Raft protocol: implementation in Akka

Distributed Systems 1 course project

Maria Pia Natale, 189160
mariapia.natale@studenti.unitn.it
Alessia Tovo, 189192
alessia.tovo@studenti.unitn.it

DISi - University of Trento

September 6, 2017

1 Introduction

This document has been written with the aim to explain the work done for the Distributed Systems 1 course project.

The project consists of implementing Raft Consensus Protocol with the ActorModel tipical of Akka, that is a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala.

In first sections the reader will find a brief description of Akka and a brief explanation of Raft protocol, then a more precise description of our work will follow.

2 Technology used

Akka is a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala. We used Java to implement our version of Raft protocol.

Akka allows to implement easily the Actor Model, a model to design and programm efficiently asynchronous systems avoiding simultaneous access to shared data that could generate a concurrency problem.

The Actor Model consists of a set of protected entities, called Actor that have an internal state and a certain logic . Actors are like Java objects but more isolated from the outside and they use messages to communicate. Inside the Actor System, Actors may run in parallel and messages that they send respect FIFO order and might be lost. Moreover, sending operation is non-blocking.

When an Actor receives a message, it starts to process it, without time boundaries, and can send another message as reply. This guarantess a process flows, even if messages might be lost or delayed.

Since Actor variables and methods are not accessible from the outside, all the informations needed are sent via message passing, but obeject passed through it are just a copy, in order to guarantees inaccessibility from other Actors.

All this characteristics are specific to Akka and allow us to implement the protocol in a more real environment, considering net congestions, messages delays, loss of messages, parallel executions of process and self-closed environment.

3 Raft overview

Raft is a leader-based algorithm used to solve the Consensus problem. A Consensus algorithm allows a cluster of machines to work together in order to reach an agreement in a system where machine failures may occur, so a Consensus protocol must be fault tolerant, resilient and it has to respect these specification:

- **Termination:** every correct process eventually decide on some value;
- **Uniform Integrity:** each process decides at most once;
- **Uniform Validity:** if a process decides v, then v must be proposed by some process;

- **(Uniform) Agreement:** No two correct (any) processes decide differently.

In Raft protocol, the Consensus problem arises during the leader election, where each server has to vote at most for one Candidate and must agree on the elected leader. At any moment of the execution, a server can be in one of these states: *follower*, *candidate* or *leader*. According to the state, a server has different behaviour:

- The *Leader* is responsible for the log replication and handles the interactions with the clients. At all times, at most one server in the state of leader is allowed.
- The *Candidate* is only used during the leader election.
- The *Follower* is a completely passive peer; it issues no requests on its own but simply respond to requests from leaders and candidates.

In Figure 1, the transition between each state is showed.

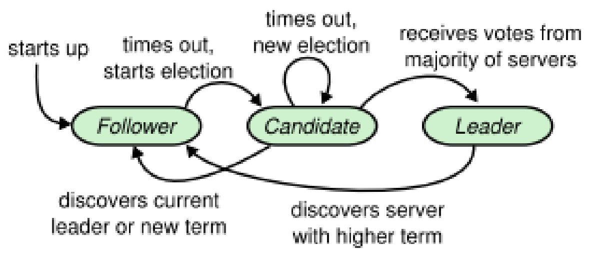
starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Figure 1: Transition between states

Time in the Raft execution is divided into *terms*. Each term can be split in two parts, one for the leader election and the other for the normal operation. A new term begins when the leader in the previous term has crashed. In some cases, it can be possible that two candidates receive the same number of votes; in this case the election is not valid and the current term ends with no leader.

In the Raft protocol, servers communicate using two types of `Remote Procedure Calls (RPCs)`, each one divided into request and reply. The first one, `AppendEntries RPC`, is used to add an entry to the log and to send heartbeats, empty messages sent by the leader to the followers to maintain the leadership. The second one, `Vote RPC`, is used by the candidate to ask votes and win the election.

Each server must keep some information about itself. Some of these information are saved in a stable storage that define the `Persistent State` of the server: *current term* to indicate the term in which the server is, *votedFor* that defines the Candidate voted by the server and *log[]* which is an array of log entry composed by the term and the command for the state machine. The other information are stored in a `Non-persistent State`. The server keeps in memory its *state*, the *leader ID*, the *commitIndex* that represents the next entry to be committed, the array *nextIndex[]* that contains the indexes of the next entries that the server must send to a peer and the array *matchIndex[]*, which contains the indexes of the highest log entry to be replicated.

