Implementing a Raft based Consensus Algorithm into the Troupe Programming Language as a Library

Mikkel Ugilt, 202106950
Victor Ask Justesen, 202107105

Bachelor Report (15 ECTS) in Computer Science
Advisor: Aslan Askarov
Department of Computer Science, Aarhus University
June 2024

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

Consensus protocols are used to achieve heightened availability and reliability by running a function on a distributed cluster of nodes. The Raft consensus algorithm works by electing a responsible node as leader, which sends out periodical heartbeats and updates to its followers. To maintain consensus, these updates will be applied when a majority of the cluster has received them.

In this thesis we will research and analyze Raft with the purpose of implementing it as a library in Troupe. Starting with an implementation of Raft made to support a basic key-value store, we will expand this further to support arbitrary state machines and handle the details of client-to-cluster communication.

Creating an implementation of Raft, that remains quick and available, will require log compaction. We will use snapshotting to achieve this by compacting all applied log entries into a single representative snapshot, which allows us to discard previous entries.

We restrict the state machines to be written in a modified version of continuation-passing style, which is used to guarantee completion of computation. This guarantee only holds under a set of rules, that restrict the available amount of functions to the state machine.

Mikkel Ugilt and Victor Ask Justesen,
Aarhus Universitet, June 2024.

Contents

Chapter 1

Introduction

When considering the threat model for distributed applications, a common problem is the application host crashing or becoming temporarily unreachable. It raises the question: How can you make an application highly available between hosts while preserving the responsiveness and consensus? Consensus protocols aim to solve this. They allow an application to continue service even with the failures of some of its hosts.

The Raft[1] consensus algorithm provides a mechanism to ensure that a state machine remains consistent on all nodes in a cluster. It uses leader elections in an asynchronous environment and depends on randomized timeouts to elect a new leader, if the old one becomes slow or unreachable.

Troupe[2] is a functional programming language with features focusing on concurrent and distributed programming. Key aspects include the processes, which are created using `spawn`, being able to be created either locally or distributed. Processes can send messages, which can be received in another process' mailbox, implementing communication and interaction between threads.

In this thesis we describe the considerations and design decisions when creating an implementation of a Raft library in Troupe. We start with an implementation of Raft supporting a key-value store, specifically to get Raft functioning in a concrete case before working towards an abstraction.

When abstracting we describe our approach to encapsulate client-to-cluster communication. This approach aims to make message sending behave similarly to the existing way that messages are sent to a process in Troupe, by implementing a dialer. It also eliminates duplicate messages and resending messages to a leader if they do not arrive.

Having an abstraction of Raft, we describe the process of turning it into a library for Troupe. We consider the design of the Troupe language and the properties of Troupe functions. This is to derive an extension of Raft that guarantees completion of computations and avoiding duplicate side effects using continuation-passing style.

Finally, we discuss what future work the project contains, specifically towards making the library compatible with more Troupe functions and the steps towards embedding the library into the runtime. Here we also compare to a different threat model and how one can use Nysiad[3] on top of Raft to create a Byzantine Fault tolerant system[4].

Chapter 2

Review of Literature

In this chapter, we will summarize and discuss the literature used to support the implementation of Raft[5] as a Troupe[2] library. At first we will provide the required background into understanding Raft, explaining the concepts of consensus algorithms, state machines and replicated state machines, and their relevance.

We will then delve deeper into Raft, its purpose, and the functionalities that we will implement throughout the project. We will also touch upon Kubernetes[6], etcd[7], and CockroachDB[8] as some practical use cases of Raft. Lastly, we will discuss *Nysiad*[3] as a method to convert a fault tolerant system to a Byzantine Fault tolerant system[4].

2.1 Replicated State Machines and Consensus Algorithms

A state machine[4] is a program containing a set of inputs and outputs, as well as a set of states. It works by receiving an input to its current state, transition to another state, and give an output. Since a state machine is deterministic, it will always go into the same state, given the same input.

The *state machine approach*[4] is an approach to replicate a state machine. It is key in practical systems where we strive to maintain stable server uptime, since a single host can become unavailable. Replicated state machines solve reliability issues, but replace them with consensus issues. This is a question of how we ensure all hosts in a cluster maintain the same state on their respective state machines, ensuring *agreement* and *order*[4]. Agreement requires that all replicated state machines in the cluster agree on the state. Order requires that all inputs received are in the same ordering on all state machines in the cluster. Consensus algorithms, such as Raft, must follow these properties.

F. B. Schneider[4] discusses two distinct fault tolerances: *Byzantine Failures* and *Fail-stop Failures*. Byzantine Failures are servers acting maliciously, and exhibiting arbitrary behavior. Fail-stop Failures describe faulty nodes that halt to avoid the state machine entering a malicious state such that other nodes realize it is halted.

2.2 The Raft Consensus Algorithm

The Raft Consensus Algorithm[5] was created to make a more understandable algorithm for consensus over Paxos[5]. The paper states: "Unfortunately, Paxos is quite difficult to understand, in spite of numerous attempts to make it more approachable. Furthermore, its architecture requires complex changes to support practical systems"[5, p. 1].

With its main goal being "understandability"[5], Raft strives to be easier to learn for a larger audience than that of Paxos while also being applicable to real world practical applications. Raft is also developed with the idea of divide-and-conquer in mind, dividing problems into subproblems.

The Raft consensus algorithm works by way of leader election. A node will through voting be elected leader of the current term. When a message is received at the cluster, the leader is in charge of replicating that message, and notifying the cluster when it is deemed safe to commit that message.

Each server in the algorithm is at all times in one of three states: *Leader, follower or candidate* [5]. When the cluster is without a leader, we need a way to elect a new leader. Because of the need for these elections, the algorithm divides itself into terms, starting at term 1 with the first leader.

2.2.1 Elections

For the purpose of beginning an election, all followers are equipped with random timeouts. The timeout is reset when a heartbeat is received from a leader. The heartbeat serves as a way to send messages, as well as ensure that the leader is still alive. If the timeout is triggered, the follower starts a new term, where it begins an election and nominates itself as a candidate. The followers vote for the candidate if the candidate's term is later than the follower's current term, and if the candidate has a log that is as up-to-date or newer than the follower's log. Followers may only vote for one candidate per term. A follower may resend votes to the same candidate in the same term to ensure they are not lost during network transmission. A candidate wins an election, if it receives votes from the majority of the nodes. If the candidate wins, it serves as the leader for the rest of the term.

The candidates are equipped with random timeouts to avoid *split votes*[5]. This is where two candidates are formed at once and manage to gather enough votes such that none of them can win the election. If this occurs, the candidates will begin a new election in a new term after another random timeout, possibly getting out of the split election.

2.2.2 Log Replication

When a node is elected leader, it handles communication between clients and the cluster. The leader keeps a log of all changes to the state machine. Leaders send *AppendEntries*-messages[5] as heartbeats to its followers, catching followers up on their log, their term, and the *commitIndex*[5], as well as ensuring them that the leader is still alive.

Each node in a cluster keeps its own log, in which it keeps note of each log entry received, its index in the log, and during which term the log entry was created.

When a message is sent to the leader, the leader appends a log entry to its own log and sends the entry to the followers, along with its term, ID, previous log index and term, and *commitIndex*. The followers check whether they have received the previous message and if their terms match. If this is fulfilled, the followers increment their log index, potentially update their term to match the leader's, and append the entries.

When a follower appends a message, it replies to the leader with a boolean message, denoting whether or not the message was appended. The leader waits until it knows that the majority of the cluster have appended the message (including itself). It will then commit the message and update its commit index. Whenever a follower receives an *AppendEntries*, it changes its own *commitIndex* to match the one attached. The followers know that all messages up to the *commitIndex* are safe to commit, since a *commitIndex* will always grow monotonically. This works because a log index within the same term will always contain the same log entry. This property is referred to as the *Log Matching property* [5].

When a message is committed, it is deemed safe by the leader such that all state machines are able to apply the entry, since it will never be rolled back. This follows from *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"[5, p. 5]

Now lets consider the case where a follower does not accept an *AppendEntries*. A follower replies with *false*, if any of the following conditions hold, and the leader will act correspondingly:

- The follower is in a later term. Only leaders can increment the term, so there exists another newer leader. Upon the leader receiving this, it demotes itself to a follower, awaiting the heartbeat of the newer leader.
- It has not seen the previous message or the previous message does not match the term of the leader's previous message (these must be different as a consequence of *Log Matching property*). The leader recognizes this, and sends the previous log entries.

2.2.3 Snapshotting

Raft also denotes the notion of a *snapshot*[5], which we can utilize to optimize the performance of our Raft implementation. Snapshotting is used to compact log entries, as the log grows bigger when the Raft algorithm stays up for longer. It works by saving the state of the state machine and discarding all committed log entries, as they have already been applied. No non-committed entries are discarded, as they have not been applied to the state machine yet, and have not affected the state. When a follower asks for log entries from before the snapshot was taken, the leader will instead send the snapshot.

2.3 Practical Uses

In this section, we describe a few ways that Raft is used in practice. We will look at CockroachDB and Kubernetes, which both uses Raft in different ways.

CockroachDB

CockroachDB[8] utilizes Raft in order to gain consensus amongst the data centers where databases are located, and keeps its log serializable such that a temporarily offline or slow server will always be able to catch up with the others.

CockroachDB adds a new role, the *non-voting replica*[8] for its multi-region reads. Also known as read-only replicas, they act as followers that are unable to vote for a candidate whenever their timeout expires. However, it may replicate its log and receive leader *AppendEntries* exactly like any other node, so its database can be contacted in the same way. The great benefit of not voting during elections is, that the non-voting nodes does not have to send anything, it just updates its leader when it receives a valid *AppendEntries* from the new leader. As a result, while the cluster is more dependent on the voting nodes, it benefits from the larger coverage.

Kubernetes and Etcd

A non-database use of Raft is through Kubernetes[6] to create consistent configurations using etcd[7], which is a distributed key-value store built on Raft. Etcd runs a Raft cluster of five dedicated nodes by default. Kubernetes is a popular "container orchestration engine"[6], that facilitates deployment, as well as scaling and management of containerized cloud applications. To simplify, Kubernetes uses the notion of a manager and nodes, whereas applications can be deployed onto the nodes, into pods. These pods are given a web address where they can be contacted. One may wish to scale an application, to ensure higher availability of the application. Kubernetes can accomodate this, by giving a unified address to a cloud application deployed on several pods, even on entirely different nodes. Kubernetes nodes have direct contact to the Kubernetes API Server.

