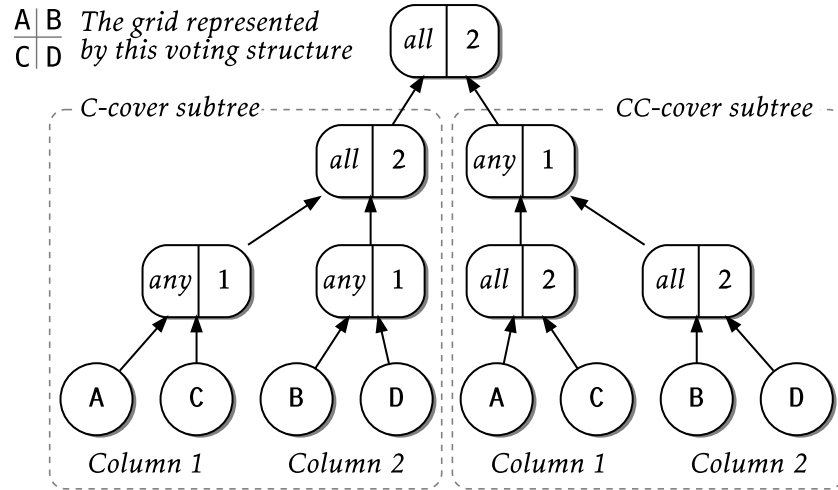


Figure 3.4: Annotated voting structure for the Grid Protocol with four processes.

The Grid Protocol specification demands that both requirements must be fulfilled in order to reach a quorum. We thus make the CC-cover root and the C-cover root the children of an all-node as in Figure 3.4, giving $\left[\bigwedge_c \bigvee_{p \in c} p \right] \wedge \left[\bigvee_c \bigwedge_{p \in c} p \right]$ as required. Note how the tree-shaped voting structure resembles the abstract syntax tree of the logical expression.

Dealing with incomplete grids

In the description of the Grid Protocol above, we assumed that the number of processes n fits nicely into a rectangular grid with r rows and c columns. We can of course always enforce this, even when the number of processes is prime, by letting either $r = 1$ or

$c = 1$. In these extreme cases, however, the Grid Protocol reduces to a voting scheme that requires the consent of *all* nodes for each operation:

$r = 1$ makes the C-cover equal to the set of all nodes;

$c = 1$ makes the CC-cover equal to the set of all nodes.

This is clearly not desirable. Instead, the grid structure imposed on the processes is the smallest almost-square grid that all processes can fit in.¹⁰ Specifically, $r = \lceil \sqrt{n} \rceil$ and

$$c = \begin{cases} \lfloor \sqrt{n} \rfloor & \text{if } \lceil \sqrt{n} \rceil \cdot \lfloor \sqrt{n} \rfloor \geq n \\ \lceil \sqrt{n} \rceil & \text{otherwise} \end{cases} \quad (3.1)$$

Note that this implies $r \geq c$; we are said to be ‘favouring rows’. Swapping r and c results in grids ‘favouring columns’.

The slots in the grid that are not taken up by a process are then considered to be populated by dead placeholder processes. As a little optimisation, we can remove the CC-covers for the columns that contain a placeholder since those columns can never be complete.

A derivation of the probability of finding a Grid Protocol quorum given k Yes votes in a cluster of n processes is given in Appendix B.

3.3.3 Tree Quorum Protocol

The Tree Quorum Protocol¹¹ comes in many flavours which allows for different trade-offs between read and write quorum size to be made.¹² For my voting structure generator, I picked the variant with the smallest write quorums called ‘log write protocol’ in which the cardinalities of write quorums are *logarithmic* in the total number of nodes.¹³

In this protocol, the processes are logically arranged in a tree of degree d (meaning that every parent has d children). For now, we assume that the tree is complete, that is, it has the maximum number of nodes. A quorum is then defined as any set of nodes forming a path from the root to any leaf of the tree.

Take the tree in the top left-hand corner of Figure 3.5 as an example. Building the voting structure bottom-up, we start by assembling a substructure representing the subtree with root B. This substructure should only vote if B and at least one of C, D or E

¹⁰Almost-square here means that c and r differ by at most one, so $|c - r| \leq 1$.

¹¹Agrawal and El Abbadi 1990.

¹²Remember that *read quorums* are minimal subsets of the cluster that have to vote Yes in order for a *read* operation to be performed; *write quorums* are defined analogously.

¹³The ‘log write protocol’ comes with the smallest write quorums and the largest read quorums of all possible Tree Quorum Protocol configurations. The latter should generally to be avoided, but for the purposes of this project, the size of read quorums does not matter since they are disregarded: only write quorums are used for the leader elections.

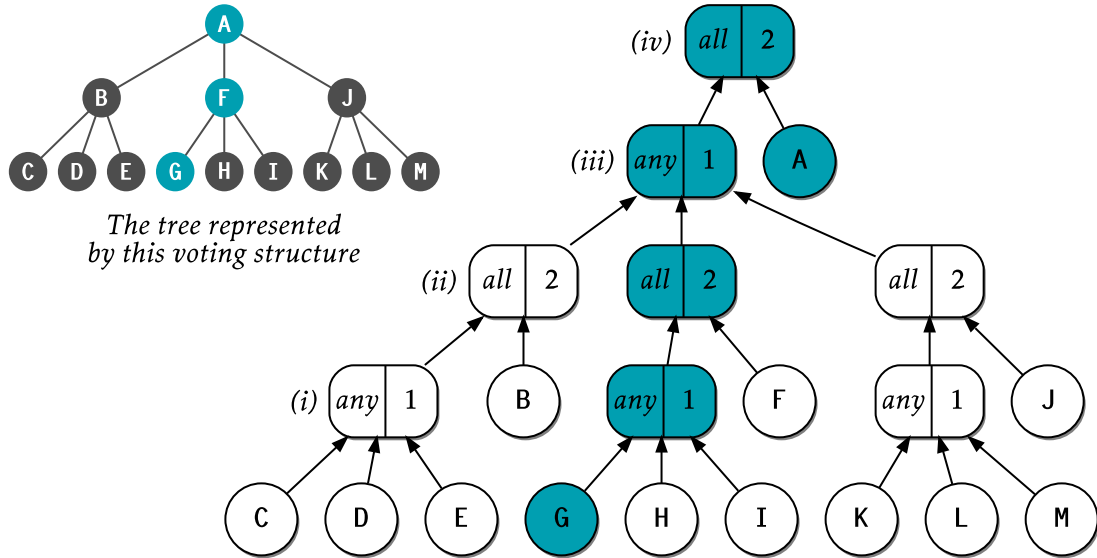


Figure 3.5: Voting structure for the Tree Quorum Protocol based on a ternary tree with 13 processes. Processes A, F and G have voted YES. They lie on a path from the root (A) to a leaf (G), so they form a quorum. (i) – (iv) refer to the description on Page 21.

have voted: $B \wedge \bigvee_{p \in \text{Children}(B)} p$. We can model this easily in terms of any-nodes and all-nodes: C, D and E are given an any-node as their parent – (i) in Figure 3.5 –, and this any-node and B are made the children of an all-node (ii).

We proceed in the same way for the subtrees rooted in F and J (iii), and then for the entire tree with A at its root and B, F and J as its children (iv).

Note that this is the only truly recursive tree-shaped voting structure: Majority Protocol voting structures always have a height of two, and any Grid Protocol voting structure is four nodes high. The height of Tree Quorum voting structures, on the other hand, is logarithmic in the number of nodes.

3.4 Integration with Rafter

Having written and tested the structured voting code, I integrated it into Rafter. This process took place in roughly the following steps:

1. Integrating my code with Rafter meant reading and understanding its source code first: 1,500 lines of code¹⁴ (see Appendix A.3 for a discussion of Erlang’s code density), spread over 11 modules and undocumented except for a few comments, in a language I was still learning. Using module and function names as a guide, I found the relevant pieces of code that needed changing. Andrew Stone, the developer of Rafter, helped me understand the purpose of a few functions whose task I could not figure out myself.

¹⁴As measured by cloc (<http://cloc.sourceforge.net>).

2. I renamed my modules and moved them into the appropriate directory to fit into the naming scheme and directory structure of Rafter. Also, the type specifications had to be pasted into the appropriate include file.
3. In order to use my structured voting code, the node configuration had to be changed from a list of peers (which is all that is needed when only the Majority Protocol is used) to a voting structure (enabling Rafter to use other protocols as well). This required updating every line of code that made use of this field from the configuration record.
4. All the places in the code which implicitly assumed that the Majority Protocol was being used had to be rewritten. In most cases, this meant making them wrappers around my generalised structured voting code.

Thanks to the modularity of both the original Rafter implementation and my own code, this part of the implementation process posed no major unforeseen difficulties.

3.5 Memcached frontend

Memcached is a high performance key-value cache designed to sit between an application and a database. When the application issues a query that has been seen before by Memcached, the result is served from the cache; otherwise, the query is executed on the database and the result stored in Memcached. Originally developed for LiveJournal, Memcached is now used by many high-traffic websites including Youtube, Twitter, Reddit, Facebook and Wikipedia.

My implementation of a Memcached frontend for Rafter takes this protocol out of the context it was originally designed for. First and foremost, Memcached caches are designed to be very fast, at the expense even of durability: clients cannot, in general, expect the cache to keep a value in memory indefinitely – dropping a key-value pair to make place for a new one is perfectly acceptable for a cache like Memcached. Rafter, in contrast, is designed to be durable: every write operation is flushed to disk at once. Therefore, building a Memcached protocol frontend on top of it results in an implementation that is orders of magnitude slower than original Memcached and has different eviction semantics.

I added a Memcached frontend to Rafter for two reasons: to showcase a useful application that could be built on top of it, and to be able to use Memcached's benchmarking tool, `memaslap`.¹⁵

Newer versions of Memcached support a binary protocol,¹⁶ which I found less ambiguous than the standard text-based protocol. Decoding and encoding binary protocols being a particular strength of Erlang, the decision over which protocol to

¹⁵`memaslap` is part of `libMemcached` (<http://libmemcached.org/libMemcached.html>), an open source library and toolset for Memcached.

¹⁶Memcached's new binary protocol is described in this document: <https://code.google.com/p/memcached/wiki/BinaryProtocolRevamped>.

