Télécom ParisTech

SLR 210 – Project

Obstruction-Free Consensus and Paxos

Prepared By:

Antonio Nassar

Cynthia Obeid.

Professor: Petr Kuznetsov

March 20th, 2023

Table of contents:

Introduction:

When a client sends a request to a server, it is important to ensure the correctness of the implementation. In other words, the property of liveness should be respected: every persistent client receives a response, as well as the safety property: responses constitute a total order with respect to the service's sequential specification. A system that verifies these properties is called a consistent and reliable system. The system should also consider the fact that some services might fail. A system that can still operate despite a partial failure or loss of communication is called fault-tolerant.

Strongly consistent replicated state machine is a common design of a distributed system to achieve fault-tolerance with servers running replicas of a service, while ensuring consistency and reliability as defined.

Consensus protocols are the basis for the state machine replication approach to distributed computing, as suggested by Leslie Lamport. Consensus refers to an agreement on any subject by a group of participants. In distributed algorithms, a process proposes a value v in a set of values and all processes should decide on a value in this set. Consensus should satisfy agreement (no two processes decide differently), validity (every decided value was previously proposed) and termination (every correct process eventually decides).

Paxos is a state machine replication protocol that is fault-tolerant and can reach consensus. It underlies today a large number of practical systems when strong consistency is needed (Google Megastore, Microsoft Azure…). Paxos is an algorithm that enables a distributed set of computers (for example, a cluster of distributed database nodes) to achieve consensus over an asynchronous network. To achieve agreement, a client starts by initiating a request, the servers then run multiple instances of consensus to reach a current leader. The client will then send the request to the leader and the leader will reply.

The goal of the project is to obtain an overview of Paxos technology by implementing the convenient algorithm and evaluate the performance of this system.

Problem statement:

To build a strongly consistent replicated state machine, the implementation should be able to solve consensus. The problem with consensus is that it cannot be solved in an asynchronous read-write shared memory system with at least one faulty process. To circumvent this impossibility, the synod algorithm is adopted, which is made of 2 components: an eventual

leader oracle and obstruction-free consensus. OF-consensus is similar to consensus except for the termination property which is relaxed: if a correct process p proposes, it eventually either decides or aborts ; if a correct process decides, no correct process aborts infinitely often ; if there is a time after which a single correct process p proposes a value sufficiently many times, p eventually decides. Synod is useful for building fault-tolerant distributed systems. It tolerates up to f < N/2 failures, where N is the number of processes in the system.

In this project, the goal was to build the OFC algorithm in an environment where we have N asynchronous processes (each one having a unique public identifier), every two processes can communicate via a reliable asynchronous point-to-point channel and up to f processes can fail with f < N/2. The failure considered in the scope of this project is a crash failure where a process stops taking steps.

Implementation:

The OFC implementation is done using the AKKA actor-based programming model to allow the communication between different processes also known as actors.

As indicated in the project description, we created a Process class that acts as a process in our system. Each instance of the Process class is an AKKA actor.

This class defines the following methods:

- Propose: takes an Integer as a parameter which represents the proposed value. This method adds N to the ballot attribute and sets the state's attributes to its initial value. It sends to all the processes a ReadMsg with its ballot value.

- readReceived: This method is called when a process receives a ReadMsg from a process pj. If the readballot attribute or the imposeballot attribute are greater than the ballot value received (newBallot), an AbortMsg is sent to pj. Else, the readballot attribute is updated to newBallot and a GatherMsg is sent to pj with newBallot, the imposeballot attribute and the estimate attribute.

- abortReceived: This method is invoked after receiving an AbortMsg. If a value has not yet been decided and the process is not on hold, the process proposes again.

- gatherReceived: This method is invoked after receiving a GatherMsg from a process pj. It increments the gather_count attribute and updates the jth value of the states attribute. If the gather_count is strictly greater than N/2 (more than the majority of processes) a potentially

decided value is chosen (the est value of the highest estballot in the states attribute). In this case, an ImposeMsg is sent for all processes with the ballot attribute and the proposed value.

- imposeReceived: This method is invoked after receiving an ImposeMsg from a process pj. If the readballot attribute or the imposeballot attribute are greater than the ballot value received (newBallot), an AbortMsg is sent to pj. Else, the estimate attribute is updated to the proposed value of the ImposeMsg and the imposeBallot attribute is updated to the ballot value. An acknowledgement message is sent to pj.

- ackReceived: This method is invoked after receiving an AckMsg. The ack_count is incremented and if ack_count is greater than N/2, a DecideMsg is sent to all processes.

