Reaching Consensus Using Raft

Joakim Antus
Abstract

Reaching Consensus Using Raft

Joakim Antus

This thesis project implements and evaluates log replication using the Raft consensus algorithm. Raft presents a new and easier to understand consensus protocol for log replication in a distributed system. This thesis aims to evaluate the correctness and robustness of Raft by implementing a scalable system that is easy to maintain and test for further development. This thesis gives an in-depth description of Raft as well as a detailed explanation of the implemented system together with an evaluation of the system performance with focus on correctness.
# Contents

Abstract

1 Introduction 1
  1.1 Objectives and Contribution ........................................... 2

2 Background 3
  2.1 Fault Tolerant System ................................................ 3
  2.2 Failures ............................................................... 3
    2.2.1 Byzantine Failures ............................................... 4
    2.2.2 Fail-Stop Failures ............................................... 4
  2.3 The State Machine Approach ........................................ 4
  2.4 Consensus Algorithms ................................................ 5
    2.4.1 Symmetric Consensus Algorithms ................................ 6
    2.4.2 Asymmetric Consensus Algorithms .............................. 6

3 Raft 7
  3.1 Raft - An Overview .................................................. 7
  3.2 Leader Election ..................................................... 9
    3.2.1 Timing and Availability ....................................... 10
    3.2.2 Safety and Election Restriction ................................ 11
  3.3 Log Replication ..................................................... 12
    3.3.1 Catch Up ........................................................ 13
    3.3.2 Conflicting Entries ............................................ 14
  3.4 Follower and Candidate Crashes .................................... 15
  3.5 Configuration Changes and Log Compaction ........................ 15

4 System Design 16
  4.1 Server Design ...................................................... 17
  4.2 Remote Procedure Calls (RPCs) .................................... 18
    4.2.1 RPC Structure .................................................. 19
    4.2.2 Internal Communication ....................................... 19
  4.3 Consensus Module ................................................... 20
    4.3.1 Leader State .................................................... 20
    4.3.2 Candidate State ............................................... 21
    4.3.3 Follower State ................................................ 22
  4.4 Log ................................................................. 22
    4.4.1 Journal .......................................................... 22
Chapter 1

Introduction

Log replication over a distributed system, has been and still is essential to many businesses. More and more of our private data and business data are moved to the cloud. When storing data on the cloud the expectations are that the data stays safe and uncorrupted for as long as needed. This puts demands on the underlying system, which must be able to provide the data uncorrupted even in the event of partial hardware failures.

The solution to this problem is to replicate the data over multiple servers in a distributed system. This reduces the risk of losing data in case of hardware failures. Replicating data in a distributed system without having requirements on the data ordering or latency is not a problem.

The problem however becomes more complex when requirements are introduced in how the data is stored. For example, if input from multiple sources is to be stored at the same time and the order of the data is important, the servers storing the data must collectively decide in what order to handle the input. Add to this a time constraint of the system’s response time on input and the complexity of the problem has increased.

Most IT-businesses handling monetary transactions will face the problem of building a highly responsive fault tolerant system. Multiple transactions can happen essentially at the same time and the order of the transaction may affect the outcome. A distributed system responsible for handling transactions must be able to instantly decide in what order to handle the transactions and then securely log each transaction on multiple servers before responding to the transaction.

An incorrect logging of a monetary transaction or loss of data due to hardware failures could potentially lead to legal repercussion, bad press and loss of market shares. That is why solution to this problem is essential to a many of businesses.
1.1 Objectives and Contribution

Svenska Spel AB handles thousands of transactions every day. In the event of an erroneous logging or hardware failure, a safe shutdown of the service is executed. A manual startup of the system has to be performed before the service can be provided again. These events are rare but costly, which is why a solution for automatic failover is of interest.

This project has been conducted on the request of Svenska Spel AB to design and implement a system for automatic failover based on the Raft algorithm. The main objectives are correctness and consistency of log replication.

The work was divided into two parts, where the first part was to identify the drawbacks and advantages of different consensus algorithms by reading scientific literature and examine various implementations. The second part involved designing and implementing a system meeting the criteria of a robustness and correctness.

The project resulted in:

- A scalable and easily maintainable implementation of the Raft algorithm.
- Creation and implementation of a log design suitable for fast editing and persistent storage.
- Improved indexing of log entries compared to the one proposed in the original description of Raft.
- Suggested improvements for further work towards a live system.

The resulting system met the requirements and proved to be an interesting implementation for further development.
Chapter 2

Background

2.1 Fault Tolerant System

Aviziens [1] defines a fault-tolerant system as "a system, which has the built-in capability to preserve the continued correct execution of its programs functions in the presence of a certain set of operational faults", provided that the system performs correct execution in absence of operational faults.

Where "correct execution" means that the program performs the execution in a timely manner without any errors, given that the data provided is without error. An operational fault is an error that occurs due to failure in one or more hardware components.

In other words, a fault-tolerant system is a system that is able to operate correctly, without external assistance, even though some parts of the hardware malfunction.

A system is said to be $t$ fault tolerant if the system can guarantee to operate correct when no more than $t$ of its components are faulty. [2]

2.2 Failures

Failures are typically separated into two types of faulty behaviors that are common when dealing with fault tolerant systems, Byzantine Failures and fail-stop failures. Byzantine failures may occur when components can send conflicting information to other parts of the system (see section 2.2.1). Fail-stop failures are failures where a component of the system halts in response to a failure (see section 2.2.2).
2.2.1 Byzantine Failures

Byzantine failures are failures that cause a component of a computer system to misbehave and send conflicting or erroneous information to other parts of the system.

Lamport, Shostak and Pease described Byzantine failures in [3] where generals in the Byzantine army were used as metaphor for communicating nodes in a computer system. The problem is described as a communication issue where the generals are trying to reach a correct decision even though some of the generals are traitors and will send conflicting information. This is analogue to a computer system, where nodes in distributed system are trying to reach consensus on an issue and some of nodes are malfunctioning.

Several solutions to handle Byzantine failures in a distributed system were proposed, and it was also proven that at a majority of more than two-thirds of the nodes in a system are required to work properly to be able reach consensus.

2.2.2 Fail-Stop Failures

A component in a system implemented for fail-stop failures is forced to halt in response to failures. Before halting, the malfunctioning component transfers to a different state enabling other components of the system to detect that a failure has occurred. It is proved that a system of $2k + 1$ nodes can handle up to $k$ failures without risking inconsistent behavior. [2]

2.3 The State Machine Approach

As defined by [4] a state machine is a system that can be in one or more states, represented by state variables. Given a command the system will atomically transfer to another state. This transition is deterministic in the means that there can only be one outcome given current state and command.

Clients utilizing a state machine can operate under the premise that commands will be processed sequentially in the same order as they were sent to the state machine. The output of a state machine is therefore only determined by the input and completely independent of time or other activity of the system.

