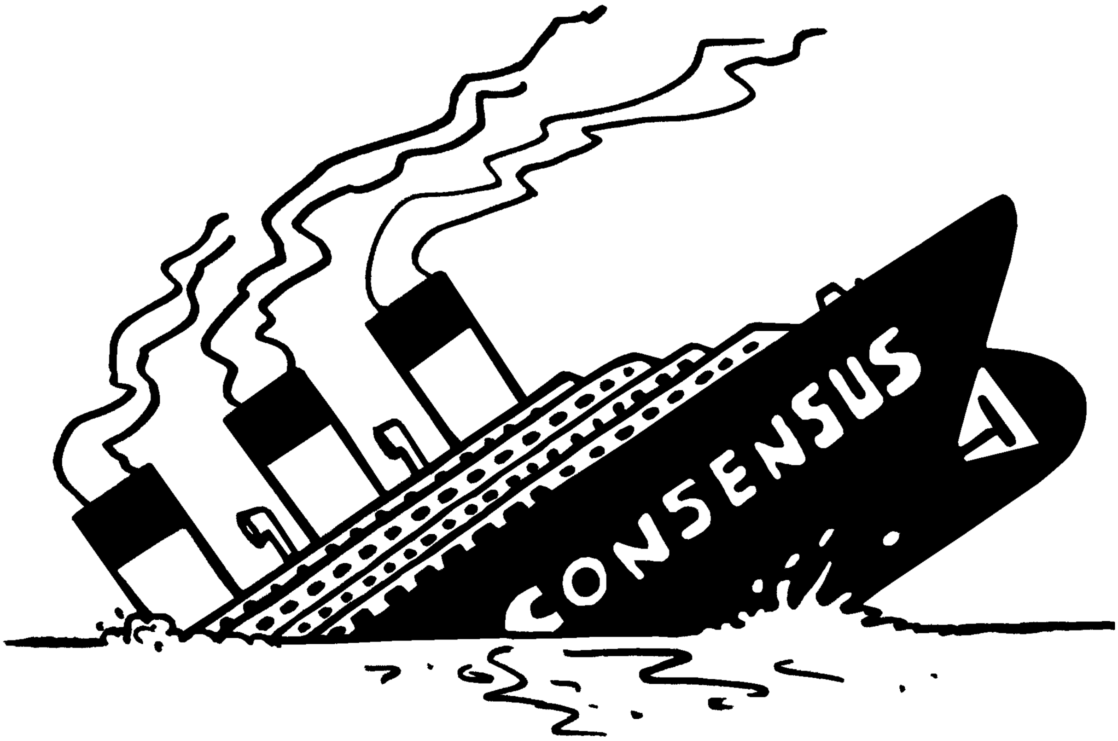
DTU Informatics — 02228 - Fault tolerant systems

FINAL REPORT
RAFT CONSENSUS ALGORITHM



Joachim Kirkegaard Friis s093256
Anders Nielsen s103457

December 7, 2014

Preface

This is the report done for the project in the course 02228 Fault-Tolerant Systems of fall 2014. The purpose of the project is to describe and evaluate the fault tolerance of a given system. In our project we have chosen to survey and try to implement a recently proposed solution to the consensus problem in distributed systems - Raft. Our approach to this, is to implemented a tool that applies Raft to a simulated distributed network and from this, derive a higher knowledge of the algorithm and it's ways of providing fault tolerance. The problem about consensus will be presented after which Raft will be described in this context. The design and implementation will follow t with notes on our experience through each step of the development. A conclusion will then summarize our result and experiences.

Contents

1. Introduction

This the final report for a project in the course "Fault tolerant systems" 02228. The purpose of this report is to document our experience implementing a consensus algorithm i.e. Raft.

The fundamental problem, when talking about consensus in a distributed system, will be presented at start. This will then show the motivation behind the project itself. And, as many solutions to this problem already exists, it should also be discussed why Raft in particular is relevant in the context of this project.

