Télécom Paris

Project Report

Obstruction-Free Consensus and Paxos

Adrien Ferrand-Laffanour
Kevin Garnier
Frédéric Srichanwit

2^{nd} year of engineering degree
Year 2025

Contents

List of Figures

1 Why Paxos?

Paxos is a consensus algorithm that allows a group of processes to agree on a single value, even in the presence of failures. It is designed to work in asynchronous distributed systems, where processes may fail or messages may be lost. Paxos is widely used in distributed systems for achieving fault tolerance and ensuring consistency. The algorithm is based on the idea of "proposers" that suggest values, "acceptors" that vote on one of the proposed values, and "learners" that learn the agreed-upon value. The algorithm ensures that even if some processes fail, as long as a majority of processes are functioning, they can still reach consensus. Paxos is particularly useful in scenarios where a distributed system needs to maintain a consistent state across multiple nodes, such as in databases, distributed file systems, and cloud computing platforms. It provides a robust and reliable way to achieve consensus, making it a popular choice for building fault-tolerant distributed systems.

Contention in distributed systems refers to the competition between multiple processes or nodes for shared resources, such as network bandwidth, CPU cycles, or access to a shared data store. High contention can lead to delays, reduced throughput, and increased latency, as processes must wait for access to the resource. Managing contention is critical for ensuring the efficiency and scalability of distributed systems, especially in scenarios where multiple processes attempt to perform operations simultaneously.

2 Java Implementation of the Robust Key-Value Store

2.1 Java classes

For our implementation with Java and the Akka framework, we have created the following classes:

- `Main`: The main class that creates the different actors and configures the system. It is also the class that computes the statistics at the end of the simulation.
- `Process`: The actor class. The obstruction-free consensus algorithm is implemented in this class.
- `AbortMsg`: The class that represents the abort message sent between the processes.
- `AckMsg`: The class that represents the acknowledgment (ACK) message sent between the processes.
- `CrashMsg`: The class that represents the crash message sent between the processes. When a process receives a crash message, it will enter in the fault-prune mode.
- `DecideMsg`: The class that represents the decide message sent between the processes.
- `GatherMsg`: The class that represents the GATHER message sent between the processes.
- `ImposeMsg`: The class that represents the IMPOSE message sent between the processes.
- `Members`: The class that contains the processes' references.
- `Pair`:
- `ReadMsg`: The class that represents the read message sent between the processes.

We didn't want to use multiple threads in the `Process` class due to the large number of processes being launched. It is more efficient to let the Akka framework's logic thread handle the processes. Therefore, we implemented a state machine in the `Process` class to manage the different states of the functions: sending the initial read request, waiting for a quorum of responses, sending the write request, and finally, waiting for a quorum of write acknowledgments.