- decideReceived: This method is invoked after receiving a DecideMsg. It outputs that a message has been decided. It sets the value_decided attribute to true.

The main function in the Main class starts by shuffling the collection of processes and sending a crash message to f of them. The number of f processes is chosen so that $f < N/2$. The function then sends launch messages to the remaining processes containing a value (0 or 1) to be proposed and a message containing the timing. This is used to calculate the latency time when a process decides. The main then sleeps for a time specified by tle: this is the initiation of the leader election mechanism. Since processes propose until a value is decided, consensus can be hard to reach. By electing a leader, the main function chooses a process that is not a faulty one and assigns him as leader. It then sends to all remaining processes a hold message so they can stop proposing. At the end, only the leader will propose and eventually the processes will decide on a value.

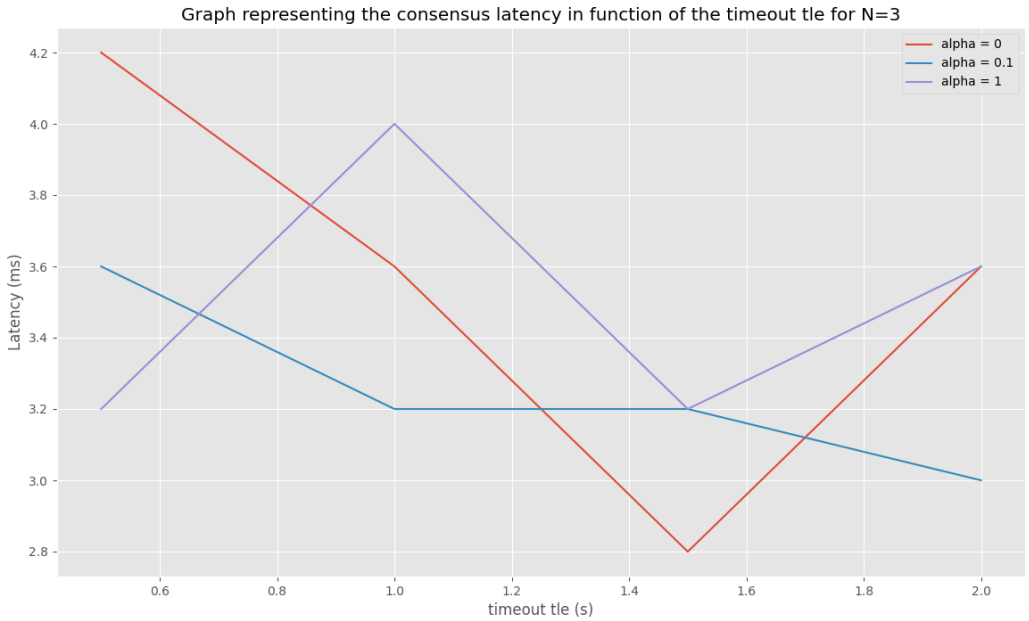
Performance Analysis:

To analyze the performance, we evaluated the consensus latency (how long it takes for the first process to decide) in function of the number of processes N for a fixed tle first (the timeout to elect a leader) and then in function of the tle for a fixed number of processes N.

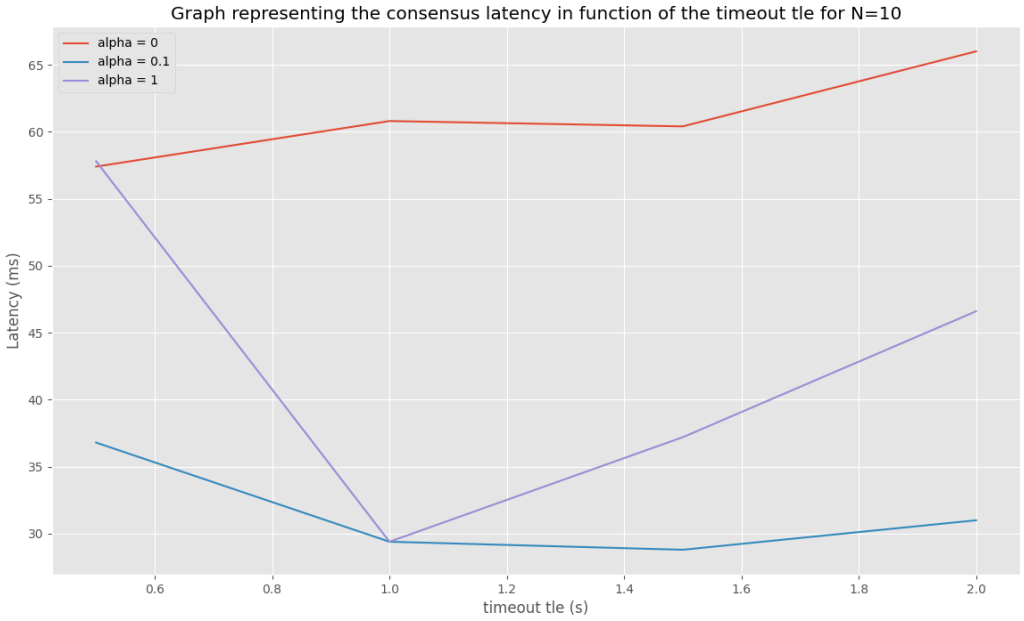
In each figure, we plotted 3 graphs to represent how different probability α of crashing for fault-prone actors affects the latency.

