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

1 Why Paxos? 3

2 Java Implementation of the Robust Key-Value Store 3
2.1 Java classes . . . 3
2.2 Akka Design . . . 4
2.3 Statistics computation . . . 5

3 Performance analysis 6
3.1 Results for a fixed timeout t_{le} . . . 6
3.2 Results for a fixed number of processes . . . 7
3.3 Results for the probability of failure . . . 8
3.3.1 Results for $N = 3$ . . . 8
3.3.2 Results for $N = 10$ . . . 9
3.3.3 Results for $N = 100$ . . . 10

4 Conclusion 10

List of Figures

1 Sequence diagram of the Akka design . . . 4
2 Mean latency comparison for fixed timeout and alpha values and different number of processes . . . 6
3 Mean latency comparison for fixed number of processes and alpha values . . . 7
4 Mean latency for $N = 3$ . . . 8
5 Mean latency for $N = 10$ . . . 9
6 Mean latency for $N = 100$ . . . 10

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.

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

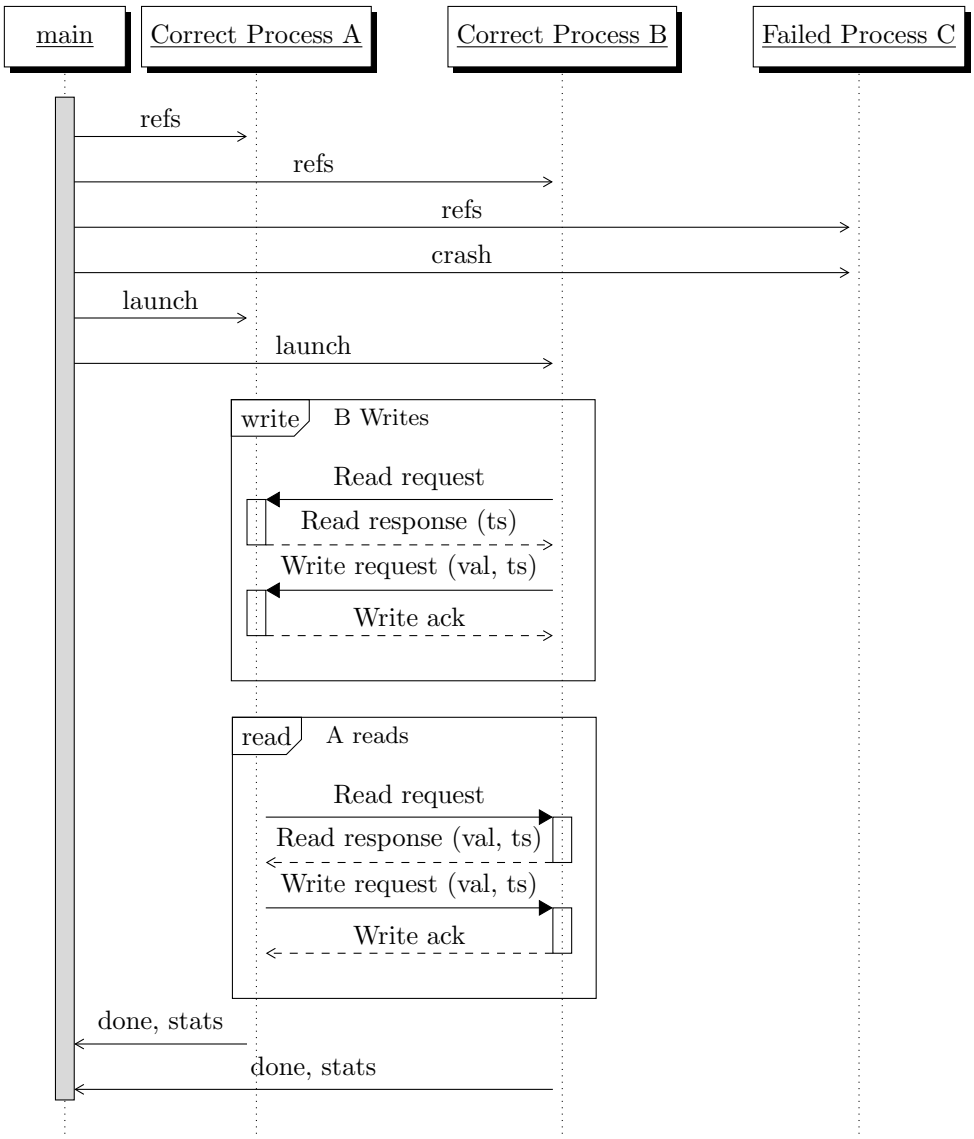


Figure 1: Sequence diagram of the Akka design

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

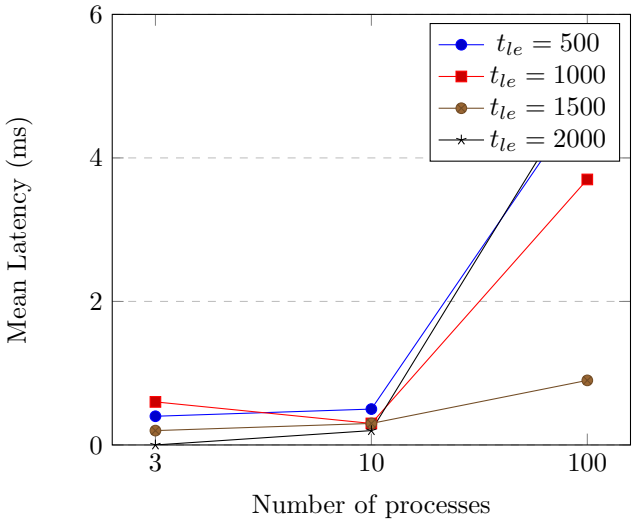
The simulation has different parameters that can be changed to test the performance of the system. The main parameters are:

- 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.

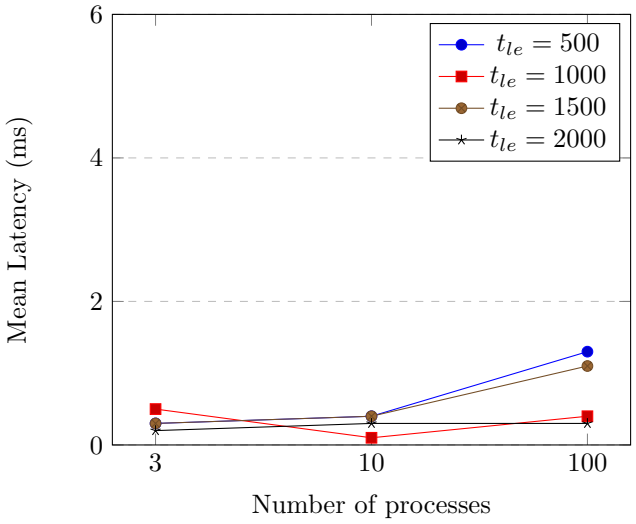
Since we need a quorum, f must be less than $N/2$. Thus, we have different scenarios to test the performance of the simulation with $(N, f) \in \{(3,1),(10,4),(100,49)\}$, $t_{le} \in \{0.5, 1, 1.5, 2\}$ and $\alpha \in \{0, 0.1, 1\}$.

