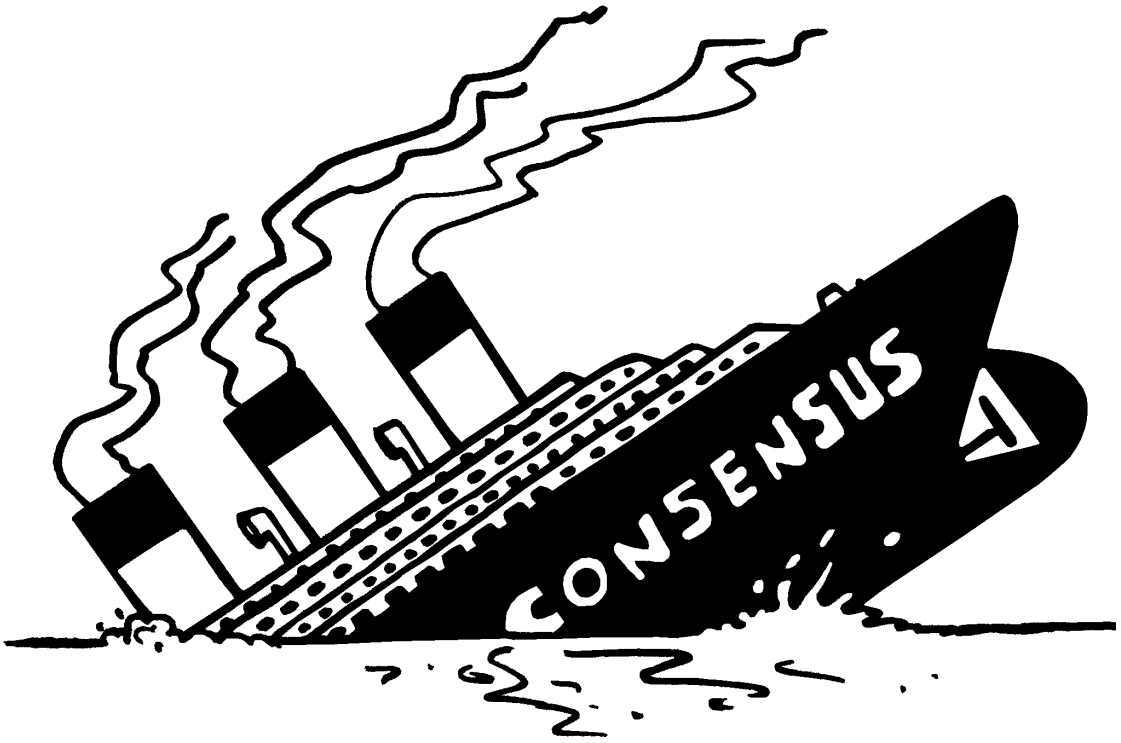
DTU Informatics — 02228 - Fault tolerant systems

FINAL REPORT
RAFT CONSENSUS ALGORITHM



Joachim Kirkegaard Friis s093256
Anders Nielsen s103457

December 4, 2014

Contents

1 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.

2 Introduction

This the final report for a project done 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.

2.1 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.
The fundamental problem behind this, is that you cannot rely on the individual process

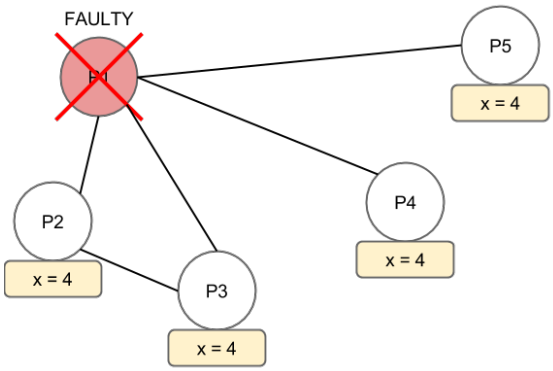


Figure 1: A distributed system consisting of a number of nodes with a faulty one.

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.

2.1.1 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 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 [**?**]

- 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 [**?**] 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.

2.2 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?