A fault-tolerant system can be constructed by replicating an implementation of a state machine on a distributed system. To maintain a correct execution on a fault tolerant state machine all state machines in the distributed system must start in the same initial
state and execute the exact same sequence of commands. The system as a whole must therefore reach consensus on what command to execute in any given state.

This kind of \( t \) fault-tolerant system can handle both Byzantine failures and fail-stop failures. To handle Byzantine failures at least \( 2t + 1 \) of the machines must be operating without any faults for the system to reach consensus and produce a correct output (see section 2.2.1). For fail-stop failures, at least \( t + 1 \) of the machines must be operating without faults to guarantee correct output (see section 2.2.2).

### 2.4 Consensus Algorithms

This section defines the meaning of consensus and what reaching consensus means in the scope of this thesis work.

A consensus algorithm is an algorithm designed to, in a group of actors; agree on one of multiple decisions. When such an agreement is met, the group is said to have reached consensus. Depending on the design of the algorithm one or more of the actors may be able to propose values for the group to reach consensus on. The algorithm must guarantee that only one at a time of the proposed values is chosen. When a value has been chosen all actors must accept that value as chosen.

For the algorithm to reach consensus it must fulfill the following safety requirements defined by Lamport.[5]

- Only one value that has been proposed may be chosen,
- Only a single value is chosen, and
- An actor never learns that a value has been chosen unless it actually has been.

If these requirements are met, the algorithm ensures that each individual machine only accepts one value and that the accepted value is the same on all machines. This enables a set of machines to continue operation as coherent group even though some of the machines fail.

There are two general approaches to consensus algorithms, symmetric and asymmetric.
2.4.1 Symmetric Consensus Algorithms

Symmetric, also called leader less, is an approach in which all servers in a cluster have equal roles and communication has to flow between all servers to be able to reach consensus. Clients interacting with the system can communicate with any server in the cluster. [6]

2.4.2 Asymmetric Consensus Algorithms

In an asymmetric approach, also called leader based, one server acts as a designated leader through whom all communication must pass. Other servers act as passive followers and only accept the decisions the leader has made. All clients interacting with the system are required to communicate directly with the leader. [6]
Chapter 3

Raft

Raft is a consensus protocol that was first briefly described by Ongaro and Ousterhout in [7] with a more rigorous explanation in Ongaro’s PhD thesis [8]. This chapter will give a detailed description of the Raft algorithm as described by Ongaro and Ousterhout.

Raft is designed to run on a distributed system consisting of a cluster of servers. Each server has a running instance of the Raft implementation consisting of a state machine, a consensus module and a log. During the rest of this chapter, server will be used when referring to a server in the distributed system running a Raft implementation.

3.1 Raft - An Overview

Raft is a consensus algorithm designed for log replication. The motivation of Raft was to create an understandable consensus algorithm. Previous attempts on designing consensus algorithm often resulted in complex systems that were hard to understand and unsuitable for real world practices. The decision to create an understandable consensus algorithm had a direct effect on the algorithm since when multiple design choices were possible; the authors always chose the more intuitive one.

In Raft, a server can be in one of three different states, leader, candidate and follower. Raft uses an asymmetric approach to the consensus protocol, which means that there can only be one server acting as leader (see section 2.4.2).

Each state has its own characteristics and a server in specific state must follow those characteristics.

**Leader**, there may only be one leader at any give time. The leader handles all incoming and outgoing communication with clients. The leader is also responsible for log
replication, making sure that all servers maintain an exact replica of the leader’s log. A server can become leader if a majority of the cluster votes for it during an election.

Candidates can request to be elected leader. A server can transfer from follower state to candidate state if it believes that there is no viable leader. A follower will automatically assume that there is no viable leader if it does not receive any indication of the opposite.

Followers are passive and only respond to requests from the leader and candidates.

The fact that a server can only be in one state at a time is a design choice by the author to simplify the algorithm compared to the Paxos algorithm where a server can be in multiple states (see section 6.1).

Raft has divided time into time-independent terms. Terms are identified by discrete consecutive numbers. Terms can be of different length and a term will last for as long as the leader can maintain its authority. A new term starts at the beginning of a leader election and ends when a new leader election starts (see section 3.2). Terms work as a global clock for the whole system, all servers must therefore keep track of what term they are in, and update their term to match the leader’s if necessary. Under normal operation, all servers must have the same term. The term must be stored persistently on the server, so that a server does restart at a lower term after a crash.

During a term, a server can be in one of the three states explained above. A server is allowed to make one state transition per term. How a server can transfer is determined by the Non-deterministic Finite Automaton (NFA) seen in figure 3.1. All servers start in follower state from which they can transfer to candidate state. A server is only allowed to be in candidate state when requesting to be elected leader. After an election, the candidate must either return to follower state or proclaim itself leader. A leader must transfer back to follower state when it detects another server with higher term and hence more up-to-date.

Servers communicate with each other by sending Remote Procedure Calls, RPCs. Basically there are only two types of RPCs, AppendEntries RPCs and RequestVote RPCs. The AppendEntries RPC is used by the leader to send log entries for followers to append to their logs. RequestVote RPCs are sent by candidates during elections requesting to become leaders.

In addition to the RPCs described above, a third type of RPC called Heartbeat RPC exists. The Heartbeat RPC, used solely by the leader to communicate with its followers, is an AppendEntries RPC without any log entries included.
For each RPC there exists a corresponding acknowledgment (ACK). The ACK is used to inform candidates if the vote request was granted or to tell the leader if the appending of entries could be performed successfully.

For best performance, RPCs should be broadcasted in parallel to all neighboring servers.

### 3.2 Leader Election

Leader election is the process taken to elect a new leader among the servers. A new election is started when a follower’s *election timeout* is triggered. The election timeout is triggered if a period of time passes without the follower receiving any communication from the leader or a candidate. If no communication is received, the follower will assume that there is no viable leader. In that case the follower will start a new leader election by increasing its term and transfer to candidate state. The new candidate starts by voting for itself to become leader and then sends out a RequestVote RPC in parallel to all servers in the cluster. If a majority of the cluster grants the candidate’s vote request, it becomes leader.

Two followers may start an election at the same time, resulting in a split vote where no leader is elected and a new election must be held. To reduce the risk of split votes, each server has their own election timer, firing after a random period of time. A random timeout ensures that a leader will be elected eventually.

Servers votes for candidates in a *first-come-first-served* approach which means that servers do not have any bias in whom it votes for, but instead votes for the server who first requests its vote. A server may vote for only one server during an election, itself included. If a server receives a second RequestVote RPC after it has voted, the request will automatically be rejected unless that new RPC has a higher term than the previous request, which would imply that a new election has started.
For each new election, there are three possible outcomes.

1. One candidate receives a majority of the votes.
2. Another server establishes itself as the leader.
3. A period of time goes by without establishing a new leader.

