UNIVERSITY OF TRENTO

Raft Simulator Evaluation

Paolo Chistè, Matteo Franzil, Andrea Stedile

February 9, 2022

ABSTRACT.

Raft is a distributed consensus algorithm for managing a replicated log. It is an efficient, understandable algorithm that is easier to implement than protocols such as Paxos, and offers equivalent performance and guarantees. In this paper, we describe the implementation of a simulator of Raft using the Java language and Akka actor toolkit. Our simulator comprises a Graphical User Interface, which provides a real-time view of the state of the servers in the cluster, allows the user to crash and resume the servers and submit new commands to be replicated.

1 INTRODUCTION

The **distributed consensus problem** is an unsolvable problem in Computer Science. Even with the most benign of assumptions, it is possible to prove that any algorithm that tries to solve the problem can enter in an inconsistent state by simply delaying the delivery of a message. [1].
However, this theoretically impossible problem is solvable "in practice", by relaxation of one or more of its requirements. Paxos [2] is one of the first of these algorithms. Many modern implementations are based on Paxos, but with major modifications to its initial description. This is in part motivated by the intrinsic complexity of the Paxos protocol. in part to the unorthodox paper where the protocol is presented [1].
Raft [4] is a protocol that aims at solving the same problem (distributed consensus), but with a specification that is easier to understand. It is similar to Paxos: both are based on the idea of a log that is replicated among the participants and on the use of a leader as a source of truth for the other participants. Distributed log replication protocols are usually used to replicate operations on multiple servers; each entry in the log is the command for a state machine. When it is safe to do so, the state machine executes an entry in the log. Non all machines execute the same command at the same time, but *eventually* all state machines reach the same state.
The report is organized as follows. Section 2 provides a brief insight on the Raft properties and how the protocol manages to successfully implement distributed consensus. Section 3 shows our implementation, outlining our key design decisions in the architecture and how they were integrated into the project. Section 4 provides information on how to downloading and starting the simulation, and is corredated with a handful of example screenshots. Finally, Section 5 contains the conclusions to the report.

[1] It outlines the algorithm as a series of archeological discoveries in the Paxos Greek island, which narrated how the island's dysfunctional parliament managed to work.

2 THEORY BACKGROUND

The Raft protocol divides the initial problem into two separate problems: leader election and log replication. The leader election is solved with a voting procedure. The log replication is based on maintaining a set of constraints on the contents on each server's log, guaranteeing the consistency of the replicated log entries. The protocol is based on two RPCs: RequestVoteRPC (for leader election) and AppendEntriesRPC (for log replication). The two phases follow each others inside a *term*.
The servers that participate in the protocol may take three roles: *leader, follower, or candidate*.

- The leader is responsible for linearizing the insertion of log entries on the other machines. It receives the entries to append from the clients, and then proceeds to replicate them.
- The followers respond to RPCs from the leader.
- The candidate role is required only when preparing to elect a new leader (see 2.1).

Each server may take only one role at the time.

2.1 Leader Election

Algorithm 1: Arguments of the RequestVoteRPC

```
bool RequestVoteRPC(
term,
candidateId,
lastLogIndex, // index of the last log entry (i.e: the most recent one)
lastLogTerm // term of the last log entry
) ;
```

The outline of the leader election protocol is simple. Any server can start the leader election procedure after a certain time has passed without receiving any messages (indicating that the previous leader is dead or not connected anymore). This "candidate" server sends a RequestVoteRPC to all other servers (which may be in any other state), and they decide to grant or not the vote to the candidate. If a majority of servers granted their votes, the candidate becomes the new leader. Otherwise, it starts a new election. The timeouts on each server are usually initialized with some random offset, to avoid simultaneous elections. Simultaneous elections do not result in more than one leader being elected, but they may split the majority and the election would not produce a leader. A server may only vote once in a single term, thus it is granted that only one leader will be elected in a single term.

2.2 Log Replication

Algorithm 2: Arguments of the AppendEntriesRPC

```
bool AppendEntriesRPC(
term,
leaderId,
prevLogIndex, // index of the last log entry
prevLogTerm, // term of the last log entry
entries[], // entries to append
leaderCommit // leader's commitIndex
);
```

After electing a leader for the cluster, the log replication phase can begin. When the leader has new entries appended to its own log, it sends an AppendEntriesRPC to all followers. The entry is appended if it satisfies

a series of checks (see 2.2.1), otherwise the RPC fails. If the RPC succeeds on the majority of the followers, the entry is considered committed. The log replication phase ends if one of the followers decides to start a new election; the protocol returns in the leader election phase, until a new leader is elected.