2.2 Akka Design

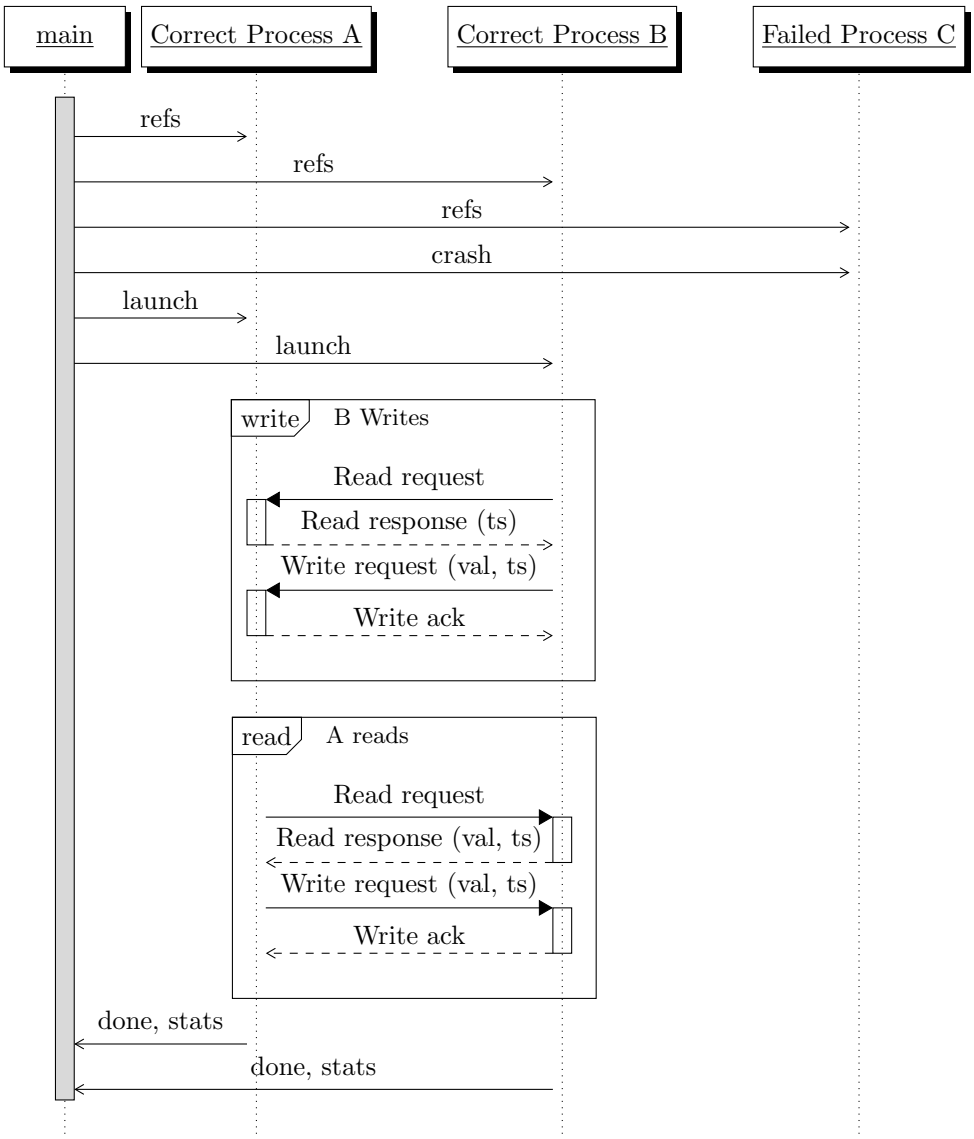


Figure 1: Sequence diagram of the Akka design

2.3 Statistics computation

TODO

Throughout the simulation, the processes continuously collect data for the statistics. At the end of the simulation, the main class will compute this data:

For the throughput, we have used the formula :

$$\text{Throughput} = \frac{\text{Number of operations}}{\text{Total time}}$$

3 Performance analysis

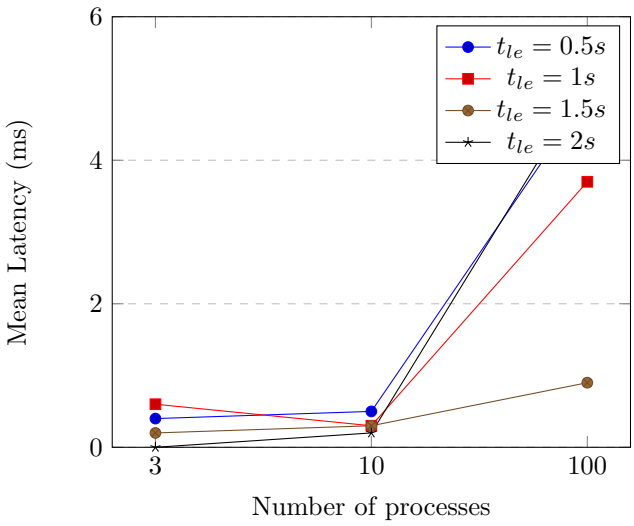
The simulation has different parameters that can be changed to test the performance of the system. The main parameters are:

- N The number of processes and f the number of processes which can fail.
- t_{le} The fixed timeout.
- α The probability of failure of a process.