In case 1, a candidate receives votes from a majority of servers in the cluster. The candidate declare itself leader by transferring to leader state and broadcasts Heartbeat RPCs to all of its followers to demonstrate authority and avoid any new elections.

Case 2, during an election a candidate may receive an RPC from another server claiming to be leader. If the other server’s term is at least as large as the candidate’s, the candidate will recognize the other server as leader and return to follower state.

In Case 3, no candidate receives enough votes to declare itself leader. If this occurs, all servers will remain stall until an election timeout is triggered and a new election is started. This can happen for two reasons, either only a minority of the cluster is viable in which case no leader will ever be elected, or more than one candidate receives a minority of the votes, resulting in a split vote. In the former case, new elections will be held until a majority of the cluster becomes viable. In the event of a split vote, all servers will remain stall until new election is triggered. Since the election timeouts are based on a randomized timer, split votes are rare and will be resolved quickly.

To maintain its authority and prevent unnecessary elections the leader periodically broadcasts Heartbeat RPCs to all of its followers. The heartbeats must be sent periodically before any server has a chance to start a new election.

3.2.1 Timing and Availability

As described above an election is triggered after a random period of time. The length of the time period does however affect the behavior of the system. If the period is too short, new elections will be held all the time, reducing the performance of the system. If, on the other hand, the time period is too long, the system will not react to leader crashes in a timely manner.

To avoid unnecessary elections and reduce the number of split votes, a server should timeout when no other server times out and do so frequently enough to reduce stall time but not to often to avoid split votes. The solution is an interval in which the timeout will be set randomly. The random timeout interval guarantees that a leader eventually will
be elected if the interval is large enough. This interval may differ depending on cluster size and what hardware the system is running on. Raft will however work correctly, electing new leaders and updating the log, as long as the election timeout fulfills the following requirement:

\[
\text{BroadcastRTT} \ll \text{Electiontimeout} \ll \text{MTBF}
\]

Where \text{Broadcast RTT} is the round-trip-time, in milliseconds, it takes to broadcast RPCs to all servers and await their acknowledgment. The broadcast RTT cannot be set by the user but is instead a factor of the hardware. The \text{Election timeout}, also measured in milliseconds, is however manually set in the implementation and should be at least a factor 10 larger compared to the broadcast time. \text{MTBF} (Mean Time Between Failures) is how often hardware failures occur, this is often hours, days or even years.

The interval above is quite large which is why a smaller interval is recommended. An interval that might be a bit conservative but still give good performance is the following.

\[
10 \times \text{BroadcastRTT} < \text{Electiontimeout} < 20 \times \text{broadcastRTT}
\]

### 3.2.2 Safety and Election Restriction

If a leader commits entries while a follower is unavailable and that follower is elected leader when it wakes up, this new leader could potentially overwrite entries already committed by a previous leader, resulting in inconsistent logs. To avoid this Raft has included a safety restriction, which ensures that a server can only be elected leader if its log contains all the committed entries from the previous term.

With this safety restriction, Raft can guarantee that the new leader’s log will contain all the previously committed entries. This holds true since a candidate must contact a majority of servers to be elected leader and if the candidate’s log is not as up-to-date as a follower’s log, the follower will not grant the vote request.

A server decides which of two logs are more up-to-date by comparing the term and id of the last entry in each log. The log with higher term on its last entry is always more up-to-date. If two log ends with the same term number then the log with a higher ID on its last entry is more up-to-date.

This up-to-date check of the log is possible since in each RequestVote RPC the candidate includes information about the term and ID of its last log entry.
This means that a candidate that is not as up-to-date as a majority of the cluster cannot be elected leader, since a majority of the servers will decline the request. This also ensures that if a leader crashes before it can commit a log entry future leaders will try to commit that same log entry.

### 3.3 Log Replication

Raft is designed for log replication. When the leader receives a command from a client, it must ensure that the command is replicated and securely stored on other servers before executing the command in its state machine.

To ensure that a command is securely stored and cannot be overwritten, a majority of servers in the cluster must accept the new command as an entry in their log. When a specific entry is stored in a majority of the logs, that entry is said to be committed. A committed entry cannot be overwritten and can therefore be safely executed on the state machine.

To accomplish this, the leader issues AppendEntries RPCs; with the new log entry included, in parallel to all of its followers.

When a follower receives an AppendEntries RPC, the follower evaluates the content of the RPC and if it contains the next expected entry, the follower persistently stores the entry in its log. When the log entry is securely stored, the follower sends an acknowledgment notifying the leader that the entry was successfully stored.

The leader waits for acknowledgments from a majority of the cluster before it marks the entry as committed and applies it to its state machine.

A leader does not send new AppendEntries RPCs to a follower before it receives an acknowledgment on last sent RPC. If a follower fails to reply on an RPC, the leader will keep sending the same AppendEntries RPC indefinitely until an acknowledgement is received from the follower. The leader can however keep sending new RPCs to other followers. This may result in one or more followers falling behind the majority of the cluster; Raft can handle these situations and will keep functioning properly as long as the leader receives replies from a majority cluster.

An entry may be deleted or overwritten unless it is committed. To make sure that all followers know what entries have been committed, every RPC (Heartbeats included) includes the ID of the last entry to be committed in the leaders log. The followers keep track of the highest committed entry in their log. If a followers log is up-to-date with the leaders log, both logs will store the same committed entries. If on the other hand
the followers log does not yet contain the last committed entry in the leaders log, the follower will mark all of its entries as committed.

A Raft log must always maintain the following two properties:

- If two entries in different logs have the same index and term, then they store the same command.
- If two entries in different logs have the same index and term, then the logs are identical in all preceding entries.

Together these two properties constitute the Log Matching Property. To be able to maintain this property, the leader keeps track of what to send next to each follower by associating a nextIndex with the each of the followers.

In each AppendEntries RPC, the leader also includes information about the term and ID of the log entry preceding the new entry included in the RPC. This allows the receiver of the RPC to make a consistency check, if the log of the receiving server does not contain an entry matching the previous one, the new entry may not be appended.

This ensures that no inconsistency in logs can occur. For consistency to work it is of importance that the leader never deletes or overwrites any entries in its own log.

Raft will keep working and apply, accept and replicate new entries as long as a majority of the cluster is up. If not then no entries can be committed.

Except for the normal case where all logs are up-to-date and new entries can be appended with out a problem, two special cases can occur.

### 3.3.1 Catch Up

When a follower receives an AppendEntries RPC, it examines the information about the new entry’s predecessor. If the follower’s log does not contain an entry matching the predecessor, it refuses to append the new entry and an acknowledgment notifying the leader of the missing entry is sent.