This is where etcd can be found. Etcd stores the configurations and properties of the Kubernetes API Server, as well as the cluster holding the cloud applications itself. It does this with a distributed key-value store, ensuring a more reliable Kubernetes Configuration.[6]

2.4 Nysiad

A fault tolerant system is a great solution for achieving availability on a state machine. In a non-trusted environment however, it is vital to look into how to tolerate Byzantine corruptions. *Nysiad*[3] is a system that provides a translation from a fault tolerant system, to a system that tolerates Byzantine failures.

A system can be broken down to a communication graph of nodes (V, E), with edges E and vertices V. The hosts are represented as vertices and their connection for communication as edges. Given an arbitrary amount of possible Byzantine failures

t, Nysiad creates a directed graph (V', E'), such that there are $3t+1$ *guards* for each host, and for each neighbor, they have $2t+1$ *guards* in common. A node is always a guard for itself, and neighbors in V, as a result neighbors will always be guards for each other. A host replicates its state machine on itself and its guards using an asynchronous reliable broadcast protocol, *OARcast*[3]. *OARcast* is specifically useful as it maintains *order* and *agreement*[4] over all non-corrupt parties, of which there are t.

Note that the notion of *state machines* in the context of Nysiad considers an entire Raft node as a host, and by extension, the state machine is the entire protocol run on that node.

Nysiad introduces a couple of extra mechanisms required for completely securing the fault tolerant system. The first one is *attestation*[3]. This mechanism ensures that the host does not forge any messages, that the state machine would not produce. Take for example, a follower f_j may be maliciously trying to send messages to f_i claiming to be a leader. f_i needs to collect $t+1$ attestations from its guards that the message is valid. This includes itself and f_j. A guard may generate these attestations, as they keep their own replica of the state machine of f_j, with which they can deduce if the state change is warranted.

Another mechanism is *credits*[3]. Credits are made to ensure that all previous messages are delivered before the next one. Credits are given by the guards and requires the host to send all messages, or no messages at all.

A Raft cluster is not a static graph configuration – During a regular term, there is a connection between the leader and all followers. When a leader fails, one of the followers will begin an election, which will result in a new leader. Now all the followers will be connected to the new leader. To control varying configuration, Nysiad introduces the *Olympus*[3]. The Olympus divides the protocol into epochs, that are switched when a host h_i tries to send a message to a host h_j for the first time (changing the configuration of communication edges in E). h_i gathers proof from its guards that it should be able to send a message to h_j, and upon receiving $n_i - t$ proof, the epoch is changed. n_i is the number of guards of host h_i.

Chapter 3

Implementing a Raft based Key-Value Store

In this chapter we will describe our implementation of a key-value store serving as a state machine, to build an implementation of Raft around. After this we will describe the path we took to implement the Raft algorithm. Lastly we will couple our key-value store together with Raft.

3.1 Implementing a Key-value Store

Since Troupe is a purely functional programming language, we had to consider how we implemented the key-value store. In theory, we wanted to create a map between keys and values, that could be updated and queried by a client. It should be replicated on multiple servers, such that it is highly available to every client.

In order to implement functional mutability, in a language that does not directly support it, we make use of recursive functions. We implement the store as a list of records to take advantage of named fields and to make the code easily understandable. It turned out to be a powerful mechanism also used when implementing and abstracting Raft.

The `get`-function finds the element in the list corresponding to the key and returns the value. The `set` either appends a new entry into the key-value store, or updates the value, such that the same key now corresponds to a new value.

3.2 Implementing Raft

In this section we will go through our implementation of Raft in Troupe, giving a detailed breakdown of a local implementation, including key differences encountered when creating the distributed implementation. Nodes are initialized as dormant followers, that wait until all nodes have started the algorithm before turning into a follower. This ensures that some nodes do not start the algorithm before everyone is ready.

3.2.1 Local Implementation

We have two areas of implementation: Local and distributed. To ease ourselves into the algorithm, we started with the local implementation.

Follower

Each node in the system keeps its own log, which is updated during the runtime of the algorithm. The state machines are created on concurrent processes of each node. To apply the committed entries of the log, the node sends the message to the state machine, ensuring loose coupling between Raft and the state machine.

When a node enters the follower state, it begins an election timeout, that, upon triggering, promotes the follower to a candidate and begins an election. The time of the election timeout E_t is derived as such:

$$E_t = E_l + x(E_u - E_l)$$

where x is a uniformly random variable over the interval $[a,b]$, and E_u and E_l are respectively an upper- and lower bound of the election timeout.
The follower can do any of the following actions:

- **Election timeout**: If the follower has not received an *AppendEntries* since the timeout was started, the follower will begin an election.
- **Vote**: If a candidate c requests a vote from a follower f, they will respond with a YES-vote if the following conditions are fulfilled:
  - c's term is strictly higher than f's, unless it already voted for c.
  - c's log is at least as up to date as f's log.
  - f has not already voted for another candidate in the same term.

  The first condition ensures that the follower does not vote for a candidate that is behind it. The second condition ensures that the logs are the same. We know this due to the *Log Matching property*[5].

  If all conditions are met, f sends a YES-vote to c, and notes the candidate voted for. Otherwise, f will respond with a NO-vote. Either way, for f's term t_f and c's term t_c, the follower sets its term to be:

  $$t_f = \max(t_f, t_c)$$

  A follower will only send YES-votes to the same candidate in the same term. Another candidate in a later term, however, *can* receive YES-votes if conditions are fulfilled.

- **Append entries to log**: When a list of entries is received from the leader, along with the leader's term, *commitIndex*, the term of the latest log entry before the list of entries, and the log index before the list of entries, the follower will do a number of checks before choosing the appropriate response:

  Accept if:

  - The leader's term is greater-than or equal to the follower's term. The follower will then update its term to match the leader's term.
  - The latest message index of the follower's log is the same as previous entries before the entries sent by the leader. This means that the follower has received all messages up until the current entries being appended.
  - The message term of the follower's latest log is the same as the term of the latest log entry before the entries. Because of the *Log Matching property*[5], this ensures that the latest message matches.

  Reject otherwise.

  If the node accepts, it will roll back its own log until the leader's previous entries of the messages and add the new entries to its log. The *commitIndex* of the follower is also updated to match the leader.

  The follower responds, letting the leader know whether it rejected or accepted the message. We return to this mechanism when discussing the leader implementation.

  If the *commitIndex* is updated, the follower applies all messages on the state machine up until the *commitIndex*. If the follower has not communicated with the leader before, it also notes down the leader's ID, such that any process contacting the cluster, can get the leader's ID from the follower. Finally, the follower resets its election timeout.

Candidate

When a follower triggers its election timeout, it increments its term and begins a new election. Afterwards, it becomes a candidate with the goal of becoming a leader.

The candidate starts by sending a request for votes to all members of the cluster. This request contains the following information:

- **Term**: The term of the candidate.
- **Log Index**: Its current log index.
- **Latest Log Term**: The term of the latest log entry.
- **Candidate ID**: A unique way to identify this node. In our concrete implementation, this is the candidate's Troupe *process-id*.

The candidate will then await r unique YES-votes where

$$r = \lceil n/2 \rceil$$

Here n is the number of nodes in the cluster. *Etcd*[7] recommends an odd-sized cluster, since adding a node to an odd-sized cluster gives the same voting constraints and fault tolerances, while increasing messages sent in the system. A candidate can receive multiple votes from the same follower. This is the follower resending a vote to ensure that it has been transmitted.

After beginning its election, the candidate can perform the following tasks:

- **Elect to leader**: When a candidate receives a YES-vote, it checks if it has gathered r unique YES-votes and, if so, promotes itself to leader. When sending its first heartbeat, the followers will recognize that the candidate has become their new leader.
- **Demote to follower**: If a candidate receives a NO-vote from a follower in a later term, it can be inferred that there exists another candidate or leader in a later term. This demotes the candidate to a follower. A candidate in term t_c will also demote to a follower if it receives an *AppendEntries* from a leader in term $t_l \geq t_c$.
- **Vote**: If a candidate receives a vote request from another candidate in a later term, it will demote itself to a follower and send a YES-vote if all other conditions hold.

Leader

When a candidate is elected, it turns into a leader. It starts a new `leader_info`, which is the *volatile state*[5] the leader keeps. It holds the following information:

- **MatchIndex**: A list of the highest index matched on each node.
  This is used to determine when a log is safe to commit. This is initialized to be 0, and will be updated as we confirm other nodes' log index.
- **NextIndex**: A list tracking the next index for each follower's log.
  This information is used by the leader to determine what to send to each follower. We start by initializing it to be the node's own log index plus one.

The leader serves as the contact between Raft and the outside world. When a client sends a message to the leader, it is appended to the leader's log. The leader sends periodical messages, *AppendEntries*[5], to replicate the message. For each follower f with *nextIndex*[5] denoted n_f and the leader with a log index i_l, the *AppendEntries* contains the following:

- **Entries**: A list of entries from index n_f to i_l in the log. This may be empty, but *AppendEntries* should still be sent to ensure that f does not begin an election.
- **Leader ID**: A unique identification of the leader.
- **Term**: The leader's term.
- **Latest Log Index**: The index which the leader believes is the follower's latest index, $n_f - 1$.

- **Latest Log Term**: The term of the entry at index $n_f - 1$.
- **CommitIndex**: Determined by taking the entries that have been acknowledged by a majority of followers, the *commitIndex* is sent along to let the followers know what entries in their log are safe to commit.

A follower will respond with either an acknowledgement or a rejection. The leader updates the *matchIndex* and *nextIndex* for the follower, as well as the *commitIndex* and applies any newly committed messages, when receiving an acknowledgement. The leader updates the *commitIndex* based on the *matchIndex* of its followers, to determine when the majority of nodes have appended a message. If the follower rejects the message, there can be three different reasons:

- **The leader is behind**:
  At some point another follower won an election. If this is the case, the leader demotes to a follower. We know this is the case based on the terms that accompanies the reject.
- **The follower is missing part of the log**:
  This happens if the follower, for some reason, has missed some *AppendEntries*. The way we solve this is by having the follower send back what their next index is. The leader updates the *nextIndex* for the follower and send the missing logs, in the next *AppendEntries*.
- **The follower contains a different log**:
  This can occur if another previous leader crashes without replicating a message to all followers. The leader will send previous messages before the follower agrees, after which it sends the missing entries as described above.