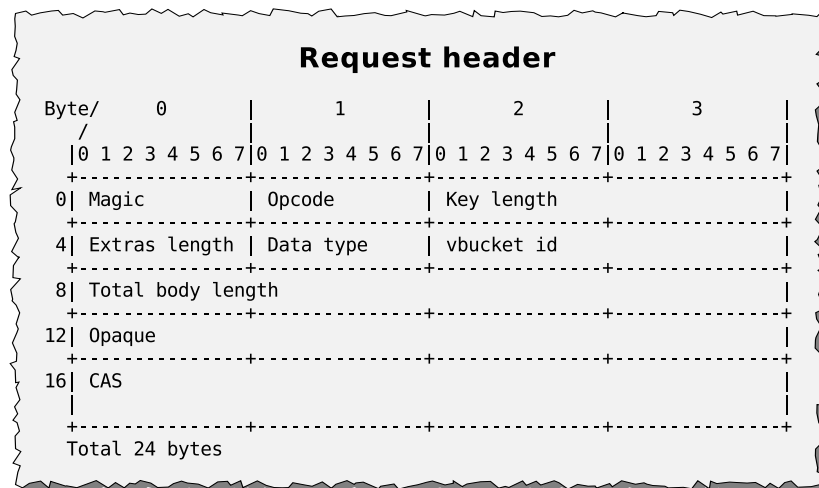


Figure 3.6: Request header format in Memcached's revamped binary protocol.
(<https://code.google.com/p/memcached/wiki/BinaryProtocolRevamped>)

```

case Packet of
  <<_Magic:8,    Opcode:8,    KeyLen:16,
    ExtrasLen:8, _DataType:8, _BucketId:16,
    BodyLen:32,
    Opaque:4/binary,      % binary: 4 bytes = 32 bits
    _CAS:64,              % request header ends here
    Body:BodyLen/binary, % request body
    Next/binary>> ->      % start of next request header
  case {Opcode, ExtrasLen, Body} of
    {?NOOP, 0, <<>>} ->
      % handle noop
    {?GET, 0, <<Key:KeyLen/binary>>} ->
      % handle get
    {?SET, 8, <<Flags:4/binary, Expiry:32, Key:KeyLen/binary,
      Value/binary>>} ->
      % handle set
  end
end

```

Listing 3.1: Memcached request decoder implementation for the binary protocol described in Figure 3.6. The structure of the case `Packet` reflects the structure of the packet header. Note how the variable `BodyLen` captured in the case statement can be used to specify the size of the `Body` variable within the same case statement. Variables preceded by an underscore are unused.

implement was easy. Listing 3.1 illustrates how the decoding of a Memcached request is implemented.

Next, I picked the commands and features I wanted the frontend to support. Since my store is meant to be a proof-of-concept, I decided to stick to the minimal set of commands required for a key-value store: `get`, `set`, and `noop`, with the obvious meaning: `get` is a read request, `set` starts a write operation and `noop` does nothing. For simplicity, I also decided not to support the data version check feature (the CAS header field in Figure 3.6).

Implementing Memcached

The actual implementation of the Memcached frontend consists of three parts: A supervisor, a protocol decoder/encoder, and a backend.

The supervisor starts and watches over the decoder/encoder processes. *Supervisors* are a *behaviour* defined in the OTP, Erlang’s set of core libraries. Crucially, they are notified when a process started by them fails. By default, the Memcached supervisor starts 20 instances of the decoder/encoder, which allows up to 20 clients to connect to the server simultaneously.¹⁷ Whenever one of them crashes, a new decoder/encoder is brought up.

The protocol decoder and encoder makes up the actual ‘frontend’: this is the component a Memcached client talks to directly via TCP.

The backend finally executes the commands decoded by the frontend by reading from and writing to a local database. In Raft terminology, this module constitutes the *state machine*. It uses Erlang’s built-in `ets` (Erlang Term Storage) module for storing the data.

3.6 Failure simulation

In order to evaluate and compare the different voting structures, I added failure simulation to Rafter’s state machine. Apart from being in leader, follower or candidate state, nodes can now also be in the *failed* state. Failed nodes do not process any incoming messages, nor do they send any messages to other nodes.

According to Jeff Dean of Google, two of the most common failure modes in a modern data centre are individual machine failures (called *repeated failures* in the following) and rack failures (correlated failures).¹⁸ During a preliminary theoretical analysis, I could find no evidence that structured voting schemes would be able to deal with rack failures better than unstructured protocols. Hence only repeated/individual failures were considered.

¹⁷My measurements show that increasing the number of decoders above 20 does not significantly improve the throughput.

¹⁸Dean 2009, sl. 10.

Here is how I implemented *repeated failures*: the idea is that a parameter $Up \in [0, 1]$ specifies the uptime ratio of each node. After running for a time $t \sim \text{Exp}(\lambda)$, nodes fail for time $t \cdot \frac{1-Up}{Up}$. This results in followers being ‘up’, that is, actively participating in the cluster, for approximately $Up \cdot 100\%$ of the time:

$$\frac{\text{uptime}}{\text{total time}} = \frac{\text{uptime}}{\text{uptime} + \text{duration of failure}} = \frac{t}{t + t \cdot \frac{1-Up}{Up}} = \frac{Up}{Up + 1 - Up} = Up$$

Since t is an inter-event time, generating it using an exponential distribution is an obvious choice. Erlang’s core libraries provide a function for generating uniformly distributed floating point numbers $U \sim U(0, 1)$. Using the formula introduced in the Part II Computer Systems Modelling course, t is calculated from U and the parameter λ as $t := -\frac{1}{\lambda} \log U$.

The actual failure simulation then proceeds as follows:

1. The benchmark runner sends a message to the leader with parameters λ and Up .
2. Immediately, the leader sends a message to some or all of its followers to start failing.
3. On receiving this message, each follower waits for time $t \sim \text{Exp}(\lambda)$ before transitioning to failed state.
4. After waiting for time $t \cdot \frac{1-Up}{Up}$, the follower goes back to *follower* state.
5. Followers repeat steps (3.) and (4.) until they receive a ‘stop’ message from the leader.

Unfortunately, I was unable to simulate failures in this way on Amazon EC2, so the failure simulation could only be run locally (that is, on my laptop – see Figure A.1 for the results).

3.7 Logging

In order to collect benchmarking data, I created a simple logging module. This module implements the *gen_server* behaviour defined in the OTP. It runs in a separate process under its own supervisor. The logger receives messages in a certain format from other components of the system, and writes the values out in a comma-separated values (CSV) file. The fields recorded in the CSV file are:

Experiment the voting scheme used and its parameters, if any;

Cluster the number of nodes in the cluster;

Operation Read or Write;

<i>Experiment</i>	<i>Cluster</i>	<i>Operation</i>	<i>TTC (μs)</i>
		\vdots	
Majority	3	Write	2920
Majority	3	Read	2166
		\vdots	
Grid	11	Write	9801
Grid	11	Read	7269
		\vdots	

Table 3.1: Example TTC log snippet.

Time to consensus (TTC) the time taken between receiving the request and returning its result to the client in microseconds.

Rafter puts information related to a specific request in a record of type `client_req`. For logging purposes, I added a field `started` to it, which is initialised to the current system time whenever a new client request record is created. I then amended the function sending the client response to also send a message to the logger containing the time difference between the current system time and the time when the request was started.

I conducted a short series of tests to see if turning on logging decreased the performance of Rafter. No statistically significant effect on the throughput could be found. There are two main reasons for this: firstly, as mentioned above, the logger runs in its own process and receives messages from other components of the system. This means that when a process wants to add an entry to the log, it sends a message to the logger. The program flow is not interrupted: the message is simply queued in the logger's message inbox and the process that sent the message continues execution. The logger acts upon the message at some later point when the Erlang scheduler decides to switch to its message handling procedure. Hence the logger automatically benefits from Erlang's scheduling and parallelism features, decreasing the time penalty.

Secondly, the CSV files are written to a folder in `/tmp`, which is mounted as a `tmpfs` file system. Under normal operating conditions, this complete file system lives in main memory and is never written to disk. This makes write operations very cheap, again keeping the time penalty for logging at a minimum.

3.8 Reporting issues and bugs

During the development of this project, I noticed several issues with the software I was using (see Table 3.2 for a summary). In some cases, this was due to a misunderstanding on my part – for example, I was having trouble setting up and using QuickCheck, so I contacted Thomas Arts of Quviq who helped me sort things out. In some cases,

<i>Project</i>	<i>Issue</i>
Rafter	CPU usage and operation execution time increase with the number of operations performed ^a
	More consistent variable naming in README ^b
Awsfabrictasks	ec2instance decorator executed too early ^c
	ec2instance login problem ^d
	Documentation correction ^e
StatsBase.jl	Corrections in README ^f

^a Issue report: <https://github.com/andrewjstone/rafter/issues/22>

^b Pull request: <https://github.com/andrewjstone/rafter/pull/17> (merged)

^c Issue report: <https://github.com/espenak/awsfabrictasks/issues/15>

^d Issue report: <https://github.com/espenak/awsfabrictasks/issues/17>

^e Pull request: <https://github.com/espenak/awsfabrictasks/pull/16> (open)

^f Pull request: <https://github.com/JuliaStats/StatsBase.jl/pull/48> (merged)

Table 3.2: Issues with third-party software that were found and reported during the development of the project.

I found mistakes in the documentation: when this happened, I forked the project, corrected the mistake, and created a pull request.¹⁹ This caused me to create pull requests for Rafter, StatsBase.jl, a statistics package for the scientific programming language Julia that I used to analyse the benchmarking data, and Awsfabrictasks.

Apart from these issues, I found two actual bugs in the software I was using. The first one concerned the performance of Rafter: I noticed that both read and write operations took more and more time the longer Rafter had been running for. I wrote a small test script, took some measurements and reported the problem to Andrew Stone, the original author of Rafter, who responded quickly and managed to fix the issue within a few days.

The second larger issue I found was a problem with a very common use case that was not well addressed in Awsfabrictasks's API. I reported this to the software's author, Espen Angell Kristiansen; eventually, this discussion turned into a proper bug report.

3.9 Summary

In this chapter, I described the implementation of the structured voting extension for Rafter, and the set-up required for benchmarking. The chapter started with a description of the data types used to represent voting structures and voting state.

¹⁹This is the development model used by most projects hosted on GitHub. *Forking* copies the project to one's personal space. The contributor creates a new branch, attempts to fix the bug, and opens a *pull request* for that branch. A pull request can then be reviewed, discussed and amended until it is either rejected or accepted and merged into the original project's master branch.

I briefly talked about the visualiser, which was used to visually debug the voting algorithms and the voting structure generators.

The voting structures for the Majority Protocol, the Grid Protocol and the Tree Quorum Protocol were derived and explained. I described the process of integrating my structured voting code into Rafter before detailing the two extensions to Rafter I implemented to make benchmarking possible: the Memcached frontend and failure simulation. Finally, I mentioned the bug reports and pull requests I created for various pieces of software I used in this project.

Chapter 4

Evaluation

In this chapter, I first compare my achievements with the requirements in the original project proposal. This is followed by a description of my unit test rig. The rest of the chapter specifies the benchmarking set-up and discusses the results, looking at time to consensus and throughput measurements to assess how my structured voting implementation compares with Rafter’s built-in Majority Protocol and how different structured voting schemes scale with cluster size.

4.1 Overall results

All the success criteria listed in my project proposal have been met, and in some areas, I have gone beyond what the criteria demanded (see Appendix C for the original proposal).