Receiving this information, the leader will instead try to append the previous entry in its log by sending it to the follower. This process will repeat, iterating back in the leader’s log, until a matching entry is found. The leader will make sure the follower catches up with the leader’s log by sending all missing entries until the log’s are equal (see figure 3.2).
Figure 3.2: Server S3’s has the most up-to-date log and is therefore elected leader. S3 will try to replicate the entry in term three to both S1 and S2. S2 can immediately append the entry but S1’s log does not contain the entry preceding the new entry. S3 will send preceding entries, one at a time, until a matching entry is found. S3 will then send the missing entries, updating S1’s log to match the leader log.

3.3.2 Conflicting Entries

If an entry in the follower’s log conflicts with the new entry sent from the leader, the leader forces the follower to replicate its log and overwrite any conflicting entries if necessary. Two entries are conflicting if they have the same ID but different terms.

For the logs to be duplicates, the follower must first revert its log back to the last entry where the logs matched (as described in section 3.3.1) and delete all entries before that. The leader then sends AppendEntries to the follower until the followers logs is as up-to-date as the leader’s log.

Figure 3.3: Server S3 is elected leader and will immediately try to replicate the entry in term five to both S1 and S2. S2 can immediately append the entry but S1’s log contains conflicting entries. S3 will send the preceding entries, one at a time, until a matching entry is found. S1 will then remove the conflicting entries in term two and three and then append the missing entries from term four and five, updating S1’s log to match the leader log.
3.4 Follower and Candidate Crashes

If a follower or candidate crashes and becomes unavailable, all AppendEntries RPCs and RequestVote RPCs sent to the unavailable servers will fail. Raft solves this by letting the leader resend the RPC until a reply is received. When the crashed server eventually restarts the RPC will complete and the sender will get a response. This also implies that only one message is sent to a follower at a time.

Due to latency, more than one copy of the RPC may be sent to a follower. If a follower or candidate receives an AppendEntries RPC that has already been logged it simply ignores that RPC.

3.5 Configuration Changes and Log Compaction

Since Raft is statically configured, meaning that all servers have to know about all other servers, adding new servers requires a configuration change. Changing configuration cannot safely be done atomically without shutting down the system. Raft instead uses a two-phase approach, where the new configuration is gradually accepted by the cluster.

Under normal operation, logs grow longer and eventually unmanageable as more space is occupied and reading back from a log takes longer time. To avoid this Raft uses log compaction by snapshotting the entire current system. Snapshots can be used to restore servers and bring back lost log entries.

Both membership changes and log compaction fall outside the scope of this project, the curious reader is instead referred to the paper by Ongaro and Ousterhout [7].
Chapter 4

System Design

The system is designed to be implemented according to the Raft algorithm but modified to suit the requirements demanded by the end user (Svenska Spel AB).

The system is designed and built after the requirements and suggestions of Svenska Spel AB. The implementation is done in C using the C POSIX standard library [9]. In addition to the POSIX library, a few libraries native to Svenska Spel AB were used to implement queues, threads and memory mapping. No other third party frameworks or software was used.

The system design and implementation is made to be flexible. The system can therefore be adjusted to fit future needs. Currently the design is based on a cluster of three nodes, which is what Svenska Spel uses and what will be used when explaining the system in the remainder of this Chapter. However, the design and implementation of the system is flexible and can adjust to a larger or smaller cluster, by only changing the configuration file. Although Raft works properly on a cluster of two or fewer nodes, a consensus algorithm would not necessary since the whole cluster must be functioning for the system to be available.

While designing this system, focus has been on creating a clean and easily maintainable system. This has lead to a system design consisting of different modules, all separated by queues to reduce the need for synchronization. Each module can easily be tested, updated or replaced without affecting the rest of the system.

This design made it possible to separate all the Raft logic into a separate module, which enhanced the development of the consensus logic but also made it easier to convince oneself that the logic was performed correctly.

Figure 4.1 shows the communication in a cluster of three nodes. Note that each server has one outgoing and one incoming connection for each of its neighbors. Not showing in
this picture is the outgoing and incoming connections each server has reserved for client connections. The cluster in figure 4.1 consists of three nodes, which is what Svenska Spel will be using in their implementation. The system is however tested and working flawlessly for clusters of up to five nodes.

Figure 4.1: An overview of the communication between servers in a cluster of three nodes.

4.1 Server Design

A server consists of different modules, all servers in the system have the same implementation and only the configuration differs among them. All servers are configured to know about all the other servers in the system.

Each module is responsible for performing a specific task. For each module, a thread is assigned to allow the work of different modules to be performed in parallel.

Figure 4.2 gives an overview of the system and how the different parts are connected to each other. The main part of the system is the Consensus Module, it is responsible for performing all the Raft logic, answer to input and produce output. The Consensus Module is also responsible for updating the log. All work in the Consensus Module is performed by a single thread. Since a single thread performs all the logic the need for synchronization is omitted, and the risk of faulty behavior is reduced.

Servers communicate with each other by sending RPCs. It is the Consensus Module’s responsibility to decide what to reply. The Consensus Module receives input by reading from the In-Queue. The In-Queue contains RPCs received from other servers as well as timeout messages from the timer. All information passed to the Consensus Module must go through the In-Queue.
Depending on the information in an RPC dequeued from the In-Queue; the Consensus Module may update the Log or send out RPCs to one or more of its neighbors. The Consensus Module sends RPCs by placing the RPC in the corresponding Out-Queue, there is one Out-Queue for each neighbor. The Sender Module dequeues the RPC from the Out-Queue and sends it to the correct location.

Each server has one incoming and one outgoing TCP connection to all other nodes in the system. Each communication route is handled by an individual thread resulting in two threads per neighbor.

Separating different parts of the system with queues offloads the need of synchronization in the system. The only critical section in this implementation are the queues. Queues are implemented as pointer queues of fixed size and are protected by a mutex lock. All queues are FIFO-queues (First-In-First-Out), guaranteeing that RPCs are handled the same order as they were received. The reason for using queues of fixed size is to be able to detect faulty behavior. If a queue is full when enqueueing an element it indicates that something in the system is wrong, leading to a controlled shut down of the server.

4.2 Remote Procedure Calls (RPCs)

Raft servers communicate with each other by sending Remote Procedure Calls (RPCs). There are different types of RPCs and depending on the type of RPC and the state of the receiving server, it will respond in different ways.

There are two main types of RPC paired with a corresponding acknowledgement (ACK) RPC.

**AppendEntriesRPC** is used when sending new entries to append to the log. Only leaders are allowed to send AppendEntriesRPCs. For each AppendEntriesRPC a
leader sends it must receive an AppendEntriesACK, if no ACK is received, the leader will resend the RPC until it receives the expected ACK.

**RequestVoteRPC** is used by candidates when starting a new election. The candidate broadcasts RequestVoteRPCs to all of its neighbors, asking for their votes in the election. The receiver of a RequestVoteRPC replies by sending a RequestVoteACK, the ACK contains information of whether or not the vote has been granted.

### 4.2.1 RPC Structure

All RPCs have the same structure, no matter if it is an AppendEntriesRPC, a RequestVoteRPC or one of the corresponding acknowledgements.