The following sections will explain in a more detailed way the different phases of the protocol.

3.1 Elections

As introduced in the previous section, Raft is a leader-based algorithm, hence the leader election is an important phase of the protocol. A leader election starts at the beginning of the protocol or when the leader of the previous term crashes. As said before, at the beginning a server start up as a `follower`. If it receives a valid `RPCs` from a candidate or a leader it remains in the `follower` state. Each server has an `ElectionTimeout` that when triggered will start a new election assuming that the old Leader has crashed.

To start a new election, a server in the Follower state increment its current term and switch to Candidate state. After that, it votes for itself and sends a new `VoteRequest RPC` to all the other servers in the cluster. A `VoteRequest` message contains information about the current term and the last committed entry of the Candidate. It becomes Leader if it receives votes from the majority of servers that are on its same term. Each server can vote for at most one candidate in a given term.

When it receives a `VoteRequest` from a Candidate it first compare the term received with its current term, if its term is older than the term received from the Candidate it will update its term with the received one, switch to `Follower` state and then choose a new `ElectionTimeout`. If its term is newer than the one received it will reply to the Candidate with a `VoteReply RPC` telling its term without granting the vote to him.

To avoid that a new Leader election modifies the entry committed by the previous Leader (Safe Requirement), a node can vote only to Candidate with an up-to-date log. Raft determines which of two logs is more up-to-date by comparing the index and term of the last entries in the logs. If the logs have last entries with different terms, then the log with the later term is more up-to-date. When a Candidate receive a `VoteReply` message it compares the term received with its current term, if the term received is newer it performs a `StepDown` operation, going back to Follower state and choose a new `ElectionTimeout`. If it receives the majority of the votes it switch to Leader state.

To maintain its leadership, a Leader periodically send an `Heartbeat` to the other server to inform them that it is still alive. An `Heartbeat` is nothing but an empty `AppendRequest`. Again, the servers that receive the `AppendRequest` will compare their term with the one received in order to neutralize old Leaders temporarily disconnected from the cluster.

3.2 Normal operation

Normal operation consists of command sent by Client and register command in Leader's log and sequentially in peer's log.

When Leader receives a command sent by Client, it adds a *term, command* pair in its log, increments its index regarding next free location in log and sends to all peers an `AppendEntries RPC`.

When a follower receives that RPC, it checks if its log's index and leader's log index have the same value and if last *term* saved in leader's log is the same in its log. This process assures that logs of all servers

are consistent. This is a key property of Raft protocol.
To decide if a command can be committed or not, leader checks if its term is saved on majority of servers' log and if servers' log contains at least one entry with its term. These rules combined with elections rules make Raft a safe protocol.

3.3 Client protocol

Client has not a central role in Raft protocol, since it is a passive protocol element. It just send a command to the Leader, if it knows exactly which server is the leader, or to a random server. This server, if is not the Leader, redirect client to the Leader. After the sending of the message, Client waits a reply to know if the command has been committed or not. If it does not receive a reply into certain time, it will send the same command to another server. This may happen if Leader crashed, for example.

4 Implementation

To implement Raft protocol we used Java as programming language and Akka, a specific toolkit to implement distributed system without concurrency problems.
Some simplifications have been adopted in developing phase. The number of server is fixed, defined in the `application.conf` file, and we did not consider the option of adding or removing servers during execution.
Another simplification adopted is that there is only one client, that respect completely the client protocol stated by Raft, exception made for command sending. More details will reported in section 4.5.
Akka allowed us to build an Actor Model, where each server and the client is an Actor, with its own private variables and methods. Both servers and client are elements of the same system, called Actor System. Each system elements has its own hostname and port. How Actors were implemented is better explained in next subsection (4.1).

4.1 Actors model

As reported in previous section, both servers and client are Actors. An Actor is an object which encapsulate state and behavior, it communicate exclusively by exchanging messages which are placed into the recipients mailbox. But the most important aspect is that it is not possible to look inside an actor and get hold of its state from the outside, unless the actor unwisely publishes this information itself.[1]

Actors configurations, like name, hostname, ip port were defined in the `application.conf` file, which contains also information about number of servers and timeout values.

When program starts its execution, all the elements (servers and client) start to run independently, but always part of the same and unique Actor System, called *RaftSystem*. Actors don't know anything about others Actors, like id, state, variables. An Actor only knows the number of Actor in the Sytsem.

Actors communicate through message passing, this explain the large number of Java Class used. Each message contains different parameters and is used in different context, depending on protocol phase and state of server.