1.1. Problem

Reaching consensus in a distributed system means that all or at least the majority of processes in the network agree on some value or state of the entire system. This is often needed when processes might be faulty thus bringing reliability and availability at stake on single-point-of-failure. Upholding these properties in a given distributed system then relies on the architecture utilised and hardware implemented in the processes.
A simple solution to this could be to initiate a vote among all correct processes on to what the value is, in which the value is the result of the majority vote. The figure 1 below illustrates an example of a distributed system consisting of a number of notes. The value x is what the system must agree upon. Though here we have faulty process *P1*, taking the fact of connectivity of system aside, the system will suffer from this because of the now unreliable messages transmitted from this process might override this value at some of the other correct ones.

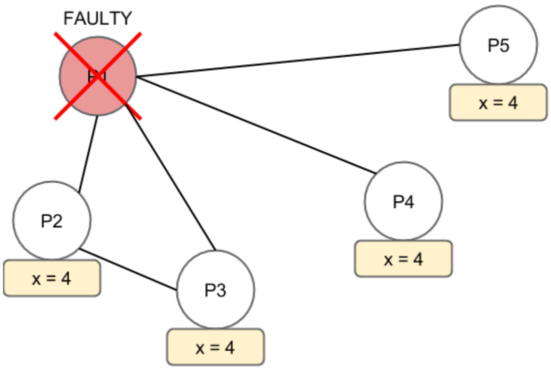


Figure 1: A distributed system consisting of a number of nodes with a faulty one.

The fundamental problem behind this, is that you cannot rely on the individual processes to be reliable and thus solely store the value. This means, that every process should store this value and should be able to be altered somehow. This boils down to a well-known problem - The Two Generals Problem.

1.1.1. Properties of distributed consensus

When want to define consensus, the goal is to satisfy a set of requirements i.e. properties that the distributed system must uphold. These are used to describe the systems fault tolerant features related to faulty processes. A faulty process can either fail by crashing or experience a Byzantine failure. Such a failure in the context of distributes system occur when e.g. a process for some reason transmits incorrect or malicious data throughout the network. Due to the arbitrary results of these kinds of failures, properties of a system are often distinguished by either tolerating them or not. A main difference in terms of properties of a system whether it tolerates Byzantine failures or not, is the validity and integrity. The integrity property for a system that does not tolerate Byzantine failures is as following [2].

- non-Byzantine failure tolerant:
  - **Validity**: If all processes propose the same value v, then all correct processes decide v.
  - **Integrity**: Every correct process decides at most one value, and if it decides some value v, then v must have been proposed by some process.
- Byzantine failure tolerant:
  - **Validity**: If all correct processes propose the same value v, then all correct processes decide v.
  - **Integrity**: If a correct process decides v, then v must have been proposed by some correct process.

The clearest commonality here, is that you should always be able to say that all correct processes must be able to decide on the same value or state, as mentioned earlier. Though the main difference is that with Byzantine processes, you must be able to say if they are all correct before stating they can derive the same value. This fact differentiates many solutions to this problem, depending on which property set they can satisfy.
Also, as discussed in [4] a consensus algorithm for a non-Byzantine system has the following properties: safety, availability, timing interdependency, and majority vote on procedure calls. The safety property can be directly related back to our validity and integrity properties, as the system's safety and availability is measured be the correctness of return result upon a request.

1.2. Motivation

The elements of the problem presented serve as motivation behind consensus in a system of unreliable components. Because how do you know for certain what a value is, when you for sure know that some components must be faulty at some point?

1.2.1. The Two Generals Problem

The basic problem of reaching consensus in a distributed network can be illustrated by the Two Generals Problem analogy.

