Graft: General Purpose Raft Consensus in Elixir

Matthew Alan Le Brun
Duncan Paul Attard
Adrian Francalanza
matthew-alan.lebrun.17@um.edu.mt
duncan.attard.01@um.edu.mt
adrian.francalanza@um.edu.mt
Department of Computer Science, University of Malta
Msida, Malta

Abstract

We present Graft, a generic tool for creating distributed consensus clusters running the Raft algorithm using state machines in Elixir. Our tool exhibits performance that is comparable to that of the Raft implementation supporting the original proposal, as well as the performance of other state-of-the-art Raft implementations running on the BEAM. The correctness of our tool is also ascertained through a number of complementary verification methods.

CCS Concepts: • **Software and its engineering** → *Formal software verification*; **Cooperating communicating processes**; **Client-server architectures**.

Keywords: Distributed consensus, Raft, Asynchronous components, Elixir, BEAM, Runtime Verification, Testing

ACM Reference Format:
Matthew Alan Le Brun, Duncan Paul Attard, and Adrian Francalanza. 2021. Graft: General Purpose Raft Consensus in Elixir. In *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang (Erlang '21), August 26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3471871.3472963

1 Introduction

Consensus algorithms describe how a distributed group of nodes can interact with one another to reach agreement in the presence of failure [39]. This, in turn, allows external entities to interact with a distributed system as if it were a single-node application. The redundancy inherent to these setups allows these consensus algorithms to provide a degree of fault-tolerance up to some assumed level of failure [24, 39] (*e.g.* as long as a simple majority of its nodes are operational without Byzantine faults).

Raft [42] is a consensus algorithm designed to simplify the complexities of Paxos [37], a widely used distributed consensus algorithm. It authors achieved this simplicity by devising an algorithm that *(i)* is decomposable into smaller, relatively *independent* sub-problems; *(ii)* uses *strong leadership*, whereby an elected node (the leader) takes decisions for the entire distributed cluster. These features streamline the logical flow of the algorithm so that it (generally) stems from the leader, making it easier to understand compared to Paxos. In their seminal paper [42], the authors also implement the algorithm in C++ as the tool LogCabin [40], and show that its performance is comparable to that of Paxos. Raft has been used in large-scale projects [28, 29], and now has a plethora of open-source implementations in a wide variety of languages available online (*e.g.,* etcd [19] and raft [27] for Go, TiKV [17] for Rust). Many of these implementations are not only single purpose applications but general purpose modules/libraries, allowing users to apply the consensus algorithm to a variety of applications without having to recode any of the consensus logic.

To the best of our knowledge, there is no general purpose implementation of Raft for the language Elixir [31] that is well-maintained. Such a library would fit the needs of a number of concurrent programs written in this language:

- Elixir's *"let it fail"* philosophy is in keeping with the fail-stop failure model used by Raft to attain fault-tolerance. There is less of an opportunity for Byzantine faults to occur if the processes running Raft exit on receiving unexpected messages.
- The battle-tested OTP Supervisor behaviour can be used to manage Raft processes and automatically restart crashed nodes if errors occur. The Raft algorithm includes a recovery protocol which can be pipe-lined into a node's restart procedure.
- Applications that employ Raft typically aim to attain *high availability*. Elixir's support for code hot-swapping allows for updates to be made to the computation running on each node without any downtime.

Erlang '21, August 26, 2021, Virtual Event, Republic of Korea
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8612-8/21/08…$15.00
https://doi.org/10.1145/3471871.3472963

We present Graft, a tool for creating distributed consensus clusters running generic state machines in Elixir. The contributions of the paper are:

1. Following a brief overview of the Raft algorithm in sec. 2, we describe the design choices that went into the construction of our tool in sec. 3.
2. In sec. 4 we showcase the *genericity* of our tool, by discussing how the Graft library can be easily instantiated to build applications that employ the Raft algorithm.
3. Sec. 5 details the verification methods used to provide correctness assurances for our implementation. Specifically, we elaborate on the two-pronged approach of test-driven development and runtime verification.
4. We conduct an extensive performance comparison of Graft in sec. 6. We measure it against the original performance data reported by Ongaro and Ousterhout [42], as well as to the state-of-the-art Raft implementations running on the BEAM.

Secs. 7–9 conclude the paper by summarising the contributions and by expanding on related and future work.

2 An Overview of Raft

Distributed consensus algorithms maintain consistency across *state machine* instances replicated on different nodes in a cluster [43]. The configuration of a state machine is stored as a *log* containing the complete *state transaction history* (*i.e.,* state machine inputs) that when applied to the start state of the machine, advance it to its latest state. The Raft [42] algorithm offers a generic method to distribute these state machine logs across cluster members. Nodes in the cluster, henceforth referred to as *servers*, are managed by one such designated server called the *leader*. A leader is elected by its peers via the *leader election* protocol that establishes how a server in the follower state transitions to the candidate state, and finally, to the leader state. Cluster members vote by communicating in rounds until a server is elected to the leader state, thereby concluding the election process. The elected leader is charged with maintaining consensus over the replicated state machine log. It accepts client requests that consist of a *command* to be executed by the state machine replicas in the cluster. Every client request that is processed by the leader is appended to its log, and subsequently forwarded to peers in the cluster for replication.

Raft organises its consensus logic into three sub-problems, namely, leader election (sec. 2.1), log replication (sec. 2.2), and safety (sec. 2.3). Cluster members participating in consensus interact via *asynchronous* Remote Procedure Calls (RPCs). Any server that issues a RPC *always* expects a corresponding reply, but does not block while waiting, making it available to handle other requests; RPCs are reissued by the server upon request *timeout*. Servers handle RPCs depending on the type of invocation and their current internal state, in connection to the Raft protocol fragments discussed next.

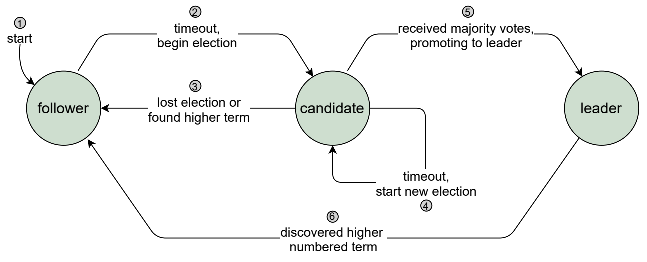

① start
② timeout, begin election
⑤ received majority votes, promoting to leader
follower
③ lost election or found higher term
candidate
leader
timeout, start new election ④
⑥ discovered higher numbered term

Figure 1. The Raft leader election protocol

2.1 Leader Election

Fig. 1 depicts the three states a server can transition to as it is elected to the leader state. We refer to a server in a particular state by its state name, *i.e., follower*, *candidate*, and *leader*, when necessary. Initially, every server in the Raft cluster defaults to the follower state, and is initialised with an *election timeout* that determines when a leader election is called, step ①. Followers only reply to RPC invocations, and *never* initiate communication with other cluster members. When its election timeout elapses, a server in the follower state assumes that there is no leader managing the Raft cluster, and triggers the election protocol, step ②. To minimise the chances of simultaneous elections being called, the election timeout is *randomised* for each follower.

Raft employs a logical notion of time via the *term number* that is incremented by the server calling a new leader election. Servers share their term number with other servers by *embedding* it in every RPC message issued. A server updates its local copy of the term number to match that of received RPCs *only when* the term number is larger than that maintained by the server, and ignored otherwise. In the case of a term number update, a server transitions back to the follower state, steps ③ and ⑥ in fig. 1. A follower calling a new election first transitions to the candidate state, increments its local term number, votes for itself, and finally, sends a `RequestVoteRPC` to every other member in the cluster. When a follower receives a valid `RequestVote` message (*i.e.,* the term number embedded in the RPC is not outdated), it replies by *granting* the vote. The candidate transitions to the leader state whenever it receives a *simple majority* concluding the election, step ⑤.