3.2.2 Distributed

There are key differences when working with local nodes, all located on the same process, and distributed nodes, that are on entirely different processes. The main ones that we had to tackle were initialization and network delay.

In order to initialize local nodes, we can spawn concurrent processes from a master process – The approach with distributed nodes is similar. All nodes start in a dormant state, where we dial into them and spawn the algorithm on the nodes. With Troupe, this can be done using aliases such that a process is spawned with a specific alias.[2]

Increased network delay is a factor whenever you deal with distributed connections. Because of this, we have to be more wary about timeouts for elections on the followers and reelections on candidates. The larger transmission time needs a larger overall timeout, but the fluctuating nature of network traffic also means, that the randomly added delay needs to have a larger variance as well. While it might slow down the algorithm a bit, a longer election timeout prevents a much larger slowdown of log growth by avoiding repeating split elections.

In order to decide the timeout parameters, we can look towards the practical use of Raft. CockroachDB uses a 2000 ms lower bound, along with a randomized factor of $1 - 2$[8]. If we apply this in our implemented solution, this gives a timeout delay of

$= 2000 + 2000x$. The heartbeat delay needs to be tweaked as well to accommodate this. CockroachDB uses 500 ms *tick interval*[8]. This gives a leader *at least* four times its heartbeat timer to transmit the message to the followers. This tells us that the election timeout must be sufficiently large compared to the heartbeat interval in a distributed system.

3.3 Coupling Key-value Store with Raft

Now with an implementation of the Raft algorithm and a key-value store, we can combine the two. Our first problem is how to run Raft and the key-value store at the same time. We choose to run the key-value store in a parallel process on each of the individual nodes in the cluster. This way we can use Troupe's built-in message system to communicate between the two processes.

We then have to find a way to couple a message, received from a client, to the cluster and pass that along to the key-value store. To do this, we create two new endpoints on the leader and follower, **GET** and **SET**. These endpoints are the precursors to the hooks that will be introduced in the next chapter.

The endpoints work in the following way:

- **GET**: This message contains a key and a callback address. The leader of the cluster will forward this to its own key-value store. The key-value store will then handle the request.
- **SET**: This message contains a key and a value. The message will first be replicated among the nodes in the cluster. After this each node will pass the message along to its own key-value store.

We create the different endpoints to solve the issue of double sending. This issue arises when a state machine sends a message to a client. In a cluster configuration with many state machines running at the same time, if they all send a value to the client, the client will receive too many answers. This is a problem that can be tackled in many ways, and at this point in development, we choose this route. This approach gives us the ability to use all of Troupe without major hurdles, as the performance of Raft is largely disconnected from the state machine.

Chapter 4

Implementing Raft based Consensus For Functions With Messages

In this section we will describe how we abstracted over our ideas from the last chapter in order to run our implementation of the Raft algorithm with any arbitrary function. Additionally, we will describe an implementation of snapshotting as a method to save large transmission time, catching up nodes that are far behind, and compressing the log to use less memory.

4.1 Implementing Arbitrary Functions

When going from a specific implementation to a general use case, there are several ways of abstracting towards an arbitrary function. One path is using the `spawn` function to run the state machine on a separate process. This created the problem of coupling the Raft algorithm to the state machine during program execution.

4.1.1 Arbitrary State Machines

While working with the key-value store, we made the distinction between the **Set** and **Get** functions. When abstracting Raft to an arbitrary state machine, we are posed with a pertinent question; How do we handle functions that sends messages? There are different approaches to solving this problem, for example if we replicate and apply every message that we receive, we run into issues with **Get** functions as each Raft node will send a message which, depending on the cluster size, is not desirable.

The approach we use exposes specific hooks to the user. This removes the issue of multiple messages, while setting user requirements on how to program the state machine. We create the following hooks:

- **RAFT-UPDATE**: This hook is defined for updating the state but does not return a value to the client. In our key-value store, this is analogous to the **Set** message. These messages will be stored, replicated, and applied on each node.

- **RAFT-GET**: With a way to update the state, we also need a way to get the state. This hook functions by applying the message to the leader's state machine and not replicating the message. It is therefore crucial that the function that the hook uses does not change the state, otherwise the leader will get out of sync with the cluster.

With this approach, we are limiting the state machines in a way, that makes the implementation of the Raft algorithm more loosely coupled.

4.1.2 Dialing

Working with the Raft algorithm, the client only communicates with the leader of a cluster. In the case where the client contacts a follower, the follower will reply with the leader's ID. We want to make communication with a Raft cluster seamless. For this we have implemented the dialer. The dialer's responsibility is to serve as middleware that handles the specifics of sending a message to the cluster and resending it, if the message was sent to a follower.

The dialer also helps with timeouts. If a leader crashes, the cluster will elect a new leader. The client will not know this and might still send messages to an unresponsive computer. We therefore attach an acknowledgement when a leader receives a message. This acknowledgement is sent back to the dialer to confirm that the message has been received. The dialer is equipped with a timeout such that, if it does not receive an acknowledgement for a message, it resends it. If a message to a previously known leader takes too long to be acknowledged, it assumes that the leader is unresponsive, and we need to find the new leader. The dialer resolves this by sending a message to a random node in the cluster. We know that either we will contact the new leader, or the follower will inform our dialer of the new leader.

If the leader receives a message but the acknowledgement does not arrive, the dialer will timeout and resend the message. To avoid the same message getting replicated twice, we send a random string as a serial key with each message. When the leader receives a message, it checks that the serial key is different from the previous serial key. If the serial key matches the previous, the message is flagged as a duplicate message and ignored.

4.2 Snapshotting

Raft presents the idea of using a *snapshot*[5] of the currently applied log in order to compact the log and avoid an infinitely growing log. The way we tackle this is in the same vein as the previously mentioned abstraction over the key-value store. First, we discuss how to snapshot, afterwards we discuss when to snapshot.

4.2.1 How to Snapshot

For specific state machines, like a key-value store, the snapshot could in theory be created from analyzing the log. However, the same is not guaranteed when supporting arbitrary functions. We therefore opt for the following approach: We introduce

two new hooks; the **Snapshot-GetHook** and the **Snapshot-SetHook** and add these as parameters to our Raft function when setting up the state machine.

- **Snapshot-GetHook** contacts the state machine on the given node and asks for its current state. The state machine replies with its state.
- **Snapshot-SetHook** contacts the state machine on the given node along with a state and overrides the current state with it.

When a leader snapshots, it is important that the leader does not delete entries that are not applied, as these are not represented in the state. Because of this, we have created snapshots with the following properties:

- **Snapshot**: The state itself. This is what the state machine outputs from the **Snapshot-GetHook**.
- **Last Included Index**: The *commitIndex* of the log at the point where the snapshot was created.
- **Last Term**: The term of the latest entry that was snapshotted.

The snapshot is added as part of the log (seen in Figure 4.1). We modify communication between leaders and followers to account for this change. Additionally, we need to make sure the Raft nodes recognize that a snapshot with **Last Included Index** i and **Last Term** t, is the same as separate committed entries from index 0 to i, with the term of entry i being t.

If a leader's log index is l and has a snapshot with a **Last Included Index** i, then if a follower f falls behind such that its next index is $n_i \leq l$, the leader does one of the following

- *If* $n_i \leq i$: Send a snapshot to f, and update its *commitIndex* to i. When accepting, the follower sets its next index to $n_i' = i + 1$. If $n_i' \leq l$, the leader sends entries from n_i' to l. A follower will only reject a snapshot if the leader is no longer a leader, which means their term is behind.
- *If* $n_i > i$: Send entries from n_i' to l.

As a message has to be committed in order for it to be snapshotted, we will never encounter a situation where a leader will be demoted to a follower, and have a snapshot that is neither replicated as a snapshot nor in the log on a majority of the cluster. Additionally, a leader, receiving a snapshot from a leader in a future term, will immediately be demoted to a follower.

4.2.2 When to Snapshot

For an abstract Raft implementation, compatible with arbitrary functions, it is different for every state machine when it is most favorable to snapshot. This is a question of when catching up a follower, by sending all previous entries, becomes more taxing than sending the entire state of the state machine. While it may not always hold in

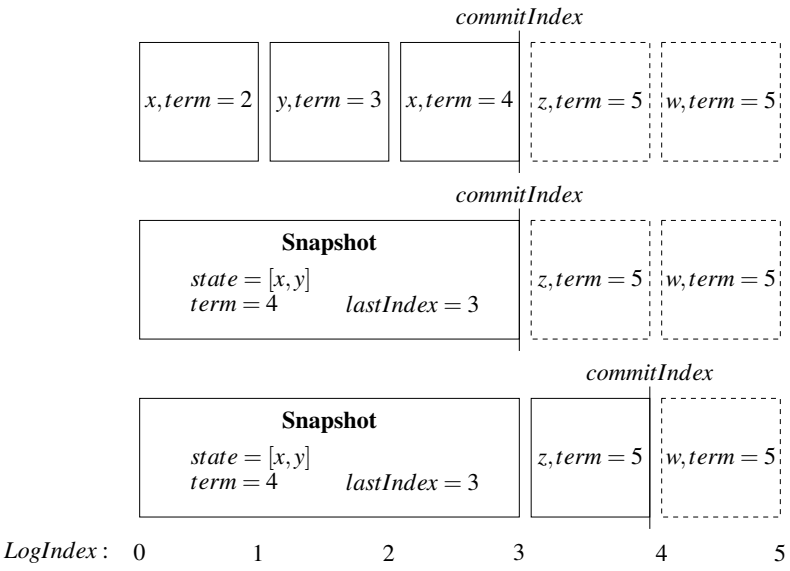

Figure 4.1: An example of how the snapshot procedure works.
From top to bottom: Boxes indicate log entries (with the wide box denoting a snapshot). Regular lined boxes are committed entries and dashed are uncommitted. The leader takes a snapshot of all committed (and applied) entries, and saves the state of the state machine. When new messages are committed, their index starts from the snapshot's last (included) index.

practice, we assume that a state is generally bigger and more taxing to send than a single log entry, and that snapshots increase computation time. This varies, and as such a hook is needed, such that the state machine can tell the Raft algorithm when to snapshot:

- **Snapshot-ConditionHook** contacts the state machine along with a representative *log summary*, and asks whether a snapshot should be performed on the log at this point. The state machine replies with a boolean.