1. *Design of the data structure to represent voting structures, and implementation and testing of the algorithm to interpret it.*

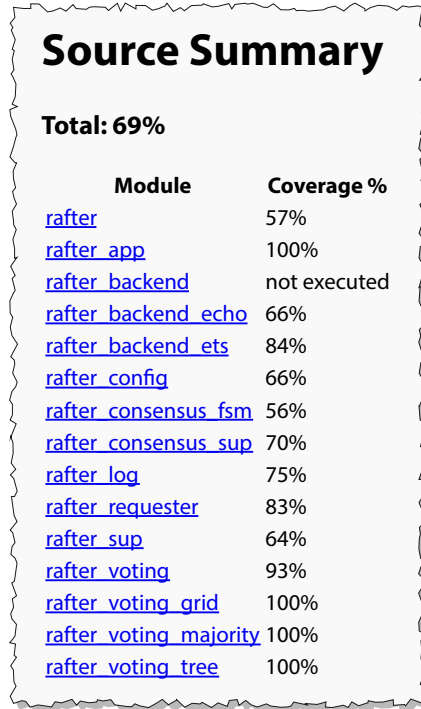
These goals were tackled first, since all other components depend on the voting structure and state type specification and the core voting algorithm being implemented successfully (see Figure 2.6). For the relevant background, see Section 2.3; the implementation is described in Section 3.1.

2. *Implementation and testing of voting structure generators for at least Majority Voting and the Grid Protocol.*

Voting structure generators for Majority Voting and the Grid Protocol were implemented successfully. Additionally, I wrote a voting structure generator for the Tree Quorum Protocol. They were tested for internal consistency using randomised unit tests (see Section 4.2), and for correctness by visual inspection using the voting structure visualiser discussed in Section 3.2.

The theory behind voting schemes is introduced in Section 2.3, the implementation of the three voting structure generators is described in Section 3.3.

3. *Incorporating the above algorithms and data structures into Rafter so they can be used to find quorums.*



Source Summary	
Total: 69%	
Module	Coverage %
rafter	57%
rafter_app	100%
rafter_backend	not executed
rafter_backend_echo	66%
rafter_backend_ets	84%
rafter_config	66%
rafter_consensus_fsm	56%
rafter_consensus_sup	70%
rafter_log	75%
rafter_requester	83%
rafter_sup	64%
rafter_voting	93%
rafter_voting_grid	100%
rafter_voting_majority	100%
rafter_voting_tree	100%

Table 4.1: Unit test coverage of the structured voting code. This is a screenshot of the test coverage report for the project generated by the EUnit library – see Section 4.2 for further details. Note in particular the high coverage of the last four modules (`rafter_voting*`), which were written as part of this project.

Integration with Rafter has been accomplished, and I have demonstrated that Rafter can run using my voting code to elect the leader. Section 2.2 provides the background on Raft, and Section 3.4 describes the integration process.

4. *Setting up the project so it can be run as a distributed key-value store in the Amazon Elastic Compute Cloud infrastructure.*

I took an Arch Linux image specially made to run on Amazon EC2 instances, built and installed all necessary tools required for benchmarking, and deployed it on a cluster.

5. *Running the benchmarks, and collecting and evaluating the results. The evaluation must include a comparison and discussion of the benchmarks collected for the different voting schemes.*

Structured voting schemes come at a performance cost due to the additional processing involved in building and traversing the voting structures and election states. But, as the more detailed analysis in Section 4.3.4 shows, structured voting schemes can under some circumstances outperform majority voting. A more optimised version of the structured voting code may increase the range of cluster sizes and voting schemes for which this is true.

4.2 Testing

In the early stages of the development of each component, testing was primarily manual. Once the code had reached a stage where testing by hand did not uncover any more bugs, I wrote a QuickCheck test for that component.

The QuickCheck tests, in their simplest form, consist of *generators* (not to be confused with voting structure generators) and a set of functions specifying the *properties* of the generated objects (examples for both are given in Appendix A.5.) QuickCheck then randomly instantiates test cases using the generators, and checks if they behave as expected.

Voting structure generators, in particular, were tested using two complementary methods: visually, I checked them for correctness using the voting structure visualiser (see Section 3.2); a QuickCheck property tested for internal integrity.

Table 4.1 shows a screenshot of the test coverage report for the project generated by the EUnit library.¹ I modified four out of the five original test modules and added another four to cover my structured voting code. Of the modules shown, I edited `rafter_config`, `rafter_consensus_fsm` and `rafter_consensus_sup` and wrote the bottom four modules (all starting with `rafter_voting`) from scratch.

4.3 Benchmarking

In order to compare the performance of my voting algorithms with those used in the original Rafter code, and to see how different voting schemes scale with the number of nodes, I designed, scripted and ran a series of benchmarks in the Amazon Elastic Compute Cloud.

4.3.1 Helper scripts

The benchmarks are automated using Fabric and the `Awsfabrictasks` plugin (see Section 2.5.2 for a brief description of the two software packages). A task called *benchmark* executes a sweep of the parameter space (= voting schemes \times cluster sizes \times failure modes), starting nodes as appropriate and collecting the logged results – see Figure 4.1 for a visualisation of the complete benchmarking procedure. It makes use of a few helper functions:

deploy Executes a series of Git commands on the remote instance to bring the project repository up to date and to a specific branch. It then runs `make` to build the project.

¹The QuickCheck tests are embedded into an EUnit test suite started by the Erlang build tool, Rebar (<https://github.com/rebar/rebar>), which is invoked via Make. This is the setup used by the original Rafter project, and I kept it so I could still run the original test suite to avoid introducing regression bugs.

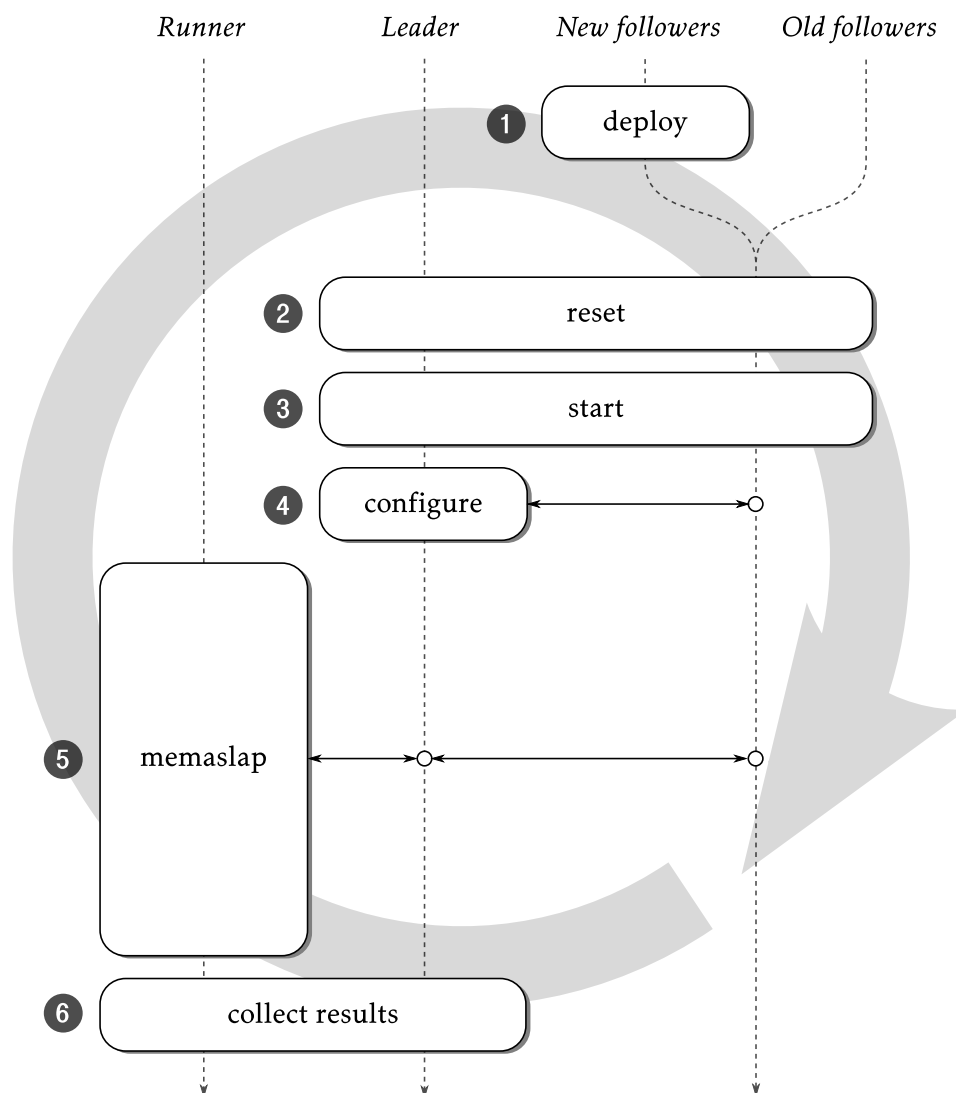


Figure 4.1: High-level overview of the *benchmark* task: (1) The latest version of the source code is pulled from the central repository and compiled on the new followers. (2) On all followers and the leader, the Erlang process is killed and the file system cache is purged. (3) Rafter is started on each node in the cluster. (4) The leader is configured. This step triggers a leader election. (5) `memaslap` is executed on the runner, targeting the leader. (6) The script downloads log files from the runner and the leader.

start_erlang_node Takes a name as its parameter and starts an Erlang node with that name on the instance.

stop_erlang_node Kills the Erlang interpreter process on this instance.

configure Creates a configuration script and executes it. The script evaluates an Erlang program that is created on the fly. It uses Erlang's remote procedure call module, to issue a call to the leader to update its configuration, which consists of the list of followers and the voting scheme to use. The script then sends a message to the leader to relay a 'restart' message to its followers so they all recover from a possible failed state. Finally, a new failure mode is sent to the leader.

memaslap Runs `memaslap`, the Memcached load tester, locally, targeting the `memaslap` frontend running on the leader node. The output of the program is diverted into a log file.

collect_results Uses `rsync` to download the Memcached and time to consensus log files to the machine running the benchmarks.

The *benchmark* task (see Figure 4.1 on Page 33) uses these helper functions as follows: For each combination of cluster size, protocol and failure mode that is to be tested, the procedure starts by deploying the latest version of the project source code to all instances as required and compiling it (using the `deploy` task). It then starts up an Erlang node with a fresh name on each of them. These nodes start in follower state and with an empty initial configuration.

Next, the leader is reconfigured using the `configure` task. Then `memaslap` is run, and its results as well as the time to consensus measurements are downloaded by the `collect_results` task.

4.3.2 Amazon EC2 set-up

The Amazon Elastic Compute Cloud was used as a testbed for the benchmarks. It provides a realistic environment for testing – after all, consensus systems like Rafter would usually run in a data centre like Amazon's. The random packet delivery timing, network congestion and packet loss experienced in the Amazon cloud make for a good stress test of Rafter's ability to cope with such issues.

New instances in the Amazon Elastic Compute Cloud are created from virtual appliances called 'Amazon Machine Images' (AMIs). Amazon's own default AMI does not have the latest version of the Erlang platform in its package sources, so I based my custom AMI on an Arch Linux image instead.² I created an instance of the Arch AMI, installed Erlang and a few standard tools and built `libMemcached` from source, as all distributions I checked did not include the `memaslap` binary in their `libMemcached` package.

²Project 'Arch Linux on EC2': <https://www.uplinklabs.net/projects/arch-linux-on-ec2/>

Users of the Amazon Elastic Compute Cloud currently have the choice between many different instance types. Preliminary experiments using a cluster of (free) ‘micro’ instances were useful for learning how to use the Amazon EC2 administration tools, but their performance turned out to be too low, and SSH connections were flaky. I therefore switched to ‘m1.xlarge’ general-purpose instances for running the benchmarks (see Table 4.2 for a comparison.)