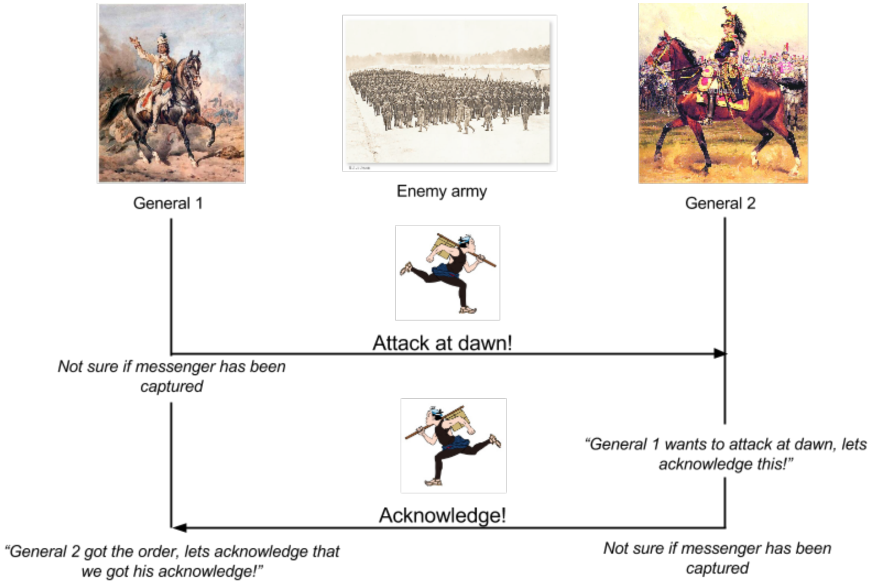

General 1
Enemy army
General 2
Attack at dawn!
Not sure if messenger has been captured
"General 1 wants to attack at dawn, lets acknowledge this!"
Acknowledge!
"General 2 got the order, lets acknowledge that we got his acknowledge!"
Not sure if messenger has been captured

Figure 2: Two generals tries to agree on when to attack the enemy by sending a messenger, but they are not sure the messenger survives his trip between their camps.

In the figure2 below we see two generals of the same army who want to attack an enemy army. But they have to attack at the same time in order to win. They cannot communicate directly to each other since they are at different fronts of the battlefield i.e. in their own camps. In the context of a distributed system, the generals here can be seen as two processes trying to agree on a value. General 1 then sends out a messenger to tell the second general, that they should attack at dawn. The second general then receives this message, but the first general cannot be sure of this (the messenger might be captured or killed by the enemy on his way to the second general and vice versa). Again, in the context of distributed system, the unreliability of messenger can directly related back to the unreliability of message transmission in a normal distributed system. In terms of Byzantine failure, this can be illustrated by the messenger being captured by the enemy and turned to spy on the generals i.e. given false information. So the second general sends the messenger back in order to acknowledge this. But the first general also has to acknowledge this, resulting in a never ending run for the poor messenger - thus the generals can never agree on when to attack the enemy.

2. Raft

As of writing this report, the most recent proposal to solving the consensus problem is Raft [4]. Diego Ongaro and John Oustershout argue that most consensus algorithms, such as Paxos [3] suffer from poor understandability and are hard to understand and implement. Diego and John introduce Raft as a simple and understandable solution to the consensus problem. The authors of Raft argue that Raft is superior in both the aspects of educational uses but also in performance gained from a less complex implementation. Their main motivation behind Raft is their point of view on the frequently used consensus algorithm Paxos [3]. They describe Paxos as being simple by its foundation but limited, resulting in many variations to solve more complex consensus scenarios. Another arguments they propose, is that Paxos might well work in theory but is hard to integrate in a real system because of its too simple foundation i.e. 'simple decree' decomposition (reaching aggreement on a single decision).

2.1. Components of Raft

Raft is described as having three main components, where the first two describes the behaviour of algorithm and the last describes how the algorithm ensures consensus safety properties.

- **Leader election**: A strong leader is elected which responsible for keeping the rest of the system in consensus for a term that ends if it fails.
- **Log replication**: The leader must accept log entries from clients and replicate them across the cluster, forcing the other servers to agree with its own log [4].
- **Safety**: The specified design provides safety such that [4]:
    - **Election Safety**: There is always at most one leader in the system.
    - **Leader Append-Only**: A leader can only append new entries to its log i.e. it cannot overwrite existing entries.
    - **Log Matching**: The logs of all servers are matching up to their latest common index and term.
    - **Leader Completeness**: No newly elected leaders will overwrite other server's log if its own log entry is not up to date.
    - **State Machine Safety**: When a server updates its log it must be up to date with the leader's log.

