SLR210
Project: Obstruction-Free Consensus and Paxos

Matteo Delfour
Yann Girey

I) Problem Statement and Objectives

Our implementation solves the problem of Obstruction-Free Consensus (OFC) in a fault-tolerant distributed system.

Consensus is the process of ensuring that a group of N asynchronous processes agree on the same value, even if they initially propose different ones.

In our case, we deal with obstruction-free consensus, which guarantees that if at least one process keeps making progress without interference (i.e., it is the only one continuously proposing a value), it will eventually reach a decision. Additionally, our system is fault-tolerant, meaning that some processes can crash, stop executing, and no longer respond to others.

More specifically, our implementation ensures that N processes reach an agreement on a binary value (0 or 1), where at most N/2 processes can crash and stop taking steps, while still respecting the obstruction-free property.

To analyze the performance, we will run the implementation with N processes, where f of them will crash with a probability α. After a certain period tle, only one correct process will keep proposing, which should eventually lead to an agreement thanks to the obstruction-free property. Finally, we will measure the consensus latency which is the time required to the first process to decide, according to these four parameters.

II) Implementation.

For the implementation, we used the AKKA framework in Java, which provides an actor-based model to handle concurrent and distributed systems efficiently. This model allows processes to communicate asynchronously through message passing, making it well-suited for implementing the Paxos algorithm and solving the Obstruction-Free Consensus (OFC) problem.

We have two main classes: the Main class, which initializes the processes and starts the algorithm, and the Process class, which handles sending, receiving, and processing messages from other processes.

Process Class:

The Process class has a main function called OnReceive, which allows it to behave differently depending on the messages it receives, using other auxiliary functions to handle specific actions.

The first message a process will receive is Members, which provides it with references to all other processes. After that, the process will receive the OfConsProposer(v) message, which instructs the process to propose the value v until it reaches a decision.

- When the process proposes, it first checks the isHolding variable. This ensures that when a process receives a *Holding* message, it stops proposing but continues responding. Next, it sets nbAck to 0 and resets the states array to its default values (which will be explained later). The process then sets proposal to v, the value it now wants to decide, and increments ballot by N to create a unique identifier for the proposal, thereby ignoring any messages from previous proposals by this process. Finally, it sends a *Read* message to all processes (including itself) to request their responses.

- After receiving a *Read* message, processes check if there is a concurrent *Read* with a higher ballot (indicating a higher priority) or a concurrent *Impose* operation with a higher ballot . If either condition is met, the process sends an *Abort* message to cancel the current proposal. Otherwise, it sets readBallot to the current ballot to avoid interference from older concurrent reads and sends a *Gather* message to the sender, including the parameters imposeBallot and estimate, which represent the value associated to the proposal with ballot imposeBallot.

- When it receives an *Abort* message, the process tries to propose the same value again.

- When the process receives a *Gather* message, it stores the values of the *Gather* message in states. If it has received a majority of responses, it sends an *Impose* message. However, if at least one ballot has already been imposed, it first updates its own value to the one that has been imposed with the higher ballot number.

- When the process receives an *Impose* message, it verifies, as it does after receiving a *Read* message, if there is a concurrent *Read* or *Impose* message with a higher ballot. If so, it sends an *Abort* message. Otherwise, it updates estimates and imposeBallot to the received value, meaning it accepts this as a potential decided value, and then sends an *Acknowledgment* message to the sender.

- When the process receives a majority of *Acknowledgment* messages, it sends a Decide message to all other processes. To determine if a majority has been reached, the process increments the nbAck counter each time it receives an *Acknowledgment* message until the required threshold is met.

- Finally, when the process receives a *Decide* message, it sends a *Decide* message with this value to all other processes, in case the original process crashes, ensuring that the value is decided across the system. It then decides on this value by updating decideValue and logs it, including the time elapsed from the first call of this process to this moment. After that, it stops responding to avoid a message storm.

