Visualizing Raft

Brugnara, Martin
#182904

Hoxha, Fatbardha

Nardelli, Andrea
#171723

Abstract

Raft is a consensus algorithm for managing a replicated log proposed by Diego Ongaro and John Ousterhout. One of its key aspects is being designed to be easily understood. Diego Ongaro (@ongardie on GitHub) created a simulator called Raft Scope in order to visualize and present it. This tool currently implements the protocol only in its basics. In this work we extend its capabilities to support all the features and we enhance the user interface to clarify the more complex aspects of the protocol.

1 Introduction

Raft is a consensus algorithm for managing a replicated log proposed by Diego Ongaro and John Ousterhout [1]. It is equivalent to the state of the art (multi-)Paxos [2] in fault-tolerance and performance. Consensus algorithms tend to be complex due to the huge problem they try to solve, and leaving space for interpretation, like Lamport did when proposing Paxos via a tale [2], does not help. The combination of Paxos' complexity and the way it was presented required Lamport to release an alternative description [3]. This complexity makes Paxos hard to implement in real world applications, which are sometimes required to relax the model described leading to unproved protocols [4]. The same reason makes it difficult to learn and study for the students and researchers.

Ongaro thus identified in *understandability* its “most important goal” [5] while describing Raft, breaking it into smaller independant subproblems and cleanly addressing all pieces needed for a pratical application. The simplest way to present and understand an interaction based protocol is visualizing it.

Raft Scope [6] is a simulator that runs in a browser (Figure 1) developed by the very same author Diego Ongaro. It simulates the *leader election* and the *log replication* protocols as described in his original Ph.D. thesis [5]. This tool is very useful to simulate easy scenarios, however it does not currently implement all the features of the protocol and lacks some graphics cues which can be key in the presentation of many corner cases.

In our project we aim to add more features to the protocol implementation and improve the user interface where needed. In particular, we plan to add functionality to handle cluster membership changes and channel noise. The code was also reorganized in order to make it clearer and easier to modify for other users.

Raft Scope is an open-source project currently hosted on GitHub and we plan to submit a pull request since this could be useful for others.

2 Raft Scope - State of the art

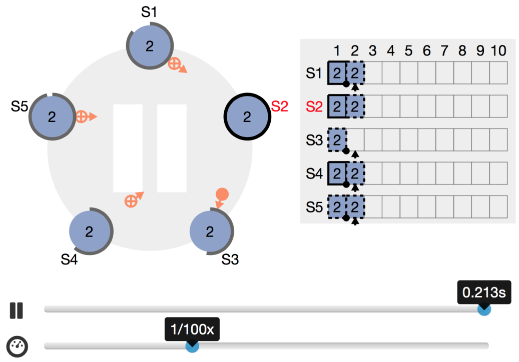


Figure 1: Original

2.1 Simulation model

Raft Scope simulates a small Raft cluster in your browser. The user can pause the simulation, seek points in time and modify its speed. It is coded in JavaScript, which is a single threaded language. The author created a sort of virtual machine (VM) where concurrency is only simulated by artificially managing time and events.

The VM has an internal clock that is advanced at each simulation cycle by an amount of time computed *w.r.t* the simulation speed. For each simulation cycle, the main loop (`raft.update`) checks each server and loops through their actions in a predefined order. If the simulated time advances past the timeout for the respective action, then that event is fired in this cycle. In the same fashion the simulation loops through each message and delivers it if its `DeliveryTime` has elapsed.

Due to this design decision and especially at very high simulation speeds (*e.g.* real time speed) many operations are executed in *bulk* in the same cycle. This does not guarantee correctness because the operations take place in their internal cycle ordering instead of being ordered by their timeouts.

For example: Suppose the cluster leader sends an `AppendEntries` message (M1) to server 2 (S2), and that the S2 `ElectionTimeout` is due to expire just after the arrival of M1. If the simulation speed is sufficiently high, it is possible that these two events take place in the same simulation cycle, thus S2 would start an election because this event check happens before the loop that delivers the messages. In a real world application S2 would receive M1, reset its `ElectionTimeout`, and reply with the `ACK` message.

2.2 Interaction

The user can interact with the Raft cluster by sending *client requests*, starting/stopping/restarting a server, forcing a server to start a new election and dropping messages. Each of these actions generates a new state in the simulation. The user can save snapshots of a particular simulation and all of its checkpoints and replay it.