2.2.1 The Two Generals Problem

The basic problem of reaching consensus in a distributed network can be illustrated by the Two Generals Problem analogy. In the figure2 below we see two generals of the same

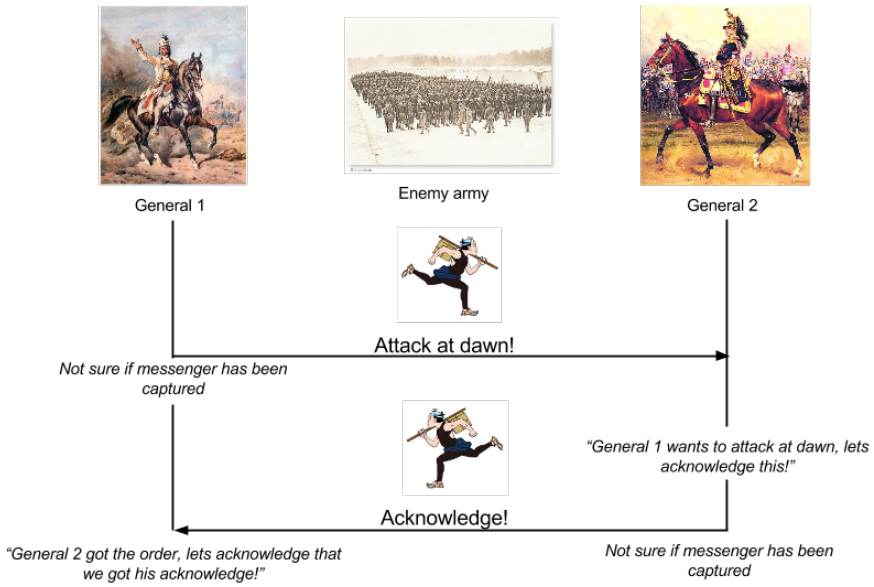


Figure 2: Two generals tries to agree on when to attack the enemy by sending a messenger, but they are not sure the messenger survives his trip between their camps.

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.

3 Raft

As of writing this report, the most recent proposal to solving the consensus problem is Raft [**?**]. Diego Ongaro and John Oustershout argue that most consensus algorithms, such as Paxos [**?**] suffer from poor understandability and are hard to teach and later implement. They then introduce Raft as a simple and understandable solution to the consensus problem.

3.1 Components of Raft

They describe Raft as having three main components, where the first two describes the behaviour of algorithm and the last describes how the algorithm ensures consensus 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 logs to agree with its own. [**?**]. The leader must also update the logs of other servers if they are not up to date.
- **Safety**: The specified design provides safety such that [**?**]:
  - **Election Safety**: There is always is at most one leader in the system.
  - **Leader Append-Only**: A leader only appends new entries to its log i.e. doesn't overwrite existing entries.
  - **Log Matching**: All logs of all servers are matching up to their latest common index.
  - **Leader Completeness**: No newly elected leaders will overwrite other servers logs 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 leaders log.

3.2 Behaviour of Raft

So the basis of this solution is the leader election after which the given leader is responsible to replicate commands it receives from a client to the rest of the network. All servers contain an implementation of Raft, having a log of entries pushed by a client.A server can be in three states: leader, candidate or follower. Figure 3 illustrated the scenario in which a leader is elected. At initial system start up every server is a follower until $P1$ times out and becomes a candidate. If this candidate receives a majority vote, it then transitions to the leader state after which it has the responsibilities described above. This leader then maintains log matching by continuously sending empty messages i.e. heart beats in order to update servers with logs that are not up to date.

Figure 4 then 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.