The above description of Raft serves as a rough overview of the algorithm. A more detailed description will be given during the implementation since the paper has implicit or unclear implementation specific components that we have to figure out ourselves - which we will also discuss their view of Paxos into account. And as we describe each component our experiences and findings in terms of implementation specific choices will thus be documented.

3. Requirements

The tool we are implementing should serve as an illustration tool, to show how the Raft algorithm works for given scenarios of faulty processes in a distributed system. The requirements are split into two different aspects i.e. functional and quality attributes:

3.1. Functional requirements

The end product must satisfy the following functional requirements given the MoSCoW method:

1. The tool must illustrate a scenario of a given set of processes in a distributed system, in which a leader is elected.
2. The tool must be an implementation of the Raft algorithm as specified in [4].
3. The user should be able to input parameters to vary the scenario, such as the number of processes.
4. The user must then be able to disconnect any process.
   a) If a leader is disconnected a new leader must be elected automatically.
5. The user must be able to request that a log entry is replicated to the system at run time.
6. The tool could be further extended with a visualisation of the given distributed system.

3.2. Quality requirements

The quality requirements of the illustration tool are inherited from the properties that are provided by the Raft algorithm.

1. The Raft implementation should have the following properties: Safety, availability, timing independence, and that commands complete as soon as a majority has responded.
   a) Safety: The properties described in section 2 about Raft.
   b) Availability: The system should be available as long it is possible to elect a leader through a majority vote.
   c) Timing independence: Safety should not depend on the timing of the system, i.e. the speed of a given process should provide it with an advantage in terms of the consensus of the system. For this, it is specified that:

$$broadcastTime << electionTimeout << MTBF$$ [4].

4. Analysis

In this section we will analyze and describe our approach on implementing Raft with relation to the purpose of this project and with relation to achieve a more fault-tolerant and robust implementation.

4.1. Existing Raft Implementations

The application of consensus algorithms such as Raft is many, but the most generic applications are databases and service discovery systems. As Raft mostly fits applications in the low level end of the stack, it will also be best suited to be implemented in a more low level and light language such as C, C++ or Go. Erlang would also be a great fit with its fault-tolerant features.

Raft have already been implemented in several versions and in many languages. Diego Ongaro himself has implemented Raft in C++ (LogCabin[1]), CoreOS[2] uses etcd[3], which is a Raft implementation in Go used for service discovery.

If a Raft implementation was needed for real usage, the solution would have been to extend some of the rich open source implementations already available in solid languages as Go and C. But as mentioned in the introduction, the scope of this project is to learn about Raft with relation to fault-tolerance by implementing it. To be able to cover as much of Raft as possible in the time scope, that language chosen for implementation is JavaScript, because it would allow more productivity.

There already exists solutions in JavaSciript, but since one of the focus of this project is to learn about Raft and the consensus problem, we have chosen to implement a new version with our own approach. Already existsing solutions in JavaScript such as raftscope and skiff have worked as inspirations during implementation.

4.2. Testing

In order to be able to verify that the implementation satisfies the requirements and that to ensure a higher level of robustness, the software will be thoroughly tested. To document requirement satisfaction, the implementation will be black-box tested on the level of acceptance and integration.

In order to achieve robustness the implementation will also be white-box tested on the unit test level in order to test the behaviour of the different objects in situations and cases that are hard to foresee.

4.3. Development Approach

Our development approach on implementing Raft has iterative with relation to the process. The Raft paper includes an informal specification [4, page 4] with behaviour of

[1] https://github.com/logcabin/logcabin

[2] A new Linux operation system fork made for server deployments.