It is possible for the election timeouts of two (or more) servers in the follower state to elapse at the same time, triggering *simultaneous elections.* To avoid this situation, a follower is limited to *one* vote per term, granting its vote to the candidate issuing the `RequestVoteRPC` on a first-come-first-served basis. If no candidate obtains a simple majority of votes, a new election is called: this is triggered via randomised candidate election timeouts, as shown in step ④. In principle, servers can timeout together and trigger an election that yields no leader (no candidate secures a simple majority of votes). However, parametrising the random

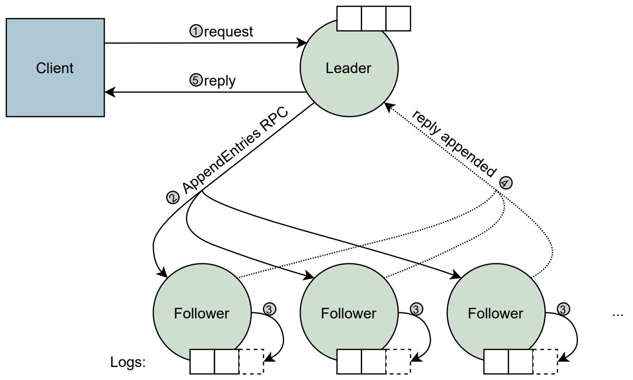


Figure 2. The Raft log replication protocol

generator with a suitable range uniformly distributes the election timeout values of servers, thereby minimising the chances of elections being called indefinitely. An elected leader periodically issues *heartbeat* RPCs to every member in the cluster. This reinitialises the election timeout of followers to *new* random values, and informs any servers in the candidate state to transition back to follower, step ③.

2.2 Log Replication

The replicated state machine log maintained across cluster members is an ordered list of entries in tuple form, that include the: *(i)* client command sent to be executed on the state machine; *(ii)* election term number at which the command was received by the leader; *(iii)* index that identifies the position of the entry in the log. Fig. 2 overviews the log replication protocol enacted by the leader. In step ①, the client issues a request consisting of the command to be executed on the state machine replica of the leader. Upon receiving the client request, the leader *appends* a new entry with the client command to its log, and sends `AppendEntriesRPC` messages to all the servers in the cluster, step ②. The RPC includes:

(i) the newly appended log entry,
(ii) the current election term number of the leader,
(iii) the index and election term number of the *previous* entry in the log of the leader, and,
(iv) the *commit index.*

On receiving the `AppendEntriesRPC`, servers in the follower state compare their latest entry against the one in the RPC. If both entries share the same index and term number, the follower appends the received entry to its log (step ③) and sends an `AppendEntriesRPC` reply to the leader, indicating success, step ④. Failing to effect this match indicates an *inconsistency* between the logs of the follower and leader. The follower rectifies this by initiating the log recovery phase:

(i) it issues a reply to the leader with unsuccessful;
(ii) the leader, in turn, replies with an `AppendEntriesRPC` containing the *previous* log entry;
(iii) the follower checks whether the index and term number of the entry in the `AppendEntriesRPC` sent by the leader correspond to the ones it holds locally;
(iv) if these match, the follower appends the entry to its log and acknowledges to the leader with success, otherwise, it repeats step (i).

When the follower replies with success, the leader transmits the remaining state machine log entries to enable the follower to repopulate its log. The leader marks a newly-appended log entry as *committed* once it receives successful replies from a simple majority of followers in the cluster. It also tracks this latest committed log entry in its *commit index*, applies the entry to its state machine replica, and concludes by issuing a reply to the client in step ⑤. The heartbeat RPC that the leader sends to members of the cluster (sec. 2.1) is encoded as an `AppendEntriesRPC` with an empty entry. This serves the purpose of updating followers with the latest commit index of the leader, informing them that it is *safe* to apply the latest log entry to their local state machine replica.

2.3 Safety

The consensus logic of secs. 2.1 and 2.2 fails to account for particular edge cases that arise due to race conditions. These may lead to inconsistencies in the replicated log. The following two rules are required to prevent these scenarios [42].

2.3.1 Election Restriction. It is possible for a server to be down whilst log entries are being committed to the cluster. *Election restriction* prevents such servers from becoming leader until they recover their log. Failing to do so can result in previously-committed log entries being overwritten by an elected leader in possession of an outdated log replica. Concretely, `RequestVoteRPC` messages also include the index and election term of the last entry in the log of a server in the candidate state. Recipient followers only grant their vote to a candidate if the log of the candidate issuing the RPC is *at least as up-to-date* as the ones followers hold locally. A log is considered more up-to-date than a second log if the *last entry* in the first log: *(i)* has a higher election term than the one in the second log, or it, *(ii)* has a larger index whenever the terms of both entires are equal [42].

2.3.2 Committing Restriction. A leader may crash before committing an entry to its log. *Committing restriction* prevents leaders from explicitly committing the entries from previous terms. Rather, once the leader commits a log entry from the *current* term, all the preceding log entries are considered committed; this follows from the fact that the commit index points to the entry with the latest term number.

3 Graft

We implement the abstract concepts introduced in sec. 2 in the Elixir language to produce Graft — a *generic purpose* Raft library. Our design choices for the implementation are motivated by a number of goals we wish the library to achieve:

- G_1 Graft should be a *generic purpose* library, offering the functionality of building custom-defined replicated state machines.
- G_2 The library should feel similar to other commonly used tools within the Elixir community (*e.g.* `GenServer`).
- G_3 Network-dependent parameters (*e.g.* election timeouts and heartbeat frequency) should be *easily configurable*.

As the original Raft, our implementation process is subdivided into three parts, handling each subproblem of the algorithm in isolation. The *leader election* and *log replication* protocols are implemented in secs. 3.1 and 3.2 respectively, whilst we account for introducing the *safety* restrictions in sec. 3.3. Before delving into these subproblems, we define the static components of the algorithm. Concretely, we declare the attributes for `AppendEntriesRPCs`, `RequestVoteRPCs`, their corresponding replies, and the internal data of a server through Elixir structs. This provides the benefit of compile-time checks on attributes attached to these components, reducing potential development errors. The list of attributes is taken from a summary of Raft provided in [42, fig. 2].

3.1 Leader Election

We begin by creating the `Graft.Server` module, encapsulating the consensus logic. It uses the `GenStateMachine` behaviour, since the actions invoked by a server depends on both its internal data and current state (*i.e.*, follower, candidate or leader). This behaviour is a standard tool in the Elixir language, and simplifies handling and sending messages within processes that can be modelled as a state machine.

Implementing leader election using `GenStateMachine` involves replicating the transitions depicted in fig. 1 as function callbacks. Examples of this are given in lst. 1. For instance, line 5 sets the starting state for the server to follower, correlating to step ① from fig. 1. On receiving a `:start` message, the follower begins a random timeout (line 9), which when triggered is handled by the callback on line 13. The function return-value on lines 15 and 16 indicates the server to transition to candidate, step ②, and to handle the `:request_votes` event afterwards. This, in turn, instructs the candidate to send `RequestVoteRPCs` to all other servers (lines 22 and 24). In spite of expecting a reply, we implement this functionality through `GenStateMachine.cast/2` (as opposed to the `call/2` variant), since RPCs are not blocking calls. Using this method, `RequestVoteRPCs` can be sent out and received in parallel. The rest of the leader election rules are implemented in similar techniques to the methods described above.

3.2 Log Replication

We add the first function to the Graft API, `Graft.request/2`, allowing client requests to be made to the consensus cluster. The function accepts as parameters the *process name* of the recipient server, and a message of *any pattern* from which an entry is created. The function attempts to send the request to the specified server, and if rejected (when the server is not the current leader), redirects the request as necessary.

Once the request reaches the leader, the operations in sec. 2.2 are initiated. By sending `Graft.AppendEntriesRPC` structs to the other servers, the leader attempts to replicate the new log entry at a majority of the cluster. Similar to leader election, callbacks are used to specify the functionality that a server should employ whenever an `AppendEntriesRPC` is received. Once the entry becomes committed, the leader calls the `apply_entry` function and waits for a result. This, in turn, communicates with the `Graft.Machine` process — a `GenServer` that handles the applying of log entries.