2.2.1 When to append log entries

During normal operation (no messages are lost or delayed) the followers can append entries to their local log just as they receive them. However, if messages are lost or delayed, the followers need to be careful in what they add. For example, if a follower receives entries with index $\{1,2,3,4\}$, but the entry with index 3 is lost, the entries in the local log of the follower will be $\{1,2,4\}$, and the logs would diverge. Instead, a follower appends an entry if it satisfies the following requirements (see algorithm 3):

1. It has a term number $\geq$ than the current term number
2. The indicated previous entry has the same term number as the previous entry on the leader

These requirements ensure that a given follower appends entries coming from the current leader and are applied in the same order as all other followers. If the follower cannot append the entries, the RPC fails. The leader does not "give up" on a failed RPC but instead sends previous entries from its own local log (see algorithm 4). By following this scheme, the leader will eventually find an entry that can be appended on the follower, and then will send the entries following that entry.
A follower needs a mechanism to correct its own log when appending new entries. When adding a set of new entries, if there is a new entry with the same index and the same term as an entry already present in the log, the new entry replaces the old one. All the entries that follow the replaced one are deleted. This allows for a follower to correct the contents of its log, if is elected a leader that has a different log from the previous one (within certain limits; see 2.2.2).

Algorithm 3: Procedure used by a follower to verify if it can add a new set of entries to its own localLog

```
addEntries(newEntries,term,prevLogIndex,leaderCommit) begin
    previousEntry ← localLog[prevLogIndex];
    if term ≥ currentTerm ∨ previousEntry.term ≠ prevLogTerm then
        RPC fails; reply false to leader and terminate;
    else
        if entryConflicts(localLog,newEntries) then
            deleteFollowers();
        end
        appendEntriesToLog(newEntries);
        if leaderCommit ≥ commitIndex then
            // if the leader already committed entries ahead of us, we can safely commit them
            commitIndex ← min(leaderCommit, newEntries.lastIndex());
        end
    end
end
```

2.2.2 Commits

The Raft protocol maintains an index to the most recent committed entry. An entry *e* is committed when a majority of servers have *e* in their logs, with the same index and with the same term. This implies that all parts of the log with `index < commitIndex` are present on all servers, all with the same term, all with the same index.
However, the phrase "a majority of servers have that entry in their logs" is misleading. On a first level, it

Algorithm 4: Procedure used by a leader to update its internal status based on the result of AppendEntriesRPC from a follower

```
updateInternalStatus(follower_id) begin
    if RPC was successful then
        nextIndex[follower_id] ← nextIndex[follower_id] + 1;
        matchIndex[follower_id] ← matchIndex[follower_id] + 1;
    else
        nextIndex[follower_id] ← nextIndex[follower_id] - 1;
        retry RPC with different nextIndex;
    end
    /* Either way, success or failure, we try to update the leader's commitIndex.
    We look if exists a majority of values of matchIndex, such that matchIndex is greater than the
      current commit index. In this way, we make sure that we increase the commitIndex only if we
      have previously received a positive RPC from the majority of followers */
    for N ← commitIndex; N < localLog.size(); N ← N + 1 do
        aheadFollowers ← 0;
        // search for followers that have a known matchIndex greater than the current N
        for i ← 0; i < matchIndex.size(); i ← i + 1; do
            if matchIndex[i] >= N ∧ localLog[N].term == currentTerm then
                aheadFollowers ← aheadFollowers + 1;
            end
        end
        if aheadFollowers ≥ (numberOfFollowers / 2) then
            commitIndex ← N;
        end
    end
end
```

seems that the leader reads the logs of the followers and then decides which entry is committed. But a more careful read of the protocol brings out an important detail: the leader uses *its own state* to decide which entry to commit, nothing else. The state of the logs is stored inside the `matchIndex[]` array; each entry contains the index of the higher entry known to be replicated on the server with id *i*. The leader knows that an entry is replicated when receives the response to the AppendEntriesRPC. By virtue of being replicated on a majority of servers, a committed entry is 'safe'; even if the leader changes, it is guaranteed that any committed entry will eventually be present on all followers.

3 Simulator/Implementation Architecture

The ultimate goal of our project was to create a simple Graphical User Interface, enabling the user to:

- Create a Raft cluster composed by the desired number of servers;
- Submit new commands to be replicated by the servers;
- Simulate a crash (and recovery) of a server;
- Visualize any change in state of the servers.

Initially, we started the development using Python and the Pykka actor library [5], but we have later rewritten the code in Java using the Akka [3] actor library. Eventually, we have settled on using Akka Typed[2] as our API of choice, which is convenient for implementing state machines.
As shown in the figure below, the codebase is organized in two packages: the `Raft` package contains the code at the heart of the protocol; the `gui` package implements the simulator.

```
src/main/it/unitn/ds2
   |- gui
   |  |- commands
   |  |- components
   |  |- model
   |  |- view
   |- raft
      |- events
      |- fields
      |- properties
      |- rpc
      |- simulation
      |- statemachinecommands
      |- states
```

