Distributed Systems Lab 10

Consensus: Leader election

During the previous lab, we discussed how to implement distributed commit, presenting 2PC and 3PC algorithms. The following two labs concern Raft, a distributed consensus algorithm. This lab presents its first part, leader election, whereas the following lab will discuss the second part, log replication. We will use files contained in this package, which you should download and extract locally.

Learning section

Consensus

Consider a distributed system which is supplied a series of operations, each to be executed by all its processes. The operations might be supplied by multiple independent clients, so to maintain a consistent state across the system, all the processes have to operate on the same initial values, execute all operations in the same way and in the same order. For example, a distributed key–value database has to assure that all its processes start with the same initial set of data, that all processes implement the `store` and `retrieve` operations in the same way, and that all processes execute all incoming requests in the same order; otherwise, clients might observe inconsistent responses (e.g., a `retrieve` operation would return an incorrect value if it was executed by some process after a consecutive `store` operation). In the field of distributed system, this problem is known as **state machine replication (SMR)**, as all processes of such a system have to implement the same automaton, and then execute the same series of its state transitions.

In practice, replicating a state machine itself can be implemented relatively straightforwardly: all processes can be built from the same, deterministic source code (thus they execute exactly the same program), and the initial values can be hard-coded. However, replicating a series of operations received from multiple clients is, in turn, a fundamental problem of distributed systems. It is named **log replication** because the series of operations can be viewed as consecutive entries in a log. If the entries are ordered the same way in every process's log, then to maintain a consistent state of the system, each process just needs to execute the entries in the order they appear in its log. In other words, the processes need to agree on the order of entries (i.e., for all n, the n-th entry in each process's log must be the same.)

To this end, distributed systems usually employ some **consensus** algorithm, which instructs their processes how to agree on a common value. Multiple consensus algorithm have been designed so far, for instance, **Paxos**, which was presented during the lectures, and **Raft**, which will be discussed during this lab.

Assumptions

In the following discussion we assume the crash-recovery failure model (individual processes crash and recover, but there are no Byzantine failures and no network failures). Moreover, we assume that

all processes in the system implement the same algorithm (e.g., they are built from the same, deterministic source code).

Raft

Raft is a consensus algorithm which manages a replicated log across all processes of a distributed system. The process of appending new entries to the log is coordinated by a single designated process, named **leader**, as this approach facilitates assuring correctness and safety of the system (although it restricts scalability as all append requests are processed by a single process). Therefore, Raft decomposes the consensus problem into two subproblems:

1. **leader election**,
2. **log replication**.

A full solution requires also considering a few implementation details, such as:

- log compaction (the log should not grow indefinitely),
- client interaction (the system's interface should provide linearizable semantics),
- membership changes in the cluster (the system should handle dynamic addition and removal of processes).

This lab presents leader election. Log replication will be discussed during the following lab, and the implementation details will be a part of the third Large Assignment.

Basic notions

From the Raft's perspective, time is divided into **terms**, numbered with consecutive integers. Each term begins with an election of a new leader, and lasts as long as the leader is alive and responsive to other processes. If the election fails, a next term is begun by launching a new leader election.

There is no physical global clock in Raft. Instead, the terms serve as a logical clock, and every process tracks the current term number itself. Processes append the term number to every message they send. When a process receives a message with a larger term number than it has locally, it updates its term number and steps down from being the leader or a candidate for the leader (if it is one).

Leader election

The process of leader election can be initiated by any process in the system. Such a process nominates itself as a **candidate**, increments its term number, votes for itself, and sends *RequestVote* messages to all other processes. In a response to the message, every process votes on the candidacy. A process votes for the candidate, if it is the first vote request it received for the term number greater than or equal to its term number. Otherwise, if it is not the first vote request the process received for this term number (so either the process already voted in this term or the message's term is older), the process votes against the candidate.

During the election process, the candidate can also receive a message from another process claiming to be the leader. If the leader's term is greater than or equal to the candidate's term, then the candidate recognizes the leader and becomes a **follower** itself; otherwise, the candidate just responds to the message and continues waiting for replies with votes. Followers only respond to messages from leader or candidates, and at boot each process starts by being a follower.

A candidate wins an election and becomes the leader if it receives a majority of votes. If it does not receive the required number of votes over an *election timeout*, and it does not receive a message

from a leader, then it restarts the election process (i.e., it increments its term number, votes for itself, and sends new *RequestVote* messages).

To notify other processes about its leadership, the elected leader periodically sends *Heartbeat* messages to them. Processes always respond to *Heartbeat* messages (if the *Heartbeat* has an outdated term, this lets its sender update its term). When some follower does not receive a *Heartbeat* over the election timeout, it assumes that the current leader is no longer active (e.g., it has crashed), and if it does not receive any message from a new candidate, it initiates election of a new leader itself.

The algorithm is not guaranteed to terminate. In practice, however, randomizing processes' timeouts is enough to make nearly all elections end after one round. The timeout should be randomized each time it is restarted, that is, each time a valid message is received.

Moreover, for the election to work smoothly, apart from the randomization, the following must hold:

$$\text{broadcast period} \ll \text{election timeout} \ll \text{mean time between failures}$$

where $x \ll y$ means that y is much bigger than x (in our context, at least 10-times bigger).

Raft requires a majority of the system to be alive and responsive. Otherwise, given the election rules, it would not be able to elect a leader.

Invariant of leader election

In any given term, there is at most one leader (could be zero when, for instance, there are two candidates and each receives a half of the votes). This is because in one term every process has one vote, and a majority of votes is necessary for a process to assume its leadership.

In a nutshell

At any give time, each Raft process can be in one of the three states:

- leader,
- candidate,
- follower.

Every process:

- at boot becomes a follower,
- if an inbound message contains a newer term, updates its term to match it and converts into a follower.

Followers:

- a follower must respond to messages from leaders and candidates,
- if a follower receives no message or does not grant vote over a timeout, it converts into a candidate.

Candidates:

- after a conversion, a candidate starts an election,
- if the candidate receives a majority of the votes, it becomes a leader,
- if the candidate receives a heartbeat from a leader, it becomes a follower,
- if the election times out, the candidate starts a new one.

Leader:

- it prevents new elections by broadcasting heartbeats periodically.

The transitions between the states are also illustrated in a diagram.

Small Assignment

Your task is to implement the leader election of Raft. You shall complete the implementation of the system provided in the template, following the interfaces and doc comments defined there.

In contrast to real-world implementations, do not add any randomization to timeouts, but use them as they are provided (this is necessary for testing). The leader shall send 10 *Heartbeats* during a timeout.

This Small Assignment is worth **2 points**. To run the system you should use the executor system you implemented as the first Large Assignment.

Additional Homework

Read the Raft paper if you wish to read the original source. In the paper the *Heartbeat* message is defined as an empty *AppendEntries* message (we renamed it here to focus solely on the leader election).

If you have problems with understanding the leader election in Raft, read this blog entry, watch this visualization, or play with a visualization here.

As an example of a real-world application of Raft we recommend browsing *TiKV*, which is a distributed key–value database. What is more, it is implemented in Rust.

When employing Raft in real-world applications, one should remember about the assumption of no Byzantine and no network failures. Without it, Raft does not guarantee liveness and Cloudflare, as they described in their blog post, learned this the hard way. The algorithm, however, can be extended to handle network failures. Read the blog post by Heidi Howard and Ittai Abraham to learn how.

During the next lab, we will discuss the remaining part of the Raft consensus algorithm, the log replication.

Authors: F. Plata, K. Iwanicki, M. Banaszek, W. Ciszewski.