In line with the goals set out for Graft, users should be able to implicitly describe how log entries are applied to the machine. The standard method of providing this functionality is through a behaviour. The `Graft.Machine` module specifies a behaviour with two callbacks: `init/1` and `handle_entry/2`.

```
 1 defmodule Graft.Server do
 2   use GenStateMachine
 3
 4   def init([...]), do:
 5     {:ok, :follower, %Graft.State{...}}
 6
 7   def follower(:cast, :start, _data) do
 8     {:keep_state_and_data,
 9      [{{:timeout, :election_timeout},
10        generate_time_out(), :begin_election}]}
11   end
12
13   def follower({:timeout, :election_timeout},
14                :begin_election, data) do
15     {:next_state, :candidate, data,
16      [{:next_event, :cast, :request_votes}]}
17   end
18
19   def candidate(:cast, :request_votes,
20       data = %Graft.State{me: me,servers: servers,
21       log: [{last_index, last_term, _} | _tl]}) do
22     for server <- servers, server !== me do
23       GenStateMachine.cast(server,
24         %Graft.RequestVoteRPC{
25           term: data.current_term+1,
26           candidate_name: me,
27           last_log_index: last_index,
28           last_log_term: last_term
29       })
30     end
31     # keep state, update term and votes in data
32     # start election_timeout
33     #...
34   end
35   #...
36 end
```

Lst. 1. Sample `GenStateMachine` callbacks

The *init* function is used to set up the initial state of the machine, whilst `handle_entry` allows users to *pattern match* to log entries and specify their response and state updates.

Fig. 3 demonstrates how the server and machine processes communicate, as well as how the user defined module is used to apply log entries. Firstly, whenever the server process identifies an entry to be applied (lines 1 and 2), it calls `apply_entry/3` (line 5). Lines 15 and 16 extract the entry

```
1  def follower(:cast,
2               %Graft.RequestVoteRPC{term: t,
3                     last_log_index: rpc_lli,
4                     last_log_term: rpc_llt,
5                     candidate_name: candidate},
6               data = %Graft.State{
7                     voted_for: voted_for,
8                     log: [{lli, llt, _} | _tail]})
9    when rpc_llt > llt and voted_for in [nil,
       ↪ candidate] do
10   # reply with vote_granted: true
11   # ...
12
13 def follower(:cast,
14              %Graft.RequestVoteRPC{term: t,
15                    last_log_index: rpc_lli,
16                    last_log_term: rpc_llt,
17                    candidate_name: candidate},
18              data = %Graft.State{
19                    voted_for: voted_for,
20                    log: [{lli, llt, _} | _tail]})
21   when rpc_llt == llt and rpc_lli >= lli and
       ↪ voted_for in [nil, candidate] do
22   # reply with vote_granted: true
23   # ...
```

Lst. 2. Election restriction

```
1  def leader(:cast,
2             rpc = %Graft.AppendEntriesRPCReply{},
3             data) do
4    # ...
5    # lli = rpc.last_log_index
6    # llt = rpc.last_log_term
7    # ci  = data.commit_index
8    # mi  = data.match_index
9    # t   = data.current_term
10
11   commit_index = if (lli > ci) and (llt === t) do
12     number_of_matches =
13       Enum.reduce(mi, 1, fn {server, index}, acc ->
14         if (server !== data.me) and (index >= lli),
15           do: acc+1, else: acc end)
16     if number_of_matches > (data.server_count/2),
17       do: lli, else: ci
18   else
19     ci
20   end
21   # ...
22 end
```

Lst. 3. Committing restriction

from the log, which is then sent as a blocking call to the machine process. The machine defines two callbacks on lines 21 and 22, which are used to create the `Graft.Machine` behaviour. To illustrate, the module implementing this behaviour is set as a module attribute on line 19. (In reality, this is extracted from configuration settings.)

Upon starting, the `GenServer` init callback is invoked (line 24), which returns the value of the *user defined* init function on line 30. After initialisation, the process waits for an `{:apply, entry}` message (line 26) and invokes the user defined `handle_entry` callback (line 27), expecting a reply and the new state to be returned. Finally, the reply is sent back to the server, where it is forwarded to the client.

3.3 Safety

The safety restrictions introduced in 2.3 were implemented by altering two of the previously defined callbacks. Firstly, the *election restriction* is enforced by adding a conditional statement that checks if the candidate is at least as up to date as the recipient of the `RequestVoteRPC` (lst. 2 lines 9 and 21). Only if the conditions hold true can the follower reply with a successfully granted vote. Otherwise, the reply is sent with `vote_granted: false`. This prohibits any followers without the latest updates from being elected leader (thereby enabling them to erroneaously override previous client requests). Secondly, the *committing restriction* is added to the leader's `AppendEntriesRPCReply` handler. Concretely, line 11 of lst. 3 restricts a leader from committing entries belonging to previous terms. If the condition holds, the leader determines whether the commit index should be updated (lines 12–17).

4 Example Application

We build a number of applications atop of our tool to showcase the genericity of Graft, and to demonstrate how effortlessly custom business logic can be integrated with the consensus algorithm, . This section presents a distributed key-value store (KV store) example, the source code for which is publicly available [14]. The online documentation of Graft[1] includes a distributed stack data structure using Graft.

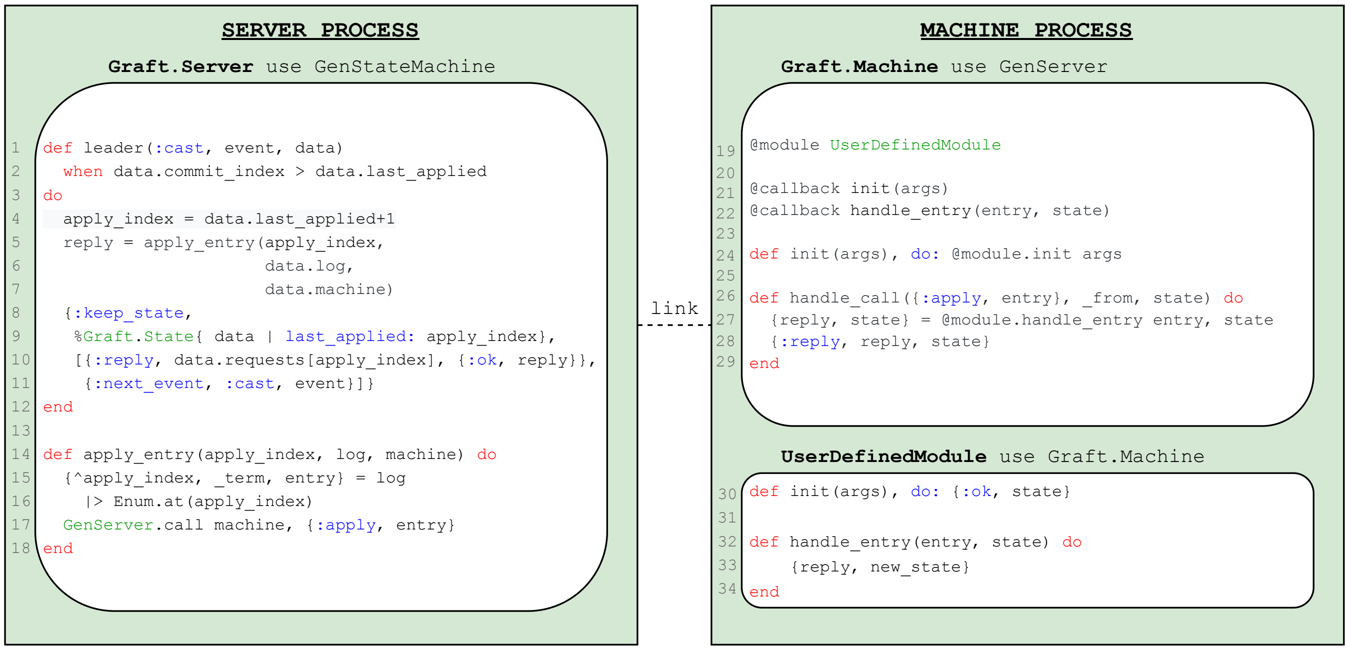
Implementing a Graft KV store involves *(i)* creating an API for client requests, *(ii)* defining how the requests are applied to the replicated state machines using the `Graft.Machine` behaviour, and, *(iii)* configuring the Graft application.