[3] https://github.com/coreos/etcd

the servers described by the state they are in or how to respond to requests. In order to have a more common language between the specification and the tests, the development approach Behaviour Driven Development (BDD)[4] [5]. The Behaviour Driven approach is iterative and have been used in the report with the following steps:

1. Derive behaviour from the specification and write a failing test of the given behaviour.

2. Write the simplest code that implements the behaviour and make the test pass.

3. Refactor the code

4. Repeat

This is close to Test Driven Development, but with focus on writing tests that describes behaviour instead of testing methods as in unit tests. One important aspect to make this approach work well is to derive the behaviour that seem most easy to implement first.

To facilitate writing tests with a BDD-aproach, we used the node lirbaries `chai` together with `mocha`, which allows us to write tests written as behaviour such as the example seen in code examplre below.

Listing 1: Example of a JavaScript test with the libraries mocha and chai

```
describe("Server", function() {
  it("responds with false if log is outdated", function() {
    var outDatedLog = new Log();
    var server = new Server(outDatedLog);
    assert(server.getResponse(), false);
  });
});
```

[4] Originally described by Dan North in the blog post `http://dannorth.net/introducing-bdd/`

5. Leader Election

As mentioned in section 2 one of the two main components of Raft is its leader election. The leader election after which the given leader is responsible of replicating commands it receives from a client to the rest of the followers. All servers contain an implementation of Raft, having a log of commands requested by clients. A server can be in three states: leader, candidate or follower.

Figure 3 illustrates the scenario in which a leader is elected:

In the beginning of the illustration, on initial system startup, every server is a follower until $P1$ times out and becomes a candidate. If this candidate receives a majority of votes it transitions to the leader state after which it has the leader responsibilities described above. The leader claims its leadership by sending out heartbeats, where if a server receives a heartbeat request from another server it is instantly converted to follower and recognizes the source as leader.

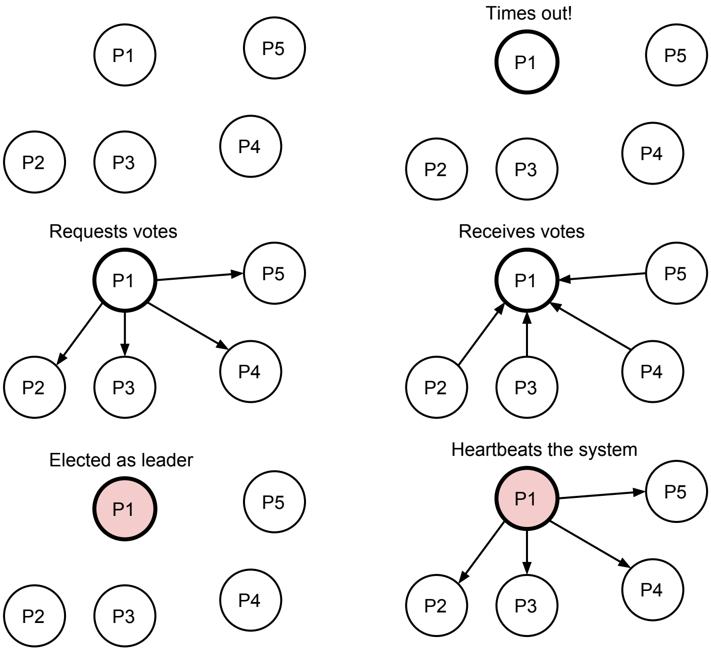


Figure 3: A simple scenario where 5 servers are to obtain consensus by first electing a leader.

Figure 4 shows how Raft will tolerate a faulty leader. Here we see that the leader $P1$ fails by crashing. $P4$ then times outs and initiates a new election.

It should also be noted that since the availability, provided by Raft, relies on the majority votes and only non-Byzantine failures, the algorithm can only uphold these properties in a network where $3F \leq N$, where F is the amount of failures and N is the amount of processes in the system. [1]