4.2 Elections

As said in Section 3.1, at the beginning each server is in `Follower` state. The change of the state is handled by the `StateChanger` class that will siwtch the server's state based on the current state and the `stepdown` variable that is used to indicate that a Candidate or a Leader has to switch back to the Follower state. At the beginning of the protocol each server receives a `StartMessage` from the cluster initializer. On the receiving of it, the Follower delete any previous `ElectionTimeout` and schedules a new `StateChanger` message to itself with a timeout chosen in a range specified by the `MinTimeout` and the `MaxTimeout` variables in the `application.conf` file. The reception of this message will change the server's state from Follower to Candidate. When in Candidate state, the server delete any

[1] http://doc.akka.io/docs/akka/snapshot/java/general/actor-systems.html

previous `ElectionTimeout`, clears the `votes` array that contains the votes received from the other server, updates the current term and votes for itself. Then it schedules a new `ElectionMessage` message that will start a new election if the previous one did not end. Finally it send to all the other peers a `VoteRequest` message containing:

- *senderID*;
- *currentTerm*;
- *lastLogIndex*, which is the index of the last committed entry;
- *lastLogTerm*, which is the term of the last committed entry;

When a server receives a `VoteRequest` message, it compares the received term with its current term. If the term received is greater than its current term, the server will call the `stepdown` procedure that updates server's term and switch the server's state to Follower sending a `StateChanger` message with the `stepdown` variable set to `true`. If the term received is less than the term saved by the server, it will inform the Candidate that its term is greater. In the two term are the same, the server checks if the log information received by the Candidate are up to date . A log is up to date if the received `lastLogTerm` is greater than the term of the last committed entry by the server or when these two are equal and the received `lastLogIndex` is greater than the log size of the server. If the received information are up to date, the server will grant the vote to the Candidate sending to him a `VoteReply` message containing:

- *votedID*, -1 if the server did not vote for the Candidate;
- *term*;
- *senderID*;
- *granted*, a boolen variable that indicates if the server voted for the Candidate;

When the Candidate receives the `VoteReply` it compares the received term with its current term calling the `stepDown` procedure if its current term is lass than the one received. Otherwise if it receives a successful reply from the majority of the server it will schedule a new `StateChanger` message to change its state to Leader.

When in Leader state, the server will start to send `Heartbeats`, empty `AppendEntries` message, to the other servers to hold its leadership. To avoid the reception of an `Heartbeat` from old Leaders temporarily disconnected from the cluster, the server will always compare the received term with its current one, informing the sender of the message if its term is greater in order to neutralize them.

4.3 Normal operations

We use the term *Normal* to indicate commands sent from Client to Leader, and log's appending by Leader and all Followers.

Log consistency is one of the main properties in Raft protocol: Leader's log is the *true* one and Followers must agree on that. As written in Section 4.2, log is used also during the Election phase, expecially when current leader crash and a new election starts.

The client initially sends a command to a random server, since it does not know which one is the leader. If the reached server is not the leader, it sends to the client the address of the leader. In this way, the client can send the next commands directly to the leader.

When leader receives a command from client, it adds it to its log in form of pairs `<term, command>`, where `term` is the *Current Term*, and `command` is a String that indicates the command received from Client, like *Command_0, Command_1, etc.* in our implementation. The Leader increments the Array List `nextIndex`, incrementing the value of its *id_position*.

Then, it calls procedure `sendAppendEntries(peer)` for each peer, excpet itself. In this procedure, leader sends to a peer a `AppendRequest` message containing:

- *leaderId*;

- *currentTerm*;
- *nextIndexLogPeer -1*, which is the index where the peer should insert new entries;
- *lastTerm*, the last term saved in the Leader's log;
- Array List `entries`, which contains all new entries in Leader's log;
- *commitIndex*, index of the last entry committed by the leader.

To create an entry, we use a procedure called `getEntriesToSend(log, index)`. This procedure takes all Leader's log entries from the last index used by peer to last Leader's log entry. If the leader receives only one command, `entries.size()` should be equal to 1 and it should contain only one entry, the one regarding the command received.
When a peer receives an `AppendRequest` message, firstly it checks if there is `term` consistency.
If the received term is greater than the current term saved by peer, it calls the procedure `stepdown(term)` to update its own term.
If the receivedc term is less than the current term saved by peer, it informs the leader that the command can not be committed through an `AppendReply` message containing a negative reply. In both cases, the command has not been committed by that peer. Then peer proceeds by checking log's index. If its next index is equal to 0 or its last term saved in log is equal to the one received by the leader contained in `prevTermReceived`, the command can be committed. The peer updates its log index by calling the `storeEntries(prevIndexReceived, entriesReceived, commitIndexReceived)` procedure. This procedure has three parameters:

- *prevIndexReceived*, the index where the peer must write the next entry;
- *entriesReceived*, the list of all new entries;
- *commitIndexReceived*, the index of the last entry committed by the Leader.

