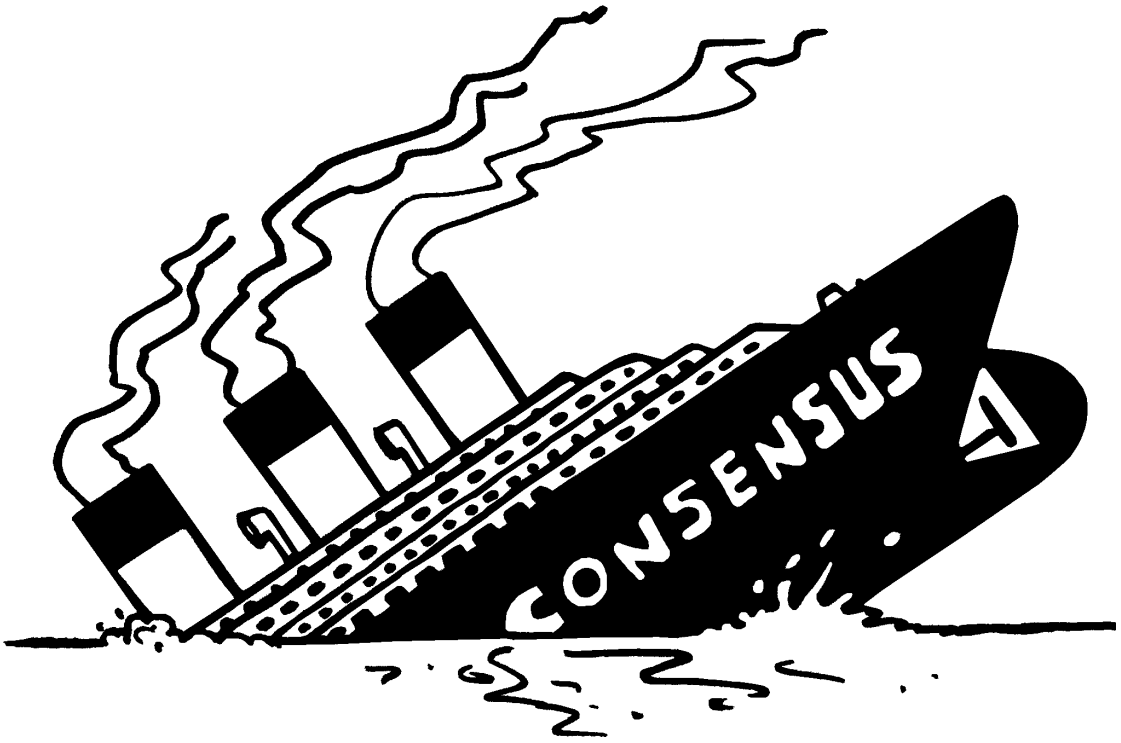
DTU Informatics — 02228 - Fault tolerant systems

FINAL REPORT
RAFT CONSENSUS ALGORITHM



Joachim Kirkegaard Friis s093256
Anders Nielsen s103457

November 15, 2014

Contents

1 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.

1.1 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

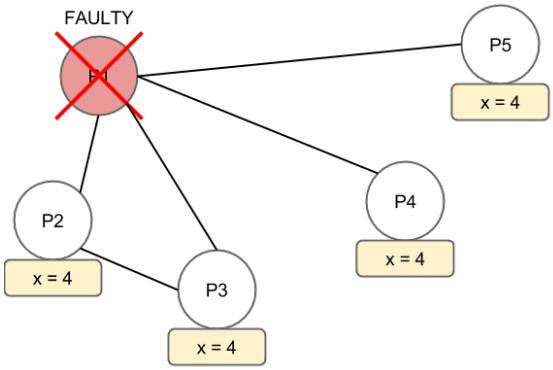


Figure 1: A distributed system consisting of a number of nodes with a faulty one.

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.

1.1.1 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[1]:

- 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.

1.2 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.

