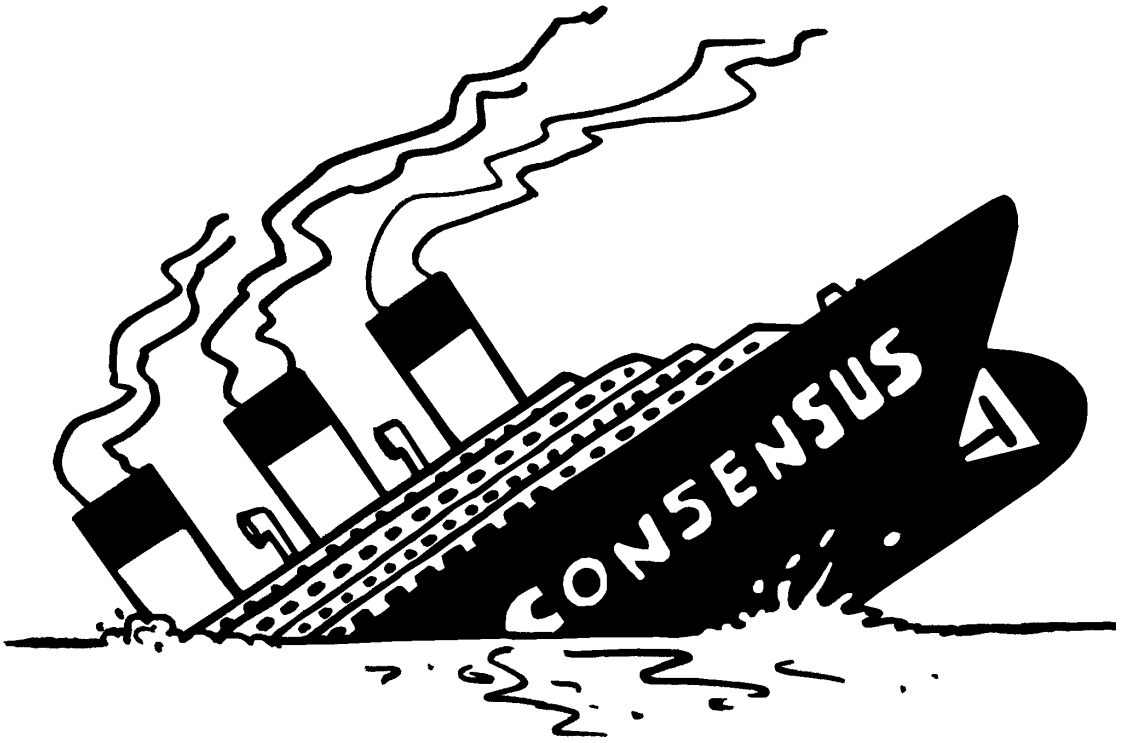
DTU Informatics — 02228 - Fault tolerant systems

FINAL REPORT
RAFT CONSENSUS ALGORITHM



Joachim Kirkegaard Friis s093256
Anders Nielsen s103457

November 22, 2014

Contents

1 Preface

In this section we will describe and motivate our project idea. We will start out by presenting the Two Generals problem in which two processes try to reach consensus about when to attack the enemy. Then we will relate this problem to modern distributed systems, in which consensus must be achievable in presence of faulty processes.

1.1 Two Generals Problem

1.2 Complexity and understandability of current solutions

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 an implementation of a consensus algorithm i.e. Raft.
Starting, the fundamental problem when talking about consensus in a distributed system, will be presented. This will then present the motivation behind the project itself. 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 processes in the network agree on some value of state of the entire system. This is often needed as processes might to be faulty and thus reliability and availability is 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. And having system that can tolerate faulty processes by maintaining a common knowledge of a value of state is preferred. 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 know 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

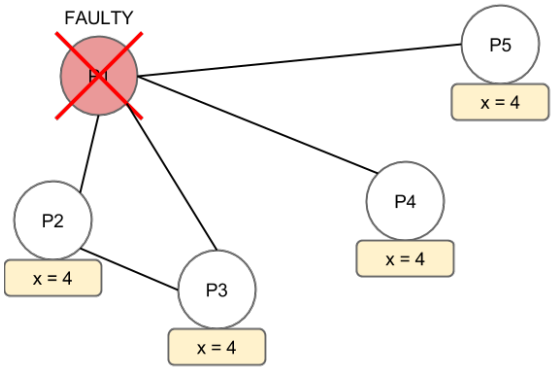


Figure 1: A distributed system consisting of a number of nodes with a faulty one.