4.1 Set-up

To implement the KV store, Graft is installed as a dependency. It is packaged with Elixir's default package manager hex, and can be installed as a dependency using Mix. Using the `mix new` command with name `diskv`, a new Mix project is created

```
$ mix new diskv
$ cd diskv/
```

[1] https://hexdocs.pm/graft/api-reference.html


SERVER PROCESS
Graft.Server use GenStateMachine
link
MACHINE PROCESS
Graft.Machine use GenServer
UserDefinedModule use Graft.Machine

```elixir
def leader(:cast, event, data)
  when data.commit_index > data.last_applied
do
  apply_index = data.last_applied+1
  reply = apply_entry(apply_index,
                      data.log,
                      data.machine)
  {:keep_state,
   %Graft.State{ data | last_applied: apply_index},
   [{:reply, data.requests[apply_index], {:ok, reply}},
    {:next_event, :cast, event}]}
end

def apply_entry(apply_index, log, machine) do
  {^apply_index, _term, entry} = log
    |> Enum.at(apply_index)
  GenServer.call machine, {:apply, entry}
end
```

```elixir
@module UserDefinedModule

@callback init(args)
@callback handle_entry(entry, state)

def init(args), do: @module.init args

def handle_call({:apply, entry}, _from, state) do
  {reply, state} = @module.handle_entry entry, state
  {:reply, reply, state}
end
```

```elixir
def init(args), do: {:ok, state}

def handle_entry(entry, state) do
    {reply, new_state}
end
```

Figure 3. Communication between the server and machine processes.

This generates the Mix default folder structure and boilerplate code. Graft is added as a dependency from the deps function in the `mix.exs` file. This example uses Graft version `0.3.0`, and should remain compatible for all future versions.

```elixir
defp deps do
  [ {:graft, "~> 0.3.0"} ]
end
```

To ensure Graft starts correctly, it can be added to the list of `extra_applications` in the `application` function.

```elixir
defp application do
  [ extra_applications: [:logger, :graft] ]
end
```

To conclude the set-up, we call the `mix deps.get` function to install the Graft dependency within the current project.

```elixir
1  defmodule Diskv do
2    def start, do: Graft.start
3
4    def put(server, key, value) do
5      Graft.request(server, {:put, key, value})
6    end
7    def fetch(server, key) do
8      Graft.request(server, {:fetch, key})
9    end
10   def delete(server, key) do
11     Graft.request(server, {:delete, key})
12   end
13 end
```

Lst. 4. API to interact with the distributed KV store

4.2 Creating an API

This KV store is to support `start`, `put`, `fetch` and `delete` functions. Lst. 4 shows how Graft is used to create a new API specific to `Diskv` with the aforementioned operations. `Diskv.start` on line 2 initiates the consensus protocol, allowing client requests to be made, whilst the remaining `Diskv.{put,fetch,delete}` specify the structure of the requests (lines 4, 7 and 10). The latter functions accept a `server` parameter, representing a participant of the consensus cluster to which the request is made. The remaining arguments are specific to the respective function.

4.3 Replicated Machine

As demonstrated in fig. 3, the supported client requests of a Graft application are described using the `Graft.Machine` behaviour. Lst. 5 demonstrates the use of the behaviour to specify how the `put`, `fetch` and `delete` requests are handled. Within `lib/machine.ex`, the `Diskv.Machine` module is set to `use Graft.Machine`, informing the compiler that the module should contain the function definitions `init/1` and `handle_entry/2`. Then, the initial state of the machine is set to an empty map on line 5. Finally, `handle_entry/2` pattern matches to the client requests defined in lst. 4 and specifies a response and state update for each. For instance, `Diskv.put` makes a request with the tuple `{:put, key, value}`. Hence, `Diskv.Machine` pattern matches to this tuple on line 8, calculates the new state of the machine by inserting the new key and value, line 9, and returns `{:ok, new_state}`, representing the response to the client and updated state respectfully.

```
defmodule Diskv.Machine do
  use Graft.Machine

  @impl Graft.Machine
  def init([]), do: {:ok, %{}}

  @impl Graft.Machine
  def handle_entry({:put, key, value}, state) do
    {:ok, Map.put(state, key, value)}
  end
  def handle_entry({:fetch, key}, state) do
    {Map.fetch(state, key), state}
  end
  def handle_entry({:delete, key}, state) do
    {:ok, Map.delete(state, key)}
  end
end
```

Lst. 5. Distributed KV store using `Graft.Machine`

```
import Config

config :graft,
  cluster: [
    {:server1, :s1@localhost},
    {:server2, :s2@localhost},
    {:server3, :s3@localhost}
  ],
  machine: Diskv.Machine,
  machine_args: [],
  server_timeout: fn -> :rand.uniform(151)+149 end,
  heartbeat_timeout: 75
```

Lst. 6. Configuration for the Graft application

4.4 Configuration

The final step is to configure the Graft application, which is done through Elixir's `Config` module. Inside a `config.exs` file, we define a key-value list for the `:graft` application. The required keys are described below:

`:cluster` A list of servers making up the consensus cluster. Each server is defined as a tuple containing the name to be assigned to the `Graft.Server` process, and the node on which it resides.

`:machine` The module with the `Graft.Machine` callbacks.

`:machine_args` A list of arguments passed to `init/1`.

`:server_timeout` Generates the leader election timeout.

`:heartbeat_timeout` Frequency at which the leader sends out heartbeat RPCs.

Lst. 6 provides the sample configuration used for diskv. This example uses a consensus cluster comprising of three distributed nodes running on a single physical machine.

4.5 Running the Project

Each node in the cluster can be started by using the command line function `iex --sname <node> -S mix` from within the project directory. Once all nodes are started, the consensus

```
iex(1)> Diskv.start
#=> [:ok, :ok, :ok]
iex(2)> [s1, s2, s3] = [{:server1, :s1@localhost},
                        {:server2, :s2@localhost},
                        {:server3, :s3@localhost}]
#=> ...
iex(3)> Diskv.put s1, :erlang, 2021
#=> :ok
iex(4)> Diskv.fetch s2, :erlang
#=> {:ok, 2021}
iex(5)> Diskv.delete s3, :erlang
#=> :ok
iex(6)> Diskv.fetch s1, :erlang
#=> :error
```

Lst. 7. Using diskv

protocol is initiated by calling `Diskv.start` from any node. Lst. 7 demonstrates `Diskv` being used within the interactive Elixir session from one of the started nodes. To demonstrate the fault-tolerance provided via Graft, one of the nodes can be terminated with `<ctrl-c>`. Client requests continue to return as expected, unless a request is made to the killed node, in which case an error is raised due to the failed connection.

5 Correctness Validation

We validate Graft using a two-pronged strategy that addresses correctness aspects of our implementation. Graft is developed following a test-driven approach [12, 45], to ensure that the consensus logic strictly conforms to the one detailed in sec. 3. We also verify a subset of the properties identified by Ongaro and Ousterhout [42, fig. 3] for our implementation of Graft. Unit testing typically focusses on the *local* functionality of the implemented logic, but makes it challenging to ascertain *global* properties such as the ones in the aforementioned paper. This partly stems from our concurrent setting, where tests are obliged to account for the *interleaved execution* of Graft components. We mitigate this issue by complementing our testing with a verification method that checks the consensus logic of Graft against these global properties *at runtime.*

5.1 Testing

Since the server implementation, `Graft.Server`, instantiates the abstract `GenStateMachine` behaviour, our suite of tests focus on the Graft consensus logic concretised as behaviour callbacks. In particular tests assert that:

(i) the `Graft.Server` state transitions that correspond to the callbacks of `follower/3`, `candidate/3`, and `leader/3`, follow those of fig. 1;
(ii) the server issues the correct responses for the messages it receives while in these states.

