# Distributed Algorithms coursework

# Implementation and analysis of Raft consensus algorithm

Marco Selvatici - ms8817@ic.ac.uk

# Table of contents

## Overview & Design choices

The code is structured around the *server.ex* module. As shown in *figure 1*, this module has the task to handle state changes for a server. The other main modules are:

- *follower.ex*: defines the behaviour of the server when in FOLLOWER state.
- *election.ex*: defines the behaviour of the server when in CANDIDATE state.
- *leader.ex*: defines the behaviour of the server when in LEADER state.

Every module contains assertions that ensure the state of the server is always as expected.

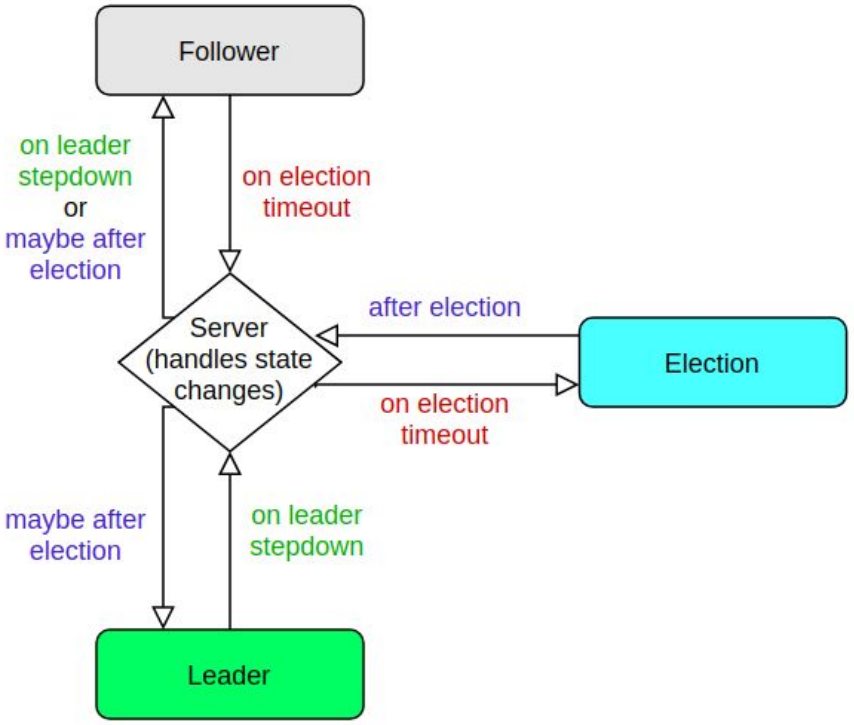


*Figure 1: high level overview of the events that lead to state changes.*

Changes among states are triggered by three main events:

- election timeout, FOLLOWER → CANDIDATE
- election done, CANDIDATE → FOLLOWER or LEADER
- leader stepdown, LEADER → FOLLOWER

I stressed on the conceptual separation of the different server states and on the clear way of moving among those, because it makes the whole system simpler to reason about. For example, when debugging some parts of the code, there is no need to ask "what is the state of the server right now?" since this information is already implied by the module itself.
Every module also contains state-specific functions (e.g. *leader.ex* contains the *broadcast_append_request* function), while the common functions like *apply_committed_entries* are in a shared module: *common.ex*.

*Figure 2* shows a typical client - server(s) interaction, with an overview of the various messages exchanged at each phase.