Quick Reminder: All values treated and analyzed each represent the mean value of 5 consecutive tests (runs) as indicated in the project description.

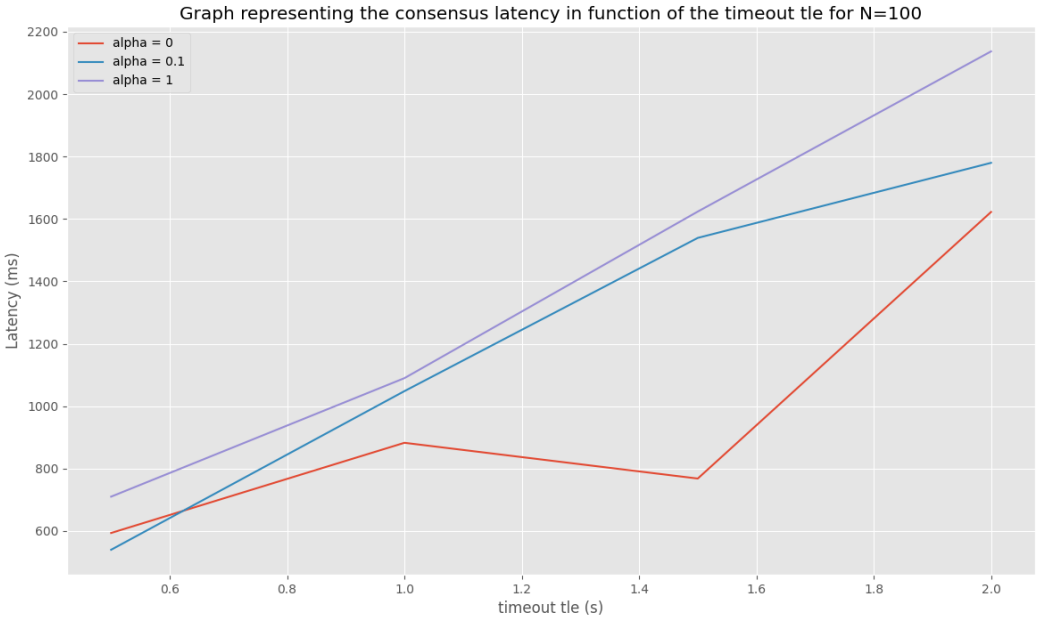
For a fixed N:



For a small number of processes (N=3), we observe a random behavior. In fact, there isn't a clear correlation between the latency and the timeout tle. This can be explained by the fact that for a small number of processes, the decision can be made quickly, even before electing a leader.