If peer's log is empty, the new entries are simply added to it. Otherwise, it starts to check from last index till the end if entries in peer's log are correct. If not, that one and following ones are removed from log, and one by one, correct entries are added. Then, peer's commit index is updated taking the minimum value between *commitIndexReceived* and actual log's index.

After this procedure call, peer send an `AppendReply` message to the leader sending:

- *id*, the peer's id;
- *currentTerm*, the term known by the peer;
- *success*, a boolean flag to inform if the command can be committed or not;
- *indexStories*, the index of the first empty position in peer's log;
- *lastTermSaved*, the term saved in the last peer's entry;
- *commitIndex*, the index of the last entry that can be committed by the peer.

When the Leader receives an `AppendReply` message, it firstly checks if peer's commitIndex is equal to its own. This has been made to understand if the received reply regards an old command or last one.

If the received reply regards the last command, the Leader saves the response in an Array `getReply`, in order to know which peer answered, and saves the received term in another Array `termsPeers`, in order to perform a consistency check before the commitment of the command.

If the received term is greater than the Leader's term, the procedure `stepDown` is called. Otherwise, if the term is consistent and the received reply contains a positive answer, the Leader updates the peer's next index.

When the Leader receives replies from more than an half of peers, two procedures are invoked:

- `checkMajorityReply`, that checks if there are more than an half positive replies about committing or not;

- `chechMajorityCurrentTerm`, that checks if its term is saved into more than an half of peer's logs.

If both procedures return a positive value, the Leader informs peers to update their commit index, since command can be committed, then it informs the client that its command has been committed.
If at least one procedure returns a negative value, the Leader waits other replies. If the Leader receives the reply from each servers and procedures's outcome is still negative, the Leader informs the client of the unsuccesfully execution of its command.

4.4 Pausing a server

Each server can be temporarily disconnected from the network. Each time a server receives a message, there is a probability of 1% to paused for 1 second. When paused, the server will skip all the messages received in the meantime except the `ResumeActor` message that will resume the peer.

4.5 Client protocol

In this Raft protocol implementation, there is only one client, sufficient to understand how clients interact with servers.
As reported in Section 4.1, the client starts to run in the same moment of the servers, since they are part of the same Actor System. The client knows the address of the Servers, so it can start sending messages to them.
When the Client sends a `SendCommand` message, it waits for a positive or negative reply from leader. Commands are just simple Strings contained in an Array. Their utility is just to show how protocol works and not to compute something defined.
At the very beginning of the protocol execution, the client does not know which server is the leader, so it sends the first command to a random server. If the contacted server is the leader, the command is forward to servers, and then leader sends an `InformClient` message about the command outcome. Then, the

client sends another command to Leader.

If the server contacted by Client was not the leader but a simple server in a state of follower, this one informs the client with a message that contains the *id* of the leader. If the initial election has not been finished yet, Client will receive as *id* a negative value, like -1, and it will know that there is not a leader yet. In this implementation, when Client receives this information, it waits 1000 milliseconds to avoid congestion of continuos `SendCommand` messages to the wrong server.

When the Client sends command to the right server, the Leader, it will wait the Leader's reply before sending another command to it. Also this choice has been made to avoid continuous messages sent from the Client, since the list that contains them has limited size.

When the Client has no other commands in its list, it sends a *FINISH* command to inform the leader that there are not other messages. This choice has been made only to know when to stop the system with `context().system().shutdown()`.

5 Conclusions

Raft is an easily understandable protocol, different from other Consensus protocols, like Paxos that is not frequently used because of its difficult implementation. There are many Open Source implementations of Raft. To develop our Raft protocol, we were based on schematic explanation given by Prof. Montresor in his lectures and the simulation given in Github repository `https://raft.github.io/` to better understand how servers interact each other. We adopted some semplifications, like static number of servers and different concept of answering timeout in *Normal operation* (4.3) context, that is the time that the Leader should wait to get the reply from followers.

Our Raft protocol execution could be defined ”static” because of static number of commands that client can send and static number of servers, that is no new server can join to the initial configuration.