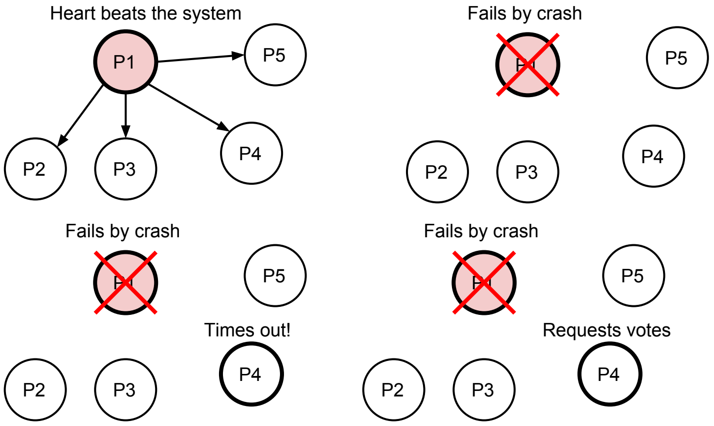


Figure 4: A simple scenario where a leader $P1$ crashes after which $P4$ times out and start a new election.

Looking back at the Two Generals Problem, Raft is only be able to solve the problem if a majority vote would be possible, which it is not. So in order to solve this problem when utilising Raft, one must add another general to the scenario such that there are an uneven number of generals in the system. We should also modify the problem such that the messenger always tells the truth (i.e. cannot suffer from Byzantine failures). Now that we have three generals and an "uncapturable" messenger, they will reach consensus by finding a leader after which he decides when to launch the attack.

6. Log Replication

In this section we will describe how Raft handles log replication, discuss some of the design decisions we have taken within Raft and describe the challenges with the implementation.

6.1. Log Replication in Raft

The general purpose of the log replication feature in a consens algorithm and in Raft is to keep a log history of commands applied to the state machine in order to compare integrity between servers.

Log entries are appended to the log when a client requests a given command to be applied. The flow of receiving the request from the client, commiting the command to the log and responding the client is illustrated in figure 5. In overall the steps of committing a command in Raft can be written as:

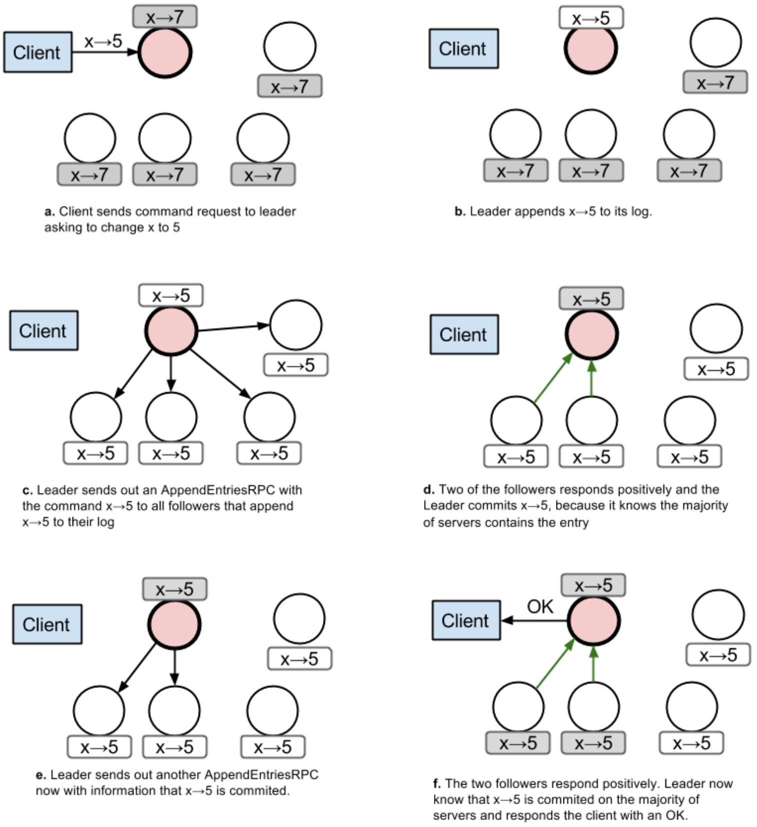
1. Client sends request to the leader with command
2. Leader appends command to own log
3. Leader replicates log to followers in parallel
4. When the command is replicated and commited to a majority of the servers in the cluster, the leader will respond positively to the client and apply the command to the state machine.