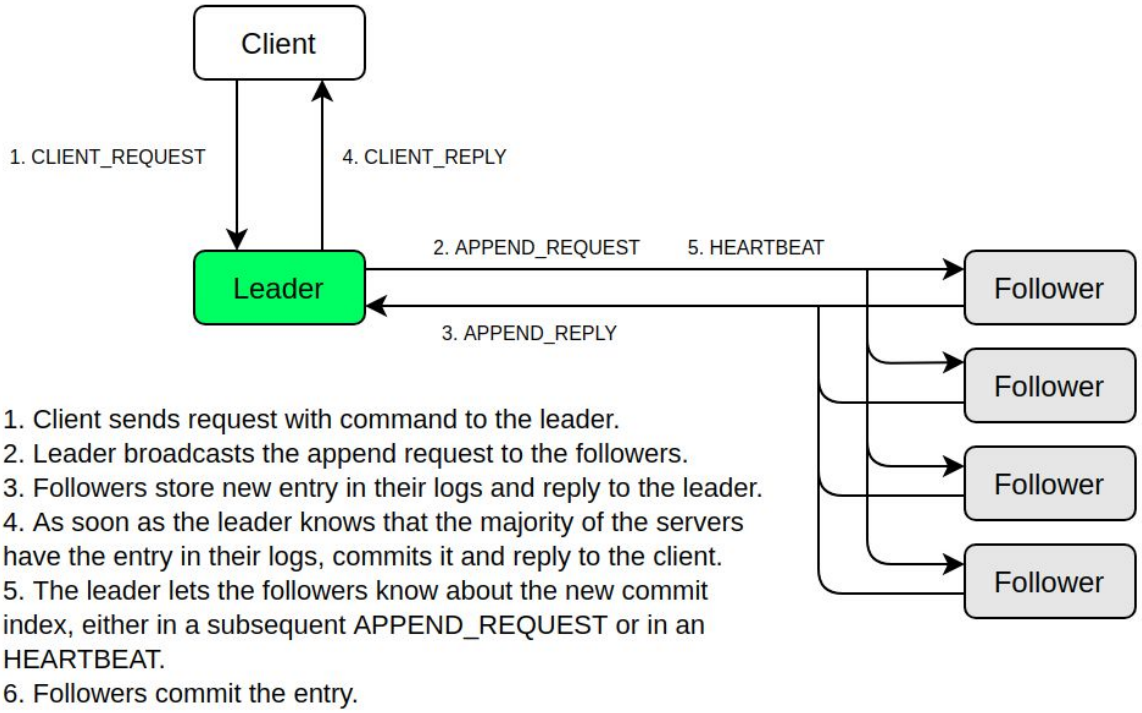


*Figure 2: typical client server(s) interaction, and messages involved.*

There is a well defined set of possible messages, and each of them can be handled (or ignored) in every state of the server:

- APPEND_REQ: append requests sent by the leader (can either contain new log entries or be a heartbeat).
- APPEND_REP: append reply sent by followers back to the server.
- VOTE_REQ: vote request sent by candidate during election.
- VOTE_REP: vote reply sent by followers back to the candidate.
- RPC_APPEND_TIMEOUT: triggers resend of an APPEND_REQ to server.
- RPC_ELECTION_TIMEOUT: triggers resend of a VOTE_REQ to server.
- ELECTION_TIMEOUT: triggers new election for followers or candidates.
- CLIENT_REQUEST: contains the information for a client request.

If the server is in a particular state, it may not be supposed to receive some types messages (e.g. follower should not receive APPEND_REP), but this still may happen due to the asynchronous nature of the system, hence they need to be handled (or ignored). For example, some messages may be queued and only handled after a change of state.

The implementation makes use of pattern matching guards in order to conceptually separate the different situations that a server may run into. This further allows to simplify logic for each message type, and result in a much more modular code. It is relatively easy to validate the correctness of each code section, and this largely reduces the risk of bugs.

In contrast, this strong modularization can lead to some code duplication and verbosity, but I believe the trade-off is worth it.

# Debugging & Testing techniques

Debugging distributed asynchronous code is notoriously a challenging task.
I tried to build my code one feature at the time, making sure everything was working before moving on. I started with the election logic, then moving onto some basic version of the follower and leader states (simply send and receive heartbeats). Once these were working, I implemented the logic for append requests, and, lastly, the handling of client requests.

While debugging, I found it extremely useful to:

- Color coding the states in the logs. This quickly allows to roughly understand what is going on, and to isolate the messages of interest.
- Change timeouts to larger or smaller values in order to consistently trigger a particular behaviour.
- Have different levels of verbosity for the log, and an easy way to switch among those. For example, debug level 0 logs the behaviour with a single-message granularity, while level 3 only shows the change of state for the servers.
- Keeping the number of servers (and clients) small, in order to make it as simple as possible to isolate the behaviour of a specific server avoiding extra clutter.

The most difficult piece of logic to debug has been one-off errors in the log consistency logic, in particular making sure that invalid log entries of an out of date follower are properly repaired by the interaction with the new leader.

# System evaluation

I am confident that the system works reliably even in the presence of server failures or under heavy load. Below are some of the experiments I carried on in order to make sure of that (the log files can be found in the *output* folder, and the steps to reproduce them are described in the *README.md* file).

## Re-elections work reliably with crashing / slow leaders

The log file *crash_leader* shows a situation where the leader crashes immediately after being elected. Ideally, this would lead the other leaders to elect a new leader.
Log shows that servers 2 and 3 become candidates in term 1. Server 3 wins and crashes immediately, without even sending a heartbeat. Server 1 candidates and gets elected for term 2 (it is aware that term 1 is over because it voted during that term). Note that server 2 could have not been elected in term 1 because server 3 was, and it steps down after seeing a vote request for term 2 from server 1. Server 1 becomes leader and crashes immediately. Similarly server 5 gets elected in term 3 and crashes.
At this point, the two remaining servers (2 and 4) cannot be elected anymore since they will never receive a majority of votes (at least 3 out of 5 needed) and they keep on candidating for new elections (each time incrementing the election term). It is interesting to see that, generally, one server candidates for term X, sends a vote request to the other server, which was a candidate for term X - 1 and hence steps down and votes. Then the election for term X terminates and the roles of the two servers "swap".