For the experiment, the process may also receive a *Holding* message, which instructs it to stop proposing but still respond to other processes. Additionally, it may receive a *Crash* message, causing the process to enter crash mode. In this state, each time the process tries to send a message, it has a probability of alpha to enter silent mode and stop responding entirely.

Main Class:

The main function first creates the AKKA system and the processes, then randomly selects f processes to send them an Init and a *Crash* message. It then initializes the remaining

processes to propose values until they reach a decision. After a specified timeout (in milliseconds), the main function randomly selects one process to be the leader and sends a *Holding* message to all other processes, causing the leader to be the only one still proposing. This ensures that a decision is eventually made, as only the leader will continue proposing.

III) Analysis

We ran our implementation with those values:

- N = 3, 10, 100 (with f = 1, 4, 49 respectively)
- t_{le} = 500ms, 1000ms, 1500ms, 2000ms
- α = 0, 0.1, 1

Each simulation where run 5 times, then we compute the mean of those.

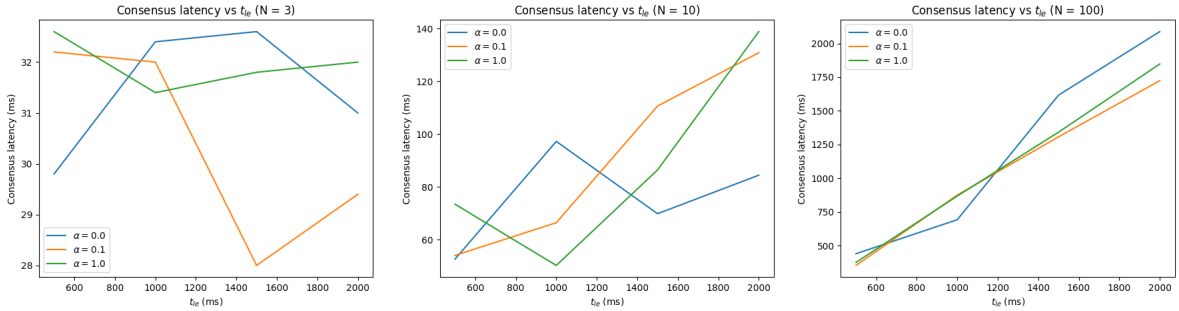

Figure 1: Consensus latency vs t_{le}

In *Figure 1*, one can see a lineal correlation between the consensus latency and the time t_{le} for large value of N.

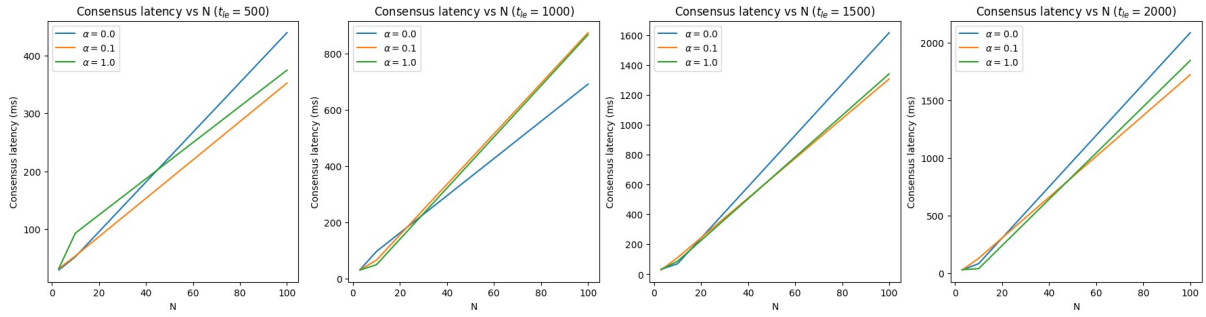

Figure 2: Consensus latency vs N

In *Figure 2*, again, a net lineal correlation can be observed between the consensus latency and the number of process N, even for same value of t_{le}.

Finaly, the probabily α for a process the crash doesn't seem to have any effect on the consensus latency, as the values obtained by the simulations are relatively close.