

# Raft Consensus Algorithm - Phase 5

Miulescu Cristina-Maria

## Abstract

Raft is a distributed consensus algorithm that was developed as an alternative for Paxos. It is mainly used to manage consistency and log replication. Compared to Paxos, Raft is claimed to be easier to understand and apply to actual systems without making compromises in performance or correctness. In this essay, I am going to analyze some of the current implementations of the algorithm and draw a comparison with the original implementation.

## 1 Introduction

Distributed consensus algorithms allow multiple nodes in a distributed system to agree on the values of some replicated log. They are essential to the development of fault tolerant services making them to act as one.

There are two main classes of distributed consensus algorithms. Byzantine consensus algorithms assume the presence of faulty components that may send wrong data to their peers. The other type of consensus algorithms will experience the fail-stop failures which assume that the components of a system will either serve or stop operating.

One of the most known non-Byzantine consensus algorithm is Leslie Lamport's Paxos [3]. This algorithm offers two main advantages of ensuring correctness and efficiency in the standard case. There are many implementations of Paxos such as Chubby [2], Megastore [1], Azure. However, the disadvantage pointed by many regarding this consensus algorithm is that it is difficult to grasp and implement.

As a result, after several struggles with Paxos, Diego Ongaro and John Ousterhout started working on an distributed consensus algorithm with the primary goal of understandability called Raft. [4]

### 1.1 How Raft works

Firstly, Raft was designed to be easier to understand by dividing the consensus problem into multiple sub problems. At any time the nodes can have one of the following state:

- Follower
- Candidate
- Leader

Initially, all the nodes in the distributed system are in the Follower state. As long as they receive the heartbeat from the leader, they remain in that state. If they do not receive a message from the leader, the followers can transition to the Candidate state and the election starts. Candidates request votes to other nodes and the one that has the majority becomes the Leader. The Leader is responsible for the changes in the system by replicating the logs to the majority of nodes in the cluster, forcing the other logs to agree with its own.

## 2 Implementation

The project was implemented in Python making use of the socket package in order to allow communication between nodes as well as with multiple clients. The main steps of the implementation will be presented in the following subsections.

### 2.1 Implementing the client and server

The first step was to implement a server and a client and make them communicate between each other.

The **client** will specify 2 arguments:

- the port it wants to connect to
- a unique id to be recognized in the network

The messages send by the client to the cluster nodes will have the following format: client + id + @ + message. An example would like: "client4@set b 5".

The **server** will have to specify also 2 arguments:

- the port
- the name

When nodes communicate with each other, they will have a similar message format as the client, for instance: "s2@AppendEntries".

### 2.2 The Log Manager

The Log Manager takes care of the commands that the client can send to the server and receive back a response. There are 4 commands allowed:

- get
- set
- delete
- show

Each node in the cluster will have its own Log Manager to keep track of the State Machine changes. When a node boots up it will recover its logs from a log file. A log will specify: the index, the term in which it has been applied and the actual command. An example of such a log file would look like this:

```
0 1 set a 4
1 2 set b 30
2 3 get b
3 3 set c 5
4 3 delete b
```

Therefore, the Log Manager takes care of executing the specified commands as long as logging them to the node's log file. Now that the communication between the server and client is set up, it's time to take care of the algorithm itself.

```

cris@cris: MINGW64 ~/Documents/Projects/DA/raft/src (raft-dev-2)
$ python client.py 10000 1
[*] Connecting to localhost port 10000

[*] Type your message:
>> get ana
[*] Sending: get ana
[*] Received:

>> GET response:
4

[*] Type your message:
>> set d 8
[*] Sending: set d 8
[*] Received:

>> SET response:
d set to 8

[*] Type your message:
>> get d
[*] Sending: get d
[*] Received:

>> GET response:
8

[*] Type your message:
>>

cris@cris: MINGW64 ~/Documents/Projects/DA/raft/src (raft-dev-2)
$ python client.py 10000 2
[*] Connecting to localhost port 10000

[*] Type your message:
>> get d
[*] Sending: get d
[*] Received:

>> GET response:
8

[*] Type your message:
>>

```

Figure 1: Handle multiple clients

## 2.3 Handle multiple client connections

By using the threading package, a node can accept multiple connections from different clients as well as from the other peers in the cluster. The mechanism works as follows: when a new connection is established, a new thread is spawned to handle the connection.