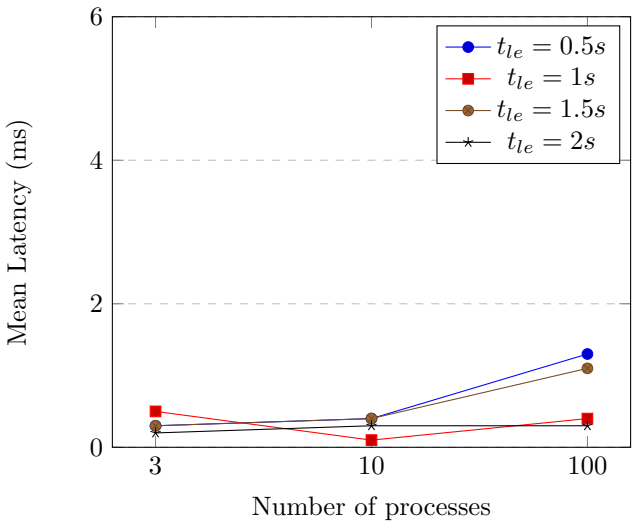
Since we need a quorum, f must be less than $N/2$. Thus, we have different scenarios to test the performance of the simulation with $(N, f) \in \{(3,1),(10,4),(100,49)\}$, $t_{le} \in \{0.5, 1, 1.5, 2\}$ and $\alpha \in \{0, 0.1, 1\}$.

3.1 Results for a fixed timeout t_{le}

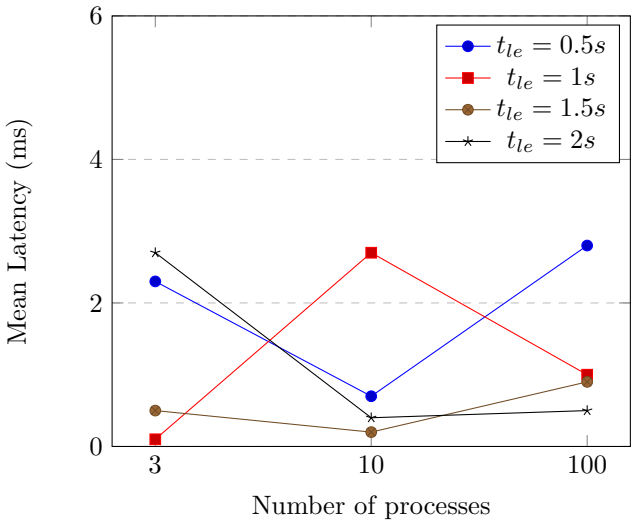
Let's now analyze the results for a fixed timeout t_{le} and different values of N.



(a) Mean latency comparison for $\alpha = 0$



(b) Mean latency comparison for $\alpha = 0.1$



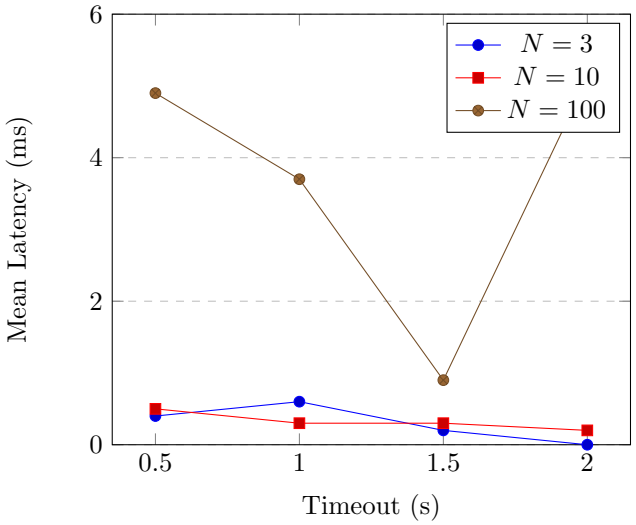
(c) Mean latency comparison for $\alpha = 1$

Figure 2: Mean latency comparison for fixed timeout and alpha values and different number of processes