With the introduction of this hook, the leader can contact its state machine, along with a *log summary* and ask if it is viable to snapshot.

We leave the log summary intentionally open, but for now we deem the following variables relevant:

- **Amount of log entries**: If the amount of stateful variables in the state machine are less than the amount of log entries, then there will be at least one overriding or irrelevant entry in the log that will be removed with a snapshot. Take for example a key-value store: The client can send n updates to key A, resulting in one stateful variable while there are n entries in the log.
- **Entries since last snapshot**: If a state machine is often valid for a snapshot, then we want to have a mechanism to avoid doing snapshots too frequently.

It will become clear when discussing the final library that we can relax this definition, but it gives an idea of how snapshot conditions can vary.

4.3 Coupling Everything Together

In this chapter, we have described the way we abstract over arbitrary functions on clusters using Raft to establish consensus. We arrive at the following functions for the local cluster:

$$\mathsf{raft\_spawn}(\mathsf{state\_machine}, n, \mathsf{hooks}) \rightarrow [\mathsf{node\ ids}]$$

and the distributed cluster:

$$\mathsf{raft\_spawn\_d}(\mathsf{state\_machine}, \mathsf{locations}, \mathsf{hooks}) \rightarrow [\mathsf{node\ ids}]$$

The state_machine denotes the state machine to be run in the Raft cluster[1]. hooks is the previously defined hooks:

- **RAFT-UPDATE** and **RAFT-GET** (Chapter 4.1.1)
- **Snapshot-GetHook** and **Snapshot-SetHook** (Chapter 4.2.1)
- **Snapshot-ConditionHook** (Chapter 4.2.2)

locations is the addresses where the cluster nodes will be spawned. In Troupe we use remote spawning to do this[2]. In a local environment these are all on the same process, so n is the amount of nodes to be spawned locally. The functions return a list of all the nodes' process-ids, such that a dialer can dial into the cluster.

Using the function:

$$\mathsf{raft\_dial}([\mathsf{node\_ids}], \mathsf{client\_id}) \rightarrow \mathsf{dialer\_id}$$

the programmer can dial into the cluster, to facilitate client-to-cluster communication. Messages from the programmer to the cluster are then sent to the dialer.

[1] In Troupe, this is any function.

[2] See the user-guide in Appendix F for more details on how this is done in practice

Chapter 5

Implementing a Raft Library into Troupe

In this section we will describe the process of turning our Raft implementation in Troupe into a library geared towards a future embedment.

Embedding a consensus protocol into a language like Troupe supports the programmer, by giving them a mechanism to run a function on a cluster, such that it maintains high availability, and is not depending on a single fallible host. To achieve this, we will analyze the way Troupe spawns functions, and look at what implications this holds. Furthermore, we will implement the library with the idea that Raft eventually will be a part of the Troupe language.

5.1 Analysis of Troupe Function Spawning

Spawning a function in Troupe equates to sending a unit function to a location, where the function will be run on that location. There exists two signatures of the `spawn`-function[2] as follows given their Troupe signature:

$$\text{spawn}(f) = \text{process\_id} \qquad \texttt{spawn fn => p\_id} \tag{5.1}$$

$$\text{spawn}(\text{alias}, f) = \text{process\_id} \qquad \texttt{spawn (alias, fn) => p\_id} \tag{5.2}$$

The main difference between (5.1) and (5.2) is the presence of the alias. The alias denotes a location of where the function will be run. In (5.1) Troupe spawns a function concurrently on the local machine. The process_id is used to send messages to a function. These messages can be received by using `receive`[2]. We want to create a similar function:

$$\text{spawn}([a_1, a_2, \ldots, a_n], f) = \text{cluster\_id} \qquad \texttt{spawn (aliases, fn) => c\_id} \tag{5.3}$$

Instead of spawning a function f on one remote machine, (5.3) creates a Raft cluster that runs f on the aliases provided in $[a_1, a_2, \ldots, a_n]$. Sending messages to the

`cluster_id` should implicitly spawn a dialer for the client sending the message, and handle client-to-cluster communication. As the scope of this project resides in making a library, we relax this variant a bit. We will discuss some properties of Troupe functions before returning to a final library.

5.2 Self-changing State Machines

Looking at Troupe, there is one gap in the library described: Concurrent functions spawned by the state machines. These functions can impact the state machine outside of any log based input from the Raft cluster. We call this *internal change* denoted $I(m)$ for message m, while the programmer-to-Raft communication is called *external input*, denoted $E(m)$ for message m.

5.2.1 Counter

With the key-value store in chapter 3, we had a state machine that was entirely driven by external input.

Assume we have implemented a state machine that monotonically increments a value at a fixed rate. This state machine raises two important questions: How do we replicate the internal changes and how do we ensure consistent logs amongst nodes?

5.2.2 Logging Internal Changes

For getting the internal changes, we introduce a new hook:

- **GetChanges** contacts the state machine and requests the internal changes since the last **GetChanges**.

We then change the heartbeat such that, at every heartbeat, the leader contacts the state machine with the **GetChanges**-hook and gets a list of internal state changes, which are added to the log. The changes in this list must be interpretable by the state machine itself, as we should be able to catch up a node using these messages.

This creates a new requirement for the state machine, as it now has to keep track of all internal changes and save them for the next **GetChanges**. We also have to reorganize the log to accommodate this change.

We add a value *internalChanges* to the log, which ensures that committed internal changes are not applied on the state machine, as they have already affected the state. *internalChanges* is decremented after each internal change is committed.

Order of Operations and Replication

To maintain the order of operations on the log, we need to insert internal changes before any non-committed external change, since the internal changes have already affected the state (see Figure 5.1). The term of internal changes in the log are derived from the log entry at the successive position. If no such entry exists, the term is derived from the leader's term.

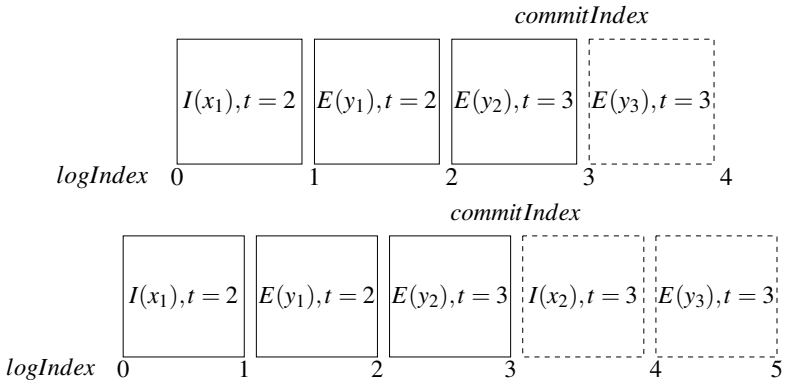

Figure 5.1: Illustration of how a new internal change is prepended to the log. The uncommitted external entry $E(y_3)$ at index 3 is moved to index 4 when $I(x_2)$ is added to the log, and $I(x_2)$ inherits the term of $E(y_3)$.

Additionally a mechanism is added such that snapshots are not made if *internalChange*>0.

5.2.3 Killing State Machines on Leadership Change

To avoid inconsistencies in the approach mentioned, the leader must be the only node running a state machine. As a result, **Snapshot-SetHook** (Chapter 4.2.1) is no longer needed. We perform the following modification on what occurs after a candidate wins an election:

- Change role to leader and make a new `leader_info`
- If *snapshot* is not empty:
  - Start the state machine with *snapshot*
  - Apply all committed log entries after *snapshot*'s **Last Included Index**
- Otherwise:
  - Start the state machine in the initial state
  - Apply all committed log entries

This approach requires proper snapshot conditions to avoid a leader having to apply a lot of entries, when it is elected.

5.3 Replicating and Confirming Computations

When working towards an implementation of Raft that, to the Troupe programmer, works identically to the `spawn`-function[2], we need to consider the inconsistencies our implementation may create. In this section we will describe how to replicate intermediary computations with the use of suspended states, and our approach to ensure that all input passed from the client to the dialer will eventually be passed to the cluster.

5.3.1 Continuation-passing Style in Troupe

The Troupe compiler is interpreting the Troupe language in Continuation-passing Style (CPS). This style works by breaking down each routine into step functions that, when run, returns the next step function of the program.

As we work towards a library that will be embedded within Troupe, it is important to examine this style of programming to identify the benefits it can offer to our implementation. CPS is useful to be able to replicate intermediary computations, such that major computations can be saved. We create the following constraints:

- State machines must be written in *Raft-CPS*
  Guarantees computation. We will return to the details of this format shortly.
- Blocking functions must **not** be used
  Will slow down log application arbitrarily, which in the worst case, can cause elections.
- No concurrent functions
  The Troupe function `spawn` can not be used. A state machine spawning a concurrent thread makes guaranteeing computations impossible, as this can create uncontrolled side effects. Notice that this immediately eliminates the problem of internal changes, since the step function is in charge of internal changes.
- Side effects are limited to sending messages
  We return to this in Chapter 5.3.4.

5.3.2 Raft-CPS Components

The state machine is defined with three components defined in the tuple

$$(\text{side\_effects}, \text{status}, \text{step\_function})$$

This is *Raft-CPS*:

- **Side effects:** The side effects of the routine. These are actions that the Raft cluster will run only on the leader. In the scope of the project, the only side effect considered is message sending.
- **Status:** Signifies to the Raft node if there is an active computation, if the state machine awaits some input, or if the state machine has finished its routine.
- **Step Function:** A function returning the next *Raft-CPS* tuple. Depending on the status, it may require an input. It can also be left empty, if there are no more states in the state machine.

5.3.3 Status and Step Functions

We have defined three statuses that a state machine can be in. They directly relate to the step function. These are their definitions and how they interact with the Raft cluster:

- **Waiting for Input** (`WAIT`): This status represents that the state machine is waiting for an input in order for it to be able to proceed to its next state. We need to make a modification to the log entries to give them a callback. This is used to notify the dialer, when messages have been committed and applied.