All RPCs consists of a header and a data buffer. The header includes information about what type of RPC it is, the ID of both the sender and the receiver, as well as parameters required by the Raft algorithm.

Since an entry can vary in size, so can RPCs. Therefore an additional parameter is included in the header, telling the receiver how many bytes of data the RPC includes. If the data buffer is empty, only a header is sent. This is case for most RPCs; only the AppendEntriesRPC may have a non-empty data buffer. If an AppendEntriesRPC does not contain any data, it is decoded as a HeartbeatRPC. The HeartbeatRPC is used by the leader to maintain its authority when all of the logs are up-to-date.

Receiving an RPC, the receiver only looks at the parameter necessary for the specific type of RPC and ignores the rest. This implementation therefore forces servers to include unnecessary information in their RPCs, resulting in larger messages. This is a trade-off for a simpler and cleaner design and the extra information included is only a small portion of the total message.

### 4.2.2 Internal Communication

The same message structure is used for elements in the In-Queue and Out-Queues, allowing for inter module communication. The most important internal message is the *Timeout* message; it is enqueued to the server’s In-Queue by the *Timer*. This message holds no information except for its type. The timeout message is used to inform the Consensus Module that a timeout has occurred.
4.3 Consensus Module

The Consensus Module is the main part of the server and it has control over all other parts of the system. It is in the Consensus Module where all the Raft logic takes place and all the decisions are made.

The Consensus Module is responsible for handling all incoming RPCs and to update the log when necessary. The decision-making differs depending on what state the server is in. The overall principle of the Consensus Module is a loop by which it first dequeues an RPC from the In-Queue, decodes it and takes action depending on its contents. Often a new message is enqueued to the appropriate Out-Queue before a new iteration is started by dequeuing the next RPC.

A server can be in three different states, leader, candidate and follower. How the Consensus Module will act is dependent on the state.

4.3.1 Leader State

Figure 4.3 gives an overview of how a server acts in leader state. The first thing a leader does after transferring to leader state is to broadcast HeartbeatRPCs to all of its followers. The leader will then enter the Consensus Module loop, which starts by the dequeuing of a message. In this implementation of Raft, a leader is required to handle at least four different message types. The main difference compared to the original Raft implementation is that the leader has to handle one additional message type, the Timeout message. The Timeout message is enqueued by the timer and tells the Leader that it is time to send out HeartbeatRPCs to maintain its authority.

![Figure 4.3](image)

**Figure 4.3:** An overview of the Consensus Module execution when the server is in leader state.

After decoding a message, the leader either completes the loop by enqueueing a new message to one or more of the followers or immediately dequeues a new message.
There is only one *End State*, where the leader stops iterating in this loop, and that is when it reverts to follower. This only happened when the leader realizes that it has lost its authority as leader.

Note that figure 4.3 does not show the logic for evaluating what to respond on an RPC.

### 4.3.2 Candidate State

The execution of a server in candidate state is very similar to one in leader state, as can be seen in figure 4.4. The main difference is that a candidate responds to RequestVoteACKs instead of AppendEntriesACKs. For each RequestVote a candidate sends it will count the number of ACKs granting the candidate’s vote request. If a majority of the cluster grants the candidate’s request to become leader, the candidate transfers to leader state.

If, on the other hand, a new election timeout is triggered while the server is still in candidate state; it will once again start a new election by transfer to candidate state and increasing its term.

If a candidate receives an AppendEntriesRPC with a higher term than its own, it will immediately return to follower state.

![Diagram of Consensus Module execution when the server is in candidate state.](image)

**Figure 4.4:** An overview of the Consensus Module execution when the server is in candidate state.

How a candidate responds on a RequestVoteRPC is not shown in figure 4.4.
4.3.3 Follower State

As can be seen in figure 4.5 the execution of a server in follower state is very simple. For the most part a follower remains passive and only replies to RequestVoteRPCs and AppendEntriesRPCs. The only thing breaking the dequeue-enqueue loop is when an election timeout is triggered and the follower transfers to candidate state.

![Diagram](image)

**Figure 4.5:** An overview of the Consensus Module execution when the server is in follower state.

The logic for what a follower will reply to a RPC is not shown in figure 4.5.

4.4 Log

The log is designed in a similar way to the log already used by the Svenska Spel.

The log is a collection of pointers (figure 4.6) to keep track of the current position in the log. The log is divided into Journals, where each journal is a file (this implementation instead of files allocates main memory, this to simplify implementation and keep focus on the Raft logic). Each journal is divided into a number of blocks, in which log entries can be appended. Entries can be of different length and the number of entries can therefore vary between blocks.

4.4.1 Journal

The main reason for dividing the log into journals is to be able to append new entries to one journal while simultaneously archiving another journal. When a journal has been archived its allocated memory can be safely reused to represent a new journal. Regularly archiving journals reduces the need for new memory allocation. The part of the log currently being edited can therefore always be kept in main memory allowing for fast insertion of new entries. Archiving falls outside the scope of this project.
Journals are indexed with unique identification numbers (IDs). The indexation are made up of consecutive integers, with the first ID, 1, assigned the first journal.

A journal can be in one of three different states, *Initialized*, *Active* or *Finalized*. At initialization, a journal enters state *Initialized*. This means that memory is allocated and assigned to the journal, but no data is yet written to it. At least one of the journals held in memory must be in state *Initialized* at any given time. This pre-allocation of memory ease the transition between journals and reduces the stall time when creating a new journal.

The first time a journal is modified, it will transfer to state *Active*. Before a journal can switch to the active state, the journal currently in state active must first change to state *Finalized*. This ensures that only one journal at a time is in state Active. Allowing only one journal to be in state active is a safety restriction, assuring there is only one journal where new entries can be appended. The active journal will switch to state *Finalized* if the new entry to append does not fit in the current journal, the entry is then instead appended in the beginning of the next journal.

When a journal is full, it switches to state *Finalized* and no more entries can be appended to it. The journal can however still be modified, if and only if, it contains any uncommitted entries. Because of this, a server must not only change the state of a journal when switching journals but also update whether or not all entries are committed. Only when all entries are committed can the journal be securely stored on disk.

All journals are divided into blocks of equal size (see figure 4.7). The first block (block 0) is reserved as the head of the journal. This head holds information about the journal’s current state and if all entries are committed or not. If the journal is in state *Active*, the
head works as the head for the whole log, holding information about the log together with parameters required by Raft to be stored persistently on the server (3.1).

![Figure 4.7: A journal is a chunk of memory divided into blocks. The first block in a journal is always used as the head of the log.](image)

The size of a journal should be decided when initializing the log. The size of a journal will be restricted to the amount of main memory; too large journals will not fit. If choosing a journal size that is too small, the number of blocks will decrease and the overhead of block 0 (the head) will increase. Typically, a log should have at least three journals (one in each state) in main memory at any time, but preferably more than one journal in state Finalized, since it may contain uncommitted entries.