3.1 Results for a fixed timeout t_{le}

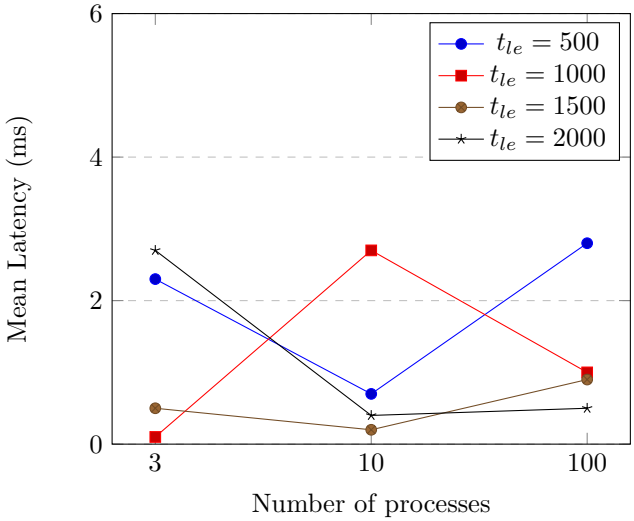
Let's now analyze the results for a fixed timeout t_{le} and different values of N.



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



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

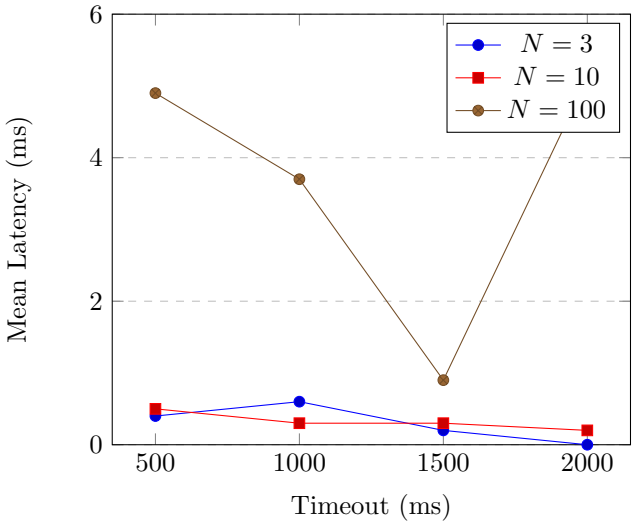


(c) Mean latency comparison for $\alpha = 1$

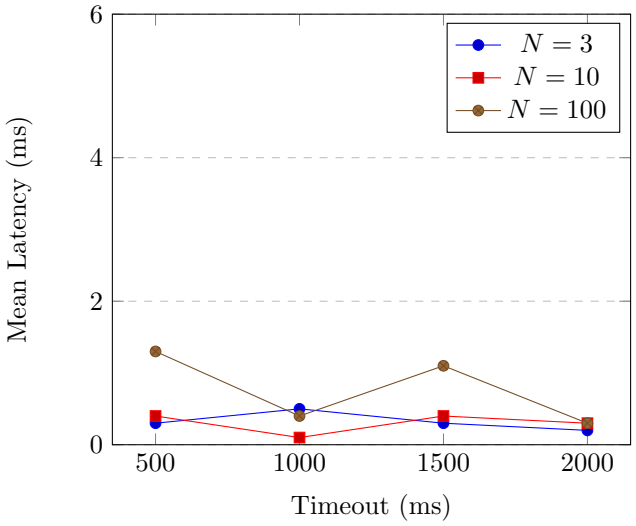
Figure 2: Mean latency comparison for fixed timeout and alpha values and different number of processes

3.2 Results for a fixed number of processes

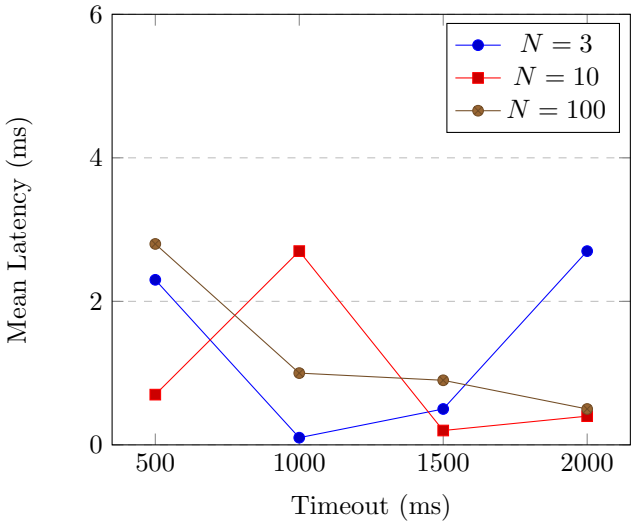
Let's first analyze the results for a fixed number of processes N and different values of t_{le}.