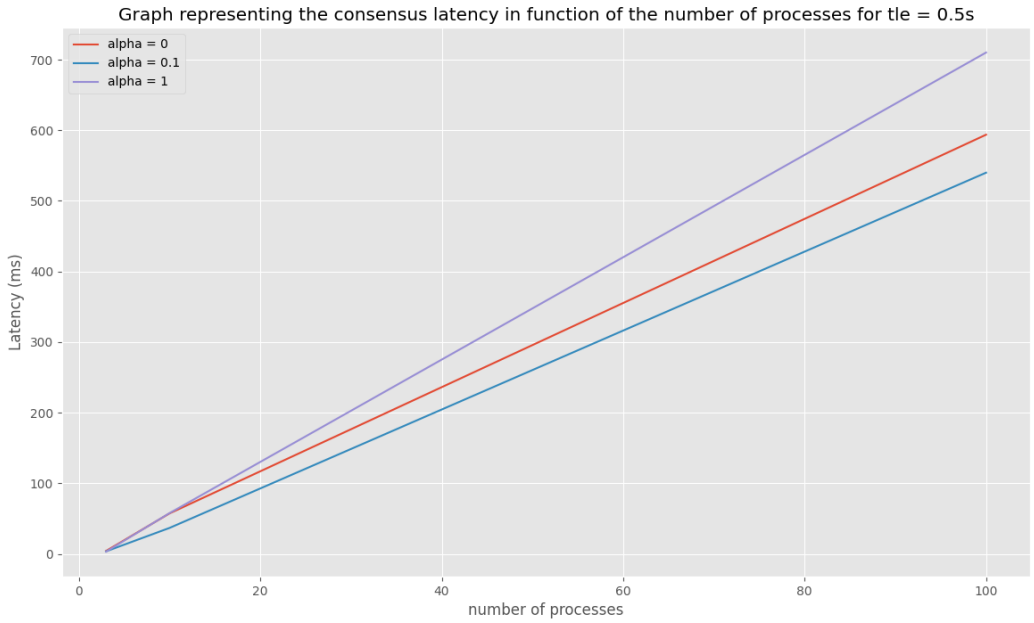


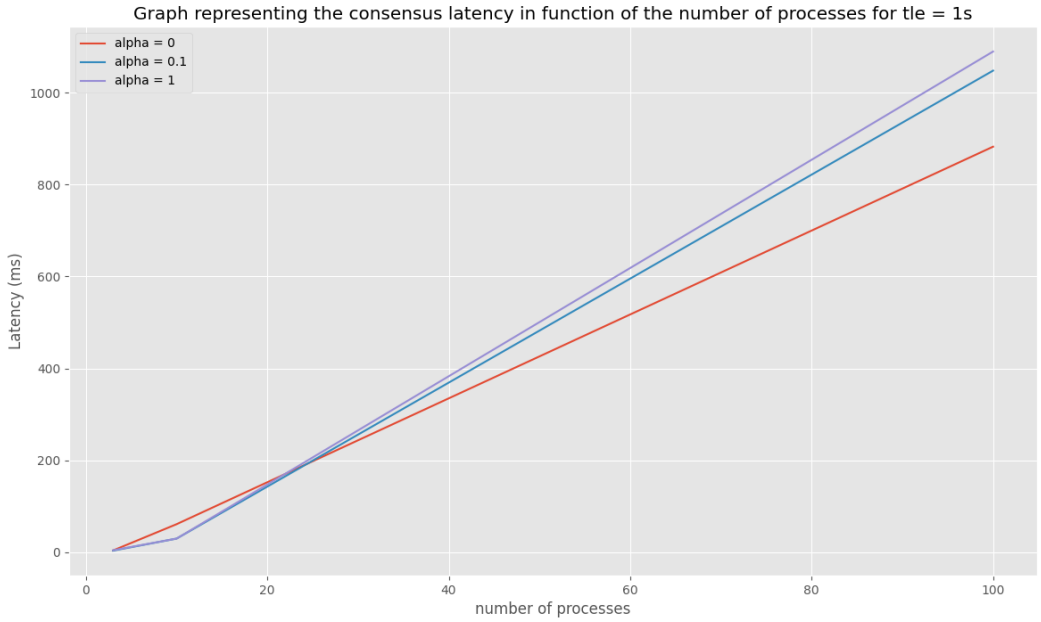
For 10 processes, we observe that for all values of alpha, the first decision is made before the election of a leader (the latency is lower than the timeout tle). So, the timeout tle is also not affecting the latency in that case.