### 4.4.2 Block

Each journal is divided into a number of blocks of fixed size, which have to be set at the initialization of the log. The reason for using blocks is to guarantee that writes can be performed atomic. This guarantee will however only hold if the blocks are smaller than a specific size. By choosing a small enough size, writes will be executed without risking data interleaving from another process writing to the same pipe. Hence, the system can assure that data will not be corrupted due to data races or hardware failures [9].

The size of a block is therefore limited by the software on which the system is running. For Linux versions before 2.6.11 the maximum block size is 4096 bytes while systems running on a later version can have blocks of size no larger than 65536 bytes [10].

Each block has a number assigned to it as ID, this number is unique within the journal, starting at zero for the first block and then monotonically increased. As mentioned in section 4.4.1 Block 0 is reserved for the journal head, the first block containing data is therefore block 1.

All blocks (except block 0) have a header. The header holds information about the block’s content and a pointer to where its data begins (see figure 4.8). The header also contains information about number of entries in the block, how many bytes of data that has been written and how many bytes remain unwritten. By keeping track of how many bytes have been written the block knows the position of the next entry to append. This position also works as the identification number of the entry within the block. This entry ID will therefore always be unique within the block.
4.4.3 Entry

An entry consists of a header followed by a byte array of size one (see figure 4.9). The header includes information about the entry’s term, the size of the entry, its ID and the ID of the entry preceding this one. With this information it is possible to iterate through a block in a similar fashion to a linked list.

Since entries can vary in size, it would be impractical to store the data in an array of fixed size. The byte array of size one works as a placeholder for the data to be stored. The array will immediately overflow, this will however not be a problem since the entry will be stored in a larger pre-allocated memory slot. As long as the header holds information of the total entry size, data can be securely written and entries stored back to back without wasting any space.

If an entry does not fit in a block, a new block is initialized and the entry is appended in the new block instead. The old block is now considered full and no new entries can be appended to it. This implementation limits the size of an entry to be at most as large as a blocks capacity to store data.

An entry’s ID differs from the one described in the Raft paper (with monotonically increasing numbers). Instead an entry’s ID is an unsigned 64 bit integer (see Figure 4.10) with the 32 most significant bits holding the ID of the journal, followed by 16 bits representing the block ID followed by 16 bit storing the position of the entry within the block. This representation of an ID does not only guarantees uniqueness within the log but also translates to the exact memory location of an entry.
Since a journal must have an ID larger than one and so does a block storing data, the smallest possible ID an entry can have is 0x100010000 (in hexadecimal representation). Thus, all succeeding entries must have larger IDs; otherwise they would be corrupt by default.

### 4.5 Communication

The communication between two servers is a connection-oriented communication using the TCP protocol. Each server has two connections open to every server in the cluster, one for receiving messages and one for sending messages. For each connection, a single thread is responsible for maintaining the connection and receiving or sending messages, depending on if it is an outgoing or incoming connection.

This design allows for simultaneous communication between servers in both directions. It increases the use of resources but eliminates the need of synchronization.

The implementation of a sender and receiver are very similar (as can be seen in figure 4.11). Both the sending and the receiving thread will try to reestablish its connection indefinitely, if the connection is lost. A lost connection can be caused by a timeout or a failure on either the receiving side or sending side.

![sender and receiver diagram](image)

**Figure 4.11:** The sender and receiver are very similar in their implementation. Both will reestablish a lost connection and keep working until the thread is terminated.

Each sending thread will have a unique queue associated with them (see section 4.6). When a connection is established the sending thread will try to dequeue a message, the dequeueing call is blocking, meaning that the thread will wait for a message to be enqueued if the queue is empty. When a message has been dequeued, the sender sends
the RPC to its destination. No matter if the send succeeds or fails the thread will delete the RPC and free its memory. A message is thus never resent, this is because it is up to the Raft algorithm and the Consensus Module (section 4.3) to decide if a message should be resent or not. Resending a message would violate the design choice to keep all logic in the same module (see beginning of Chapter 4).

The receiving thread, on the other hand, starts by trying to receive a message when a connection is established. If a message is received, the thread enqueues the message and waits for a new one, otherwise, if no message is received the receiving thread evaluates if the connection has been lost or not and acts accordingly. Contrary to the sending thread, the receiving thread does not have a queue reserved for it, but instead the queue is shared among all receiving threads.

4.6 Queues

Queues are a vital part of the system. Queues are used to separate the individual parts of the system. This simplifies synchronization and eases the inter-process communication.

All queues are implemented as First-In-First-Out pointer queues. This means that the queue is only holding a reference to the object placed in the queue and not the object itself. All queues have fixed size, meaning that the maximum number of elements the queue can hold is predetermined, making it impossible to enqueue elements when the queue is full. This works as a sanity check. If a queue is full when enqueueing, the part of the system responsible for reading the queue is malfunctioning. Since correctness is crucial to the system, a malfunctioning part could be devastating which is why a server shuts down if it detects that a queue is full.

All queues are implemented using mutex locks, therefore it is only possible to manipulate a queue while holding the lock associated with it. Queues are the only part of the system requiring synchronization.

Queues are used for communication between modules in the system. There are two types of queues, queues handling incoming communication and queues handling outgoing communication.

4.6.1 In-Queues

In-Queues are queues holding all incoming communication of a server. There may only be one In-Queue on every server, and the part of the system responsible for reading this
queue is the Consensus Module (section 4.3). The size of the In-Queue is equal to the number of servers in the cluster, making sure that the server will not shut down even if all neighbors sent messages at the same time.

The In-Queue is the only interface to the Consensus Module, simplifying the implementation of the Consensus Module.

### 4.6.2 Out-Queues

A server has one Out-Queues for every neighbor. Only the Consensus Module is allowed to enqueue elements to the Out-Queues and only the Sender Module associated with the queue is allowed to dequeue elements.

Out-Queues have a fixed size of one, this gives fast feedback to the Consensus Module if something is wrong with the Sender Module. This also eliminates the risk of sending a new RPC without receiving a response of the previous one.

### 4.7 Timer

The original Raft implementation uses two timers, one leader specific timer for sending Heartbeats and another running on all servers for election timeouts. The two timers had the following relationship:

\[
\text{Heartbeat Timeout} \ll \text{Election Timeout}
\]

Meaning that the Heartbeat timer would fire multiple times before the Election timer would fire.

In the implementation of this project, this relationship is exploited to implement only the Heartbeat timer and letting the Election timer be a function of it. The following formula clarifies the relationship between the two timers:

\[
\text{Election Timeout} = n \times \text{Heartbeat Timeout}
\]

Where \( n > 0 \) is a random number acting as a threshold, making the Election Timer fire after \( n \) Heartbeat timeouts. The timers are implemented the same way, no matter if in leader, candidate or follower state.
When the timer fires it enqueues a timeout message to the In-Queue of the server (see section 4.6.1). If the server is leader, it will act on the timeout message by sending out heartbeats to all of its followers. If the server does not receive responses from a majority of its followers before a random number of timeouts are triggered the leader will assume it lost its leadership and revert to follower state. This is equivalent to the Election timeout in the original Raft design.