  The cluster will only accept an input to the state machine, if the state machine is in a waiting state. The cluster will otherwise ignore the message. If a dialer waits too long before receiving an acknowledgement, it will resend the message as before. The existing serial key system avoids duplicates.

  A `WAIT`-status is in the following format:

  $$(\text{side\_effects}, \text{WAIT}, f(x) \rightarrow (\text{side\_effects}', \text{status}', \text{step\_function}))$$

- **Suspended** (`SUS`): The `SUS` status is used so heavy computations can be organized into smaller steps, to ensure that a node crashing does not undo large computations. It does not require any user input, as it is a breakpoint in the computation.

  We replicate this by checking if the state machine is in the `SUS` state, and if the log is committed, meaning that the state machine has applied all changes. If so, the leader will add a unit entry to the log, which is replicated before being applied to the state machine.

  If a dialer contacts the cluster with a state machine in this status, the cluster lets the dialer know and the dialer waits an arbitrary amount of time, and tries to resend the message. A `SUS`-status is in the following format:

  $$(\text{side\_effects}, \text{SUS}, f() \rightarrow (\text{side\_effects}', \text{status}', \text{step\_function})) \tag{5.4}$$

- **Finished** (`DONE`): The final status of the state machine. The step function is empty and no more inputs should be passed into the state machine. No entries can be added to the log beyond this point.

5.3.4 Side Effects

State machines can have several side effects when run on a normal computer, but many of these do not make sense when run on a cluster. One side effect that does make sense is sending messages. This should be tied to the leader to avoid all nodes in a cluster sending the same message.

When we run a step function, any messages that need to be sent from the cluster is output in a list of side effects. If a leader crashes while side effects are being sent, and another node gets elected, it might send the messages twice. To handle this, we added deterministic serial keys to messages based on log index and index in the list of side

effects. The dialer checks this serial key, before passing the message onto the client, to eliminate duplicates.

By giving the programmer a mechanism for sending messages through the cluster, we also facilitate a way to send messages from cluster-to-cluster. Since clusters do not keep a dialer themselves[1], the Raft leader creates a temporary dialer, dials up the cluster, sends all messages designated to that cluster, and shuts that dialer down. We attach the previously mentioned serial keys to messages sent by the temporary dialer. This ensures that if the leader crashes while the list of side effects is being handled, the same message will not get a different serial key, and the recipient cluster can discard duplicate messages. The dialer presented in Chapter 4 works to ensure eventual delivery.

5.3.5 Consequences

Adding a state machine, that is not a concurrent thread but instead a step function, allows us to remove most of the previously described hooks in Chapter 4, except for the snapshot condition. We simplify it to be the maximum size that the committed log can grow to before a snapshot is performed, and add it as an arbitrary constant of the function.

A major consequence of using Raft-CPS for state machines is the direct tie of log replication to the progress between states using suspended states. This creates a delay during runtime, since each message has to be replicated before the leader can append a new entry to the log. We alleviate this by sending unit messages to the up to date followers instantly after adding it in a suspended state. We discovered issues running this locally, so here we opt for our previous solution, tying it to the heartbeat.

This system disallows state machines spawning concurrent threads, as Raft-CPS state machines spawning new threads can create inconsistencies. As a result, we cannot combine the solutions for Raft-CPS and internal changes.

5.4 Integrating into the Troupe Library

When choosing one solution over the other we revisit our goal:
To make an abstract Raft implementation to further be developed into a library, which is then set to become an embedding of Raft in the future.

We would like to create a `raft_spawn` function that works to the user as the other `spawn` functions present in Raft. Writing a state machine in CPS can range from being a slight inconvenience to annoying compared to writing it in plain Troupe – Using Raft-CPS will therefore be more demanding of the user. The library is however developed towards eventually becoming embedded into Troupe, which uses CPS. The programmer will not be required to write their program in Raft-CPS. It is up to the compiler to convert a function to Raft-CPS.

The main reason for choosing the Step Functions over Internal Changes, is that Internal Changes can introduce too many inconsistencies, violating safety. It breaks Raft's *Leader Append-Only* property[5], by changing already appended entries in the leader's log. With the approach of using CPS, we can guarantee that the client can

[1] As it would create a single point of failure

interact with the cluster in the same way it would with any other distributed concurrent running state machine. Internal changes are still relevant for discussing how a system supporting *all* of Raft's features could be made. Implementing step function with CPS, we get the final signature of the Raft library.

Let *state_machine* be a state machine written in Raft-CPS:
For local Raft with n nodes in the cluster:

$$\text{raft\_spawn}(\text{state\_machine}, n) \rightarrow \text{cluster\_id}$$

For distributed Raft with *locations* being the locations of the distributed nodes:

$$\text{raft\_spawn}(\text{state\_machine}, \text{locations}) \rightarrow \text{cluster\_id}$$

Just like before, the client needs to dial into the cluster using

$$\text{raft\_dial}(cluster\_id, client\_id) \rightarrow \text{dialer\_id}$$

for client-to-cluster communication. The final implementation can be found at Appendix A. [2]

[2]This requires the modified `lists`-library found at Appendix B. For a simpler setup, look at the userguide at Appendix F. In the userguide, we introduce a variable *settings*, for defining different Raft and dialer behaviours, such as heartbeat interval.

Chapter 6

Evaluation on the Implementations of Raft

In this chapter we will evaluate the performance of our Raft library. We will describe the testing environment we used for testing the integrity of the Raft library, as well as testing the overhead of running Raft.

6.1 Integrity

The Raft algorithm is crash safe, as long as a majority of the cluster is still alive. In order to test this, we built a system designed to crash as many nodes as possible at random intervals.

This function is named `bad_actor`. The idea behind it is to randomly choose a node in the cluster and either take it offline, or put it back online. The function will keep track of how many nodes it has taken offline, as Raft makes no guarantees for a cluster where less than the majority of the cluster is online.

To run `bad_actor` it needs the following arguments:

- **Cluster**: The cluster to disrupt. `bad_actor` can only disrupt one cluster at a time.
- **Amount of Nodes**: The amount of nodes in the cluster. This is to calculate a threshold for the amount of failures. `bad_actor` also uses this to create its overview of the cluster.
- **Stress Interval**: This is the interval at which it will toggle nodes. This can be useful to dial with depending on which state machine is running on the cluster and the level of stress we want to simulate.
- **Verbose**: To print extra debugging information. It will print whenever `bad_actor` tries to toggle a node

With this tool in hand, we were able to properly test snapshotting and side effects, as these functions are impacted by nodes crashing and coming back online.

A ping-pong server[1] is a program which, when given a callback id and an integer value, increments the value by one and sends it back, along with its own callback id. We created a test, where an honest client will keep pinging the server, as long as it receives a correct response. We chose to use `bad_actor` on the ping-pong server, as it is a computationally lightweight state machine with a lot of log activity.

This setup was run multiple times to provoke errors in the cluster. This could manifest itself in two different ways, the cluster could repeat a number previously sent to the client, or the cluster could skip side effects. We coded a client in a simple way, as to not accidentally create an error that is not due to our implementation of Raft.

Some of these tests ran without major errors. Some of the tests did however expose flaws in the way we handled side effects. This was due to how the leader could crash before sending a reply to the client. When this happened, the next leader would not redo the side effects, which lead to the client waiting forever on a message that would not come. We fixed this, by always sending side effects upon leader election. This will result in double sending some messages, but because of our serial keys, we know that any recipient is able to recognize this.

6.2 Overhead

Part of evaluating Raft, as an approach to create highly reliable and available state machines, is also testing the impact of the increased overhead that Raft creates. It is not surprising (rather expected) that running a state machine in a Raft cluster is slower, but we aim to research and figure out the margins, and relevant factors.

For testing the overhead that Raft creates, we have implemented an $O(n)$ Fibonacci-function in three ways: standard recursive, CPS-style, and Raft-CPS. Their Troupe implementations can be found in Appendix D. We traverse the CPS-function in a loop until it returns a value.

We ran the programs on the inputs $n = 5, 8, 10, 12, 15$ and recorded the time before and after to determine how long they took. The results of the two non-Raft implementations are shown in Figure 6.1. CPS is a tiny bit slower, as it requires a function to step through it and has to save the next function before running it.

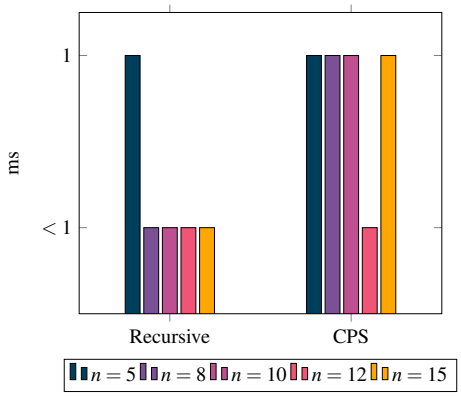

Figure 6.1: Comparison between the time non-Raft Fibonacci solvers will take with different inputs in milliseconds.

We researched how big of an impact the amount of states in a Raft-CPS function has as well as the amount of nodes in the cluster. All tests on Raft-CPS were performed on a Raft cluster with distributed nodes on the same machine to avoid random transmission time affecting the tests.