In this exposition, we limit ourselves to the tests in lsts. 8 and 9; for the full ExUnit suite, readers are referred to the Graft code repository [15].

```
1  test "candidate transitions to follower on receiving
       ↪  an AERPC from a new leader" do
2    leader = self()
3
4    rpc = %Graft.AppendEntriesRPC{ # Emulated ldr RPC
5      term: 1,
6      leader_name: leader,
7      prev_log_index: 0,
8      prev_log_term: 0,
9      entries: [],
10     leader_commit: 0
11   }
12
13   c_data = %Graft.State{ # Mocked candidate state.
14     me: {:test_server, :nonode@nohost},
15     current_term: 1,
16     voted_for: {:test_server, :nonode@nohost},
17     votes: 1
18   }
19
20   assert {:next_state, :follower, _, _} =
21     Graft.Server.candidate :cast, rpc, c_data
22 end
```

Lst. 8. Asserting server state function inputs/outputs

```
1  test "follower replies with false to vote request
       ↪ containing outdated term number" do
2    candidate = self()
3
4    rpc = %Graft.RequestVoteRPC{ # Emulated can. RPC
5      term: 1,
6      candidate_name: candidate,
7      last_log_index: 0,
8      last_log_term: 0
9    }
10
11   f_data = %Graft.State{ # Mocked follower state.
12     me: {:test_server, :nonode@nohost},
13     current_term: 2
14   }
15
16   Graft.Server.follower :cast, rpc, f_data
17
18   assert_receive {_, %Graft.RequestVoteRPCReply{
19     term: 2,
20     vote_granted: false
21   }}
22 end
```

Lst. 9. Asserting message requests received by server

Lst. 8 tests the case where a "candidate transitions to the follower state when receiving an AppendEntriesRPC from a *new* leader". On line 2, the invocation to `self/0` obtains the process ID (PID) of the test runner process that emulates the Graft server in the leader state. This PID is used to fabricate the AppendEntriesRPC message (line 4) that the leader issues to the candidate, whose internal state is mocked on line 13. Note that the mocked candidate server process state and RPC payload of line 4 share the *same* term number, *i.e.*, 1. Function `candidate/3` on line 21 simulates the reception and processing of the AppendEntriesRPC message sent by the leader to the server in candidate state. The assertion on line 20 confirms that the next state returned by `candidate/3` obliges `Graft.Server` to transition to the follower state.

Whereas lst. 8 operates on the arguments and return values of the GenStateMachine state callbacks implemented by `Graft.Server`, lst. 9 tests the exchange of messages between the server components of Graft. Lst. 9 replicates the scenario where a "follower replies with `false` to voting requests containing an outdated term number", *i.e.*, the term in the RPC < follower term. The PID of the test runner obtained on line 2 emulates the candidate process. Line 4 shows the creation of the voting request that the candidate submits to the follower; the internal state of the follower is mocked on line 11. The invocation of `follower/3` on line 16 simulates the server in follower state receiving the voting request of line 4 issued by the candidate. The corresponding statement on line 18 asserts that the reply to the voting request does not grant the vote.

5.2 Runtime Verification

Our difficulty in testing scenarios comprised of interacting Graft server components is reflected in the relatively low line coverage of $\approx 51\%$ (obtained using the utility ExCoveralls). We thus complement tests with a post-deployment verification technique called Runtime Verification (RV) to redress this shortcoming. RV [11, 20] is a lightweight approach that uses *monitors* to analyse the *current* system execution observed as a stream of runtime events, known as the *trace*. RV monitors[2] are encapsulated software components that are typically synthesised automatically from formal descriptions of correctness properties. These are instrumented with the system under scrutiny to analyse its execution, processing trace event sequences to reach a *verdict* (relating to the property being checked). The monitors we consider for verifying Graft are synthesised from *safety properties*, *i.e.,* properties stating that "nothing bad ever happens" [5], flagging either *rejection* or *inconclusive* verdicts [1, 3, 9, 22]. A rejection verdict corresponds to a property *violation* (similar to a failed assertion in unit testing); inconclusive verdicts are issued when the execution trace does not provide enough evidence to conclude that the system violates the property.

We use the RV tool called detectEr [7, 8] to verify a subset of the properties identified by Ongaro and Ousterhout [42]. detectEr is developed in Erlang, and targets asynchronous component systems that execute on the Erlang Virtual Machine [6, 16] (BEAM). To the best of our knowledge, it is the only RV tool available for this ecosystem that is still supported. detectEr synthesises executable *monitor code* from

[2]These are unrelated to OTP monitors.

properties specified in terms of the logic sHML [2, 4, 21], the safety fragment of the modal μ-calculus [35, 38]. sHML describes the properties of systems based on the events they exhibit; the logic has been shown to be maximally expressive for describing safety properties [21, 22]. At runtime, monitors observe system events in the form of an *execution trace.* detectEr instruments the monitor code it synthesises to execute with the system, collecting trace events by leveraging the tracing infrastructure of the BEAM. Monitors analyse these events incrementally to reach a verdict. In the sequel, we omit the full detail regarding detectEr and limit ourselves to the particulars that illustrate how the verification of Graft is conducted. For a comprehensive exposition, the interested reader should consult Attard et al. [8].

Our analysis of the Graft implementation presented in sec. 3 considers the following subset of properties identified by Ongaro and Ousterhout [42]:

- P_1 *Election safety:* at most one leader can be elected in a given term;
- P_2 *Leader append-only:* leaders never overwrite or delete entries in their logs — they only append new entries;
- P_3 *Log matching:* if two logs contain an entry with the same index and term, then the logs are identical in all entries up to the given index;
- P_4 *Leader completeness:* if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-number terms.

The remaining fifth property, stated below, is not verified using detectEr because we encountered observability issues when attempting to monitor the invariant.

- P_5 *State machine safety:* if a server applies a log entry for a particular index, no other server ever applies a different entry for the same index.

Runtime verifying this property necessitates the monitoring of function calls and function returns, which are not supported by the current version of the RV tool. This prohibits the observation of key state changes of a process in relation to the property. We leave the verification of this property for future work.

Consider property P_1, formalised as the sHML formula φ_1 below:

$$\max \underline{x}.\ \bigwedge \begin{pmatrix} \overbrace{[\mathit{Ldr}_1 : \_ ! \langle :\mathsf{AE}, \mathit{Trm}_1\rangle \text{ when true}]}^{①} \max \underline{y}.\ \bigwedge \underbrace{\begin{pmatrix} [\mathit{Ldr}_2 : \_ ! \langle :\mathsf{AE}, \mathit{Trm}_2\rangle \text{ when } \mathit{Ldr}_1 \neq \mathit{Ldr}_2 \wedge \mathit{Trm}_1 = \mathit{Trm}_2]\, \mathsf{ff}, \\ [\_]\,\underline{y} \end{pmatrix}}_{③}, \\ \underbrace{[\_]\,\underline{x}}_{②} \end{pmatrix} \qquad (\varphi_1)$$

The key construct in sHML is the modal formula $[p \text{ when } c]\,\varphi$, where: *(i)* p is an *event pattern* that corresponds to events the system exhibits; *(ii)* c is a *constraint* on the variables bound by p; *(iii)* φ is the continuation formula. In formula φ_1 we only use the event pattern $\mathit{Snd} : \mathit{Rcv} ! \mathit{Msg}$ that matches send actions. Variables inside patterns are *instantiated* with data values learnt at runtime, *e.g.* Snd and Rcv are assigned the PIDs of the sender and receiver processes, whereas Msg is instantiated with the message payload of the send action. Pattern variables also *bind* the free variables inside constraints, and this binding scope *extends* to the continuation formula φ. To facilitate specification, pattern matching in detectEr follows that of Elixir and Erlang, where atoms are matched directly, and the 'don't care' pattern (_) matches any value. We say that a system satisfies the formula $[p \text{ when } c]\,\varphi$ *whenever* it exhibits an event that matches pattern p, fulfils the constraint c, and the ensuing system behaviour satisfies φ.