Also, on the Log Manager side, a client lock will be put in order to execute an operation one at a time.

Listing 1: LogManager lock

```

import threading

def LogManager:
    self.client_lock = threading.Lock()
    ...
    def handleOperation(self, command):
        operands = command.split("_", 2)
        operation = operands[0]

        with self.client_lock:
            if (operation == "get"):
                key = operands[1]
                response = ">>_GET_response:_\n" + self.get(key)
            ...

```

## 2.4 Implementing RAFT

I will start by describing the implementation for AppendEntries RPC and RequestVote RPC.

### 2.4.1 RequestVote RPC

The RequestVote RPC is used by a node in the Candidate state to gather votes from the cluster peers. The RequestVote contains information about the candidate's currentTerm, previousIndex and also previousTerm.

### 2.4.2 AppendEntries RPC

The AppendEntries RPC is used by the leader for two main reasons. The first is to send heartbeat signals to the other nodes in order to keep the leadership. The second reason is to maintain the log replication consistent across all the nodes. This means that the leader will make sure that its followers will have the same logs as the leader. Also, if a follower is not up to date it will inform its last index to the leader during the heartbeat.

Now that the server-client connections are taken care of, we can move on to implementing the actual algorithm. I will structure the logic of Raft from three points of view:

- Follower
- Candidate
- Leader

### 2.4.3 Follower

First, in the beginning the state of each node is set as Follower. Also, when a node is started the first thing it does is to recover its logs from the log file through its Log Manager.

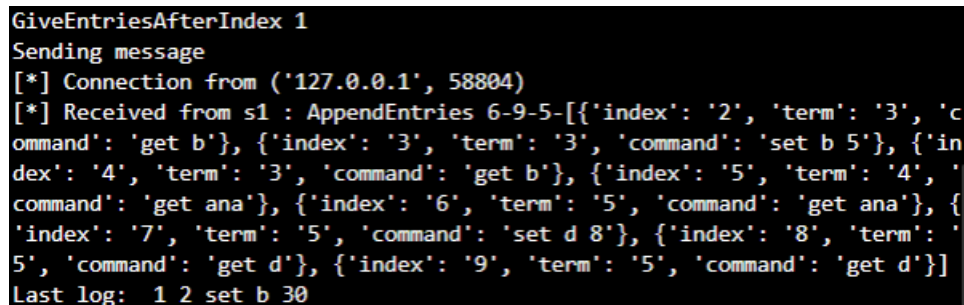
Next, it will start a countdown till starting a new election. As long as the leader reaches out before the election timer is up, the node will stay in the following state.

When a leader sends the heartbeat to its followers it will send an AppendEntries RPC with the following information:

- currentTerm
- previousIndex
- previousTerm
- entries

The heartbeat will send this RPC with empty entries. The follower will receive the message and will compare the leader's information with its own and send back one of these responses:

- I am up to date
- GiveEntriesAfterIndex



```
GiveEntriesAfterIndex 1
Sending message
[*] Connection from ('127.0.0.1', 58804)
[*] Received from s1 : AppendEntries 6-9-5-[{ 'index': '2', 'term': '3', 'c
ommand': 'get b'}, { 'index': '3', 'term': '3', 'command': 'set b 5'}, { 'in
dex': '4', 'term': '3', 'command': 'get b'}, { 'index': '5', 'term': '4', '
command': 'get ana'}, { 'index': '6', 'term': '5', 'command': 'get ana'}, {
'index': '7', 'term': '5', 'command': 'set d 8'}, { 'index': '8', 'term': '
5', 'command': 'get d'}, { 'index': '9', 'term': '5', 'command': 'get d'}]
Last log: 1 2 set b 30
```

Figure 2: Follower sends GiveEntriesAfterIndex request to leader and receives back AppendEntries RPC

This sums up the role of a Follower in the context of Raft.