We tested the impact of suspended states in a cluster by spawning five distributed

[1] Code found in Appendix C

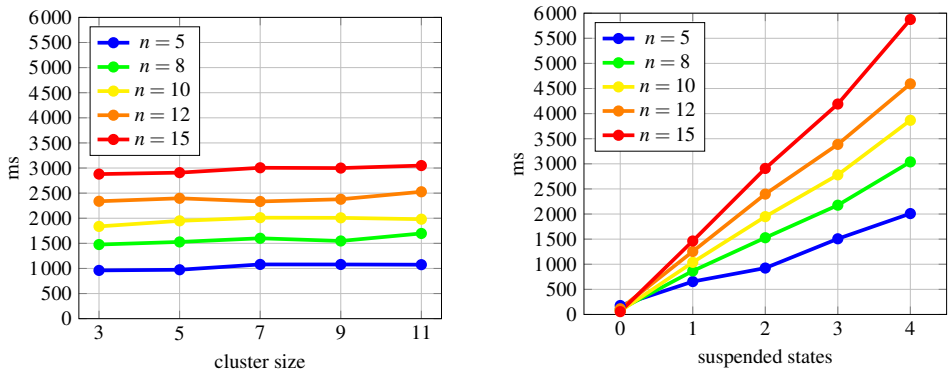

Figure 6.2: Left: Comparison of Raft Fibonacci solver for different cluster sizes.
Right: Comparison of Raft Fibonacci solver for different amount of suspended states. Time for both is given in milliseconds.

nodes and added suspended states, such that they would be traversed during a non-terminating loop of the Fibonacci-function. All five different configurations are found in Appendix D.3. When reducing or increasing the amount of suspended states, you effectively add another message to be processed by each node. It is notable how critically the runtime of the Raft algorithm implemented gets increased for all inputs when adding additional suspended states (Figure 6.2, right). While it is potentially safer to avoid redoing computations, this puts a responsibility on the compiler, when this library is eventually embedded, to ensure not to add superficial suspended states on simple executions like arithmetics and returning a value. For an $O(n)$ Fibonacci-function, if you add suspended states to each iteration, you multiply the amount of messages to be replicated in the Raft cluster.

From our tests on the cluster size, we had the two suspended states present in the code at Appendix D.3.3. We see a tiny increase in the time it took, as more nodes are present in the system, but it is tiny and very linear. This highlights some of the optimality of Raft – The leader only needs a majority of messages before sending the next one. This does not mean that the programmer should be careless when defining the size of the cluster. The *etcd* documentation[7] suggests not to use a cluster of a size larger than 7. Of course this depends on the reliability of your nodes, but in any case it is important to know that there is a considerate amount of latency added to a system when increasing the cluster size. Cockroach DB's *non-voting replicas*[8] add the functionality for this: adding more availability while removing the increased messages being sent back and forth.

You should not make a cluster larger just to be safe, because it can impact performance. Simultaneous elections are more likely and in a delayed network, unlike the one tested in, there can be a lot of messages sent back and forth. Similarily, when designing the state machine, every suspended state added should be carefully considered to avoid increased unnecessary latency. It is a more concrete and harsh increase of overhead set by Raft-CPS. A Raft managed state machine is clearly not going to win over any local run state machine in terms of runtime. The recursive implementation stays under 1ms for almost all inputs and, in the tests performed, Raft goes anywhere from 56ms to 5875ms. The exact data is located in Appendix E.

Chapter 7

Comparison to related work and ideas for future work

In this section we will compare our Raft implementation to a possible implementation using Nysiad[3] to avoid Byzantine Failures[4]. Lastly, we will reflect on the implemented Raft library and further discuss what future work it entails.

7.1 Comparison to Nysiad

Since the described implementation works given a fully trusted cluster, it is interesting to compare the system to a variant in a Byzantine Faulty[4] environment using Nysiad[3]. The major downside of Nysiad is that it worsens the performance and increases messages sent. All messages have to be approved by $t+1$ guards before being delivered to the state machine. The messages sent during an election from and to the Olympus, to allow candidates to request votes and finally become leaders, given the number of nodes in the cluster being n, and guards for each node being $3t-1$, grows quite a bit. Candidate c_i sending out a request-vote m_r connects a candidate to $n-1$ nodes (as it is already connected to the leader), and in order for the Olympus to deliver to all $n-1$ nodes, it needs to get proof of the state of c_i from c_i's $\geq 3t-1$ guards for each node. That is $(n-1)(3t-1)-1$ additional messages sent to the nodes.

Nysiad has some benefits however of contributing to a more flexible system, supporting a weak trust base. We have noticed the benefits of membership change policies, and it is something that would be ideal in future work. It allows us to change slow or defective nodes, take a node offline to update it, and optimize the overall responsiveness of the system. Nysiad could be essential in such a system that does not necessarily trust a given number of its nodes. It supports any failures on t nodes, making it possible to support dangerous threat models.

7.2 Future Work

What we have achieved in this project is a library for the programming language Troupe, that gives users the ability to spawn a process on a cluster of machines. The

process is limited to only be written in Raft-CPS, and there are some functions in Troupe that are not supported. Before this can be used as a part of the Troupe runtime, there are some hurdles to overcome, which we will discuss in the following sections.

7.2.1 Extensions to Snapshot Conditions and Log Summary

It is clear that there is no regular way to snapshot efficiently that will work for every state machine. Since the snapshotting condition will, in the embedment, end up being hidden away from the programmers into the runtime, and the runtime can gather the required information about the state from the step function, we can leave it there, as an optional setting to be set by the programmer[1].

7.2.2 Unsupported Functions

At the moment the library does not support using functions that block or spawn other threads. This is something that has to be handled on a case by case basis. Receiving messages is trivial, as it is implemented through the WAIT status. It is not at complete feature parity with how `receive` works in Troupe – The programmer needs to pattern match on the input and implement their own mailbox to save messages for later, unlike `receive` in Troupe, which handles this implicitly.

The main unsupported function in Raft-CPS is `spawn`. The issue with spawning concurrent threads, is that we still need to be able to replicate the state of the entire program running on the cluster. If `spawn` was delegated to spawn a thread on only the leader, it is no longer truly distributed. A potential way to handle `spawn` on the cluster is substituting `spawn` with a Raft specific spawn function, where the cluster has to keep track of a list of state machines instead of just one. This would come with some design decisions: How should the cluster handle input to the different machines, and how can two state machines on the same cluster communicate with each other? There is also the consideration of state machines spawning remotely and state machines spawning Raft clusters.

7.2.3 Embedding into Runtime

We have made our library in such a way, that embedding it into the runtime itself, should be doable without a major refactorization. This would entail that the runtime will handle the Raft specific functions in the background without the user knowing it. Things like sending a message to a cluster can be abstracted away, if the runtime can see that the process-id is a cluster and not a single computer or thread. It will then use the dialer to communicate with the cluster, and since the dialer we created already handles all the quirks of communicating with a cluster, the runtime does not need to do more in that regard.

The bigger tasks in the embedding is going to be the interpretation of programs running on the cluster. The Troupe runtime already compiles the code into a form of CPS, but when running on the cluster it needs to substitute the unsupported functions

[1] Practical use described in Appendix F

with patterns, which are able to be run by the cluster. This will again have to be handled on a case by case basis, as the different unsupported functions have different workarounds to be able to run in a cluster.

Chapter 8

Conclusion

In this thesis we have researched and analyzed Raft with the goal of developing a library in the Troupe programming language. We have presented relevant terminologies and properties regarding replicated state machines and clusters, and reviewed concrete practical use cases of Raft. We have described Raft as a leader based consensus algorithm utilizing three different roles and explained the election- and logging process.

We have implemented a key-value store, laying the foundation for Raft to begin running on a concrete state machine. We described the logic for a follower to begin an election and become a leader, as well as how the leader transmits messages to the log to modify a key-value store and receive values.

Next, we abstracted away from the key-value store and moved towards running an arbitrary state machine on the cluster using explicitly defined hooks. To facilitate client-to-cluster interaction, we implemented a dialer, ensuring that each message is eventually processed by the cluster, and that messages are not sent to the client twice using serial keys. We implemented snapshotting based on the Raft specification as a mechanism for log compacting.

Afterwards, we looked into implementing a library for the Troupe programming language, made for the purpose of becoming an embedment in the future. Here we presented two approaches: Making it possible for state machines to run concurrent sub-processes that can change the state, and securing computation using suspended states. In the end, we opted for securing computation as we felt that it is important with regards to our initial goal. This had the impact that state machines are now to be written in a modified version of continuation-passing style, Raft-CPS. Furthermore we implemented cluster-to-cluster communication.

We tested the integrity using our `bad_actor`, responsible for turning Raft nodes on and off, to discover bugs and issues. We also tested the impact of the additional overhead induced by the Raft cluster. Here we observed the impact of cluster size and suspended states as a method to save computation. We found that suspended states must be utilized with great care to optimize the speed of the state machine.

Finally, we discussed how the system could be adapted to support Byzantine corruptions with Nysiad. For possible future work we covered the different unsupported Troupe functions in the Raft library, and how the embedment can make them compatible.

Bibliography

[1] Diego Ongaro. “Consensus: bridging theory and practice.” PhD thesis. Stanford University, USA, 2014. URL: https://searchworks.stanford.edu/view/10608105.

[2] *Troupe programming language user guide*. May 2024. URL: https://troupe.cs.au.dk/userguide.pdf.

[3] Chi Ho et al. “Nysiad: Practical Protocol Transformation to Tolerate Byzantine Failures.” In: *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*. Ed. by Jon Crowcroft and Michael Dahlin. USENIX Association, 2008, pp. 175–188. URL: https://www.usenix.org/legacy/events/nsdi08/tech/full_papers/ho/ho.pdf.

[4] Fred B. Schneider. “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial.” In: *ACM Comput. Surv.* 22.4 (1990), pp. 299–319. DOI: 10.1145/98163.98167. URL: https://doi.org/10.1145/98163.98167.

[5] Diego Ongaro and John K. Ousterhout. “In Search of an Understandable Consensus Algorithm.” In: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. Ed. by Garth Gibson and Nickolai Zeldovich. USENIX Association, 2014, pp. 305–319. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[6] The Kubernetes Authors. *Kubernetes*. May 2024. URL: https://kubernetes.io/.

[7] etcd Authors. *v3.5 docs | etcd*. May 2024. URL: https://etcd.io/.

[8] *CockroachDB - Replication Layer*. May 2024. URL: https://www.cockroachlabs.com/docs/stable/architecture/replication-layer.

Appendix A

raft_troupe.trp

See attached file with filename “raft_troupe.trp” or look in the User-Guide at Appendix F .

Appendix B

lists.trp

See attached file with filename “lists.trp”, or look in the User-Guide at Appendix F .