(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

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.

3.3.1 Results for $N = 3$

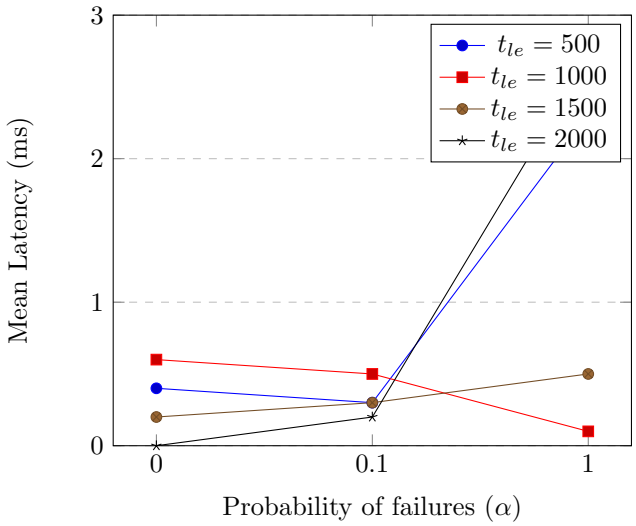


Figure 4: Mean latency for $N = 3$

3.3.2 Results for $N = 10$

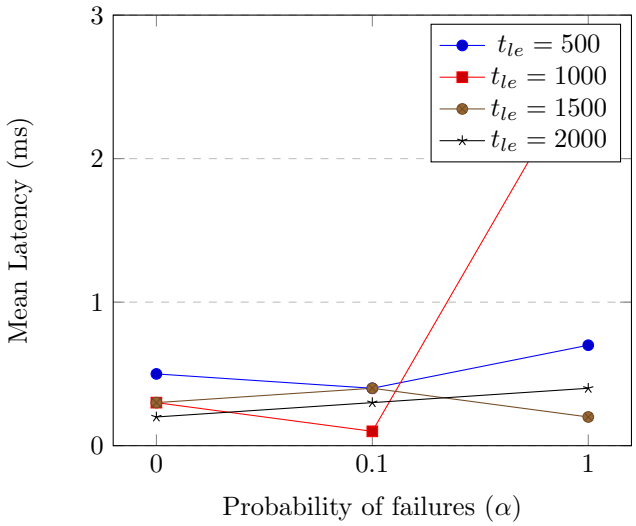


Figure 5: Mean latency for $N = 10$

3.3.3 Results for $N = 100$

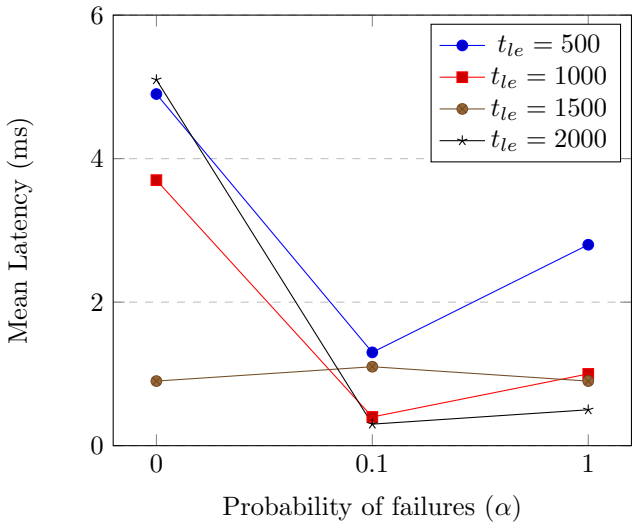


Figure 6: Mean latency for $N = 100$

4 Conclusion