If on the other hand, the server is not in a leader state, each timeout message received will increase a counter until it reaches a randomly set threshold and an Election Timeout will be triggered.

### 4.8 Client

A client was developed to provide an easy way to append new entries when the system was up and running. Building a client was not prioritized, resulting in a very simple client made to test system behaviours that would otherwise be impossible to test.

Only the leader is able to handle RPCs from the client and the client is implemented to be static. Having a static client means it cannot dynamically change server it is connected to at run time. This results in having to restart the client with a connection to the new leader if the current leader crashes. This is unsuitable for a live system but developing a more advanced client falls outside the scope of this project.
Chapter 5

Evaluation

The final system has been rigorously tested to confirm the robustness and correctness of the algorithm. However, only one physical machine was provided for development and testing. Therefore all tests have been conducted on a single node running multiple instances of the system. Hence, the performance tests may only be evaluated in comparison to each other since network latency will be a much bigger factor on a multi-node test.

5.1 Test Environment

All tests were performed on a single machine running 64 bit Red Hat Enterprise Linux Server release 7.2 (Maipo)[11] with two quad core Intel Xeon Processor E5-2637 v3s (30M Cache, 3.50 GHz)[12]

The single machine was provided by Svenska Spel for development and testing which limited the opportunities to stress test the system.

There are two main reasons for why the tests could not be executed on multiple nodes. The main reason is that only one machine was provided during the development and no more machines were available, mainly due to relocation to a new office. The second reason for not being able to test on multiple nodes was that the system utilizes libraries and source code native to Svenska Spel, which limits the distribution of the source code to machines hosted by Svenska Spel. For security reasons, no source code is allowed to be redistributed on machines outside Svenska Spel, which is why the system could not be tested on a multi-node cluster like the available at Uppsala University.
5.2 Performance

The performance tests were performed with multiple instances of the server running on the same physical machine. The time measured will therefore not be representable for a real world implementation since network latency in a multiple node implementation will be much higher. The test can however give an insight to how well the system scales and performs when resolving leader elections.

Under normal condition, a single leader with multiple followers, the performance of the system will vary with the network latency. A slow follower might fall behind but will not affect the overall performance of the system, as long as a majority of the cluster is working properly.

During leader election the system performance will however vary depending on the number of servers in the cluster and how fast an election timeout is triggered. Resolving leader elections in a timely manner is crucial to system performance and testing on a single machine will give some insight to how well the system resolves leader elections. A re-calibration of timeout thresholds might be necessary when running the system on multiple physical machines, since network latency will affect the performance.

5.2.1 Initial Leader Election

How fast the system can start up depends on the number of elections needed before a leader can be established. The fewer elections needed to establish a new leader, the better the overall performance of the system will be.

Figure 5.1 shows the number of elections needed with different configurations. The x-axis is showing the interval in milliseconds in which followers might timeout and start a new election. The y-axis shows the number of elections needed before a leader could be established. If two elections are started at the same time it will result in a split vote and a new election will be held.

The size of an interval clearly seems to be the major factor when reducing the number of elections needed.

5.3 Robustness

How well the system can handle server crashes gives an indication of how robust the system is. The potential crash of a server in follower or candidate state should not affect the system’s performance unless a majority of the cluster becomes unavailable.
But when a leader crashes a new leader must be elected before any new entries can be appended. If electing a new leader takes too much time, it will affect the performance of the system and in the end make the system seem stall.

To avoid unnecessary stall time new leaders have to be elected fast and with as few re-elects as possible.
Since the all tests were conducted on a single physical machine, the actual time to elect a new leader does not represent how the system would act on a distributed system. However, the test still showed an interesting characteristic. Figure 5.2 shows the system’s ability to elect a new leader after a leader crash. The interesting thing to note here is that it almost always takes longer time to elect a new leader on a cluster of 4 servers compared to a cluster of three or five. This can be explained by how large the majority has to be for a candidate to win the election.

In a cluster of three or five servers the majority makes up for 66% and 60% respectively. This is considerable lower than the majority in a cluster of four servers, where 75% is needed. This implies that elections in cluster consisting of four nodes would more often result in split votes, which aligns well with the results of this test.

### 5.4 Correctness

Correctness is the most important subject when it comes to evaluating the Raft algorithm and this implementation. For the algorithm to be correct it may under no circumstances allow two servers to commit conflicting information to their logs. Conflicting logs resulting in faulty response to the end user would be devastating for Svenska Spel as a business, therefore this issue cannot be stressed enough.

The following subsections highlights three different scenarios that were tested and might occur during runtime. The squares in figure 5.3 through figure 5.5 represent entries in the log. The numbers in each square tell in what term the entry was appended to the leader log.

The following test cases can occur when the leader or one or more of the followers crashes. In all tests the servers start in follower state and the outcome of the test will therefore be dependent on which of the three servers is elected leader.

#### 5.4.1 Catch Up

There may occur situations where one or more followers are lacking committed entries, those followers have to catch up with the leader by appending the missing entries.

In figure 5.3 server 3 (S3) wakes up after a crash and has not appended the last committed entry. Server 3 cannot be elected leader since neither server 1 (S1) nor server 2 (S2) would grant a vote request from server 3. Server 3 will receive an AppendEntriesRPC with the missing entry when connection to the leader is established.
The outcome of this test showed a correct behavior, where the leader always forced followers to replicate its log.

5.4.2 Uncommitted Entries

Logs may contain uncommitted entries if a leader crashes before it is able to replicate the entry on majority of its followers. Depending on the outcome of the leader election following a crash, the uncommitted entry may be committed on all servers or omitted from all logs where it exists.

Figure 5.4 shows the test of a scenario where server 2 and server 3 crashed before server 1, who was the leader, had a chance to replicate the entry in term 2 to the other servers. Server 1 returns to follower state before server 2 and server 3 wake up. The entry in term two is not committed and the resulting log may therefore differ depending on which of the three servers is elected leader. If server 1 becomes leader the last entry will be replicated to all logs. If on the other hand either server 2 or server 3 is elected leader the last entry in server 1 will be omitted.

Both cases were replicated in tests and the outcome was as expected.
5.4.3 Conflicting Entries

Logs may contain conflicting entries if new leaders keep crashing before they can replicate their log on the followers.

Figure 5.5 shows the test of an unlikely scenario where server 1 and server 2 both have been elected leader but crashed before they could replicate their log to the other servers, resulting in a situation of conflicting entries. Both server 1 and server 2 contain uncommitted entries (only entry of term 1 is committed). The outcome of this situation depends on which one of server 1 and server 2 is elected leader, note that server 3 (S3) cannot be elected leader since neither server 1 nor server 2 will grant a vote request from server 3.