The m1.xlarge instances provided enough compute power and much more network bandwidth than the micro instances. Connections to them were reliable once established.

<i>Name</i>	<i>ECUs</i>	<i>vCPUs</i>	<i>Memory (GiB)</i>	<i>Storage (GB)</i>	<i>Network performance</i>
t1.micro	up to 2	1	0.613	EBS only	very low
m1.xlarge	8	4	15	4×420	high

Table 4.2: Comparison of the two Amazon EC2 instance types used. ECU stands for ‘Elastic Compute Unit’, roughly equivalent to one 1.0 GHz 2007 Opteron; a vCPU is a virtual CPU; EBS means ‘Amazon Elastic Block Storage’.

4.3.3 Choice of parameters

Due to time constraints, benchmarking every combination of parameters was unfeasible. I benchmarked all voting structures. The Grid Protocol was tested with the ‘favouring rows’ option (explained in Section 3.3.2); benchmarks were also run on binary and tertiary trees created by the Tree Quorum Protocol.

As for cluster sizes, I tried to find configurations for which at least one protocol was optimal.³ My Amazon EC2 account allowed me to run experiments with up to 19 nodes (plus one separate ‘runner’ instance). This seems like a reasonably large range of cluster sizes – the authors of the Zookeeper paper, for instance, give benchmarking results for up to 13 servers.⁴

Majority Protocol Any odd number of nodes is optimal.

Grid Protocol Clusters of size 4, 9 and 16 are square; 6 and 12 form full almost-square rectangles. However, grids of size 4 and 6 require more nodes than the Majority Protocol for a quorum, which makes them less interesting for this analysis.

Tree Quorum Protocol Trees of degree d and height h with $n = \sum_{i=0}^h d^i$ nodes are full. Considering cluster sizes less than 20, binary trees of size 3, 7 and 15 are full, and so are ternary trees with 4 or 13 nodes.

³A configuration is *optimal* when it allows for the largest cluster size such that its minimal quorum size equals some given number. With the Majority Protocol, for instance, clusters of size 4 and 5 both have a minimal quorum size of 3, so for the Majority Protocol with quorum size 3, a cluster size of 5 is optimal.

⁴Hunt et al. 2010.

Taking the union of the ‘interesting’ optimal cases, we get 3, 4, 5, 7, 9, 11, 12, 13, 15, and 17 as the cluster sizes to benchmark.

The `memaslap` command line arguments and the corresponding configuration file can be found in Appendix A.6.

4.3.4 Results and analysis

This section discusses the results gathered from running the benchmarks with failure simulation turned off.⁵ The purpose of the benchmarks was twofold: to find the overhead caused by adding additional complexity in the form of the structured voting layer, and to compare the performance of different structured voting schemes.

While the benchmarks ran, two types of measurements were taken: time to consensus (TTC) and throughput.

Time to consensus was logged by a Rafter module I wrote for this purpose (see Section 3.7 for more details on the logger). It is the time a request spent inside the system in total, from when it was received by the leader to when the leader sends the response back to the client.

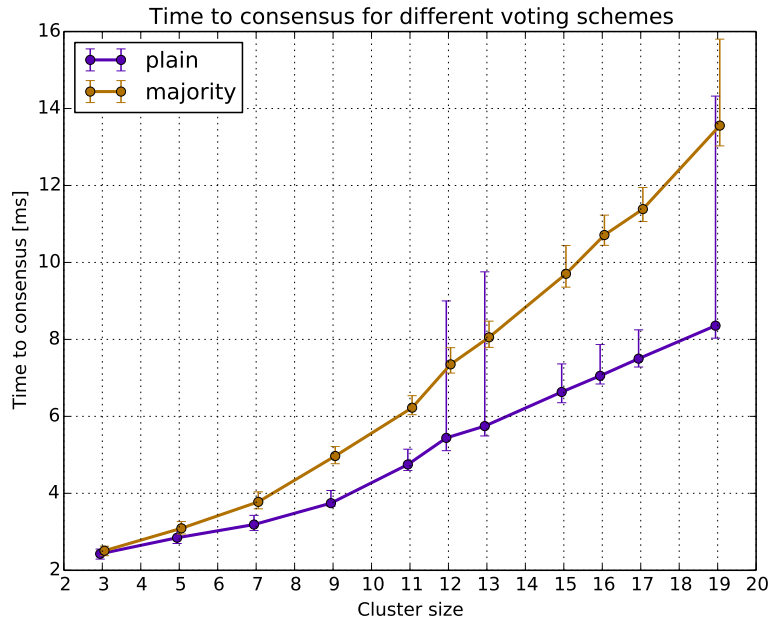
Total throughput was captured by `memaslap`, the Memcached benchmarking tool. Throughput is measured in operations per second.

Comparison with the original Rafter implementation

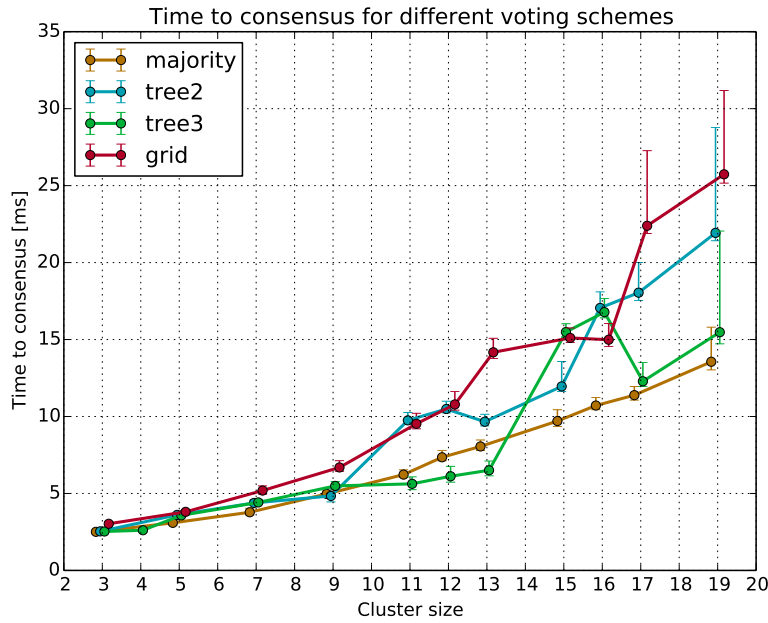
Figure 4.2a shows time to consensus (TTC, y-axis) as a function of the cluster size (x-axis). The two curves have the same shape: this is to be expected, since the quorum finding mechanism works in precisely the same way. The time to consensus for the original Rafter implementation without structured voting schemes (the ‘plain’ curve in the graph) is consistently smaller, and the difference between the two curves increases with the number of nodes in the cluster. In other words, for clusters with less than 20 nodes at least, computing quorums under the Majority Protocol takes roughly 300 μ s longer per node than with the original Rafter implementation (as can be seen on Figure 4.2a).

This was to be expected: the voting scheme which is implemented by original Rafter is the same as the Majority Protocol, but the implementation is very different. In the original Rafter code, checking if a quorum has been found amounts to checking whether or not the number of YES votes received is greater than half the number of servers. With the Majority Protocol, on the other hand, a new voting state is created whenever a new vote takes place, and a tree-walking algorithm updates it whenever a new vote comes in. It should come as no surprise that this takes longer.

⁵Figure A.1 shows the cumulative density functions for the Majority Protocol and the Grid Protocol under a repeated failure simulation with different uptimes. The measurements shown in this graph were taken locally and are therefore not comparable with the distributed benchmarks discussed in this chapter.



(a) Time to consensus results for original Rafter and the Majority Protocol.



(b) TTC measurements for different structured voting schemes

Figure 4.2: Time to consensus measurements for the original Rafter implementation ('plain'), the Majority Protocol ('majority'), the Tree Quorum protocol ('tree2' and 'tree3' with branching factors $d = 2$ and $d = 3$, respectively), and the Grid Protocol ('grid'). Each point in the graph represents 5000 write operations, all executed in a single run. The lines connect the median measurements and the error bars indicate the 25th and the 75th percentile.

A more interesting aspect is the seemingly linear relationship between the size of the cluster and the difference in time to consensus between the Majority Protocol and original Rafter. This can be explained as follows: whenever a new vote comes in, the root node ‘accumulates’ all votes, that is, it updates the number of Yes and No votes it has seen by looking at its children. But the number of children of the root node in the Majority Protocol voting structure is precisely the size of the cluster, which means that for every incoming node, updating the voting state takes $O(n)$ time.

Comparison between different structured voting schemes

The time to consensus versus cluster size graph for all four structured voting schemes implemented in this project in Figure 4.2b has some interesting features.

Consider the graph of the Grid Protocol time to consensus (the ‘grid’ dataset in Figure 4.2b). The TTC measurements increase by about 3.5 ms between 12 and 13 nodes but then stay within about 1 ms until the cluster size reaches 16; then there is a jump by 7.5 ms as the cluster size hits 17. Why the plateau between 13 and 16? By Equation (3.1), cluster sizes 13, 14, 15, and 16 all generate quadratic 4×4 grids. Clusters with 12 nodes fit into a 4×3 grid, and those with 17 nodes require a 5×4 grid. Thus, it seems that the TTC times change mainly when the grid shape changes. See Figure A.2 for a combined plot of TTC and minimum quorum cardinality for the Grid Protocol as a function of cluster size.

This behaviour can be explained by looking at the number of nodes required to vote YES, which depends on the shape of the grid: for a $r \times c$ grid it is $r + c - 1$.⁶ For cluster sizes 13, 14, 15 and 16, the grid has the same 4×4 shape, so in each case, seven votes are required. See Figure 2.3 on Page 8 for a plot of minimum quorum cardinalities for different voting schemes and cluster sizes.

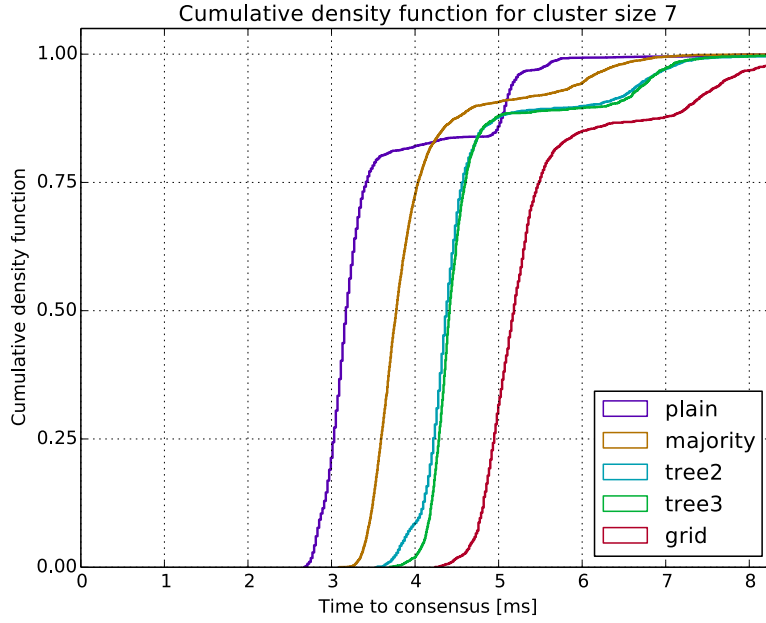
The Tree Quorum Protocol graphs show a similar behaviour. A cluster of 15 nodes exactly fills a binary tree of height 3. Add one more node, and a non-full binary tree of height 4 is required to hold all nodes. Analogously, 13 nodes fit exactly into a ternary tree of height 2; more nodes require at least height 3. This helps explain the rapid increase in time to consensus between 15 and 16 nodes in the binary (‘tree2’) case and the similar effect between 13 and 15 nodes in the ternary (‘tree3’) case.

