SLR210 Project - Obstruction-Free Consensus and Paxos

Yassine Benbihi - Tristan Perrot - Benjamin Ternot

Table of Contents

I. The Initial Problem

A. Specification

This project aims to get initial experience in designing a fault-tolerant distributed system. Here we focus on a state-machine replicated system built atop a consensus abstraction with a message-passing system.

We need to implement an obstruction-free consensus (OFC) algorithm to achieve that. The obstruction-free consensus must meet these properties:

- Validity: every decided value is a proposed value.
- Agreement: no two processes decide differently.
- Obstruction-free termination:
  - If a correct process proposes, it eventually decides or aborts.
  - If a correct process decides, no correct process aborts infinitely often.
  - If there is a time after which exactly one correct process p proposes a value sufficiently many times, p eventually decides.

B. Environment

We use this environment to implement OFC:

- N asynchronous processes (N actors if we use the AKKA names). Every process has a distinct identifier, and the identifiers are publicly known.
- Every two processes can communicate via a reliable asynchronous point-to-point channel.
- Up to $f < N/2$ of the processes are subject to crash failures: a faulty process prematurely stops taking steps of its algorithm. A process that never crashes is called correct.

II. Our implementation

Our implementation is based on the algorithm provided, using Akka actors, with a main class instancing all our processes.

We defined a class for each type of message: *AbortMsg, AckMsg, CrashMsg, DecideMsg, GatherMsg, HoldMsg, ImposeMsg, LaunchMsg and ReadMsg*. Each type of message is sent in a specific order through the decision process.

- When a process wants to propose a value, it sends a *ReadMsg* to all other processes, with a ballot number. A process *i* only send multiples of *i* as a ballot number. Therefore, no two processes can have the same ballot number.
- When a process receives a *ReadMsg*, it sends a *GatherMsg* to the sending process, containing the ballot number of the "ReadMsg", the largest ballot number it has seen so far, and the estimate it has for the proposed value.
- When a process receives a *GatherMsg*, it stores the estimate and the ballot number it received, and if it has received *GatherMsg* from more than *N/2* processes, it computes the estimate with the highest ballot number and sends an "ImposeMsg" to all processes with the new estimate and ballot number.
- When a process receives an *ImposeMsg*, it checks if the ballot number is larger than any previous ballot number it has seen, and if it is, it updates its estimate and imposes the new estimate and ballot number by sending an *AckMsg* to the sending process.
- When a process receives an *AckMsg* from a majority of processes, it decides on the proposed value and sends a *DecideMsg* to all processes with the decided value.
- When a process receives a *DecideMsg*, it stops listening for further messages.
- When a process receives a *CrashMsg*, it means that it is subject to failure and for every other event, it has a fixed probability to crash and stop reacting to anything;
- When a process receives a *HoldMsg*, it means that the *main* (that is the client) waited its limited time t_{le} and decided to put a term to the debate by selecting a leader and stopping the others.

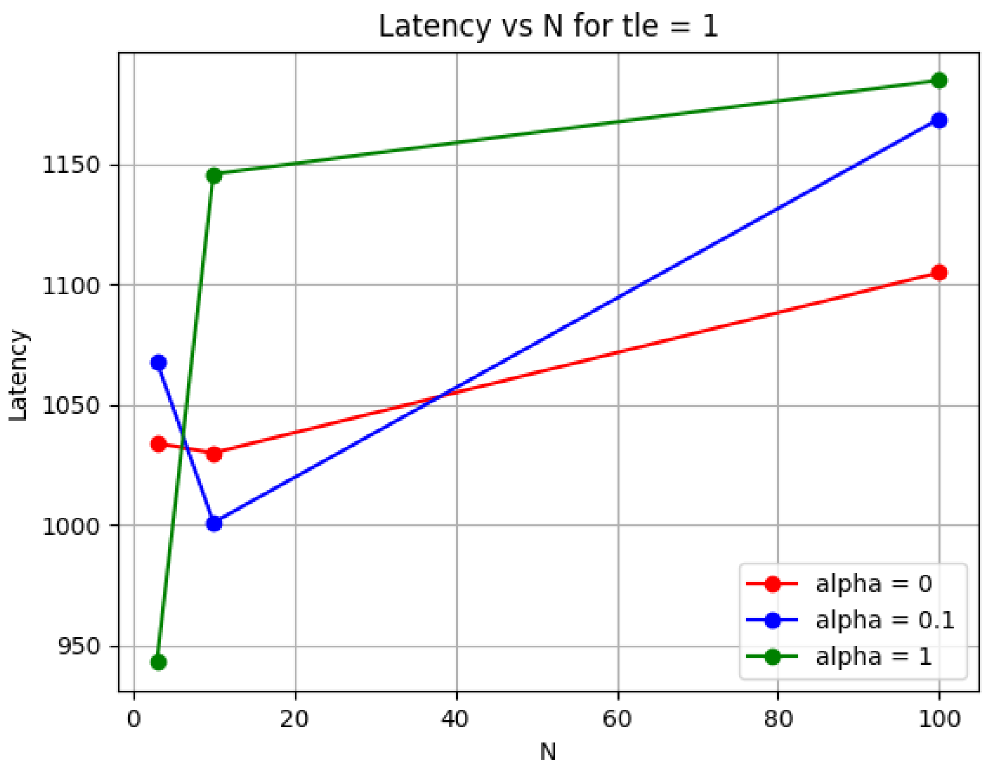
Each process proposes a value (either 0 or 1) by sending a *ReadMsg* to all other processes. Here let's talk about process P to make it easier. P then waits until the other processes reply, and stores the replies. When the majority of processes replied, P chooses a value to impose on other processes and sends this value along with a ballot number through an *ImposeMsg* to all of the other processes (for instance Q). Q can either send an *AckMsg* to P if it uses the imposed value (if the ballot number is greater than the previous ballot number received) or it can send an *AbortMsg*, meaning that a concurrent *propose* operation with a higher ballot number is being executed. When P received sufficiently many *AckMsg*, it means that a majority of processes accepted the imposed value. Therefore P notifies all processes that a value has been decided with a *DecideMsg* containing the decided value.