When either of server 1 (S1) or server 2 (S2) becomes leader, it will force the other server to delete all conflicting entries and instead append the entries of the leader log.

<table>
<thead>
<tr>
<th></th>
<th>Initial</th>
<th>Final with S1 as leader</th>
<th>Final with S2 as leader</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>1 2 4</td>
<td>1 2 4</td>
<td>1 3 3</td>
</tr>
<tr>
<td>S2</td>
<td>1 3 3</td>
<td>1 2 4</td>
<td>1 3 3</td>
</tr>
<tr>
<td>S3</td>
<td>1</td>
<td>1 2 4</td>
<td>1 3 3</td>
</tr>
</tbody>
</table>

Figure 5.5: Server 1 (S1) and server 2 (S2) contain conflicting entries. The final log will depend on the outcome of the leader election.

Results from the tests aligned with the expected output.
Chapter 6

Related Work

Fault tolerant distributed systems have been of interest for more than thirty years. Many different approaches for creating fault tolerant systems been developed, with Raft as the first one focusing on an understandable algorithm for reaching consensus. Even though Raft has been the main inspiration for this thesis, other algorithm and approaches to the problem have affected the end result.

6.1 Paxos

The original proposal of the Paxos algorithm by Lamport [13] was inspired by the parliament system of the island Paxos in ancient Greece. The legislators of Paxos did not want to spend all their time in the parliament and could therefore come and leave as they wished. In order to reach consensus on a proposed decree each legislator had a ledger where he recorded each proposed decree. A decree could only be accepted if a majority of the legislators were present in the parliament during a period of time. This inspired the author to come up with a consensus algorithm for fault-tolerant systems.

Ever since the Paxos algorithm was first published it has been the state-of-the-art approach when it comes to consensus algorithms. Paxos is therefore widely spread and has been the inspiration of many implementations. Despite that, it is considered to be difficult to understand and ill suited for real world implementations. Even simplified explanations are hard to comprehend. [8]

Paxos allows a machine to take one or more of three different roles: proposer, acceptor, and learner. A proposer proposes a value by sending it to a set of acceptors. An acceptor is allowed to accept more than one proposal; assigning a natural number to it identifies proposals. A proposal can only be accepted as value when a majority of the acceptors
has accepted it. When a majority of the acceptors have agreed on a proposed value the learners can take action and store the value.

The Paxos algorithm is designed with a symmetric approach, which means that there is no designated leader (see section 2.4.1). All machines have equal roles and in order to reach consensus, each machine must communicate with every machine in the system. In a real world implementation one proposer is distinguished to be the leader and is the only one that is allowed to make proposals [5].

6.2 Viewstamped Replication

Viewstamped Replication is a replication algorithm to provide highly available services in a distributed system. It was developed in the late 1980’s around the same time, but unknowingly about Paxos. Unlike Paxos it is designed as replication protocol instead of a consensus protocol. [14]

Viewstamped Replication is designed for highly available services, which it achieves by replication on a distributed system. The principle is that one server is acting as a primary performing all computation and all the other servers are acting as backups. The primary informs all backups of what computations it has done so in the event of a primary crash a backup server can take over if necessary.

Viewstamped Replication assumes that a server that is running is working correctly without causing any corrupt data. Thus it does only handle non-byzantine failures. [15]

6.3 ZooKeeper

ZooKeeper is an example of a replicated state machine. It was developed to provide a service for maintaining configurations and coordinating processes on a distributed system.

ZooKeeper works by providing an interface through clients to a cluster of ZooKeeper nodes called znodes. One of the znodes is elected leader while the rest remains followers. A client interacting with the cluster may connect and send requests to any of the znodes, leader as well as follower. Servers will handle requests and respond to clients locally, unless the request is a write request, which results in a state change. These state changing requests are forwarded to the leader as part of an agreement protocol. It is the leaders responsibility to execute the request and broadcast the resulting state change to
all its followers. The follower originally issuing the request may respond to the client after transferring to the state proposed by the leader.

ZooKeeper is very well established, used in various contexts and by major actors such as Yahoo! among many [16].

6.4 ARC: Analysis of Raft Consensus

ARC was the first published assessment of the Raft protocol and was actually published before the Raft paper was published for peer review. After the publication of the ARC paper changes were made to the Raft protocol, probably as an effect of the ARC paper.

ARC provides a more extensive explanation of the algorithm and implementation compared to the original Raft paper. The ARC author was able to replicate the results produced by the original Raft implementation.

The authors also suggested some modification to the algorithm to enhance performance. Some of the modifications were later adopted as part of the Raft algorithm [17].

The list below lists a few of the modification recommended by the ARC author:

- Separation of follower and candidate timeouts. A faster timeout for candidates reduce the time it takes to elect a new leader.
- Read Commands, a command whose execution guarantees not to change the state machine does not have to be replicated across all nodes.
- Leader discovery protocol, a discovery protocol for a client to find a leader instead of randomly connecting to servers until the leader is found. A leader discovery protocol was later adapted to the Raft algorithm.

6.5 Other Raft Implementations

There are numerous different Raft implementations available on the Internet. Most of them are still under development or limited in their implementation. LogCabin [18] developed by Diego Ongaro is one of the more extensive implementations.

A short list of different Raft implementations is available on Raft’s official website. [19]
Chapter 7

Conclusion

The objectives of this thesis project, to implement, test and evaluate the correctness of the Raft algorithm are reached. Raft proved to be easy to understand and implement, and is an interesting algorithm with great potential for log replication.

The resulting implementation shows very promising results. The separation of the system into modules is an appealing design for further implementation and testing. The Raft logic seems to be implemented correct resulting in a correctly behavior of the system in all test cases. Correctness of the log can with most certainties be guaranteed when no unexpected behavior has been present. Only valid leaders are elected and logs replicated as expected.

The system remains operating in correct manner for as long as a majority of the cluster is available. The system proves to be \textit{t fault tolerant} in a system of \(2t + 1\) nodes.

The system also proves to be robust and can handle failures of multiple nodes in different states without any erroneous behavior. A new leader was always elected in a timely manner and split votes could be resolved quickly.

The performance tests looks promising, however the system has to be more rigorously tested on multiple servers before any conclusions can be drawn.

7.1 Future Work

The outcome of this thesis work has met the expectations. Plans for further development of this system design are in progress.

The main concerns for further development are listed below without any internal ordering.
• Dynamic Client, improve the client to be able to handle leader crashes.

• Configuration changes, implement dynamic configuration changes to ease configuration changes.

• Log, update the implementation to use a replica of the existing log at Svenska Spel.

• Communicate using SCTP protocol instead of TCP protocol to improve robustness.

The main priority is to test the system on multiple servers to confirm the expected behavior and calibrate the configurations to maximize performance.
Bibliography