Formula φ_1 expresses the invariant P_1 via the recursion binder $\max x$. The action pattern in the modality of sub-formula ① matches send actions whose message payload consists of *tagged* pairs of the form $\langle :\mathsf{AE}, \mathit{Trm}_1\rangle$. Variables Ldr_1 and Trm_1 are respectively instantiated with the PID of the Graft server process in the leader state, and the term number embedded in the RPC message. Receiver PIDs are unimportant in formula φ_1, and disregarded via the pattern _. Sub-formula ② acts as a filter that 'eats up' non-matching events by recursing on variable x to unfold formula φ_1.

The *nested* recursion binder $\max y$ guarding sub-formula ③ serves a similar purpose to the outer $\max x$. When a send action satisfies modality ①, its continuation $\max y\ \bigwedge(\ldots)$ either leads to a violation or to an unfolding of sub-formula ③. This depends on the subsequent events analysed. The pattern $\mathit{Ldr}_2 : \_ ! \langle :\mathsf{AE}, \mathit{Trm}_2\rangle$ in the first modality of sub-formula ③ matches send actions, instantiating variables Ldr_2 and Trm_2 as before. Ldr_1 and Trm_1, whose binding scope in modality ① *extends* to ③, are compared against Ldr_2 and Trm_2 in the constraint when $\mathit{Ldr}_1 \neq \mathit{Ldr}_2 \wedge \mathit{Trm}_1 = \mathit{Trm}_2$. Since no system can satisfy falsehood, ff, formula φ_1 is *violated* when the constraint is satisfied, *i.e.,* there are two different leader PIDs for the same term number. Property formula φ_1 is *not* violated when the system action that satisfies the modality $[\mathit{Ldr}_2 : \_ ! \langle :\mathsf{AE}, \mathit{Trm}_2\rangle \text{ when } \mathit{Ldr}_1 \neq \mathit{Ldr}_2 \wedge \mathit{Trm}_1 = \mathit{Trm}_2]$ is not performed. This case is handled by $[\_]\,y$.

For the sake of our exposition, formula φ_1 omits the auxiliary clauses that account for the interleaved execution of Graft components, and extraneous trace events due to OTP libraries. We also simplify the Graft RPC message payloads in formula φ_1, denoting them by tagged tuples, and use the shorthand AE in lieu of `AppendEntriesRPC`. The full sHML specification of formula φ_1, together with the formalisations of properties P_2–P_4 can be found in the Graft code repository [15]. The present Graft implementation has never violated any of these properties at runtime; the only exception is when faults were intensionally injected for testing purposes.

6 Performance

We evaluate Graft by comparing its performance against two implementations. The first, LogCabin, is developed by Ongaro and Ousterhout [42] and features as the C++ implementation behind their performance metrics. This comparison allows us to expose any performance benefits or detriments resulting from architectural and paradigmatic differences between the implementations. The second comparison is against Ra, a battle-tested state-of-the-art implementation developed by RabbitMQ in Erlang. Their project is well maintained, thoroughly tested, and (it records 150,000+ downloads to date). Since Ra runs on the BEAM, the potential biases of the first comparison are eliminated, allowing for a more precise evaluation of performance.

6.1 Methodology

A key performance parameter for Raft is the time a cluster takes to elect a leader. This metric is important because it correlates closely to the period in which the cluster is unresponsive (*i.e.,* unable to handle client requests). Ongaro and Ousterhout [42] measure how quickly elections in their implementation converge based on a variety of election timeout ranges. These ranges refer to a minimum and maximum time a follower waits before starting an election (*e.g.* a timeout range of 150–300ms implies that followers will choose a timeout value randomly and uniformly within this range). The experiments of Ongaro and Ousterhout [42] are run on five nodes, use heartbeat timeouts equal to half the minimum election timeout value, and make some nodes ineligible for leadership through log inconsistencies.

We replicate each of these parameters for both Graft and Ra, but simulate ineligible nodes by crashing them, as opposed to tampering with their logs; Ra does not offer the functionality of forcefully manipulating the log of a server. Concretely, our experiments run multiple simulations for each timeout range, where the consensus cluster *(i)* elects a leader, *(ii)* crashes it, and *(iii)* records the time taken to notice the crash and elect a replacement. More precisely, leader downtime is measured by taking timestamps at specific points in the algorithm's operations. The start time is recorded exactly prior to calling `Supervisor.terminate_child/2` (used to simulate a crashing node), whereas the end time is taken at the entry point for the leader callback in the module using the `GenStateMachine` behaviour. The difference is calculated and persisted by a separate node to minimise performance overheads. This described process reflects the actions of a single iteration within its respective simulation. Our simulations are conducted with a cluster of 5 servers and repeated 1000 times for each timeout range — destroying and rebuilding the cluster after each iteration. Servers run in distributed OTP nodes on a single machine housing a 6-core CPU with 12 threads of execution.

6.2 Results

Fig. 4 plots leader downtimes for the Graft and Ra experiments against a cumulative probability distribution (average case results are seen at the 50% mark). The results for our experiments follow the same curve as those presented in Ongaro and Ousterhout [42]. We note that for both LogCabin and Graft, a number of results indicate that a leader is elected *before* the minimum election timeout value (*e.g.* for Graft, this occurs with 55% probability for the 12–24ms range, 74% for 150–155ms, and 10% for 150–300ms). This behaviour depends on the arrival of the last heartbeat before the initial leader is crashed. In the best case scenario, half a follower's election timeout elapses before the leader is halted. Hence, this phenomenon occurs more often for timeout ranges with a small difference between upper and lower bounds.

Ra never elects a new leader before the minimum timeout value, unlike Graft and LogCabin. This is due to Ra not using heartbeat RPCs to maintain the leadership of a server, instead opting for the use of OTP monitors. With this approach, each time a leader is elected, an exit-trapping Erlang monitor process is spawned to inform the remainder of the cluster whenever the leader goes down. This modification helps to prevent unnecessary elections in the presence of network latency, at the cost of delayed elections. The drawback of delayed elections is primarily pronounced in simulations with a small upper and lower bound difference, since these are the most likely to be elected before the minimum timeout value; for timeout ranges with a larger interval, the performance difference becomes negligible. We conjecture that this observation may be due to the optimisations introduced by RabbitMQ over the years of Ra's development, in order to make up for the shortcomings of their monitor-based approach.

From the experiments reported in Fig. 4, we can conclude that Graft performs comparably to both LogCabin and Ra. In the average case, Graft performs better for each timeout range when compared to the former, while exhibiting comparable results when compared against the latter.

7 Future Work

Graft provides a foundation on which a more robust generic purpose library can be built. The following gives some insight into possible future work and upgrades for the tool.

7.1 Implementation Correctness

This paper employed rudimentary methods of providing correctness assurances to Graft that are geared more towards uncovering errors as opposed to ascertaining their absence (*i.e.,* testing and RV). Although effective, the next step would be to employ more exhaustive verifications methods. These are typically based on model-checking techniques [10, 26, 34, 44], logic and theorem provers [13, 18, 46] and process