The dent in the cumulative density function

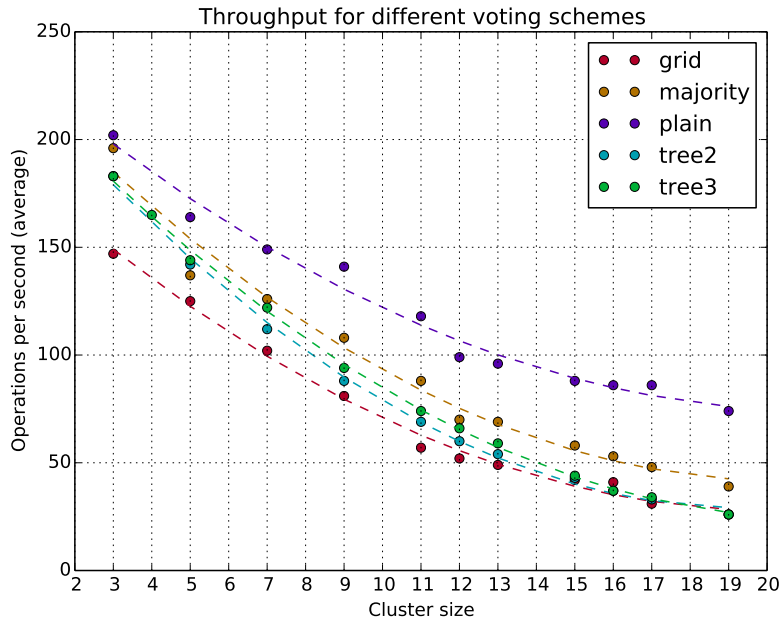
Cumulative density functions allow us to easily compare the shapes of the latency distributions of different voting schemes. All cumulative density functions (Figure 4.3a shows the one for a cluster of size 7) of the time to consensus measurements I took exhibit one or more ‘dents’ between the 75th and the 100th percentile. These show up as large upper error bars in Figure 4.2 as well. Where do they come from?

Preliminary experiments showed that the time to consensus increases by orders of magnitude as message queues build up when the load on the system is high. The system load is primarily a function of the frequency at which the client – memaslap

⁶A Grid Protocol quorum needs votes from each column (c votes) and one entire column (r votes). Those two sets intersect in one process, and so the number of processes in a Grid Protocol quorum is $r + c - 1$.



(a) Cumulative density function of time to consensus measurements for a cluster with seven nodes. Each lines in the graph represents 5000 write operations.



(b) Throughput (set and get combined). Each point in the graph represents the average throughput over 5000 read and 5000 write operations. The dashed lines indicate the second order polynomial best fit.

Figure 4.3: TTC cumulative density and throughput plots for the original Raft implementation ('plain'), the Majority Protocol ('majority'), the Tree Quorum protocol ('tree2' and 'tree3' with branching factors $d = 2$ and $d = 3$, respectively), and the Grid Protocol ('grid'). All measurements were taken in a single run.

for the purposes of the benchmarks – sends requests. By carefully reducing this throughput by varying the `-tps=X` command line parameter, I made sure to operate the system in a regime just below overloading for most of the requests (see Listing A.4 for a specification of the command line parameters used in the benchmarks). This is the part of the curves below the dents, roughly the region below the 75th percentile.

Throughput

Throughput is highly correlated with time to consensus: The lower the TTC, the higher the throughput. But it depends on a number of other factors as well, most importantly the degree of parallelism in the system: if the number of operations that can be executed simultaneously is doubled, then (*ceteris paribus*) the throughput doubles while TTC stays constant.

I used the output of `memaslap` (which Figure 4.3b is compiled from) to confirm that my time to consensus values were reasonable. It is important to note that while Figures 4.2 and 4.3a show time to consensus for write operations only, Figure 4.3b indicates the throughput averaged over both read and write operations. The reason behind this is that in Rafter, the leader has the canonical log, and can therefore serve any read request directly. There is no quorum-building involved in a read operation, and so the response time is independent of the voting structure or the cluster size. Unfortunately, `memaslap` only outputs the overall average throughput, so I had to make do with this one value.

4.4 Summary

This chapter started by showing that all the requirements of the project proposal have been fulfilled. I described my testing set-up using QuickCheck and demonstrated good software engineering practice through very high test coverage.

The remainder of the chapter was devoted to a discussion of the benchmarking measurements. Two main results have been found. Firstly, the implementation of structured voting schemes is computationally more expensive than hard-coding majority voting and therefore slower, as demonstrated by Figure 4.2a. This was to be expected.

Secondly, I have shown that structured voting schemes are nevertheless worth considering in a practical setting. Figure 4.4 shows that given the right cluster size, structured voting schemes can be faster than unstructured voting schemes in finding quorums, as evidenced by the circled region of the graph. Here, the (ternary) Tree Quorum Protocol, a structured voting scheme, beats the Majority Protocol, the default unstructured voting scheme, in the time to consensus measurements.

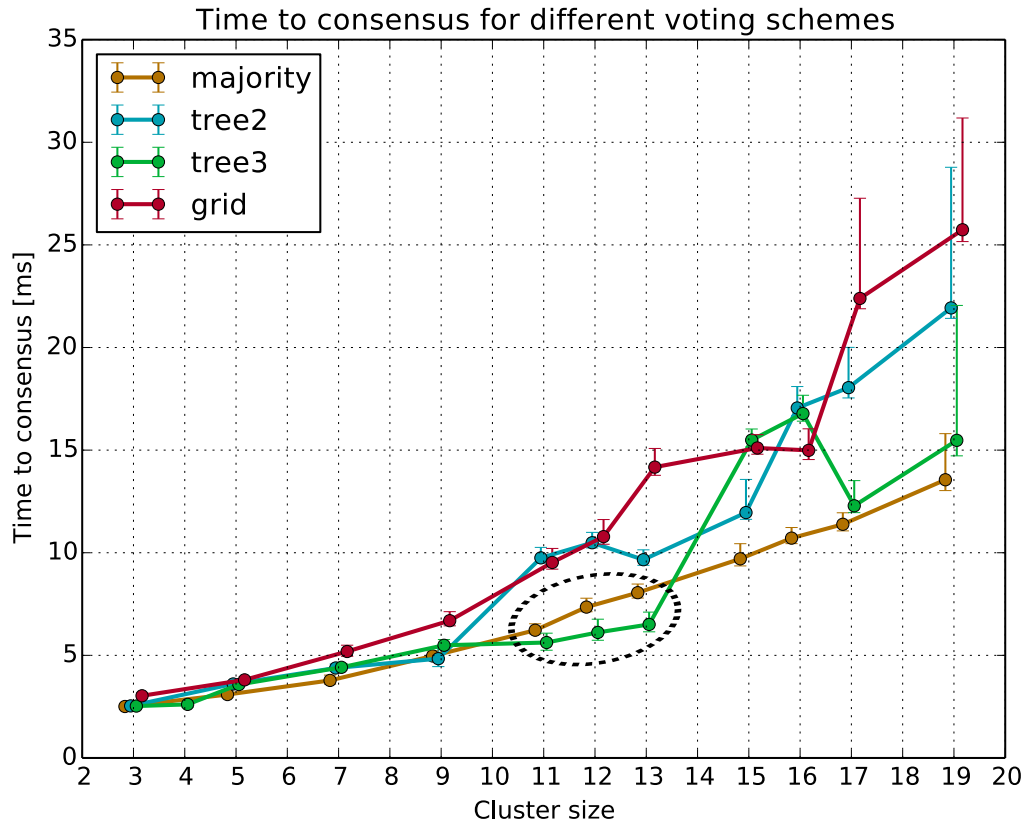


Figure 4.4: Time to consensus measurements for the original Raft implementation ('plain'), the Majority Protocol ('majority'), the Tree Quorum protocol ('tree2' and 'tree3' with branching factors $d = 2$ and $d = 3$, respectively), and the Grid Protocol ('grid'). This graph shows the main empirical evidence that structured voting schemes can have practical advantages. Within the circled area, the (ternary) Tree Quorum Protocol found quorums faster than the Majority Protocol. This graph is an annotated version of Figure 4.2b. Each point in the graph represents 5000 write operations, all executed in a single run. The lines connect the median measurements and the error bars indicate the 25th and the 75th percentile.

Chapter 5

Conclusions

This dissertation has described how I designed, implemented, tested and benchmarked a structured voting extension to Rafter, an existing implementation of the new Raft consensus algorithm. All goals outlined in the original proposal have been met, and a few optional extensions have been added.

Results

The result of the project is twofold: firstly, the implementation of the extension itself. I have successfully built upon an existing piece of software, Rafter, that is still in an early phase of development, largely undocumented and written in a language I was entirely unfamiliar with when I started this project. The extension works and is stable. Adding more structured voting schemes is easy.

Secondly, I have shown that structured voting schemes have practical value. The main evidence for this can be found in Figure 4.4, which shows that under certain conditions, structured voting schemes can be more efficient than unstructured ones. Optimising the voting structure data types and algorithms would decrease the penalty for using structured voting schemes, and therefore result in structured voting schemes being more efficient than unstructured ones under a larger range of configurations, especially as the size of the cluster grows big.

Takeaway

Doing this project and writing the dissertation has been very educational. I have learnt to administer cloud servers, analysed big datasets with a variety of different tools, and fought battles with \LaTeX to get just the layout I wanted. Last but not least, I have learnt to appreciate just how much time and care it takes to benchmark a distributed system.

Conclusion

I have tried to structure this dissertation in way that makes sense to the reader, and I hope that each part of the project was given appropriate space for explanation and discussion. Structured voting schemes are not a new idea, but maybe after decades of being confined to academic papers, they will finally be picked up by implementers of distributed systems – the potential for efficiency gains are real, as this dissertation has shown.