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 talking about consensus, the goal is to satisfy a set of requirements i.e. properties that the distributed system must uphold. These properties 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 system, if it tolerates Byzantine failures, is the validity and integrity. The integrity property for a system that does not tolerate it 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, whether on which property set that can achieve to 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 by this problem serves 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

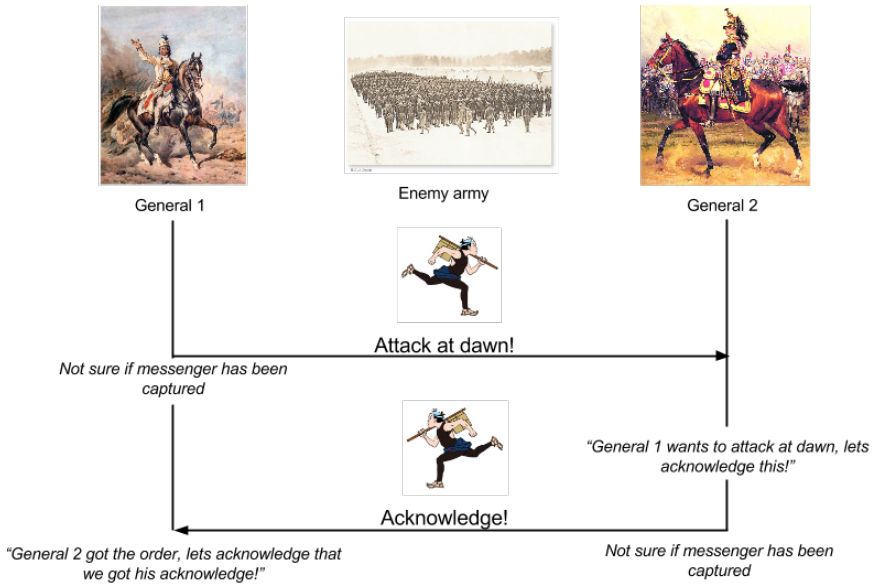


Figure 2: Two generals tries to agree on when to attack the enemy by sending a messenger, but they are not sure the messenger survives his trip between their camps.

army who want to attack an enemy army. But they have to attack at the same time in order to win. They cant communicate directly to each other since they are at different fronts of the battlefield i.e. in the 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. 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.

2.3 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 thus are hard to teach and later implement. Raft is is constructed by solving these three fundamental problems:

- **Leader election**: a new leader must be chosen when an existing leader 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. [**?**]
- **Safety**: is ensured by the servers state machine safety property [**?**]. This state machine makes sure that no other server can apply a different command at a given log index, that has already been issued.

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

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.

a. The tool must illustrate a scenario of a given set of processes in a distributed system, in which a leader is elected.

b. The user must be able to add new processes to the system.

c. The user should be able to input parameters to vary the scenario, such as the number of processes.

d. The tool must be an implementation of the Raft algorithm.

   a) The Raft implementation should have the following properties: Safety, availability, timing independence, and that commands complete as soon as a majority has responded.

e. The tool could be further extended with a visualisation of the given distributed system.

TBD:

a. The user must then be able to disconnect the leader.

   a) If a leader is disconnected a new leader must be elected automatically.

b. The system should be able to have the parameters changed at run-time.

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 acheive 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.

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 throughly tested. To document requirement satisfaction, the implementation will be black-box tested on the level of acceptance and integration.
In order to acheive 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.

[1] https://github.com/logcabin/logcabin
[2] A new Linux operation system fork made for server deployments.
[3] https://github.com/coreos/etcd

5 Leader Election

6 Conclusion