Appendix C

Ping-Pong Raft-CPS State Machine

```
fun main cluster x = case x of
(callback, x) =>
    ([(callback, (cluster, x + 1))], WAIT, main cluster)
|_ => ([], WAIT, main cluster)

fun ping_server cluster = ([], WAIT, main cluster)
```

Appendix D

Fibonacci test implementations

D.1 Standard Recursive

```
fun standard_fib n =
    if n <= 1 then n
    else
        (standard_fib (n - 1) + standard_fib (n - 2))
```

D.2 Continuation-passing Style

```
fun cps_fib n =
    let fun loop n f =
        if n <= 1 then f n
        else (fn () =>
            loop (n - 1) (fn x => fn () =>
                loop (n - 2) (fn y => f (x + y))))
    in loop n (fn x => x)
end
```

D.3 Raft-CPS Fibonacci on Different Suspended States

D.3.1 0 Suspended States

```
(* Calculates the nth Fibonacci number*)
fun fib_raft cb n  =
    (* O(n) recursive solution*)
    let fun loop cb n a b = case n of
    (* Send result of recursive loop to cb *)
    1 => ([(cb, b)], WAIT, fib_input)
    (* Add suspended states to ensure computation when doing
       recursion*)
    | n => loop cb (n - 1) b (a + b)

    in if n <= 1 then ([(cb, n)], WAIT, fib_input)
        else loop cb n 0 1
end

and fib_input x = case x of
(* Compute the nth fibonacci number if input matches *)
(callback, n) => fib_raft callback n
(* Ignores otherwise*)
| _ => ([], WAIT, fib_input)
```

D.3.2 1 Suspended State

```
(* Calculates the nth Fibonacci number*)
fun fib_raft cb n  =
    (* O(n) recursive solution*)
    let fun loop cb n a b = case n of
    (* Send result of recursive loop to cb *)
    1 => ([(cb, b)], WAIT, fib_input)
    (* Add suspended states to ensure computation when doing
       recursion*)
    | n =>
        ([], SUS, fn () => loop cb (n - 1) b (a + b))

    in if n <= 1 then ([(cb, n)], WAIT, fib_input)
        else loop cb n 0 1
end

and fib_input x = case x of
(* Compute the nth fibonacci number if input matches *)
(callback, n) => fib_raft callback n
(* Ignores otherwise*)
```

```
| _ => ([], WAIT, fib_input)
```

D.3.3 2 Suspended States

```
(* Calculates the nth Fibonacci number*)
fun fib_raft cb n  =
    (* O(n) recursive solution*)
    let fun loop cb n a b = case n of
    (* Send result of recursive loop to cb *)
    1 => ([(cb, b)], WAIT, fib_input)
    (* Add suspended states to ensure computation when doing
       recursion*)
    | n =>
        ([], SUS, fn () =>
            let val n = n - 1
            in ([], SUS, fn () => loop cb n b (a + b)) end)

    in if n <= 1 then ([(cb, n)], WAIT, fib_input)
        else loop cb n 0 1
end

and fib_input x = case x of
(* Compute the nth fibonacci number if input matches *)
(callback, n) => fib_raft callback n
(* Ignores otherwise*)
| _ => ([], WAIT, fib_input)
```

D.3.4 3 Suspended States

```
(* Calculates the nth Fibonacci number*)
fun fib_raft cb n  =
    (* O(n) recursive solution*)
    let fun loop cb n a b = case n of
    (* Send result of recursive loop to cb *)
    1 => ([(cb, b)], WAIT, fib_input)
    (* Add suspended states to ensure computation when doing
       recursion*)
    | n =>
        ([], SUS, fn () =>
            let val n = n - 1
            in ([], SUS, fn () => loop cb n b (a + b)) end)

    in if n <= 1 then ([(cb, n)], WAIT, fib_input)
        else loop cb n 0 1
```

```
end

and fib_input x = case x of
(* Compute the nth fibonacci number if input matches *)
(callback, n) => fib_raft callback n
(* Ignores otherwise*)
| _ => ([], WAIT, fib_input)
```

D.3.5 4 Suspended States

```
(* Calculates the nth Fibonacci number*)
fun fib_raft cb n  =
    (* O(n) recursive solution*)
    let fun loop cb n a b = ([], SUS, fn () => case n of
    (* Send result of recursive loop to cb *)
    1 => ([(cb, b)], WAIT, fib_input)
    (* Add suspended states to ensure computation when doing
       recursion*)
    | n =>
        ([], SUS, fn () =>
            let val n = n - 1
            in ([], SUS, fn () =>
                let val c = a + b
                in ([], SUS, fn () => loop cb n b (a + b)) end
                   ) end))

    in if n <= 1 then ([(cb, n)], WAIT, fib_input)
        else loop cb n 0 1
end

and fib_input x = case x of
(* Compute the nth fibonacci number if input matches *)
(callback, n) => fib_raft callback n
(* Ignores otherwise*)
| _ => ([], WAIT, fib_input)
```

Appendix E

Fibonacci Data Tables

E.1 Non-Raft Data

	n=5	n=8	n=10	n=12	n=15
Recursive	1	0	0	0	0
CPS	1	1	1	0	1

E.2 Raft Cluster Size Data

	n=5	n=8	n=10	n=12	n=15
Cluster size=3	961	1476	1837	2338	2879
Cluster size=5	974	1528	1948	2397	2908
Cluster size=7	1080	1601	2011	2334	3005
Cluster size=9	1079	1547	2008	2378	2999
Cluster size=11	1075	1697	1981	2527	3049

E.3 Raft Suspended States Data

	n=5	n=8	n=10	n=12	n=15
Suspended states=0	178	99	67	113	56
Suspended states=1	655	866	1038	1254	1463
Suspended states=2	924	1528	1948	2397	2908
Suspended states=3	1507	2176	2780	3387	4191
Suspended states=4	2009	3038	3867	4593	5875

Appendix F

User Guide

F.1 Requirements

There are a few requirements to using the Raft library:

- A fully set up Troupe install with the fork located at TroupeLang/Troupe.
- Remember to switch to the 'dev'-branch
- Build and add file as library: From Troupe source folder: `cd` into `examples/raft-troupe`. Run `make` (Requires Python3). From the `build`-folder, move `raft_troupe.trp` into `$TROUPE/lib` and modify Makefile in `$TROUPE` to add library.
- Import the library by writing `import raft_troupe` at the top of the file.

F.2 Usage

F.2.1 Setting up Distributed Nodes

In order to spawn a distributed node, you need your cluster of nodes to be running on an empty script such that it can be contacted to spawn the Raft nodes. This also requires the node to be run as follows:

```
1    $TROUPE/network.sh zero.trp --id=ids/node.json  --rspawn=
        true --aliases=aliases.json --stdiolev={} # --debug --
        debugp2p
```

With `zero.trp` being a Troupe program containing only 0, `node.json` being the id for that node, `--rspawn=true` enabling remote spawning and `--aliases=aliases.json` being the generated aliases for the cluster. See the Troupe userguide for more in-depth explanation of this procedure.

F.2.2 Cluster

`raft-spawn (state_machine, n)`

Spawns a Raft cluster with default settings, containing n nodes (making a majority of n), running a state machine in the consensus algorithm. Snapshots when the log grows to size > 50.

Parameters:

- `state_machine`: A function written in our CPS-style, such that it always return a 3-tuple, $(\text{side effects}, \text{status}, \text{step})$. May be left as a unit if one wishes to spawn a cluster before loading a state machine.

Returns: A list of process-ids of the nodes.

`raft-spawn (state_machine, n, settings)`

Spawns a Raft cluster with custom settings, containing n nodes (making a majority of $\lceil n/2 \rceil$), running a state machine in the consensus algorithm. Snapshots when the log grows to size $>$ snapshot_cond.

Parameters:

- `state_machine`: A function written in our CPS-style, such that it always return a 3-tuple, $(\text{side effects}, \text{status}, \text{step})$. May be left as a unit if one wishes to spawn a cluster before loading a state machine.
- `n`: An integer, denoting the amount of nodes to be spawned in the cluster. To avoid frequent split votes it is recommended that n is uneven.
- `settings`: Settings defining constants and some behaviour of the cluster.

Returns: A list of process-ids of the nodes.

`raft_spawn_alias (state_machine, aliases)`

Spawns a local Raft cluster with the nodes located on the aliases, containing $|\text{aliases}|$ nodes (making a majority of $\lceil |\text{aliases}|/2 \rceil$), running a state machine in the consensus algorithm. Snapshots when the log grows to size > 50.

Parameters:

- `state_machine`: A function written in our CPS-style, such that it always return a 3-tuple, $(\text{side effects}, \text{status}, \text{step})$. May be left as a unit if one wishes to spawn a cluster before loading a state machine.
- `aliases`: A list of strings denoting the aliases of machines running with remote spawning enabled (explained above).

Returns: A list of process-ids of the nodes.

`raft_spawn_alias (state_machine, aliases, settings)`

Spawns a local Raft cluster with custom settings with the nodes located on the aliases, containing $|\text{aliases}|$ nodes (making a majority of $\lceil |\text{aliases}|/2 \rceil$), running a state machine in the consensus algorithm.

Parameters:

- `state_machine`: A function written in our CPS-style, such that it always return a 3-tuple, $(\text{side effects}, \text{status}, \text{step})$. May be left as a unit if one wishes to spawn a cluster before loading a state machine.
- `aliases`: A list of strings denoting the aliases of machines running with remote spawning enabled (explained above).
- `settings`: Settings defining constants and some behaviour of the cluster.

Returns: A list of process-ids of the nodes.

F.2.3 Cluster Settings

To create custom cluster settings, you must make a record in the following format:

```
1  val settings = {
2        ELECTION_TIMEOUT_LOWER = 2000 (* Some integer value *)
             ,
3        ELECTION_TIMEOUT_UPPER = 4000 (* Some integer value *)
             ,
4        HEARTBEAT_INTERVAL = 500 (* Some integer value *),
5        TIE_COMMITS_TO_HEARTBEAT = true (* Some boolean value
             *),
6        MAXIMUM_LOG_SIZE = 50 (* Some integer value *),
7        leader_dialer_settings = dialer_settings (* Record
             containing dialer settings *)
8    }
```

Here is an explanation of the fields:

- `ELECTION_TIMEOUT_LOWER`: The lower bound of an election timeout. Must be smaller than `ELECTION_TIMEOUT_UPPER`.
- `ELECTION_TIMEOUT_UPPER`: The upper bound of an election timeout. Default value is 4000 in `raft_spawn` and `raft_spawn_alias`
- `HEARTBEAT_INTERVAL`: The time interval between each heartbeat sent by a leader. Default is 500 in `raft_spawn` and `raft_spawn_alias`.
- `TIE_COMMITS_TO_HEARTBEAT`: Denotes whether or not suspended states are sent from leader to (up to date) followers as soon as they are appended. In a multithreaded environment, when running a cluster distributed, it is recommended that this is set to `true`, and `false` otherwise. Default is `true` in `raft_spawn` and `false` in `raft_spawn_alias`.