III. <u>Performance analysis</u>

We have run our implementation with different values:

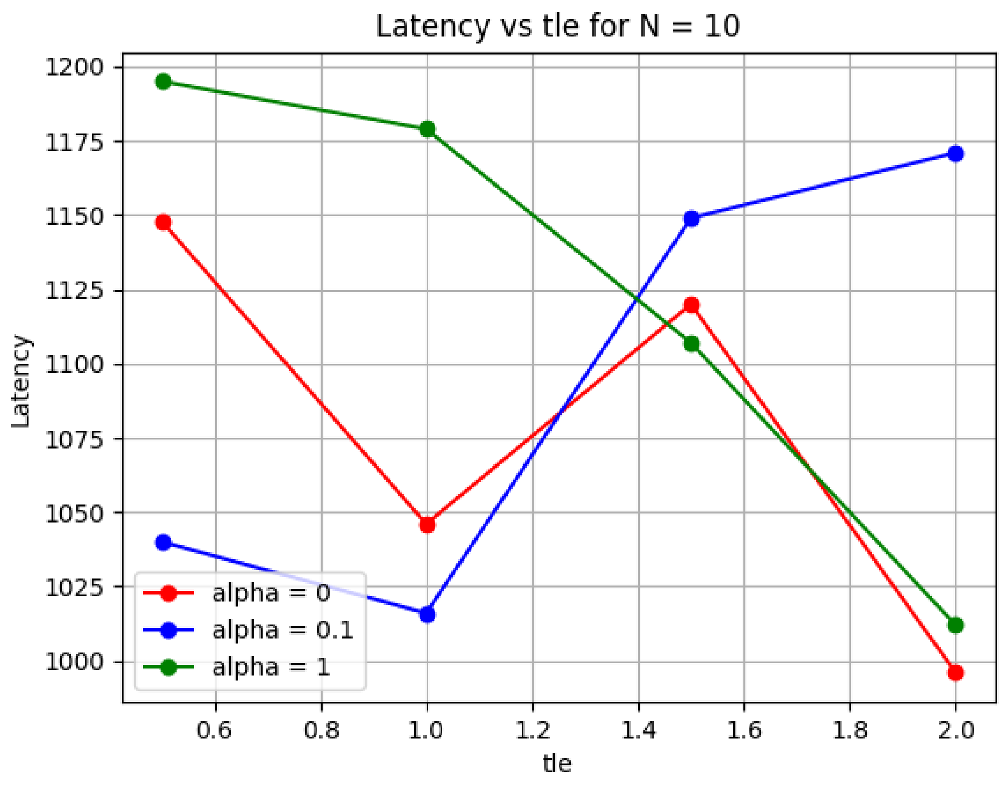
- N = 3, 10, 100 (with f = 1, 4, 49, respectively)
- t_{le} = 0.5s, 1s, 1.5s, 2s
- α = 0, 0.1, 1

To get the values, we have run 5 times every different execution and after we computed the mean to have values quite representative. Then, we plotted the latency as a function of N or as a function of t_{le} to see how the latency evolves depending on these parameters.



In this graph, we can clearly see that the more there are processes, the more the latency will be higher. Indeed, it can be explained by the fact that there are much more messages transiting and in the queue for each process that when the leader is elected, no process can act on it, and the consensus is delayed.

Also, it’s good to see that the curves of alpha = 0 and alpha = 0.1 are quite the same. It’s certainly due to the fact that the probability of 0.1 is quite low and processes don’t crash every time like when alpha = 1.



This graph is particular. Indeed, we clearly see that for alpha = 1, the more t_{le} is higher, the less will be the latency. It can possibly be due to the fact that when we have a high t_{le}, the decision process is over before even have to send *HoldMsg*, whereas with a t_{le} smaller, the sending of *HoldMsg* interferes with decisions processes that might have been in progress, and then resulting to forcing newer decision process to begin from the plain.

Again, we see that curves alpha = 0 and alpha = 0.1 are quite the same for the same reason as before.