6.1.1. AppendEntries RPC/Heartbeat

The log replication mechanism explained in these steps are in Raft the same heartbeat mechanism used for claiming leadership as explained in section 5. The hearbeat/log replication request is a Remote Procedure Call (RPC) called *AppendEntries RPC*. The leader invokes the RPC with information for the follower duplicate the leaders log and be in consensus. The follower then responds the leader with success if it is fully replicated or else failure.

6.1.2. Commiting Log Entry

The first time a server receives an AppendEntries RPC it will just append it to the log. A log entry is committed by the leader when the leader knows that a majority of the servers have appended the given log entry to their log. The log entry is then commited on a follower, when the leader sends out information (through *AppendEntries RPC*) that the index of a given log entry is commited. The log entry is safe to apply to the state machine, when a majority of the servers have commited the entry.



a. Client sends command request to leader asking to change x to 5

b. Leader appends x→5 to its log.

c. Leader sends out an AppendEntriesRPC with the command x→5 to all followers that append x→5 to their log

d. Two of the followers responds positively and the Leader commits x→5, because it knows the majority of servers contains the entry

e. Leader sends out another AppendEntriesRPC now with information that x→5 is commited.

f. The two followers respond positively. Leader now know that x→5 is commited on the majority of servers and responds the client with an OK.

Figure 5: An illustration of how the leader succesfully apply a command from the client. At first *(a)* the client requests the leader to apply the command x→5 to the state machine, then *(b)* the applies the command to its own log and *(c)* replicates it to the followers. *(d)* The followers respond positively if the command is applied to their log and the leader commits the entry from the client, when a majority of the servers have appended the log. *(e)* The followers will receive information that the entry is commited and *(f)* when the leader knows that a majority of the servers have applied the command, it will respond to the client positively and append it to the state machine

6.2. Handling Log Inconsistency

The scenario explained this far is the sucessful one, but in distributed systems failures such as network partitions and single server crashes are not rare and should be handled. Raft have behaviour for handling different kind of log inconstencies, which compared to the true log of the leader can be:

1. A follower is missing entries
2. A follower has extra entries

3. A follower is both missin entries and has extra entries

All situations of inconcistencies are handled by forcing the followers to duplicate the leaders log. The first situation is handled through the AppendEntries RPC by finding the largest common commited index and replicate the missing entries from the leader. The second scenario is handled by finding the largest common commited index and delete all entries following. Thereafter the situation is as the first and handled so. The last scenario is handled as the second.

6.3. Implementation

7. Evaluation and Testing

As mentioned in the analysis section4 we are to evaluate our solution with white- and black box tests. Their description will thus be provided and followed by the results.

7.1. White box tests

The development of our solution was to iterate based on the behaviour of Raft. This included writing a test for each behaviour described in the condensed summary of Raft [4]. For each of these tests its required functionality was then implemented in order to satisfy it.

7.1.1. Tests

Although the functional tests, constructed during implementation, can be directly read in the code, a small example will be presented for completeness's sake.

The following code snippet shows the test for the AppendEntries RPC attribute that a receiver should reply false if the given RPC call contains a term ¡ currentTerm.

Listing 2: The test for the AppendEntries RPC Receiver implementation

```
describe("Receiver AppendEntries Implementation", function() {
    // Rule 1
    it("replies false if term < currentTerm", function() {
      var server1 = new Server(2, 'leader');
      var server2 = new Server(3, 'candidate');
      server2.currentTerm = 2;
      updatePeers([server1, server2]);
      var result = server1.invokeAppendEntries(server2.id);
      assert.equal(result.success, false);
    });
});
```

7.1.2. Results

The results for all of the whitebox tests can be seen in appendix TODO.