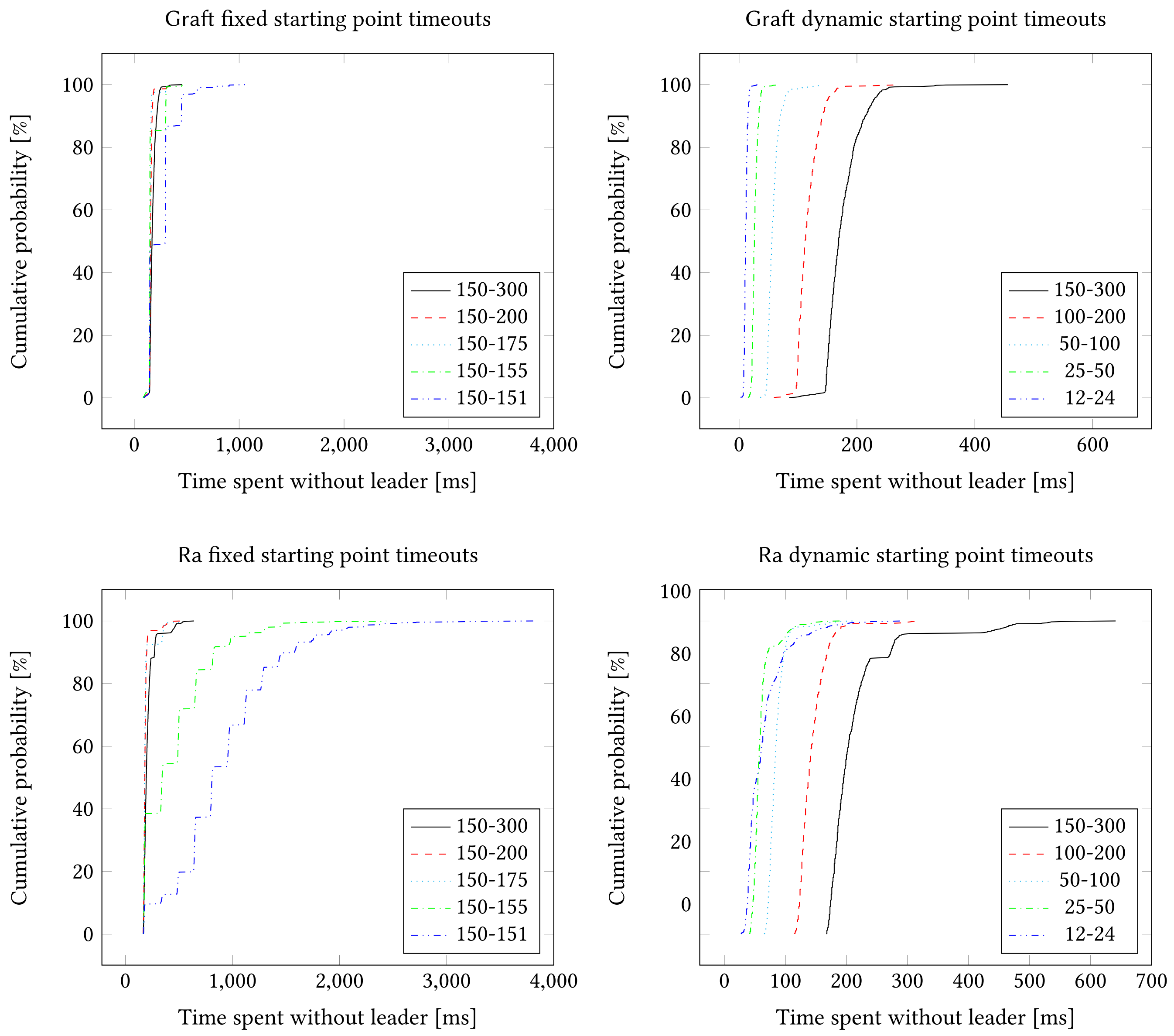


Figure 4. Performance evaluation results

calculi modelling [23, 25, 36]. We believe that our current implementation lends itself well to this extended analysis.

7.2 Raft Extensions

In Ongaro and Ousterhout [42], two extensions to the algorithm are provided that aim to increase Raft's practicality. The membership changes extension allows the cluster to be dynamically reconfigured during runtime, and log snapshots provide a safe method for logs to be persisted to secondary storage. Graft's practicality would benefit from implementing these extensions. One could also integrate log snapshots with Ecto [47] to provide customisation for data persistence.

7.3 Monitor Overhead

Although RV is a lightweight verification technique, the instrumentation of runtime monitors comes with a performance overhead. We believe this overhead to be insignificant when compared to the execution of the election process for small clusters. However, it may be beneficial to determine at what cluster size the observed overhead begins to significantly interfere with elections (which rely on timeouts). Additionally, future work may consider taking performance metrics based on the time taken to handle client requests, both with and without the instrumented monitors. This should give further insight into any performance overhead introduced by the RV tool.

7.4 Machine Fault-tolerance

By placing the machine process under a supervisor, Graft can very quickly re-spawn it if runtime errors occur whilst applying entries. However, Raft's specification dictates that entries be committed before they are applied, thus a fault-causing entry would already be replicated across a majority of nodes before the system suffers from its fault. A further degree of fault-tolerance may be provided if Raft were extended to cater for these errors (possibly by devising a method for leaders to warn other servers of potentially harmful entries).

7.5 Parallelising Processes

Currently, Graft spawns two processes per node, these being the server and machine. So far, communication amongst these processes is synchronous. Future work may consider exploring the safety and performance difference of non-blocking calls between these processes.

8 Related Work

Graft is not the only Elixir implementation of Raft. A list of open-source implementations [41] of the algorithm identifies at least three other Elixir implementations. Rafute [30] implements Raft's leader election and log replication protocols for a *specific* key-value store purpose. RaftKV [33] is another *special purpose* implementation of Raft, that it is widely used (8,000+ downloads) and extensively tested. Raft [32] (not to be confused with Raft, the algorithm) is an Elixir implementation allowing for *custom defined* replicated machines, similar to Graft; they also achieve this genericity through a behaviour that allows users to define how client requests are handled. However, this project appears *not* to be maintained any longer: it appears *not* to have been heavily tested, nor is there any information available regarding its performance.

As previewed in Sec. 6, Ra is a notable general purpose implementation in Erlang. Apart from providing leader election and log replication, it also implements extensions of the algorithm supporting cluster membership changes (one server at a time), log compaction (tailored for RabbitMQ-specific use) and snapshot installation. This project is thoroughly tested, has 150,000+ downloads and is used internally by RabbitMQ to maintain replicated queues. As a consequence, we treated Ra as our benchmark for BEAM Raft implementations. When compared to Ra, Graft provides post-deployment guarantees (through the instrumentation of runtime monitors) which, to our knowledge, is not offered by any other BEAM implementation. Moreover, since the Graft.Machine behaviour is natively constructed in terms of existing Elixir behaviours, we believe that our implementation is more readily accessible to the Elixir community.

The benchmark data used for our performance evaluation of sec. 6, taken from Ongaro and Ousterhout [42], has been validated by other studies. In Howard et al. [29], the authors successfully reproduce the original performance metrics obtained from [42] using an OCaml implementation. This work supports the claim that Raft is easier to understand than Paxos at a high level, but also states that the description provided in Ongaro and Ousterhout [42] omits subtleties that hinder the development process. Finally, Verdi [46] is a framework that extracts OCaml implementations out of formal specifications of such algorithms, and checks for their correctness against a fault-model. This was used to create the first formally verified implementation of Raft [48], a process which involved discovering and proving 90 system invariants. Graft, although not as robust, attempts to provide similar correctness guarantees for an implementation built on the BEAM, whilst maintaining performance levels comparable to the state-of-the-art.

9 Conclusion

This paper presents Graft, a generic tool for creating distributed consensus clusters running the Raft algorithm using state machines in Elixir. Our work describes the process of developing such an algorithm using the actor paradigm adopted by Elixir, and details our main design choices and use of OTP behaviours (sec. 3). This is coupled with a demonstration showcasing how the tool can be used to implement a distributed key-value store (sec. 4).

We employ a two-pronged approach to verify aspects of Graft's correctness — consisting of a test-driven development phase (sec. 5.1), and a formal verification by means of runtime monitors (sec. 5.2). Finally, we evaluate the performance of the tool by comparing it against two other implementations (sec. 6). Graft is observed to perform comparably against both the C++ implementation developed by Ongaro and Ousterhout [42], and Ra, a state-of-the-art implementation built in Erlang by RabbitMQ.

References

[1] Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir. 2017. Monitoring for Silent Actions. In *FSTTCS (LIPIcs, Vol. 93)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:14. https://doi.org/10.4230/LIPIcs.FSTTCS.2017.7

[2] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Sævar Örn Kjartansson. 2020. Determinizing monitors for HML with recursion. *J. Log. Algebraic Methods Program.* 111 (2020), 100515. https://doi.org/10.1016/j.jlamp.2019.100515

[3] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. 2019. Adventures in monitorability: from branching to linear time and back again. *Proc. ACM Program. Lang.* 3, POPL (2019), 52:1–52:29. https://doi.org/10.1145/3290365

[4] Luca Aceto, Ian Cassar, Adrian Francalanza, and Anna Ingólfsdóttir. 2018. On Runtime Enforcement via Suppressions. In *CONCUR (LIPIcs, Vol. 118)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 34:1–34:17. https://doi.org/10.4230/LIPIcs.CONCUR.2018.34

[5] Bowen Alpern and Fred B. Schneider. 1987. Recognizing Safety and Liveness. *Distributed Comput.* 2, 3 (1987), 117–126. https://doi.org/10.1007/BF01782772

[6] Joe Armstrong. 2007. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf.