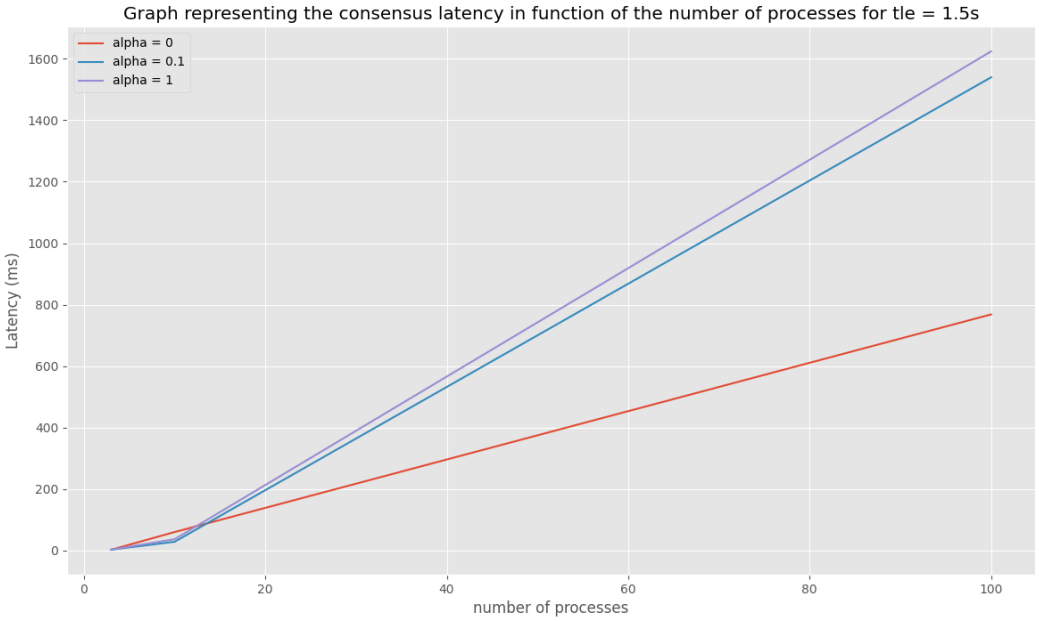


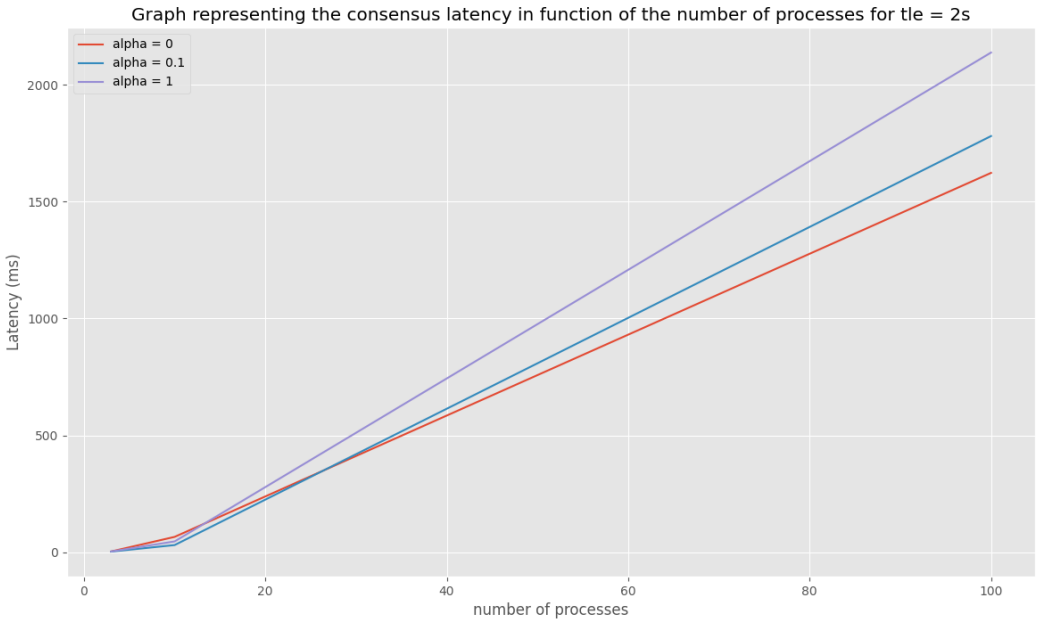
For a large number of processes (N=100), we observe that the latency increases when the timeout tle increases. This is coherent, because for a larger number of processes, the decision could take time to be made (wait till all the processes receive messages and the majority agree on it). Therefore, when we elect a leader, the processes stop proposing and the decision can be made faster.

For a fixed tle:









For a fixed Leader Election Time ‘tle’, it can be seen that increasing the Number of Processes ‘N’ also increases the consensus latency, this is straightforward as more acknowledgments are needed in order to reach a decision.

We can also observe that after a certain number of processes N, the latency increases when alpha increases. This is normal, because it could take more time to reach the acknowledgement of N/2 processes when we have a lot of processes that crash.

Conclusion:

In conclusion, the paragraph discusses the evaluation of the consensus latency in a fault-tolerant distributed system. The analysis is conducted by varying the number of processes and the timeout for electing a leader. The impact of the probability of crashing for fault-prone actors on the latency is also studied. For a small number of processes, a random behavior is observed, while for a larger number of processes, the latency increases as the timeout tle increases. Furthermore, increasing the number of processes also increases the consensus latency for a fixed Leader Election Time 'tle'. These findings highlight the importance of considering the number of processes and timeout values in designing fault-tolerant distributed systems.