3.1 The `raft` package

The `raft` package is the core of our project and contains all the required classes for the protocol to run without intervention. As a design decision, we chose not to implement a client: instead, commands and events are sent and propagated through a `CommandBus` and `EventBus` provided by the `gui` package. Implementing an actual client with logic (e.g. for making requests to the current leader, retrying, etc...) is left as future work.
The main subfolder of the package is the `states` folder. To truthfully implement Raft's states and the three different roles (`Candidate`, `Leader`, and `Follower`), we implemented an abstract class, `State` and a base class, `Server`, that are subclassed by each of the three roles in order to get the correct subset of functionality for each of them.

```
public abstract class State {
    // Persistent state on all servers:
    public final CurrentTerm currentTerm;
    public final VotedFor votedFor;
    public final Log log;

    // Volatile state on all servers:
    public final CommitIndex commitIndex;
    public final LastApplied lastApplied;
}
```

[2] https://doc.akka.io/docs/akka/2.5/typed/index.html

Concretely, the `State` class is an abstract class that implements volatile and persistent states that pertain to all three roles. Each instance variable is a different class in itself, implementing a `ContextAware` interface to allow observability.
On the other hand, the `Server` class is a full-fledged class. It implements `Behaviors` that are used by all three subclasses.

```
public class Server {
    (...)
    protected static Behavior<Raft> \
        onVote(ActorContext<Raft> ctx, State state, RequestVoteRPC msg);

    protected static Behavior<Raft> \
        stop(ActorContext<Raft> ctx, Servers servers, State state);

    protected static Behavior<Raft> \
        crash(ActorContext<Raft> ctx, Servers servers, State state, Crash msg);
```

In the Akka Typed API, Actors represent finite state machines and must be modeled as such.[3]. Static methods in the `Server` class and subclasses return `Behaviors` that represent what the actor will be able to do once the method has finished executing. In other words, actors execute actions, and at the end, the actor will move in another state, represented as a set of `Behaviors`.
This model is very easy to adapt to the Raft protocol. In this case, the generic `Server` behaviors include primitives for sending out leader votes, stopping, and crashing.
As mentioned before, each role subclasses both the `State` and `Server` classes. For example, the `Leader` role is represented by the `LeaderState` and `Leader` classes. Sections 3.1.1 to 3.1.3 detail more clearly what each role can and can not do in the simulation.
Next, Sections 3.1.4 and 3.1.5 discuss more in depth about two key parts of the implementation: the usage of RPCs and the `AppendEntries` message exchange.
Finally, a special role was inserted for simulation purposes: the `Offline` role. It represents either a crashed or non-started server, and supports behaviors that allow it to get informed of newcomers in the cluster and then start. This implementation is discussed in Section 3.1.6.

3.1.1 Follower

The `Follower` class contains most of the basic logic required for a Raft follower to operate.

```
public final class Follower extends Server {

    public static Behavior<Raft> waitForAppendEntries(ActorContext<Raft> ctx, Servers
                                                servers, FollowerState state);

    private static void startElectionTimer(ActorContext<Raft> ctx, TimerScheduler<
                                                Raft> timers);

    private static Behavior<Raft> onElectionTimeout(ActorContext<Raft> ctx, Servers
                                                servers, FollowerState state);

    // (...implementation of onAppendEntries, crash and stop from superclass
}
```

When a server becomes a follower, it defaults to the usual behavior of waiting for `appendEntries` heartbeats from the current leader. This is defined in the `waitForAppendEntries` method, which returns the following Behavior:

```
return Behaviors.withTimers(timers -> {
    startElectionTimer(ctx, timers);

    return Behaviors.receive(Raft.class)
```

[3] https://doc.akka.io/docs/akka/current/typed/fsm.html

```
            .onMessage(AppendEntriesRPC.class, msg -> onAppendEntries(ctx, timers,
                                                    servers, state, msg))
            .onMessage(ElectionTimeout.class, msg -> onElectionTimeout(ctx, servers,
                                                    state))
            .onMessage(RequestVoteRPC.class, msg -> onVote(ctx, state, msg))
            .onMessage(Crash.class, msg -> crash(ctx, timers, servers, state, msg))
            .onMessage(Stop.class, msg -> stop(ctx, timers, servers, state))
            .onAnyMessage(msg -> Behaviors.ignore())
            .build();
};
```

This behavior comprises the following actions. First and foremost, it is a behavior with timers, and as the protocol requires, it sets a timer that when expired will start a new election. During this timer, clearly, the follower will need to respond to different types of messages. Excluding the obvious `Crash` and `Stop`, a follower can receive vote requests (`RequestVoteRPC`), election timeout messages (which are auto-sent by the expiring timer), and `appendEntries` messages.
These methods implement exactly the protocol requirements. On a `RequestVoteRPC`, the Follower calls his parent class's `vote` method. On a `ElectionTimeout`, the follower becomes a candidate and starts an election. `AppendEntries` messages comprise probably the most important part of the entire project, due to their vital role in several key parts of the protocol. More information on their implementation can be found in Section 3.1.5.