Bibliography

- Agrawal, Divyakant and Amr El Abbadi (1990). 'The Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data.' In: *Proceedings of the 16th Very Large Data Bases Conference*. Vol. 90, pp. 243–254.
- Agrawal, Divyakant and Amr El Abbadi (1992). 'The Generalized Tree Quorum Protocol: An Efficient Approach for Managing Replicated Data'. In: *ACM Transactions on Database Systems* 17.4, pp. 689–717.
- Apache Software Foundation (2014). *Apache HBase: a distributed, scalable, big data store*. URL: <http://hbase.apache.org> (visited on 17/03/2014).
- Basho Technologies (2014). *Riak: an open source, distributed database*. URL: <http://basho.com/riak> (visited on 17/03/2014).
- Brewer, Eric (2000). 'Towards Robust Distributed Systems'. In: *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*. Invited talk.
- Brewer, Eric (2012). 'CAP Twelve Years Later: How the "Rules" Have Changed'. In: *Computer* 45.2, pp. 23–29.
- Cheung, Shun Yan, Mostafa H Ammar and Mustaque Ahamad (1992). 'The Grid Protocol: A High Performance Scheme for Maintaining Replicated Data'. In: *IEEE Transactions on Knowledge and Data Engineering* 4.6, pp. 582–592.
- Dean, Jeff (2009). *Designs, Lessons and Advice from Building Large Distributed Systems*. Keynote at the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware. URL: <http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf>.
- DeCandia, Giuseppe et al. (2007). 'Dynamo: Amazon's Highly Available Key-value Store'. In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 6. ACM, pp. 205–220.
- Fox, Armando and Eric Brewer (1999). 'Harvest, yield, and scalable tolerant systems'. In: *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*. IEEE, pp. 174–178.
- Gilbert, Seth and Nancy Lynch (2002). 'Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services'. In: *ACM SIGACT News* 33.2, pp. 51–59.
- Hale, Coda (2010). *You Can't Sacrifice Partition Tolerance*. URL: <http://codahale.com/you-cant-sacrifice-partition-tolerance/> (visited on 17/03/2014).

- Hébert, Fred (2013). *Learn You Some Erlang for Great Good!: A Beginner's Guide*. No Starch Press.
- Hunt, Patrick et al. (2010). 'ZooKeeper: Wait-free coordination for Internet-scale systems'. In: *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*. Vol. 8, pp. 11–11.
- Kumar, Akhil, Michael Rabinovich and Rakesh K Sinha (1993). 'A Performance Study of General Grid Structures for Replicated Data'. In: *Proceedings of the 13th International Conference on Distributed Computing Systems*. IEEE, pp. 178–185.
- Lakshman, Avinash and Prashant Malik (2010). 'Cassandra: A Decentralized Structured Storage System'. In: *ACM SIGOPS Operating Systems Review* 44.2, pp. 35–40. URL: <http://cassandra.apache.org>.
- Lamport, Leslie (1998). 'The part-time parliament'. In: *ACM Transactions on Computer Systems* 16.2, pp. 133–169.
- Ongaro, Diego and John Ousterhout (2013). 'Safety Proof and Formal Specification for Raft'. Last updated on 28 March 2013. URL: <https://ramcloud.stanford.edu/~ongaro/raftproof.pdf> (visited on 17/03/2014).
- Ongaro, Diego and John Ousterhout (2014). 'In Search of an Understandable Consensus Algorithm'. Last updated on 2 February 2014. URL: <https://ramcloud.stanford.edu/raft.pdf> (visited on 17/03/2014).
- Storm, Christian (2011). 'Specification and analytical evaluation of heterogeneous dynamic quorum-based data replication schemes'. PhD thesis. University of Oldenbourg.
- Storm, Christian and Oliver Theel (2006). 'Highly Adaptable Dynamic Quorum Schemes for Managing Replicated Data'. In: *Proceedings of the 1st International Conference on Availability, Reliability and Security*. IEEE, pp. 245–253.
- Theel, Oliver (1993). 'Ein vereinheitlichendes Konzept zur Konstruktion hochverfügbarer Dienste'. PhD thesis. Technische Universität Darmstadt.
- Thomas, Robert H (1979). 'A majority consensus approach to concurrency control for multiple copy databases'. In: *ACM Transactions on Database Systems* 4.2, pp. 180–209.
- Underwood and Underwood (1906). *Log Rafts, 1902*. Oregon History Society Oregon History Project. URL: http://www.ohs.org/education/oregonhistory/historical_records/dspDocument.cfm?doc_ID=231CB437-D102-C4A1-86D6D971C095C8D4 (visited on 23/04/2014).

Appendix A

Additional resources

A.1 Raft safety guarantees

Figure 3 of the Raft paper lists the following safety properties which the algorithm guarantees to be true at all times:¹

Election Safety: at most one leader can be elected in a given term.²

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 server 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.

A.2 Raft correctness proof: intersection requirement

The Raft correctness proof uses a variable *Quorums*. It is introduced by the following comment: ‘The set of all quorums. This just calculates simple majorities, but the only important property is that *every quorum overlaps with every other*.’ This is what originally inspired me to look into replacing majority voting by structured voting schemes in Raft.

Later in the proof, the intersection property is used multiple times. One example is in Lemma 2, *There is at most one leader per term*: step three of its proof says, ‘Let *voter* be an arbitrary member of $e.votes \cup f.votes$. Such a member must exist since any two quorums overlap’ (here, e and f are two elections taking place in the same term.)

¹Ongaro and Ousterhout 2014.

²Raft divides time into *terms* of arbitrary length. Each begins with an election, in which one or more candidates attempt to become leader.

A.3 A note on functional programming

Most of the code written for this project is in Erlang. Being a functional programming language, Erlang supports a very dense programming style. Its core libraries (which are imported by default) include modules for working with lists, dictionaries (hashmaps), sets, string, timers, I/O and databases as well as random values, each of which is used in my code at some point, among others.

Just to illustrate this point, I picked a typical line of Erlang code (Listing A.1) and ‘translated’ it into relatively idiomatic C++03 (Listing A.2). Note that the Erlang code only makes use of the built-in module `dict`, so no imports are required. As a dynamically typed language, Erlang does not need type declarations either, although they *can* be added as annotations to functions for clarity. Thus, the code in Listing A.1 really does stand on its own, as long as it is wrapped in a function.

```
NewIndices = dict:map(fun(_, Paths) -> [[0|P] || P <- Paths] end, Indices)
```

Listing A.1: A typical line of Erlang code, taken from the Grid Protocol voting structure generator. Note how the use of higher-order functions like `dict:map`, anonymous functions and list comprehensions allows for very dense code.

```
#include <map>
#include <deque>

typedef int id;
typedef std::deque<std::deque<std::size_t> > paths_t;

// ...

for(std::map<id, paths_t>::iterator indices_it = indices.begin();
    indices_it != indices.end(); ++indices_it) {
    paths_t paths = indices_it->second;
    for(paths_t::iterator paths_it = paths.begin();
        paths_it != paths.end(); ++paths_it) {
        paths_it->push_front(0);
    }
}
```

Listing A.2: The Erlang code from Listing A.1, translated into relatively idiomatic C++.

A.4 Voting structure and voting state type specifications

```

-type path() :: list(non_neg_integer()).
-type vote() :: boolean() | pending.

% voting structure records
-record(vstruct_p, {
    votes = 1 :: pos_integer(),
    id :: peer()
}).
-record(vstruct_v, {
    votes = 1 :: pos_integer(),
    thresh :: pos_integer(),
    children :: list(#vstruct_v{} | #vstruct_p{})
}).
-record(vstruct, {
    tree :: #vstruct_v{},
    indices :: dict()
}).

% voting state records
-record(vstate_p, {
    votes = 1 :: pos_integer(),
    vote = pending :: vote()
}).
-record(vstate_v, {
    votes = 1 :: pos_integer(),
    yes_votes = 0 :: non_neg_integer(),
    no_votes = 0 :: non_neg_integer(),
    thresh :: pos_integer(),
    children :: list(#vstate_v{} | #vstate_p{})
}).
-record(vstate, {
    tree :: #vstate_v{},
    indices :: dict()
}).

```

Listing A.3: Definitions of the data types used in the structured voting algorithms.

A.5 QuickCheck testing examples

A.5.1 Generator example

In `test/rafter_gen.erl`, the function `vstruct()` generates a voting structure, proceeding as follows: It (a) creates a list of unique identifiers of random length uniformly distributed between three and 40, (b) picks a voting structure generator at random (Majority Protocol, Grid Protocol, or Tree Quorum Protocol), (c) generates additional parameters depending on the voting structure generator chosen, (d) calls the chosen voting structure generator with the shuffled list of identifiers and any additional parameters as arguments, and (e) returns the resulting voting structure.

This generator function is used wherever a test requires a voting structure. As the tests are run a large number of times, the probability increases that all relevant combinations of parameters are tested at some point.

A.5.2 Property example

The file `test/rafter_voting_majority_eqc.erl` defines a property of the Majority Protocol in the function `prop_majority_quorum()`. In essence, this property states that a vote succeeds if more than half of the nodes agree ($v \geq \lceil n/2 \rceil$, where v is the number of votes and n the total number of processes.)

A.6 Memaslap configuration

```

memaslap \
  --servers=${leader_address}:11211 \      # server address and port
  --binary \                               # use binary protocol
  --execute_number=5000 \                  # run for 5000 operations
  --verify=1.0 --exp_verify=1.0 \         # verify all responses
  --cfg_cmd=memaslap.conf \               # use memaslap.conf
  --tps=0.05k \                           # aim for 50 ops/sec
  --concurrency=1 \                       # only use one thread

```

Listing A.4: memaslap command line arguments used for the benchmarks.

```

key
64      64      1  # all keys are 64 bytes long

value
1024    1024    1  # all values are 1024 bytes long

cmd
0  0.5          # 50% set
1  0.5          # 50% get

```

Listing A.5: The memaslap configuration file used for the benchmarks.

A.7 Additional graphs

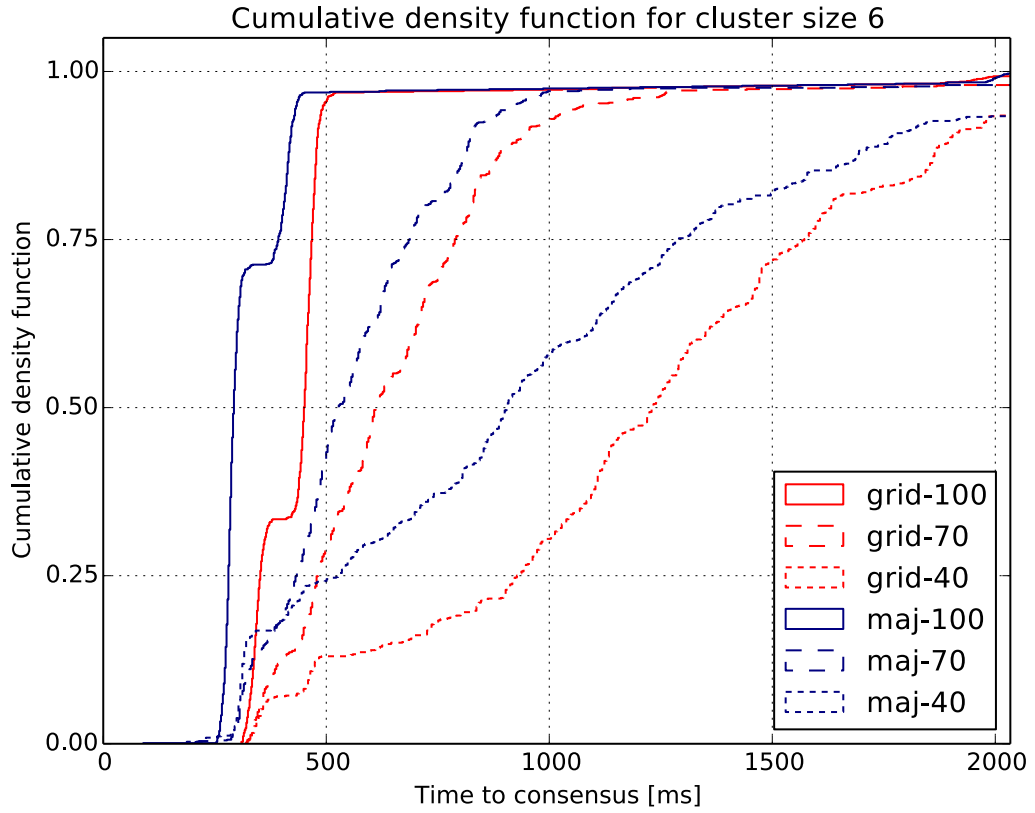


Figure A.1: Cumulative density functions for different voting schemes and uptimes. The red lines represent the Grid Protocol and the blue lines correspond to the Majority Protocol. Solid means Uptime = 100%, dashed Uptime = 70%, dotted Uptime = 40%. Each line represents 1000 write operations executed in a single run. In contrast to all other graphs in this dissertation, the measurements for this graph were taken on a single machine. See Section 3.6 for an explanation of the failure simulation used to produce these plots.