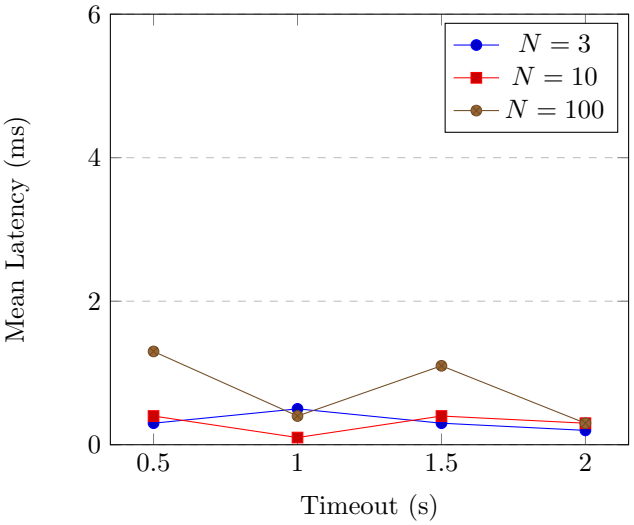
Since consensus is reached before the call to *hold*, the differences in the t_{le} values do not significantly impact performance. In a larger system, having a timeout is very useful, because it allows the system to still be able to reach a consensus by reducing the number of proposing process to only one. The choice of timeout should balance failure detection and system stability. A timeout that is too long delays the election of a new leader when a failure occurs, increasing recovery time. On the other hand, a well-chosen timeout ensures that the system remains responsive without unnecessary leader use.

3.2 Results for a fixed number of processes

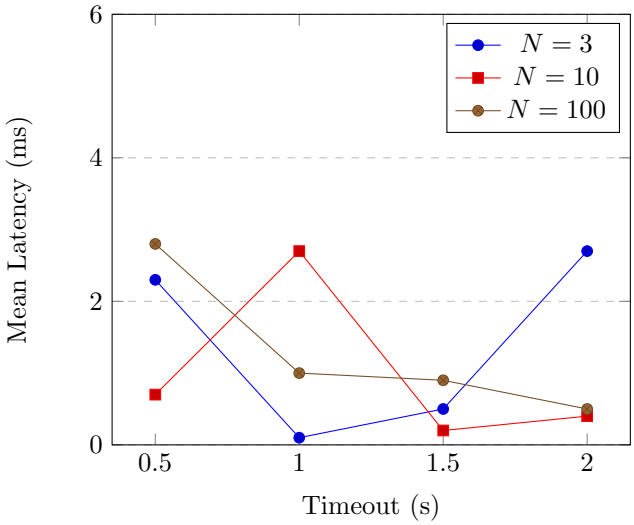
Let's first analyze the results for a fixed number of processes N and different values of t_{le}.



(a) Mean latency comparison for $\alpha = 0$



(b) Mean latency comparison for $\alpha = 0.1$



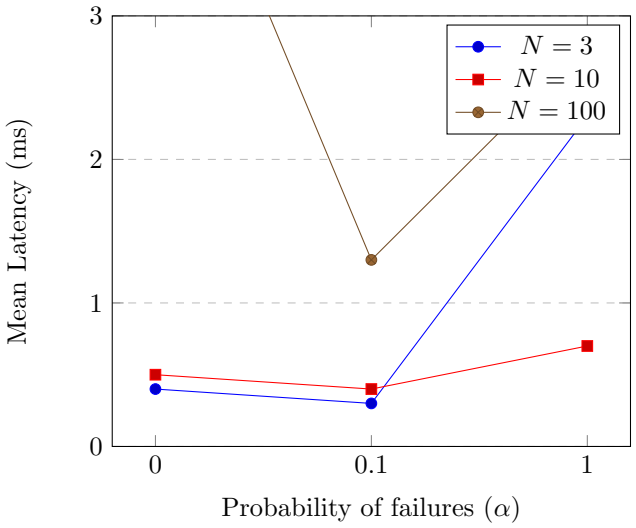
(c) Mean latency comparison for $\alpha = 1$