- `MAXIMUM_LOG_SIZE`: The maximum size a committed log will be before log compacting (snapshotting). Default is 50 in `raft_spawn` and `raft_spawn_alias`.
- `leader_dialer_settings`: Settings controlling the temporary dialer spawned to faciliate cluster-to-cluster communication. How this is defined is specified below.

F.2.4 Client-to-State-Machine Communication

In order to facilitate client-to-cluster communication, the client should make use of a dialer, to handle the overhead of removing duplicate message delivery, ensuring that all client messages are eventually delivered and so on. A cluster can be dialed into using:

`raft_dial (cluster, client_id)`

Spawns a dialer with default settings for `client_id` which dials into `cluster` and sends all received (non duplicate) message to the client.

Parameters:

- `cluster`: A list of process-ids generated by `raft_spawn` or `raft_spawn_alias`.
- `client_id`: The process id of the client that should receive messages sent from the state machine to the dialer.

Returns: A process-id of the dialer.

`raft_dial (cluster, client_id, dialer_settings)`

Spawns a dialer with custom settings for `client_id` which dials into `cluster` and sends all received (non duplicate) message to the client.

Parameters:

- `cluster`: A list of process-ids generated by `raft_spawn` or `raft_spawn_alias`.
- `client_id`: The process id of the client that should receive messages sent from the state machine to the dialer.
- `dialer_settings`

Returns: A process-id of the dialer.

F.2.5 Dialer Settings

To create custom dialer settings for configuration cluster-to-cluster communication or client-to-cluster communication, you must make a record in the following format:

```
1  val dialer_settings = {
2        DIALER_NOLEADER_TIMEOUT = 500,
3        DIALER_NOMSG_TIMEOUT = 2000,
4        DIALER_SM_BUSY_TIMEOUT = 1000
5    }
```

Here is an explanation of the fields:

- `DIALER_NOLEADER_TIMEOUT`: The time the dialer waits, before attempting to re-send a message when the leader is unknown. Default is 500.
- `DIALER_NOMSG_TIMEOUT`: The time the dialer waits without an acknowledgement, before resending the message. Default is 2000.
- `DIALER_SM_BUSY_TIMEOUT`: The time the dialer waits for a Raft cluster with its state machine in a SUS/suspended state, before attempting to send the message again.

Contacting the state machine/dialer

A dialer ensures that the message received from the client are delivered into the cluster and replicated. This is how the client may communicate with the state machine through a dialer with process-id `dialer_id`:

```
1 |     send(dialer_id, (RAFT_UPDATE, msg))
```

The message `msg` will eventually be replicated and forwarded to the state machine to change its state. Note: If the state machine will never reach a WAIT-state the dialer will infinitely continue to try sending the message.

F.2.6 Writing State Machines

As described in the report, our final implementation requires the state machines to be written in a CPS-like format such that each function contains a step-function to the next one. However, for Raft to be able to determine the correct course of action, a couple of extra mechanisms are added. Therefore, a state machine must be in the following 3-tuple-format:

```
1 |(side_effects, status, fn _ =>
2 |  (side_effects, status, fn _ =>
3 |    (side_effects, status, fn _ => ...)))
```

side effects:

`side-effects` denotes a list of 2-tuples, containing the adress of who to sent the message to, and what the message is, in the format:

```
1 |(callback_pid, msg)
```

The callback_pid is not the native Troupe process-id. Because a state machine can communicate with a client (through the dialer), or a cluster (through a temporary cluster-created dialer), we require process-ids to be a record of either:

```
1 |{ type = CLIENT, id = pid}
```

, where `pid` is the process-id of the dialer, or

```
1 |{ type = CLUSTER, id = [pid]}
```

, where `[pid]` is a list of process-ids the nodes to send the message to created by the `raft_spawn` or `raft_spawn_aliases`-functions. The `msg` is simply any message.

Status and step

There are three different statuses and the step function is closely related in the following ways: The statuses must be picked based on their corresponding step function, and,

Status	Step
`WAIT`	`fn x => ...`
`SUS`	`fn ()=> ...`
`DONE`	`()`

by nature of CPS, the step function must always (unless status is `DONE`) return a new `(side_effects, status, fn _ => ...)`. `SUS`/suspended-states will be automatically progressed and replicated by the Raft cluster, `WAIT`-states require input from a dialer, and `DONE` is the state machine ending its execution. No messages will be passed from the Raft cluster to it afterwards.

Fibonacci Example

Here is an example of a recursive Fibonacci state machine written in Raft-CPS:

```
import raft_troupe

(* Calculates the nth Fibonacci number*)
let fun fib_raft cb n  =
  (* O(n) recursive solution*)
  let fun loop cb n a b = case n of
  (* Send result of recursive loop to cb *)
  1 => ([(cb, b)], WAIT, fib_input)
  (* Add suspended states to ensure computation when doing
     recursion*)
  | n =>
    ([], SUS, fn () =>
      let val n = n - 1
      in ([], SUS, fn () => loop cb n b (a + b)) end)

  in if n <= 1 then ([(cb, n)], WAIT, fib_input)
    else loop cb n 0 1
  end

  and fib_input x = case x of
  (* Compute the nth fibonacci number if input matches *)
  (callback, n) => fib_raft callback n
  (* Ignores otherwise*)
  | _ => ([], WAIT, fib_input)
```

And here is how one can spawn a Raft cluster on remote processes named "node1, node2, ..." and use a dialer to have the state machine calculate the 15th fibonacci number and receive it:

```
  val pid = self()
  val cluster = raft_spawn_alias (([], WAIT, fib_input), ["
     @node1", "@node2", "@node3", "@node4", "@node5"])
  val dialer = raft_dial(cluster, pid)
in  send(dialer, (RAFT_UPDATE, ({type = CLIENT, id = dialer},
    15)));
  receive[hn x => print x]
end
```

F.2.7 Bootstrapping the Cluster

Due to the limitations of the library, and without any way for a function in a state machine to gather the ids of the cluster it is hosted on, one can bootstrap the cluster-id as follows:

- Start a Raft cluster without a state machine.

- Spawn a dialer on the cluster.
- Send a message with the function, given the cluster-id (list of node's ids).

Here is an example of how this can be done:

```
import raft_troupe

let (* Define some arbitrary state machine. This one simply
    sends the IDs of the cluster. *)
  fun some_sm_input cluster input =
      ([(input, cluster)], WAIT, some_sm_input cluster)

  (* Define bootstrapping function. This defines the cluster-
     id and goes into the main state of the state machine. *)
  fun bootstrap_sm cluster =
    ([], WAIT, some_sm_input cluster)

  (* Spawns a cluster of size 5 on an empty state machine. Any
       function sent via RAFT_UPDATE will be run and should
     return a Raft-CPS tuple. *)
  val cluster = raft_spawn ((), 5)
  val pid = self ()
  (* Dial into cluster *)
  val dialer = raft_dial (cluster, pid)

in
  (* Bootstrapping cluster-id to state machine *)
  send(dialer, (RAFT_UPDATE, fn () => bootstrap_sm cluster));
  (* Printing our local cluster ID, and the cluster ID from
     the cluster. These should match *)
  print "local:";
  print cluster;
  (* Requesting cluster-id *)
  send(dialer, (RAFT_UPDATE, {type = CLIENT, id = dialer}));
  print "state_machine␣response:";
  receive [hn x => print x]
end
```

In the following section is an example of bootstrapping state machines to make a ping-pong server.

F.2.8 Ping-Pong Example

Here is an example of how to use bootstrapping in order to facilitate cluster-to-cluster communication:

```
import raft_troupe

```

```
3  let (* Main loop of ping-function *)
4    fun ping_main cluster client_dialer x = case x of
5      (* Whenever receiving a message with a callback and a
          value... *)
6      (callback, x) =>
7        (* Send a message to callback and the client, with the
            value of x + 1, and return to the main loop *)
8        ([(callback, (cluster, x + 1)), (client_dialer, x + 1)],
             WAIT, ping_main cluster client_dialer)
9      (* Otherwise, ignore and return to main loop *)
10     |_ => ([], WAIT, ping_main cluster client_dialer)
11
12   (* Function used to bootstrap client dialer and the state
        machine's own cluster-id *)
13   fun ping_server cluster client_dialer =
14     ([], WAIT, ping_main cluster client_dialer)
15
16   val pid = self()
17   (* Spawn Raft cluster of ping-cluster with empty state
        machine *)
18   val ping_cluster = raft_spawn_alias((), ["@node1", "@node2",
         "@node3", "@node4", "@node5"])
19   (* Define cluster-id of ping-cluster *)
20   val ping_id = {type = CLUSTER, id = ping_cluster}
21   (* Spawn Raft cluster of pong-cluster with empty state
        machine *)
22   val pong_cluster = raft_spawn_alias((), ["@node6", "@node7",
         "@node8", "@node9", "@node10"])
23   (* Define cluster-id of pong-cluster *)
24   val pong_id = {type = CLUSTER, id = pong_cluster}
25   (* Dial into the two clusters *)
26   val ping_dialer = raft_dial(ping_cluster, pid)
27   val pong_dialer = raft_dial(pong_cluster, pid)
28
29   (* Function to continuously read incoming messages *)
30   fun read_pingpongs () =
31     receive [ hn x => print x; read_pingpongs ()]
32 in
33   (* Bootstrap the ping-cluster, giving it a function with its
         ID and the dialer's ID *)
34   send(ping_dialer, (RAFT_UPDATE, fn () => ping_server ping_id
         {type = CLIENT, id = ping_dialer}));
35   (* Bootstrap the pong-cluster, giving it a function with its
         ID and the dialer's ID *)
36   send(pong_dialer, (RAFT_UPDATE, fn () => ping_server pong_id
         {type = CLIENT, id = pong_dialer}));
```

```
37    (* Start an infinite ping-pong *)
38    send(ping_dialer, (RAFT_UPDATE, (pong_id, 0)));
39    read_pingpongs ()
40  end
```