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. 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 cant communicate directly to each other since they are at different

[1] http://en.wikipedia.org/wiki/Consensus_(computer_science)

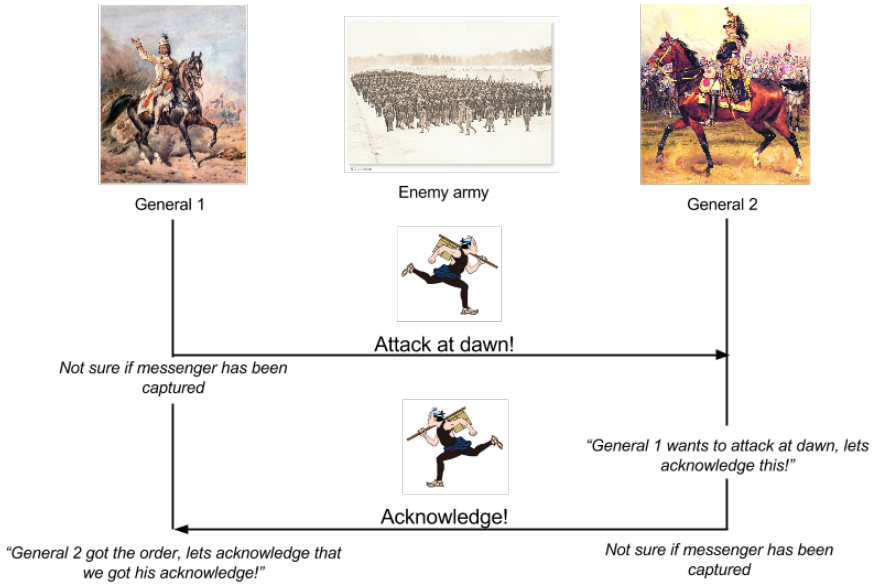


Figure 2: Two generals tries to agree on when to attack the enemy by sending a messenger, but they are not sure the messenger survives his trip between their camps.

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.

1.3 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 composed with three main components:

- **Strong leader**: *Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.*
- **Leader election**: *Raft uses randomized timers to elect leaders. This adds only a*

small amount of mechanism to the heartbeats already required for any consensus algorithm, while resolving conflicts simply and rapidly.

- **Membership changes**: *Rafts mechanism for changing the set of servers in the cluster uses a new joint consensus approach where the majorities of two different configurations overlap during transitions. This allows the cluster to continue operating normally during configuration changes.* [**?**]

Below is a description of the terms they use in this case:

- Log entries: A list of values that should be the same among all processes in the network.
- Log replication: A client pushes a new value to the network to the leader, which then replicates this to the rest of the network.
- Server: A process in the network.
- Heart beat: An empty message repeatedly send to all the followers of the leader, in order to maintain a knowledge of failed process.

2 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 then be able to disconnect the leader.
    a) If a leader is disconnected a new leader must be elected automatically.

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

d. The user should be able to input parameters to vary the scenario, such as the number of processes, and the heart rate of the system.
    a) The system should be able to have the parameters changed at run-time.

e. 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.

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

3 Analysis

In this section we will come back to the Two Generals problem and further analyse and realize that reaching consensus without a leader/authority, who can issue commands, is impossible. Then we will present Raft as a solution to this problem and compare it to other solutions and discuss why we have chosen it.

3.1 Impossibility of the Two Generals Problem

3.2 Raft as a solution

4 Design

In this section we will relate to our requirements of the tool, and from this design the implementation.

5 Implementation

6 Tests

6.1 Deterministic vs. in deterministic tests

a. Simulation and validation of different scenarios described in the requirements.
   a) Leader election:
      i. The system must choose a new leader if the current leader is disconnected.
      ii. The system must be able to choose a leader if the system is partitioned.
   b) Timing and availability
      i. The system must not depend on timing of new elections, thus not producing incorrect results because of unknown events.

7 Conclusion