#### 2.4.4 Candidate

A leader election is started by a candidate server. A server becomes a candidate if it receives no communication by the leader over a period called the election timeout, so it assumes there is no acting leader anymore. It starts the election by increasing the current term, voting for itself as new leader, and sending a message to all other servers requesting their vote. A server will vote only once per term, on a first-come-first-served basis. If a candidate receives a message from another server with a term number larger than the candidate's current term, then the candidate's election is defeated and the candidate changes into a follower and recognizes the leader as legitimate. If a candidate receives a majority of votes, then it becomes the new leader. If neither happens, e.g., because of a split vote, then a new term starts, and a new election begins.

Raft uses a randomized election timeout to ensure that split vote problems are resolved quickly. This should reduce the chance of a split vote because servers won't become candidates at the same time: a single server will time out, win the election, then become leader and send heartbeat messages to other servers before any of the followers can become candidates.

```
$ python server.py s1 10000
C:\Users\crist\Documents\Projects\DA\raft\src\logs\server-configuration.tx
t
[*] Recovering logs...
[*] Current term: 5
[*] Server started on ('localhost', 10000) with timeout 15.0 seconds
True
[*] Starting election...
Sending message
[*] Connection from ('127.0.0.1', 58651)
[*] Received from s2 : Count on me
[*] I am Leader
{'s1': True, 's2': True, 's3': False, 's4': False}
Sending heartbeat...
Sending message
[*] Connection from ('127.0.0.1', 58653)
[*] Received from s2 : I am up to date
```

Figure 3: Candidate becomes leader after receiving a majority of votes

#### 2.4.5 Leader

Once the leader is chosen, it will start sending heartbeat messages to the other nodes.

The leader is also responsible for the log replication. It accepts client requests. Each client request consists of a command to be executed by the replicated state machines in the cluster. After being appended to the leader's log as a new entry, each of the requests is forwarded to the followers as AppendEntries messages. In case of unavailability of the followers, the leader retries AppendEntries messages indefinitely, until the log entry is eventually stored by all of the followers.

```
[*] Received from client : get ana
Sending heartbeat...
Sending message
Sending message
Sending message
```

Figure 4: Leader receiving from client

#### 2.4.6 Client

The client can connect to any of the cluster nodes by specifying the port. If the node is not the leader, it will inform the client that it cannot execute the operation. If the client connects to the leader, then the command will be executed and a response will be sent to the client.

```
crist@cristina MINGW64 ~/Documents/Projects/DA/raft/src (raft-dev-2)
$ python client.py 10001 1
[*] Connecting to localhost port 10001

-----
[*] Type your message:
    >> get ana
[*] Sending: get ana
[*] Received:

Sorry, I am not the leader. Last leader I heard from is: s1

-----
[*] Type your message:
    >>
[*] Closing socket
```

Figure 5: Client sends command to Follower

```
crist@cristina MINGW64 ~/Documents/Projects/DA/raft/src (raft-dev-2)
$ python client.py 10000 2
[*] Connecting to localhost port 10000

-----
[*] Type your message:
    >> get d
[*] Sending: get d
[*] Received:

>> GET response:
8

-----
[*] Type your message:
    >> 
```

Figure 6: Client sends command to Leader

#### 2.4.7 When leader fails

When the current leader fails or the algorithm starts up, a new leader must be chosen.

A new term begins in the cluster in this scenario. On the server, a term is an arbitrary length of time during which a new leader must be elected. Each term begins with the election of a leader. If the election is successful (i.e. a single leader is elected), the term continues with the new leader orchestrating usual operations. If the election fails, a new term begins, complete with a new election.

```
<class 'str'>
[*] Connection from ('127.0.0.1', 61227)
[*] Received from s1 : AppendEntries 7-11-6-[]
Last log: 9 5 get d

<class 'str'>
True
[*] Starting election...
█
```

Figure 7: Follower becomes Candidate after Leader fails

#### 2.4.8 Conclusion

In conclusion, implementing the Raft Consensus algorithm was very approachable as the primary goal of the authors was understandability. Also, the main idea of the algorithm has been developed but there may be tested more scenarios in the future, cases such as: multiple leaders.

## References

- [1] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. 2011.
- [2] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, 2006.
- [3] Leslie Lamport. Generalized consensus and paxos. 2005.
- [4] 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.