3.1.2 Candidate

Once a server receives an `ElectionTimeout`, it becomes a candidate and starts querying other servers actively for their roles. In our candidate implementation, this is achieved via the `sendRequestVote` method, which broadcasts a `RequestVoteRPC` to the cluster and waits for a response from at least the majority of the servers.

```
private static long randomElectionTimeout() {
    public static Behavior<Raft> \
        beginElection(ActorContext<Raft> ctx, Servers servers, CandidateState state);

    private static void \
        startElectionTimer(ActorContext<Raft> ctx, TimerScheduler<Raft> timers);

    public static void  \
        sendRequestVote(ActorContext<Raft> ctx, SeqNum seqNum,
        ActorRef<Raft> recipient, CandidateState state, boolean isRetry);

    private static Behavior<Raft> \
        onRPCTimeout(ActorContext<Raft> ctx, SeqNum seqNum,
        CandidateState state, ActorRef<Raft> recipient);

    private static Behavior<Raft> \
        onElectionTimeout(ActorContext<Raft> ctx, Servers servers,
        CandidateState state);

    private static Behavior<Raft> \
        onVote(ActorContext<Raft> ctx TimerScheduler<Raft> timers,
        Servers servers, SeqNum seqNum, Votes votes,
        CandidateState state, RequestVoteRPCResponse msg);

     // (...implementation of onAppendEntries, crash and stop from superclass
}
```

As votes arrive, the candidate – which preserves its behavior across methods with the `Behaviors.same()` shorthand – calls the `onVote` method and dynamically verifies if the majority requirements are met. Clearly, elections may or may not conclude successfully. Often, the election might timeout, prompting the candidate to start another one with `beginElection`, or it might receive an `appendEntries` from a new leader,

effectively reverting it to the follower state.
If the candidate were to be elected, then it invokes the leader's `elected` method, and switches role. Section 3.1.3 discusses what happens at this point.

3.1.3 Leader

The leader role is arguably the most powerful in the protocol, yet it is the most delicate. Yielding absolute power over the log, operations over it must be carefully assessed before proceeding. Eventual crashes and delays complicate the protocol, requiring some changes and some architectural choices in order to make the protocol work correctly. Details of these issues can be found in Section 3.1.6.
Once the crashes are dealt with, however, the leader role becomes surprisingly simple to implement.

```
private static long randomElectionTimeout() {
    public static Behavior<Raft> \
        beginElection(ActorContext<Raft> ctx, Servers servers, CandidateState state);

    private static void \
        startElectionTimer(ActorContext<Raft> ctx, TimerScheduler<Raft> timers);

    public static void  \
        sendRequestVote(ActorContext<Raft> ctx, SeqNum seqNum,
        ActorRef<Raft> recipient, CandidateState state, boolean isRetry);

    private static Behavior<Raft> \
        onRPCTimeout(ActorContext<Raft> ctx, SeqNum seqNum,
        CandidateState state, ActorRef<Raft> recipient);

    private static Behavior<Raft> \
        onElectionTimeout(ActorContext<Raft> ctx, Servers servers,
        CandidateState state);

    private static Behavior<Raft> \
        onVote(ActorContext<Raft> ctx TimerScheduler<Raft> timers,
        Servers servers, SeqNum seqNum, Votes votes,
        CandidateState state, RequestVoteRPCResponse msg);

     // (...implementation of onAppendEntries, crash and stop from superclass
}
```

Leader logic starts with the aforementioned `elected` method. Once elected, the leader's log is "the truth", and other followers must obey to this rule and have their logs changed, updating their entries and eliminating extraneous ones. Moreover, eventual leaders are deposed by the same `appendEntries` messages sent. Leaders send their `appendEntries` messages with `appendEntriesRPC` and process their responses with `onAppendEntriesResult`. These methods closely follow the paper's implementation and guidelines. Section 3.1.5 deals with this matter more in depth.
Finally, leaders have a special method, `onCommand`, which allows receiving new entries from the GUI and adding them to the log. It implements stashing to correctly process incoming commands in a FIFO-like fashion.

3.1.4 RPCs

A whole section must be reserved to the implementation of communication between servers and the implications that our implementation has.
As anticipated through previous sections, servers use RPCs to communicate between each other. However, this approach was not the first that we took into account. Indeed, in the very first versions of the project, we opted to use regular `tell` primitives.
In Akka, `tell` methods can be used to send messages in a "fire-and-forget" fashion, disregarding the status of the recipient. Instead, `ask` methods send a message and return a future, which can be awaited until

timeout or a reply is received. This allows a finer control over the behaviors of the actors.[4].

```
ctx.ask(AppendEntriesResult.class,
    recipient,
    isRetry ? Duration.ofMillis(100) : Duration.ofMillis(properties.rpcTimeoutMs),
    (ActorRef<AppendEntriesResult> replyTo) -> \
        new AppendEntriesRPC(ctx.getSelf(),
                             seqNum.computeNext(recipient),
                             replyTo,
                             appendEntries),
    (response, throwable) -> {
        if (response != null) {
            return new AppendEntriesRPCResponse(
                recipient,
                seqNum.expectedSeqNum(recipient),
                appendEntries,
                response);
        }
        return new RPCTimeout(recipient);
    }
);
```

The above code – written with the functional paradigm – is implemented at the end of the `appendEntries RPC` method of the `Leader` class. It specifies the target of the RPC, the response timeout, what to send, and how to act once the response arrives (or times out).