A slightly different situation is presented in *slow_leader*. Here, a leader takes a long time to send heartbeats (longer than the election timeout). This implies that, when a server gets elected, the other servers have no way to know it and another election is started.
In the log, you can see server 2 becoming leader for term 1, and then some time after stepping down to follower for term 2 (after seeing the vote request from server 4). The log clearly shows how a server, even though it is elected as leader, cannot keep such a role and gets replaced soon after.
There are no updates done to the database since the leader can never send data to the followers, and hence it can never commit any log entry (even though server 3 receives 2 client requests while leader, after around 6 seconds).

## Logs are consistent across servers (same entries in the same order)

In this experiment there are 5 clients, each sending up to 100 requests. The experiment runs the system under "normal" conditions for 30 seconds, with no exceptional delays nor server crashes. The logs can be found in the file *checksums*.
In order to make sure logs are replicated correctly across the servers (same committed entries in the same order), every server prints a checksum calculated on top of the committed entries in their log. All servers are sequentially crashed after 30 seconds (well after they are done processing client requests), and upon crashing they display their checksum. It is easy to see that the value of the checksum is 74578118 in every case, hence the committed log entries are consistent.

## Logs of correct processes are consistent even if leader crashes

The setup of this experiment is similar to the previous one, except that now, servers 1 and 3 are artificially forced to crash after 10 and 15 seconds respectively. The logs can be found in *log_consistent_on_leader_crash*.
A standard election happens at the beginning and server 3 becomes leader. The leader starts to receive client requests and updating followers logs. After 4 seconds, the leader received 69 client requests, and committed 66 of them. The followers have not yet performed 66 updates because the leader informs them of its current *commit_index* only in a subsequent append request (or heartbeat). 3 log entries (69 - 66 = 3) are not yet replicated in the majority of the servers, and therefore cannot be considered committed.
The number of db updates done grows similarly in all the servers, with some lag with respect to the leader. After sever 1 crashes, the number of its updates stopped growing.
After server 3 crashes, server 4 is elected and handles the remaining 297 client requests.
After around 22 seconds, all client requests have been processed, and each correct server (2, 4 and 5) have performed 500 updates to the database, as expected.
After around 30 seconds all the servers are crashed, so they can display their checksum, which is 73498699 in every case.

## System response with slow append replies

This experiment explores how the system behaves when followers are slow, how performance degrades in such scenarios and what could possibly be done to mitigate that.

In this test there are 5 clients, and each of them sends up to 5 requests. There is a delay of 80 milliseconds before each append reply is sent, and the leader's append request rpc timeout is 50 milliseconds. No servers are artificially crashed.
The relevant log file is *slow_append_replies*.
The log shows a "double election" (more on this later) and server 5 becomes the leader for term 2. It starts to handle client requests, but can only do so very slowly, since the followers are relatively unresponsive. Logs show that after 8 seconds, only 14 requests have been replicated on the majority of the servers, and the system appears to be slowing down more and more as time passes (you can see how the number of updates done for server 5 grows slower and slower). This is because the leader is filling up the followers queues with append requests, which are sent every 50 milliseconds. The followers can only reply to an append request after a delay of 80 milliseconds. In queueing theory, a system with arrival rate bigger than throughput is called an unstable system, and it is easy to show that the size of its queue of requests tends to grow to infinity (it just can't "keep up" with the load).

At the end of the 38 seconds the processes have not finished processing yet. Queues of the followers are still saturated with append requests.
Increasing the rpc timeout for append requests from 50 to 100 milliseconds improves performance a lot. This is because now the throughput of a follower is higher than its arrival rate, hence the system's queue should not grow to infinity (even though the average number of queued elements may still be high, but bounded).
*slow_append_replies_high_timeout* shows that now all clients manage to send their 5 requests and the system has handled all of them after 20 seconds.
Clearly, it is important to strike a balance: on one side you want to keep timeouts low in order to quickly retransmit information if necessary, but on the other side it is crucial to not overload the followers, as this would heavily degrade the performance of the system. Even better would be an adaptive system, where the leader can measure the responsiveness of a follower and vary each rpc timeout consequently.

The "double election" at the beginning of *slow_append_replies* is an interesting event that occurs sometimes (more often with small election timeouts). Server 2 candidates for term 1, and requests votes from the other servers (including server 5). Server 5, votes for server 2 but before server 2 can send a heartbeat as newly elected leader, server 5 starts elections for term 2. The previous leader therefore steps down and server 5 becomes the new leader.

## System response under heavy load

Running the system with 10 servers, 5 fast clients (sleep for a short time between two requests) and a maximum of 1000 requests in 5 minutes, yielded that the system can reliably cope with stress. This experiment brings my laptop to its computational limits, the steady state is roughly 5 client requests processed every two seconds, but I expect this to be way higher on the much more powerful lab machines. In the end, all correct processes report the same checksum, and the system handled all the requests that the clients sent (870 in total).
The relevant logfile is *heavy_load*.