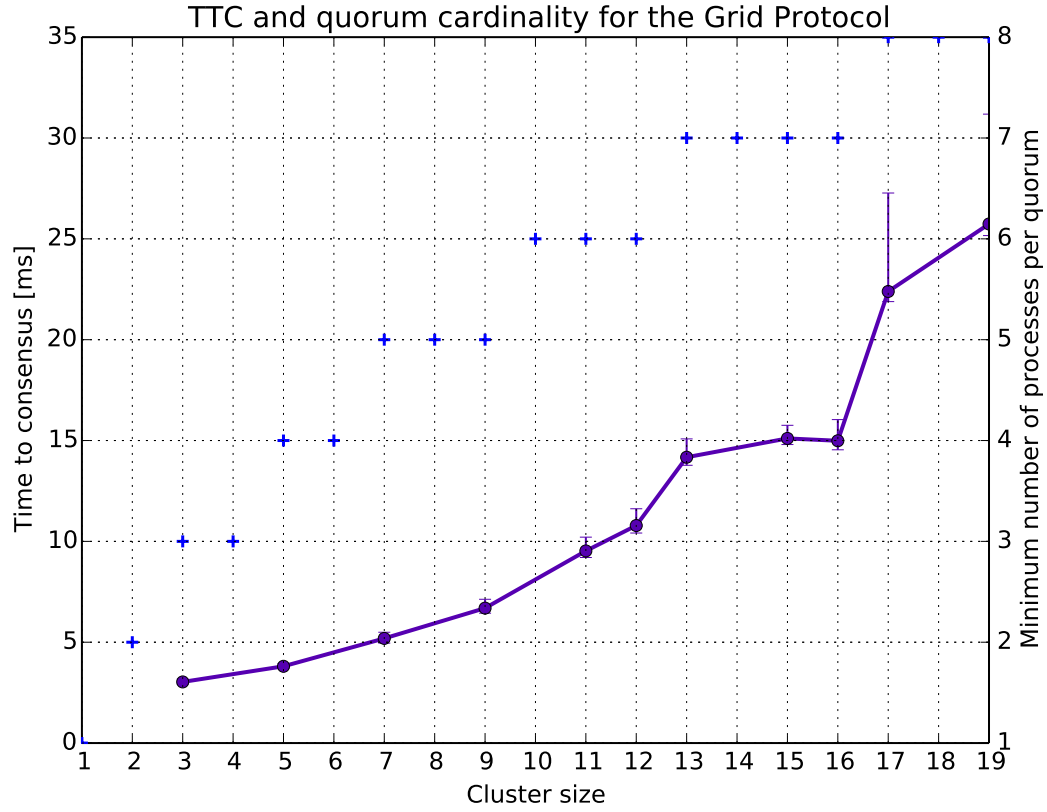


Figure A.2: Time to consensus and minimum quorum cardinality (plus signs) for the Grid Protocol with up to 20 processes. Each dot represents 5000 write operations executed in a single run. The line connects the median TTC measurements and the error bars indicate the 25th and the 75th percentile.

Appendix B

Grid Protocol probability analysis

Given $k \in \{0, 1, \dots, n\}$ YES votes, what is the probability $\mathbb{P}(Q_k^{c,r}) \stackrel{\text{def}}{=} \mathbb{P}(Q^{c,r} \mid V = k)$ that the set of votes forms a quorum?

We assume here that the grid we are considering has r rows and c columns (let $n = rc$ for convenience), therefore a complete column cover (CC-cover) consists of r nodes and a column cover (C-cover) consists of c nodes.

Let $\mathbb{P}(CC_k^{c,r}) \stackrel{\text{def}}{=} \mathbb{P}(CC^{c,r} \mid V = k)$ and $\mathbb{P}(C_k^{c,r}) \stackrel{\text{def}}{=} \mathbb{P}(C^{c,r} \mid V = k)$ be the probability of getting a CC-cover and a C-cover, respectively, given k YES votes. Then $\mathbb{P}(Q_k^{c,r}) = \mathbb{P}(CC_k^{c,r}, C_k^{c,r}) = \mathbb{P}(C_k^{c,r} \mid CC_k^{c,r}) \mathbb{P}(CC_k^{c,r})$.

B.1 C-cover probability

In this section and the next, I will assume that all the binomials that occur are defined, which is not necessarily the case. This issue will be taken care of when we put everything together at the end.

The number of ways in which we can get a C-cover equals the number of ways in which c columns can be covered by k nodes. We divide by the total number of ways in which k nodes can be distributed over n slots to get

$$\mathbb{P}(C_k^{c,r}) = \binom{k}{c} \div \binom{n}{k}$$

Conditioning on the existence of a CC-cover, we find that there are c ways in which a CC-cover can come about, and so $\mathbb{P}(C_k^{c,r} \mid CC_k^{c,r}) = c \mathbb{P}(C_{k-r}^{c-1,r})$.

B.2 CC-cover probability

Consider how a CC-cover could be built. The first node could go anywhere in the grid. The second node would have to go into the same column, an event with probability $\frac{r-1}{n-1}$. By similar reasoning, the third node would go into the same column with probability $\frac{r-2}{n-2}$, and so forth. There are $\binom{k}{r}$ different ways in which the k nodes at our disposal could cover those r slots.

This gives

$$\begin{aligned}\mathbb{P}(CC_k^{c,r}) &= \frac{rc}{n} \cdot \frac{r-1}{n-1} \cdot \frac{r-2}{n-2} \cdot \dots \cdot \frac{r-(r-1)}{n-(r-1)} \cdot \binom{k}{r} \\ &= c \cdot \frac{r!}{n! \div (n-r)!} \cdot \binom{k}{r} \\ &= c \cdot \binom{k}{r} \div \binom{n}{r}\end{aligned}$$

B.3 Quorum probability

Putting everything so far together, we have

$$\begin{aligned}\mathbb{P}(Q_k^{c,r}) &= \mathbb{P}(C_k^{c,r} \mid CC_k^{c,r}) \mathbb{P}(CC_k^{c,r}) \\ &= c \mathbb{P}(C_{k-r}^{c-1,r}) \mathbb{P}(CC_k^{c,r}) \\ &= c \left\{ \binom{k-r}{c-1} \div \binom{n-r}{k-r} \right\} \left\{ c \binom{k}{r} \div \binom{n}{r} \right\} \\ &= c^2 \frac{\binom{k-r}{c-1} \binom{k}{r}}{\binom{n-r}{k-r} \binom{n}{r}}\end{aligned}$$

By the definition of the binomial, this is only defined if $n \geq k$ (which is true by assumption), $k \geq r + c - 1$ and $n \geq r + c - 1$ (which is implied by transitivity). We know that $r + c - 1$ nodes are needed in order to get a C-cover as well as a CC-cover, so if $k < r + c - 1$ there can be no quorum.

The second special case is when we are sure to get a quorum. Starting from a full grid, we can take $r - 1$ nodes away and still get a C-cover, or we can take $c - 1$ nodes away and still get a CC-cover. Thus if $n - k < \min(r, c)$ we will always find a quorum.

This gives the final expression for the probability of finding a quorum given k out of n possible votes,

$$\mathbb{P}(Q_k^{c,r}) = \begin{cases} 0 & \text{if } k < r + c - 1 \\ 1 & \text{if } k > n - \min(r, c) \\ c \cdot \binom{n - (r + c - 1)}{k - (r + c - 1)} \div \binom{n}{k} & \text{otherwise} \end{cases}$$

Figure B.1 shows a plot of this probability $\mathbb{P}(Q_k^{c,r})$ as a function of k with $r = 4$ and $c = 5$ fixed.

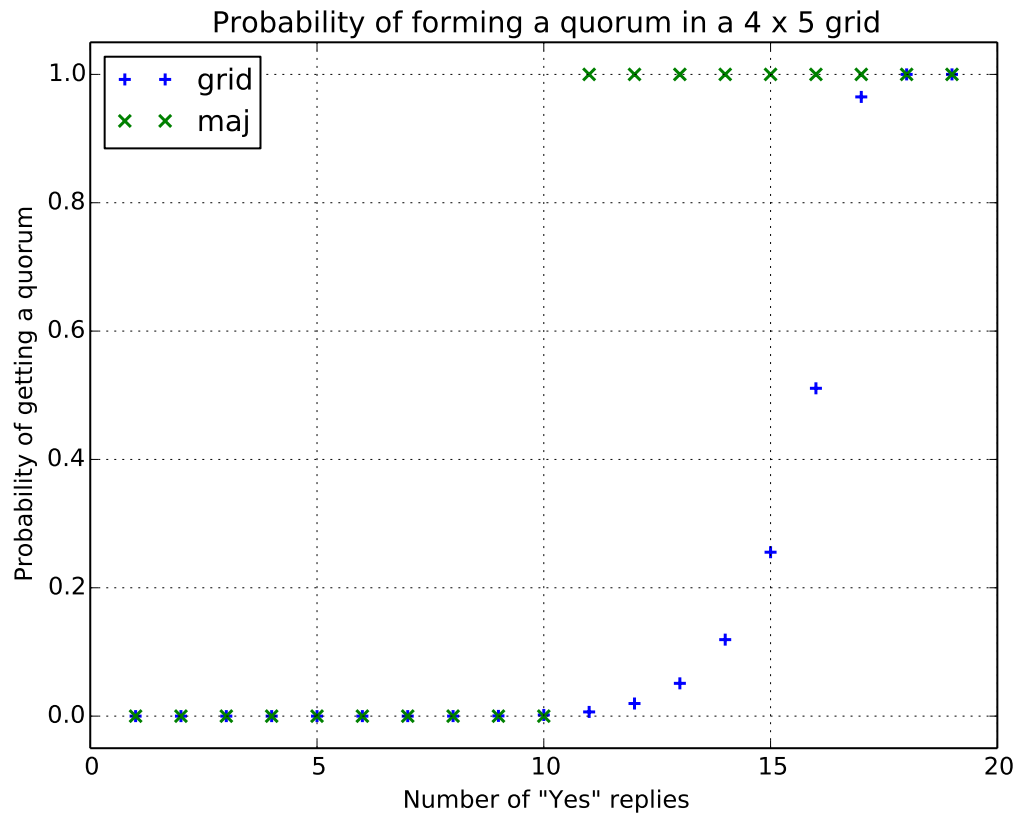


Figure B.1: Probability of forming a quorum as a function of the number of YES votes in a cluster of 20 processes for the Majority and the Grid Protocol.