[7] Duncan Paul Attard. 2021. *detectEr Tutorial.* Retrieved May 24, 2021 from https://duncanatt.github.io/detecter/

[8] Duncan Paul Attard, Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. 2021. Better Late Than Never or: Verifying Asynchronous Components at Runtime. In *FORTE (LNCS, Vol. 12719)*. Springer, 207–225. https://doi.org/10.1007/978-3-030-78089-0_14

[9] Duncan Paul Attard and Adrian Francalanza. 2016. A Monitoring Tool for a Branching-Time Logic. In *RV (LNCS, Vol. 10012)*. Springer, 473–481. https://doi.org/10.1007/978-3-319-46982-9_31

[10] Christel Baier, Nathalie Bertrand, and Philippe Schnoebelen. 2006. Symbolic Verification of Communicating Systems with Probabilistic Message Losses: Liveness and Fairness. In *FORTE (LNCS, Vol. 4229)*. Springer, 212–227. https://doi.org/10.1007/11888116_17

[11] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to Runtime Verification. In *Lectures on Runtime Verification*. LNCS, Vol. 10457. Springer, 1–33. https://doi.org/10.1007/978-3-319-75632-5_1

[12] Kent Beck. 2002. *Test Driven Development by Example.* Addison-Wesley.

[13] Martin Biely, Pamela Delgado, Zarko Milosevic, and André Schiper. 2013. Distal: A framework for implementing fault-tolerant distributed algorithms. In *DSN*. IEEE Computer Society, 1–8. https://doi.org/10.1109/DSN.2013.6575306

[14] Matthew Alan Le Brun. 2021. diskv. https://github.com/MatthewAlanLeBrun/diskv

[15] Matthew Alan Le Brun. 2021. Graft: A Raft Implementation in Elixir. https://github.com/MatthewAlanLeBrun/graft

[16] Francesco Cesarini and Simon Thompson. 2009. *Erlang Programming: A Concurrent Approach to Software Development.* O'Reilly Media.

[17] CNCF. 2021. TiKV: A Distributed Transactional Key-Value Database. https://github.com/tikv/tikv

[18] Cezara Dragoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. 2014. A Logic-Based Framework for Verifying Consensus Algorithms. In *VMCAI (LNCS, Vol. 8318)*. Springer, 161–181. https://doi.org/10.1007/978-3-642-54013-4_10

[19] etcd Team. 2021. etcd: A Distributed, Reliable Key-Value Store for the Most Critical Data of a Distributed System. https://github.com/etcd-io/etcd

[20] Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. 2017. A Foundation for Runtime Monitoring. In *RV (LNCS, Vol. 10548)*. Springer, 8–29. https://doi.org/10.1007/978-3-319-67531-2_2

[21] Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. 2015. On Verifying Hennessy-Milner Logic with Recursion at Runtime. In *RV (LNCS, Vol. 9333)*. Springer, 71–86. https://doi.org/10.1007/978-3-319-23820-3_5

[22] Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. 2017. Monitorability for the Hennessy-Milner logic with recursion. *Formal Methods Syst. Des.* 51, 1 (2017), 87–116. https://doi.org/10.1007/s10703-017-0273-z

[23] Adrian Francalanza and Matthew Hennessy. 2007. A Fault Tolerance Bisimulation Proof for Consensus (Extended Abstract). In *ESOP (LNCS, Vol. 4421)*. Springer, 395–410. https://doi.org/10.1007/978-3-540-71316-6_27

[24] Adrian Francalanza and Matthew Hennessy. 2007. A theory for observational fault tolerance. *J. Log. Algebraic Methods Program.* 73, 1-2 (2007), 22–50. https://doi.org/10.1016/j.jlap.2007.03.003

[25] Rachele Fuzzati, Massimo Merro, and Uwe Nestmann. 2007. Distributed Consensus, revisited. *Acta Informatica* 44, 6 (2007), 377–425. https://doi.org/10.1007/s00236-007-0052-1

[26] Jan Friso Groote and Jeroen J. A. Keiren. 2021. Tutorial: Designing Distributed Software in mCRL2. In *FORTE (LNCS, Vol. 12719)*. Springer, 226–243. https://doi.org/10.1007/978-3-030-78089-0_15

[27] Hashicorp. 2021. Raft: Golang Implementation of the Raft Consensus Protocol. https://github.com/hashicorp/raft

[28] Heidi Howard and Richard Mortier. 2020. Paxos vs Raft: have we reached consensus on distributed consensus?. In *PaPoC@EuroSys*. ACM, 8:1–8:9. https://doi.org/10.1145/3380787.3393681

[29] Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. 2015. Raft Refloated: Do We Have Consensus? *ACM SIGOPS Oper. Syst. Rev.* 49, 1 (2015), 12–21. https://doi.org/10.1145/2723872.2723876

[30] Yuki Ito. 2016. *Rafute: An implementation of Raft Consensus Algorithm in Elixir.* Retrieved May 6, 2021 from https://github.com/mururu/rafute

[31] Sasa Juric. 2019. *Elixir in Action.* Manning Publications.

[32] Chris Keathley. 2018. *Raft: An Elixir implementation of the raft consensus protocol.* Retrieved May 6, 2021 from https://github.com/toniqsystems/raft

[33] Shunsuke Kirino. 2019. *RaftKV: An Elixir library to store key-value pairs in a distributed, fault-tolerant, self-adjusting data structure.* Retrieved May 6, 2021 from https://github.com/skirino/raft_kv

[34] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. 2019. TLA+ model checking made symbolic. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 123:1–123:30. https://doi.org/10.1145/3360549

[35] Dexter Kozen. 1982. Results on the Propositional μ-Calculus. In *ICALP (LNCS, Vol. 140)*. Springer, 348–359. https://doi.org/10.1007/BFb0012782

[36] Morten Kühnrich and Uwe Nestmann. 2009. On Process-Algebraic Proof Methods for Fault Tolerant Distributed Systems. In *FMOODS/-FORTE (LNCS, Vol. 5522)*. Springer, 198–212. https://doi.org/10.1007/978-3-642-02138-1_13

[37] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169. https://doi.org/10.1145/279227.279229

[38] Kim Guldstrand Larsen. 1990. Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion. *TCS* 72, 2&3 (1990), 265–288.

[39] Nancy A. Lynch. 1996. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[40] Diego Ongaro. 2021. Log Cabin. https://github.com/logcabin/logcabin

[41] Diego Ongaro. 2021. *The Raft Consensus Algorithm.* Retrieved May 6, 2021 from https://raft.github.io/#implementations

[42] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference*. USENIX Association, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[43] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (1990), 299–319.

[44] Ilina Stoilkovska, Igor Konnov, Josef Widder, and Florian Zuleger. 2019. Verifying Safety of Synchronous Fault-Tolerant Algorithms by Bounded Model Checking. In *TACAS (2) (LNCS, Vol. 11428)*. Springer, 357–374. https://doi.org/10.1007/978-3-030-17465-1_20

[45] David Thomas and Andrew Hunt. 2019. *The Pragmatic Programmer: Your Journey to Mastery.* Addison-Wesley.

[46] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*. ACM, 357–368. https://doi.org/10.1145/2737924.2737958

[47] Darin Wilson and Eric Meadows-Jonsson. 2019. *Programming Ecto: Build Database Apps in Elixir for Scalability and Performance.* Pragmatic Bookshelf.

[48] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. 2016. Planning for change in a formal verification of the raft consensus protocol. In *CPP*. ACM, 154–165. https://doi.org/10.1145/2854065.2854081