3.1.5 `AppendEntries` messages and logs

`AppendEntries` messages are the core of the Raft protocol. They have multiple functions: they double as heartbeats, sent by leaders to reinforce their power over the current term, inform old leaders that they were deposed, and expire other servers' election timeouts. However, their first and foremost usage is to update the local log of the other servers.
`AppendEntries` messages are exchanged with RPC as described in Section 3.1.4, and are equipped with the following parameters:

```
public final class AppendEntries extends AbstractRaftMsg implements RaftRequest {
    public final ActorRef<Raft> leaderId;
    public final int prevLogIndex;
    public final int prevLogTerm;
    public final List<LogEntry> entries;
    public final int leaderCommit;
}
```

When a leader is elected, an `AppendEntries` message is sent to all the cluster. As mentioned in previous sections, this message will:

- depose old leaders, should they realize that their `currentTerm` is greater than the leader's;
- restart all other servers' election timeouts;
- inform them of the last committed entry by the leader;
- what the leader expects the client to have in their log (`prevLogIndex` and `prevLogTerm`).

Followers receiving this message with `ask` will invoke their `onAppendEntries` method, which implements the protocol's logic as detailed in Figure 2, p. 4 [4] of the paper.
This means that the receiver will respond **false** if their term < `currentTerm` (they weren't aware of the new term) or if their log doesn't contain an entry at `prevLogIndex` whose term matches `prevLogTerm` (they were lagging behind newer additions to the log). It will check for conflicts in the entries, eventually deleting them and decreasing the log index.

[4] `https://doc.akka.io/docs/akka/current/typed/interaction-patterns.html`, section "request-response with ask between two actors"

The receiver will then construct a response (also in the form of a RPC), which will be then sent to the leader. This ping-pong will continue until the follower will respond true to the leader, meaning that now the two logs match.
All of this logic is made possible and easier thanks to subclassing. `LogEntries` are a class on their own, containing the command for the state machine and the term.

3.1.6 Failure handling

We conclude the section about the `raft` package by briefly discussing about the failure model we chose to take into account and how it is implemented.
As mentioned before, we supplemented the three status presented in the paper – `Candidate`, `Leader`, and `Follower` – with a fourth one, `Offline`.
The `Offline` role represents either an unstarted server, or a crashed one. From a logical perspective, we can consider these two "states" as a single one, one in which the corresponding server lost its transient state but retained its permanent one. Moreover, an offline server must ignore all messages but the ones required for it to restart.

```
ctx.ask(AppendEntriesResult.class,
    recipient,
    isRetry ? Duration.ofMillis(100) : Duration.ofMillis(properties.rpcTimeoutMs),
    (ActorRef<AppendEntriesResult> replyTo) -> \
        new AppendEntriesRPC(ctx.getSelf(),
                             seqNum.computeNext(recipient),
                             replyTo,
                             appendEntries),
    (response, throwable) -> {
        if (response != null) {
            return new AppendEntriesRPCResponse(
                recipient,
                seqNum.expectedSeqNum(recipient),
                appendEntries,
                response);
        }
        return new RPCTimeout(recipient);
    }
);
```

This is achieved by making use of the `Behaviors.ignore()` shorthand and a `Behavior` builder that only accepts `Start` messages. When in this state, RPCs directed at the server will fail, prompting them to retry such RPCs until they either give up or the server comes back online.

3.2 The `gui` package

This package contains code for the application that allows to run the simulation, and to interact with it. The GUI is written with `JavaFX`, basically the standard for implementing GUIs in Java. The entry point for the application is the `App` class.

```
public class App extends Application {
    private final static double WINDOW_WIDTH = 1200;
    private final static double WINDOW_HEIGHT = 800;

    private ApplicationContext applicationContext;

    @Override
    public void start(Stage primaryStage) {
        applicationContext = new ApplicationContext();

        var mainView = new MainView(applicationContext);
```

The class creates an instance of `ApplicationContext`, which then creates a new `ActorSystem`, which is the component responsible for actually simulating our cluster.

```
public class ApplicationContext {
    public final CommandBus commandBus;
    public final EventBus eventBus;

    public ApplicationContext() {
        var actorSystem = ActorSystem.create(SimulationController.create(this), "raft
                                                -cluster");
        commandBus = new CommandBus(actorSystem);
        eventBus = new EventBus();

    }
```

All buttons in the GUI generate an event when interacted with. The events are passed through an `EventBus`; this class is used to relay all events towards the class `SimulationController`, which is responsible for changing the simulation state. Control over the simulation is implemented with special messages that are sent to the actors inside the simulation. The actor receives the control message and acts accordingly.
For example, this is the code that responds to the event to stop the simulation:

```
    private Behavior<Raft> onStopSimulation(StopSimulation command) {
        getContext().getLog().info("Stop simulation command");
        var stop = new Stop();
        servers.forEach(server -> server.tell(stop));
        return this;
```

which results in this method getting called on the `Server`:

```
    protected static Behavior<Raft> stop(ActorContext<Raft> ctx, Servers servers,
                                         State state) {
        ctx.getLog().info("Received stop command. State is:\n" + state);

        return Offline.waiting(ctx, servers, state);
    }
```