Appendix C

Original project proposal

Computer Science Part II Project Proposal

Extending Raft with Structured Voting

October 24, 2013

Project Supervisors *Malte Schwarzkopf and Ionel Gog*

Director of Studies *Dr Jonathan Hayman*

Overseers *Dr Markus Kuhn and Dr Neal Lathia*

Introduction and Description of Work

Finding quorums is a key functionality of any strongly consistent distributed system. Raft¹ is a new consensus algorithm to be used instead of Paxos,² aiming to be easier to understand and implement correctly. It models the distributed system as a replicated state machine using a replicated log. Raft features a strong leader, an explicit membership change mechanism and a (partial) correctness proof.³

By default, Raft uses Majority Consensus Voting to find quorums. Both the initial paper and the separate correctness proof suggest that it should be possible to replace Majority Consensus Voting by alternative schemes.

Structured voting schemes⁴ impose a logical structure on the set of processes and use structural properties to specify quorum systems. Each voting scheme defines a way to construct *write quorums*, any two of which overlap. The Grid Protocol, for instance, logically arranges all nodes in a rectangular grid. A write quorum in the Grid Protocol consists of the union of one entire row and one entire column of nodes. Clearly, by construction, any two such write quorums intersect. Since this intersection property is the only requirement for the correctness proof, replacing Majority Consensus Voting by a write quorum computed from a structured voting scheme does not affect the correctness of the Raft algorithm.

Structured voting protocols have the drawback that they may fail to generate a quorum, even though more than half of the nodes are available (i. e. neither failed nor busy). They may, on the other hand, find a quorum even when fewer than half of the nodes are available. Majority voting will always succeed in finding a quorum in the former case, and will necessarily fail to do so in the latter case. Thus structured voting schemes involve a trade-off, being less reliable than Majority Consensus Voting if most nodes are available, but more reliable if the majority of nodes is not available.

There are at least two scenarios under which this last point becomes relevant: firstly, under high load a majority of nodes may be busy processing requests. Using a structured voting scheme, it may still be possible to assemble a quorum, avoiding an expensive retry of the operation that would be necessary when a Majority Consensus Voting scheme was employed. Secondly, it may be the case that node failures are not independent, as is often assumed. In this case, the structured voting scheme could be specifically designed for resilience against certain failure modes like the power outage of entire racks or connection problems between data centres.

The aim of this project is to add structured voting protocols to an existing implementation of the Raft consensus algorithm, and to compare the performance of different structured voting protocols with the performance of Raft's default Majority Consensus Voting.

¹Ongaro and Ousterhout 2014.

²Lamport 1998.

³Ongaro and Ousterhout 2013.

⁴Storm 2011.

Starting Point

In this project, I will extend an existing implementation of the Raft consensus algorithm that is written in Erlang.

Raft consensus algorithm. I have read the Rafter paper,⁵ listened to a recorded lecture on Raft by one of its creators, and studied the separate correctness proof.

Implementation of Raft. *Rafter*⁶ is an open source implementation of Raft written in Erlang by Andrew J. Stone at Basho, Inc. It still being developed, but is mostly feature-complete. I shall use its code as the basis for this project.

Concurrent and Distributed Systems. The Part IB lecture courses Concurrent Systems and Distributed Systems have given me an appreciation for the challenges that such systems pose. During my work placement this summer, I implemented a minimalistic distributed key-value store that used structured voting protocols to guarantee consistency as a proof of concept in Node.js/JavaScript.

Erlang programming language. Most of the code I shall be writing for this project will be in Erlang, a programming language I have no prior experience with. However, I am familiar with functional programming from both the Part IA Standard ML exercises and from small personal projects in Haskell and OCaml.

Substance and Structure of the Project

The objective of this project is to add structured voting protocols to Rafter, and to benchmark different structured voting protocols both against each other and Raft's default Majority Consensus Voting algorithm.

All the unstructured and structured voting schemes that will be implemented can be specified by voting structures,⁷ which describe the set of quorums. For each voting scheme, a generator will have to be written that maps the set of nodes and potentially some further options to a voting structure. I will also require an interpreter algorithm that decides whether a certain set of nodes constitutes a quorum within a given voting structure.

In order for Rafter to use voting schemes for quorum finding, the data structure that specifies a node's configuration will have to include the current voting structure.

To obtain meaningful benchmarks and demonstrate the usefulness of this project, an application on top of Rafter will be required. A simple in-memory key-value store state machine already exists, but so far the only way to communicate with the cluster is via Erlang's own message passing protocol.

⁵Ongaro and Ousterhout 2014.

⁶Andrew J. Stone. Rafter: An Erlang library application which implements the Raft consensus protocol (Git repository), accessed October 17, 2013. <https://github.com/andrewjstone/rafter>.

⁷Storm 2011.

This means that there are two ways of benchmarking the system once structured voting schemes have been implemented: either by writing and running an Erlang benchmarking tool that communicates with the Rafter cluster using Erlang's message passing primitives, or, if time allows, by first adding a compatibility layer on top of Rafter that implements a popular key-value store protocol, and then running an existing benchmarking tool that supports this protocol.⁸ In both cases, the latency distribution for both read and write operations (in units of time) and throughput (in operations per time unit) will be measured.

Success Criteria

The following should be completed:

1. Design of the data structure to represent voting structures, and implementation and testing of the algorithm to interpret it. Implementation and testing of voting structure generators for at least Majority Voting and the Grid Protocol.
2. Incorporating the above algorithms and data structures into Rafter so they can be used to find quorums.
3. Setting up the project so it can be run as a distributed key-value store in the Amazon Elastic Compute Cloud (EC2) infrastructure.
4. Running the benchmarks, and collecting and evaluating the results (the benchmark metrics are further specified above). The evaluation must include a comparison and discussion of the benchmarks collected for the different voting schemes.

Optional Extensions

Probability analysis. Given both the total number of nodes and the number of nodes that are not available, compute the probability of finding a quorum for different voting schemes.

More voting schemes. Investigate other voting schemes, for example the Tree Quorum Protocol.

TCP or HTTP interface. Extend the project with a CRUD (Create, Read, Update, Delete) API that can be accessed via TCP or HTTP. This API should model the protocol of some existing key-value store as closely as possible so that the respective benchmarking tools can be used.

⁸Popular key-value stores with well-defined protocols include Memcached (<http://memcached.org>) and Redis (<http://redis.io>); benchmarking tools exist for both systems.

Exactly-once semantics. Use unique client IDs to guarantee exactly-once semantics.

Heterogeneous voting scheme. Allow for a mapping from the number of nodes to a voting scheme to be defined as part of the configuration before Rafter starts up. Rafter should then switch to the appropriate voting scheme whenever a membership change occurs.

Advanced key-value store. Many extensions to the existing simple key-value store state machine are possible. One idea would be to use a persistent database for data storage.

Profiling and performance tuning. The voting structure interpreter will incorporate some non-trivial tree walking operations. Trying to make these operations, and therefore quorum finding, as fast as possible would be a useful extension.

Timetable and Milestones

Before Proposal Submission

Discussion with Overseers and Director of Studies. Allocation of and discussion with Project Supervisors, preliminary reading, and writing the Project Proposal. Discussion with Supervisors to arrange a schedule of regular meetings for obtaining support during the course of the year.

Milestones: Phase 1 Report Form on October 14, 2013, then a Project Proposal complete with as realistic a timetable as possible by October 17, 2013, approval from Overseers and confirmed availability of any special resources needed. Signatures from Supervisors and Director of Studies.

Weeks 1 and 2 (Oct 21 to Nov 3)

Familiarisation with Erlang: I will read the relevant parts of the introductory Erlang book *Learn You Some Erlang for Great Good*,⁹ get an overview over the Erlang documentation and ecosystem, and write small example programs in Erlang.

Milestones: An Erlang implementation of a non-trivial algorithm that uses records, tuples, lists and recursion. There should be unit tests, and *Dialyzer*¹⁰ should be employed for static code checking.

Weeks 3 to 6 (Nov 3 to Dec 1)

Designing the voting structure data representation and implement generators for Majority Consensus and the Grid Protocol, as well as an algorithm to interpret voting

⁹Hébert 2013.

¹⁰Dialyzer, the Discrepancy Analyzer for Erlang programs, can be used to statically type check Erlang programs. <http://www.erlang.org/doc/man/dialyzer.html>.

structures. For debugging, I will write a tool that allows a visual representation of the generated voting structures.

Milestones: Diagrams of the voting structures generated by the implemented voting schemes.

Weeks 7 and 8 (Dec 2 to Dec 15)

Incorporate voting structures into Rafter. This involves changing the per-node configuration data to include the current voting structure.

Weeks 9 to 11 (Dec 16 to Jan 5)

Buffer time to catch up with the timetable in case I have fallen behind the schedule at this point.

Week 12 (Jan 6 to Jan 12)

Finish incorporating voting structures into Rafter and testing.

Milestones: A demonstration of Rafter successfully running the existing key-value store state machine using the Grid Protocol to find quorums.

Weeks 13 and 14 (Jan 13 to Jan 26)

Write progress report and create presentation.

Milestone: Handing in the progress report and delivering the presentation.

Weeks 15 and 16 (Jan 27 to Feb 9)

Design and write the benchmarking tool.

Milestones: Running the benchmarkings locally and collecting and interpreting the results.

Weeks 17 and 18 (Feb 10 to Feb 23)

Set-up and configuration for Amazon EC2. Running the benchmarks.

Milestones: Collecting and interpreting the benchmarking results.

Weeks 19 and 20 (Feb 24 to Mar 9)

Buffer time: can be used for core work if I have fallen behind the schedule at this point; alternatively, this time can be used to work on one of the optional extensions.

Weeks 21 to 25 (Mar 10 to Apr 13)

Write Dissertation.

Milestones: Discussion of the dissertation draft with my supervisors.

Weeks 24 to 26 (Apr 14 to Apr 27)

Clean-up of code and dissertation.

Resources Required

Development will predominantly be on my laptop running Linux. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure: I will use Git to keep track of changes in my code and reports, and I will push these changes to my GitHub repository¹¹ every day. In addition, I will take weekly backups to an external hard drive.

In order to benchmark the project under realistic conditions, I will require Amazon EC2 compute time on a sufficient number of nodes (at least 100). My supervisors have assured me that the Systems Research Group has credits available that I can use.

¹¹GitHub is a Git repository hosting service. The most current version of my code and reports will always be available at <http://github.com/curiousleo/rafter>.