2.3 Code organization

The Raft Scope code base is organized mainly in three different files:

- `state.js`
- `raft.js`
- `script.js`

`state.js` manages the state of the simulation and keeps track of its configuration and its entities(servers, messages, logs, etc...). For each simulation cycle, the previous state is appended to a list of checkpoints and a new one is generated. This allows the user to rewind and playback the simulation by jumping to a previous state.

`raft.js` manages the rules of the simulation. Rules are actions defined by the protocol, such as *BecomeLeader*, *StartNewElection*, *SendRPC*, etc... These are connected to the time of the simulation and are checked automatically by the main loop.

`script.js` handles everything else: connecting entities to their graphical representation, generating the log table, initializing the simulation. This code was particularily tricky to understand at first.

3 Extending Raft Scope

3.1 Simulation model

Fixing the problem in section 2.1 is not straight forward: swapping the two checks would break other scenarios (`e.g.` M1 is due just after the `ElectionTimeout`). The only definitive fix would have been rewriting the entire simulation system. Given the use case and applications of Raft Scope we decided to limit the maximum speed to prevent such problems.

Network noise was integrated into the simulation and the state object, dropping each message with a probability set by the user.

3.2 Interaction

Cluster membership changes are an important part of a consensus protocol. Raft initially handled this with a procedure called *joint consensus* but Ongaro expanded on this matter in his Ph.D. thesis [5](Chapter 4) with a new simpler algorithm that only allows single server changes at a time (both removal and addition), however any arbitrary change that could be expressed with joint consensus can be also managed with a series of single server changes that is easier to visualize. Each configuration in the series must be committed before the next one can take place. With the intention to clarify this series of changes, we added a *cluster configuration changes queue*. This queue keeps track of all the cluster reconfigurations submitted by the user while they are being processed. The original dissertation is not clear on what action to take regarding pending configurations when leadership is lost.

A bug in the single server membership change algorithm was later found and discussed in the raft-dev mailing list [7] which could rewrite committed history. The bug occurred whenever leadership was lost and two pending configurations were happening at the same time, submitted by different leaders. As described there, we implemented the fix by having leaders append a `NOOP` to their log.

We implemented a safety check for disruptive(removed) servers as described in section 4.2.3 of the author's thesis. Servers, when removed, could disrupt cluster functionality by starting new elections whenever a pending change for their removal is being processed. The fix makes servers ignore VoteRequest RPCs if they have received an heartbeat within `MIN_ELECTION_TIMEOUT` from an active leader.

3.3 Integrating the new features

In order to allow transparent playback we modified the rendering phase in the main loop to take in account the possibility for membership changes and automatic message drop due to network noise. It was particularly challenging to determine when it is safe to remove a server from the simulation: it is not correct to remove it when its corresponding configuration change is committed because other servers, which have not yet received the entry, might still interact with it. We decided to remove it when every other server has removed it from its `peers` list.

3.4 Code reorganization

We added the following files:

- `render.js`
- `presenter.js`

`render.js` handles all the graphic, in particular SVG generation, servers realignment, sliders and generating the new log table. It also contains the code that generates modal and context menu when clicking messages or servers.

`presenter.js` contains utilities to save and load a particular state, hotkeys to quickly interact with the system with functions that simulate interesting situations such as split votes.

`script.js` now only contains intialization code and the *system clock*.

3.5 Interface

The log table has been rewritten in plain `html` instead of SVG to allow support for an infinite log with autoscrolling and a high number of servers. To avoid confusion the "leader only" fields in the modal menu of the follower servers have been hidden and a legend was added to explain the otherwise cryptic symbols in the log table. Moreover, each log entry is categorized depending on its type (`NOOP`, configuration change entry, user request).

We added controls for the features we introduced in the protocol: a slider to set the *network noise*, a *server add* button, a button to the server context menu for removal and finally a new table to visualize the cluster configuration changes queue.

3.6 Future work

The *leadership transfer extension* described in Section 3.10 of the thesis was not implemented in the original Raft Scope or our work because it was not evaluated by the authors.

As said in the introduction we plan to submit a pull request to the original Raft Scope and discuss our changes (while fixing any bugs we might have introduced) with the original author.

4 Conclusions

This project helped us learn how Raft works because we had to understand how the model and simulation are structured in order to make improvements.

In particular we implemented the cluster change functionalities, we simplified and clarified the user interface to allow the understanding of all aspects of the algorithm and corner cases.

We hope our work will easy the learning curve for others to come, as the original Raft Scope did for us.