7.2. Black box test scenarios

Verifying that our solution actually is an implementation of Raft requires more in-depth tests in the form of validating that we uphold the properties that are argued to provide consensus. This is done by constructing a set of scenarios that reflect the behaviour of the components of Raft: Leader Election, Log Replication, and Safety. The scenarios are as following with success criteria and a simple use case:

7.2.1. Tests

- **Leader Election**:
  1. A leader is elected upon initiate start up of the system.
     a) The user starts the program.
     b) Server S times out.
     c) S converts to candidate.
     d) S receives a majority vote and converts to leader.
  2. A new leader must be elected if the current leader fails.
     a) The user starts the program.
     b) A server L is elected as leader.
     c) The user crashes L at run-time.
     d) Server S times out and starts a new election.
- **Log Replication**:
  1. Log entry is replicated through the system.
     a) The user starts the program.
     b) A server L is elected as leader.
     c) The user requests that a value x is replicated.
     d) L replicates x such that each log of each server contains x in their log at the correct term and index.
  2. Awoken follower get its log updated.
     a) The user starts the program.
     b) A server L is elected as leader.
     c) The user requests that a value x is replicated.
     d) The user crashes a server S.
     e) The user requests that a value y is replicated.
     f) The user wakes up S.
     g) L updates S's log such that it contains y in its log at the correct index and term.
- **Safety**:
  1. There is at most one leader at any given term.
     a) The user starts the program in which two servers S_1 and S_2 have the same election time out.
     b) A leader L, being the only one with an election time out smaller than both S_1 and S_2's, get elected.

    c) The user crashes L.

    d) Both S_1 and S_2 start their elections.

    e) Either S_1 or S_2 is elected as the new leader.

2. Leader never overwrites its own log but only appends on to it.

    a) The user starts the program.

    b) A server L is elected as leader.

    c) The user requests that a value x is replicated.

    d) The user requests that a value y is replicated.

3. Logs of every servers match each other up to their latest common index in their logs.

    a) TODO

4. The log of a leader contains all previous log entries of leader of previous terms.

    a) The user starts the program with a server L with the lowest election time out and S with the second lowest.

    b) L is elected as leader.

    c) The user crashes a server S.

    d) The user requests that a value x is replicated.

    e) The user wakes up S

    f) The user crashes L

    g) S has an up to date log.

5. A server's log must be up to date with the leaders upon updating it.

    a) TODO

And referring back to requirements in section 3. The solution should also satisfy the requirements about availability and timing independence. The success criteria and scenarios for these requirements are as following:

- **Availability**:

    1. As long as a majority vote is possible, a leader must always be elected. If not the system should become unavailable.

        a) The user starts the program with five servers.

        b) A server L is elected as leader.

        c) The user crashes L and another server.

        d) A new leader L' is elected as leader.

        e) The user crashes L'.

        f) The system becomes unavailable.

The last scenarios about safety, availability, and especially timing independence are hard to informally validate since the program simulates a concurrent system, in which the utilisation of a formal verification tool would be necessary, which is out of scope of this project.

7.2.2. Results

The scenarios are implemented and test in the same way as done for the whitebox tests, in which we describe the each scenario with its underlying use case as the real test. Please refer to appendix TODO to see the results of these tests.

8. Discussion

In this section we will look back at the problem introduced in section 1 and evaluate the implementation of Raft. This will be done by comparing our test results with the requirements stated in section 3. Doing this, we can reflect on the overall goal of the project i.e. to gain deeper knowledge of Raft. Further extensions and improvements to the end product will then be proposed and discussed.

8.1. Reflection on goal

The goal of this project was to learn about what is the state of the art in terms of solving the consensus problem. In order to do this, our approach was to gain knowledge about Raft through implementing it in a simulated environment. The requirements for the implementation stated that it should provide a tool for visualising the behaviour of Raft. TODO

8.2. Experiences

Implementing Raft required some inspiration from other current implementations in order to get a complete and working that reflected the specification in the paper [4]. TODO

8.3. Further work

TODO

9. Conclusion

References

[1] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[2] Tim Kindberg George Coulouris, Jean Dollimore and Gordon Blair. *Disitrbuted Systems, Concepts and Design.*

[3] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[4] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[5] C. Solis and Xiaofeng Wang. A study of the characteristics of behaviour driven development. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 383–387, Aug 2011.

A. Results