Figure 3: Mean latency comparison for fixed number of processes and alpha values

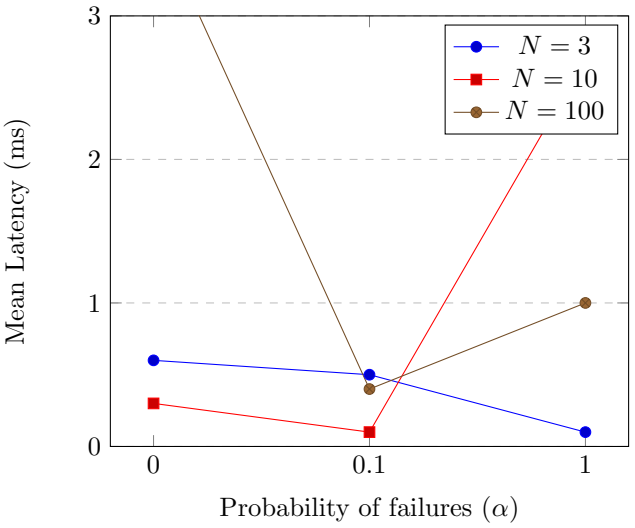
We can see here that the latency increases as the number of processes grows. This is due to the required quorum size needed to reach consensus. However, while having fewer processes can be faster, it is hardly scalable for more general use cases where a high number of requests may occur. With too few processes, the system risks overloading. Furthermore, with more processes, and given that at least $\frac{N}{2} + 1$ correct processes are required, the system can tolerate a higher number of failures.

3.3 Results for the probability of failure

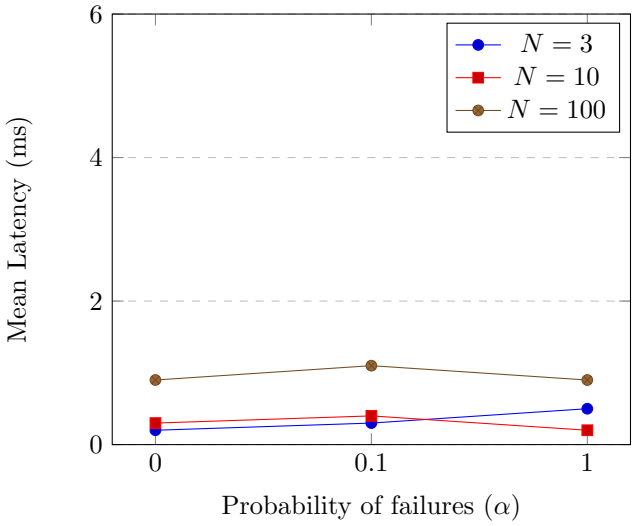
Let's now analyze the results for fixed numbers of processes N and timeout t_{le}, with different probabilities of failure α.



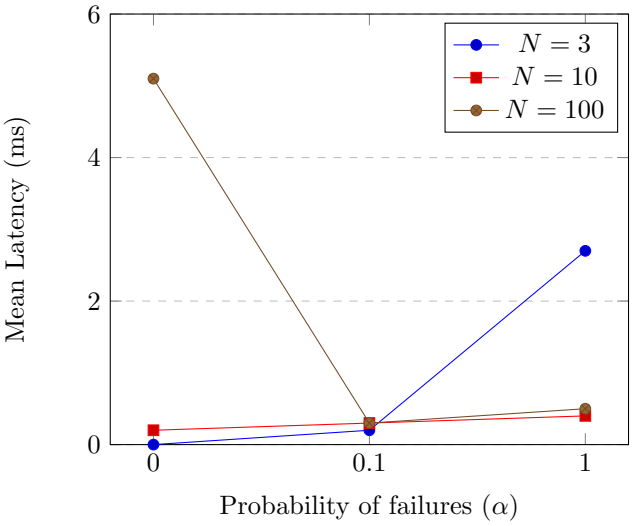
(a) Mean latency for $t_{le} = 0.5s$



(b) Mean latency for $t_{le} = 1s$



(c) Mean latency for $t_{le} = 1.5s$



(d) Mean latency for $t_{le} = 2s$

Figure 4: Mean latency comparison for fixed number of processes and alpha values

We can see in this part that the latency increases as the probability of failure grows. This is due to the fact that the processes are more likely to fail, so they can't respond to reqyests and thus in order to reach a consensus, the process need to call all of the correct processes $(N/2 + 1)$.