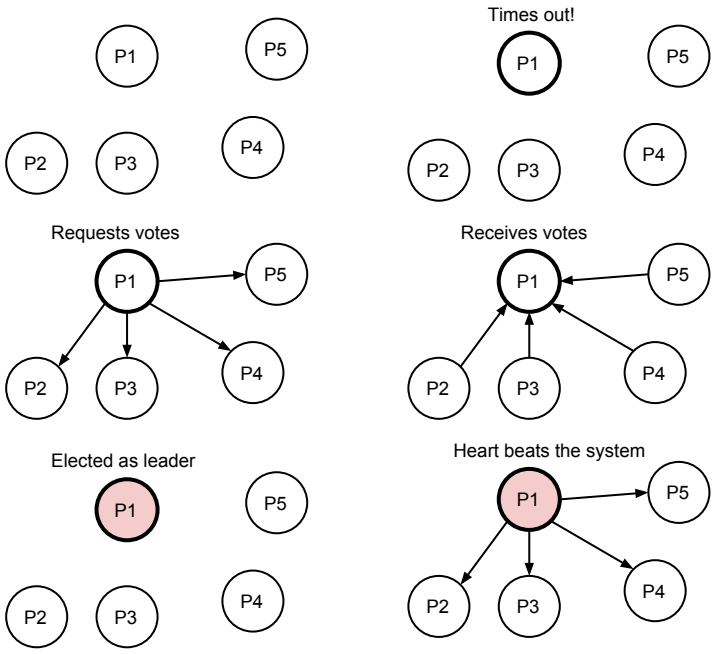


Figure 3: A simple scenario where 5 servers are to obtain consensus by first electing a leader.

It should also be noted that, since the availability provided by Raft relies on majority votes and non-Byzantine failures only i.e. failure is a server stopping and not giving sending incorrect messages, 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.

Looking back at the Two Generals Problem, Raft would 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 - such that there is now three generals instead of two. We should also modify the problem, such that the messenger always tells the truth (i.e. cannot suffer from Byzantine failures). So 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.

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. So as we every component is describes our experiences and findings in terms of implementation specific choices will thus be documented.

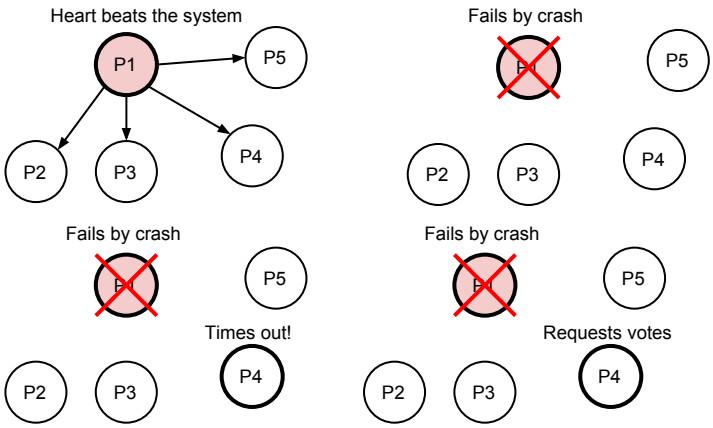


Figure 4: A simple scenario where a leader $P1$ crashes after which $P4$ times out and start a new election.

4 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:

4.1 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 [**?**].
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.

4.2 Quality requirements

The quality requirement 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: There must always be at most one leader in the distributed system.
    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:

$$broadcastTime << electionTimeout << MTBF \text{ [?]}.$$

5 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.

5.1 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.

5.2 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.

5.3 Development Approach

Our development approach on implementing Raft has iterative with relation to the process. The Raft paper includes an informal specification [**?**, 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] [**?**]. 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/`

6 Leader Election

7 Log Replication

8 Evaluation and Testing

As mentioned in the analysis section5 we are to evaluate our solution with white- and black box tests. Their description will thus be provided followed by the results.

8.1 White box tests

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

8.2 Black box test scenarios

Verifying that our solution actually is an implementation of Raft requires more cohesive 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 reflects the behaviour of the components of Raft: Leader Election, Log Replication, and Safety. The scenarios have the following success criteria:

- **Leader Election**:
  1. A leader is 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 start 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 every log 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 also.

- **Safety**:
  1. There is at most one leader at any given term.
  2. Leader never overwrites its own log but only appends on to it.
  3. Logs of every servers match each other up to their latest common index in their logs.
  4. The log of a leader contains all previous log entries of leader of previous terms.
  5. A server's log must be up to date with the leaders upon uptading it.

)

The last scenarios about safety 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.

9 Discussion

10 Conclusion