To update the contents of the server status screen, we use an `EventBus` that relies events from the simulation to the GUI code. The components subscribe to an event, and when the event is fired they execute a listener to update the contents of the interface. For example, the following lines in the declaration of `LocalLogView`:

```
        applicationContext.eventBus.listenFor(Spawn.class, this::onSpawn);
        applicationContext.eventBus.listenFor(LogAppend.class, this::onLogAppendEvent
                                                );
        applicationContext.eventBus.listenFor(LogRemove.class, this::onLogRemoveEvent
                                                );
        applicationContext.eventBus.listenFor(CommitIndexIncrement.class, this::
                                                onCommitIndexIncrement);
        applicationContext.eventBus.listenFor(CommitIndexDecrement.class, this::
                                                onCommitIndexDecrement);
```

subscribe to the events that regard the spawn of a new actor and the updating of the local log on an actor.

4 Demo

In this section, we describe how to run the project demo and get started with the simulation of the Raft protocol.

4.1 Requirements

The following requirements must be met in order to run the project:

- Java 17.0.1 (or later)
- Apache Maven 3.8.4 (or later)

Project dependencies (such as JavaFX) are managed by Maven, and will be pulled and installed once the project is imported and set up. At the moment, we do not provide an executable `jar` file.
To get started, obtain the source code from the GitHub repository:

```
git clone git@github.com:andreastedile/raft-simulator.git
```

Then, move into the directory, and use Maven to start the project:

```
mvn javafx:run
```

Maven stores all the information about the project in the `pom.xml` file, which is available in the project root. This includes the JavaFX plugin, which enables Maven to automatically link JavaFX libraries with the compiled Java code.
Once the aformentioned command is run, Maven will query the repository, pull the relevant dependencies, build the project, and finally open a pop-up Java window, as in Figure 1.

4.2 The GUI

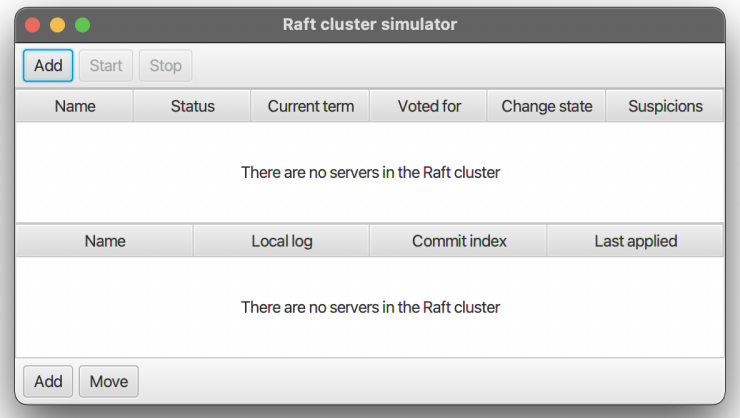


Figure 1: The Raft simulator window at start.

The GUI is composed of three sections: the *toolbar*, the *server view*, and the *log view*.
The *toolbar* sits at the top of the window, and is the entrypoint for starting the Raft cluster. It has three buttons, which are dynamically enabled and disabled depending on the context. To get started, press the *Add* button to add a Raft server. Repeat until a satisfactory number of servers is inserted in the system. Note that, once a server has been added, it cannot be removed.

Once at least a server is added to the cluster, the *Start* button is enabled, allowing the system to be started. Since one of the assumptions of the Raft protocol is that the number of servers in the cluster shall not change, once the simulation is started adding more servers is not supported, and the relevant button is grayed out.
In the *server view*, a sortable list of servers is populated as the *Add* button is repeatedly clicked. Fields presented included the name, the status (`FOLLOWER/CANDIDATE/LEADER`), the current term, and who the server voted in this term. Finally, a button is available on the far right of each right. Figure 2 shows an example of the GUI once some servers are added.

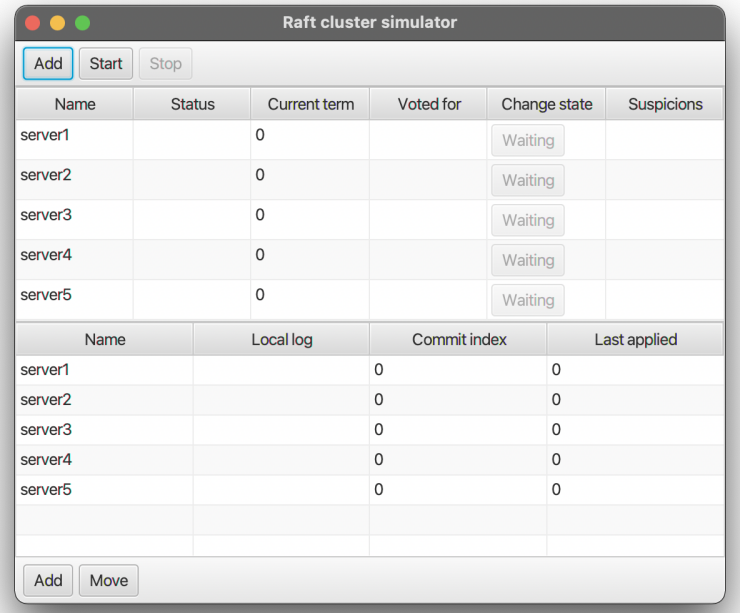


Figure 2: Adding a handful of servers to the simulation.