References

[1] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[2] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

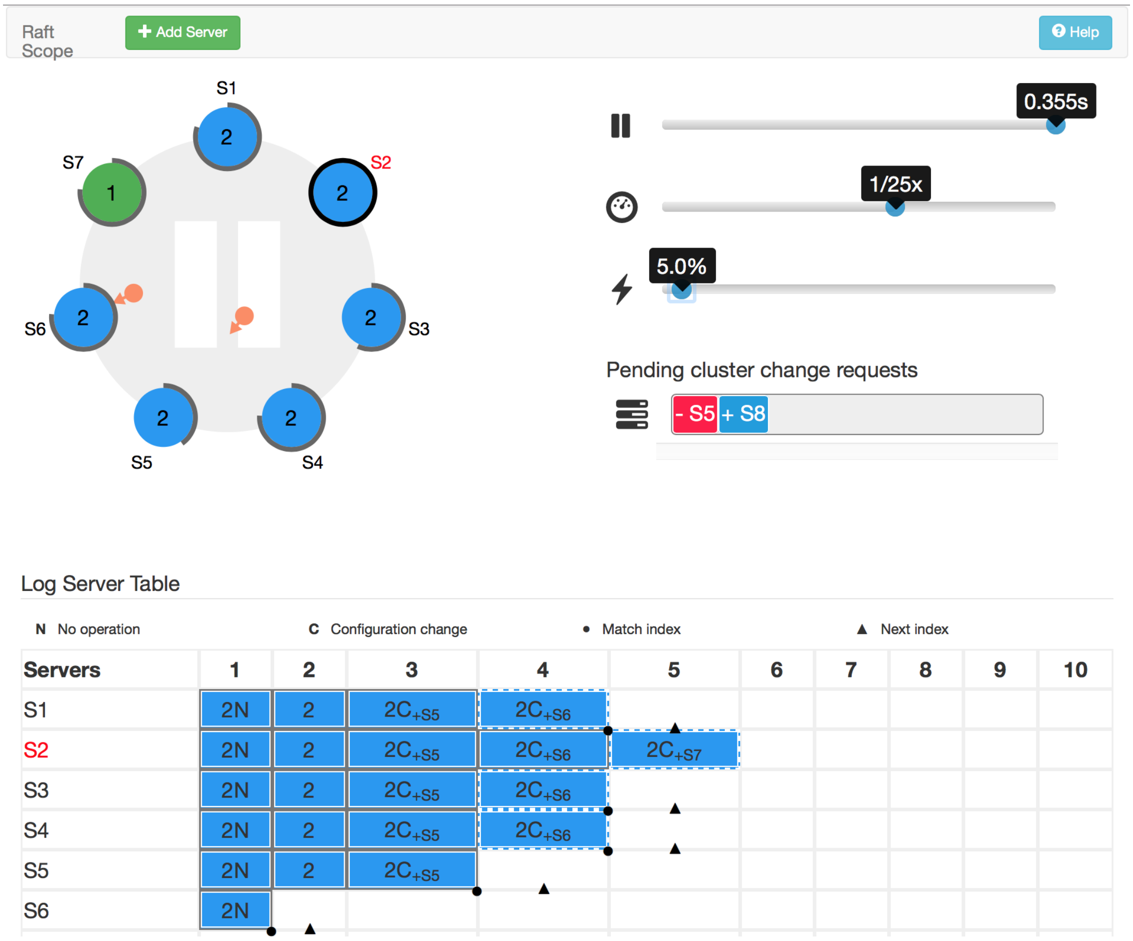
[3] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[4] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[5] Diego Ongaro. *Consensus: Bridging theory and practice*. PhD thesis, STANFORD UNIVERSITY, 2014.

[6] Diego ongardie Ongaro. Super hacky visualization of raft, 2014.

[7] Ongaro. Bug in single-server membership changes, 2015.



Pending cluster change requests

- S5 + S8

Log Server Table

N No operation   C Configuration change   ● Match index   ▲ Next index

Servers	1	2	3	4	5	6	7	8	9	10
S1	2N	2	$2C_{+S5}$	$2C_{+S6}$ ●	▲					
S2	2N	2	$2C_{+S5}$	$2C_{+S6}$	$2C_{+S7}$					
S3	2N	2	$2C_{+S5}$	$2C_{+S6}$ ●	▲					
S4	2N	2	$2C_{+S5}$	$2C_{+S6}$ ●	▲					
S5	2N	2	$2C_{+S5}$ ●	▲						
S6	2N ●	▲								

Figure 2: Extended