This button, which is active only when the simulation has been started, allows the user to programmatically crash the related node. The node will stay in a crashed state (i.e. will ignore any incoming message and forget its non-persistent state) until the user presses the button again, effectively rebooting it.
On the bottom *log view*, another table is available which shows the local log, the commit index, and when the commit was last applied. Two additional buttons are available, which allow the user to query the current leader and ask to add an entry to the log.

4.3 Starting the simulation

Once the user is satisfied with the amount of servers added to the simulation and has started it, the Raft cluster will let every server know each other and let them interact freely using the primitives described in section 3. The servers will set an `electionTimeout`, and after a short time, one will prevail and will be elected leader. For showcasing purposes, we can crash it with its button. Figure 3 shows an example of what happens in the GUI: as the leader is killed, the other servers' timeouts will expire and will elect a new leader, as they will still be able to meet the majority criteria.
If the user were to crash more than the majority of the servers, then the remaining servers will start panicking and, as their election timeouts repeatedly expire, start increasing their terms. Figure 4 (left) shows an example of this effect. This is expected behavior, as two servers out of five should never be able to elect a new leader. On the other hand, Figure 4 (right) shows the result of a clean comeback from a crash. The previous leader (`server2`) came back and was swiftly deposed by the new one, and with the `appendEntries`

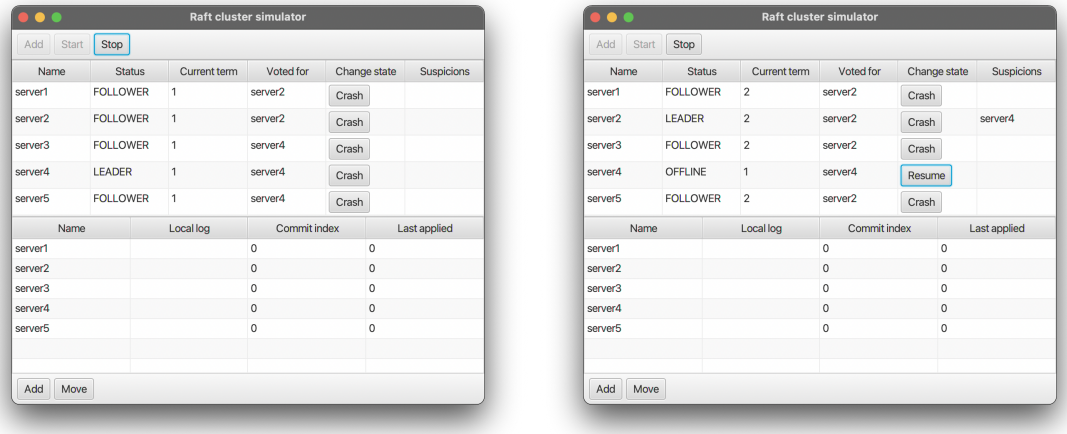


Figure 3: The start of the simulation, and crashing the current leader.

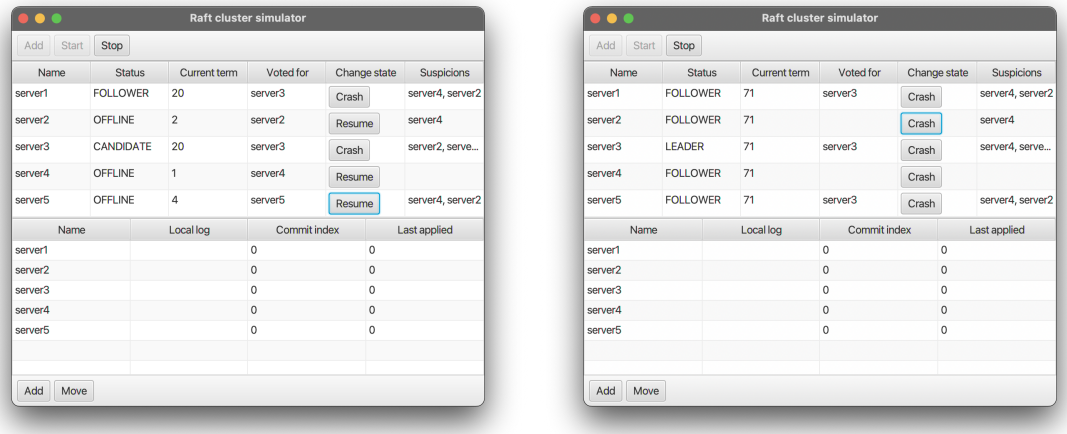


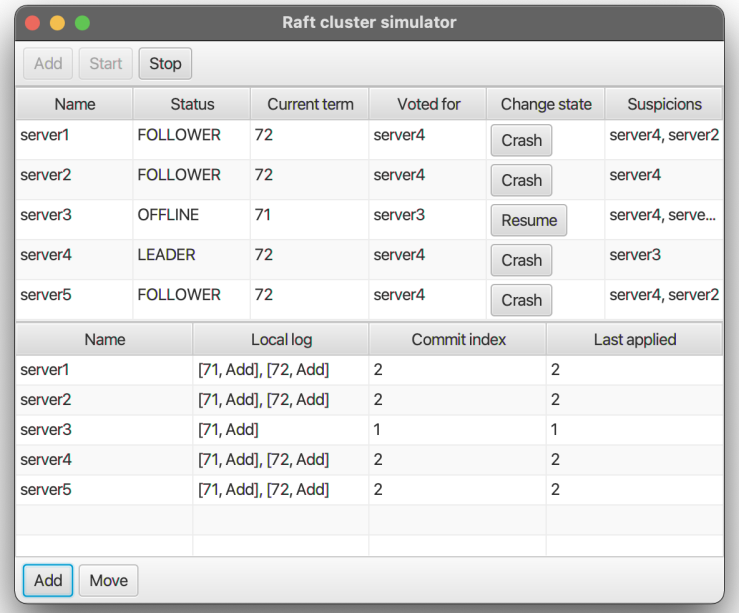
Figure 4: Crashing more than the majority of the servers: to the left, terms start increasing drammatically. To the right, term parity is achieved when they resume from the crash.

messages was informed of the new term and leader. We can observe that `server2` did not vote for anyone in this term. Again, this is to be expected, as the server was crashed during the election.
Finally, Figure 5 shows an example of an addition of log entries to the cluster. At first, all servers have recovered from the crashes in previous screenshots. One request is made to the leader of term 71 while all servers are alive. This results in a swift addition to each server's local log. Then, `server3` is crashed, and an additional request is made to the new leader. Now, every server but `server3` is aware of the request and has applied it, as the majority was still met even without him. When `server3` will come back, he will learn with `appendEntries` messages of the new entries and will update its local log accordingly.

4.4 Simulation parameters

The simulation supports fine tuning of some parameters. This is done via the `simulation.properties` file. The provided file looks like this:

```
timeScale=1
minElectionTimeoutMs=150
maxElectionTimeoutMs=800
heartbeatMs=50
```



ht

Figure 5: Adding some logs.

```
rpcTimeoutMs=15
maxCrashDuration=10
```

The configurable parameters are as following:

- `timeScale`: an integer ≥ 1. It is used as a factor for other parameters, and allows slowing down the simulation;
- `minElectionTimeoutMs` and `maxElectionTimeoutMs`: the floor and ceiling values used by the PRNG when computing election timeouts after each `appendEntries` message;
- `rpcTimeoutMs`: the milliseconds waited by servers before retrying an RPC.

5 Conclusion and future work

This report illustrated the capabilities of Raft, a powerful and straightforward protocol for applications requiring a relaxed version of distributed consensus. Using the Java language and the Akka library, we implemented a full-fledged Raft cluster simulation that closely follows the paper's theory and guidelines. Halfway through the project, we realized that the protocol would be easier to approach with a functional paradigm. Thus, we switched our implementation to the newer Akka Typed library and rewrote our project to closely match the behavior of a state machine.
We supplemented our cluster with a GUI written in JavaFX, which allowed continuous monitoring of the state of each server in the cluster and allowed direct command sending. All of this was done while keeping the amount of source lines of code relatively low (around 2500 SLOCs), while allowing for a strong decoupling between the GUI and the protocol. By doing such work, we confirmed the Raft paper's results on understandability and ease of use, which were at the core of their research in providing an approachable alternative to other distributed consensus algorithms such as Paxos.
Future work for this project may include the addition of dynamic membership and log compaction, two features that were present at the end of the paper that we chose not to implement due to time constraints. Moreover, the inclusion of a client actor to the cluster could be an interesting addition, with the GUI being changed to act as an interface to the client itself.

REFERENCES

[1] J. Fischr and A. Lynch, "Impossibility of Distributed Consensuswith One Faulty Process," p. 9, 1983.

[2] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998. [Online]. Available: https://dl.acm.org/doi/10.1145/279227.279229

[3] Lightbend, Inc. Akka. Lightbend, Inc. [Online]. Available: https://akka.io/ Accessed 2022-01-31.

[4] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," *Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference*, p. 16, 2014.

[5] Stein Magnus Jodal et al. Pykka. [Online]. Available: https://pykka.readthedocs.io/en/stable/ Accessed 2022-01-31.