Télécom Paris

Project Report

Robust Key-Value Store

Adrien Ferrand-Laffanour
Kevin Garnier
Yann Girey
Frédéric Srichanwit

2^{nd} year of engineering degree
Year 2025

Contents

List of Figures

1 Java Implementation of the Robust Key-Value Store

1.1 Java classes

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

- `SLR206`: 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 algorithms given in the pseudo-code are implemented in this class.
- `MyMessage`: The class that represents the string messages sent between the processes. It is used to launch processes or trigger crashes.
- `ReadRequest`: The class that represents the read requests sent between the processes.
- `WriteRequest`: The class that represents the write requests sent between the processes.
- `ReadResponse`: The class that represents the read responses sent between the processes.
- `WriteAck`: The class that represents the write acknowledgments 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.

1.2 Akka Design

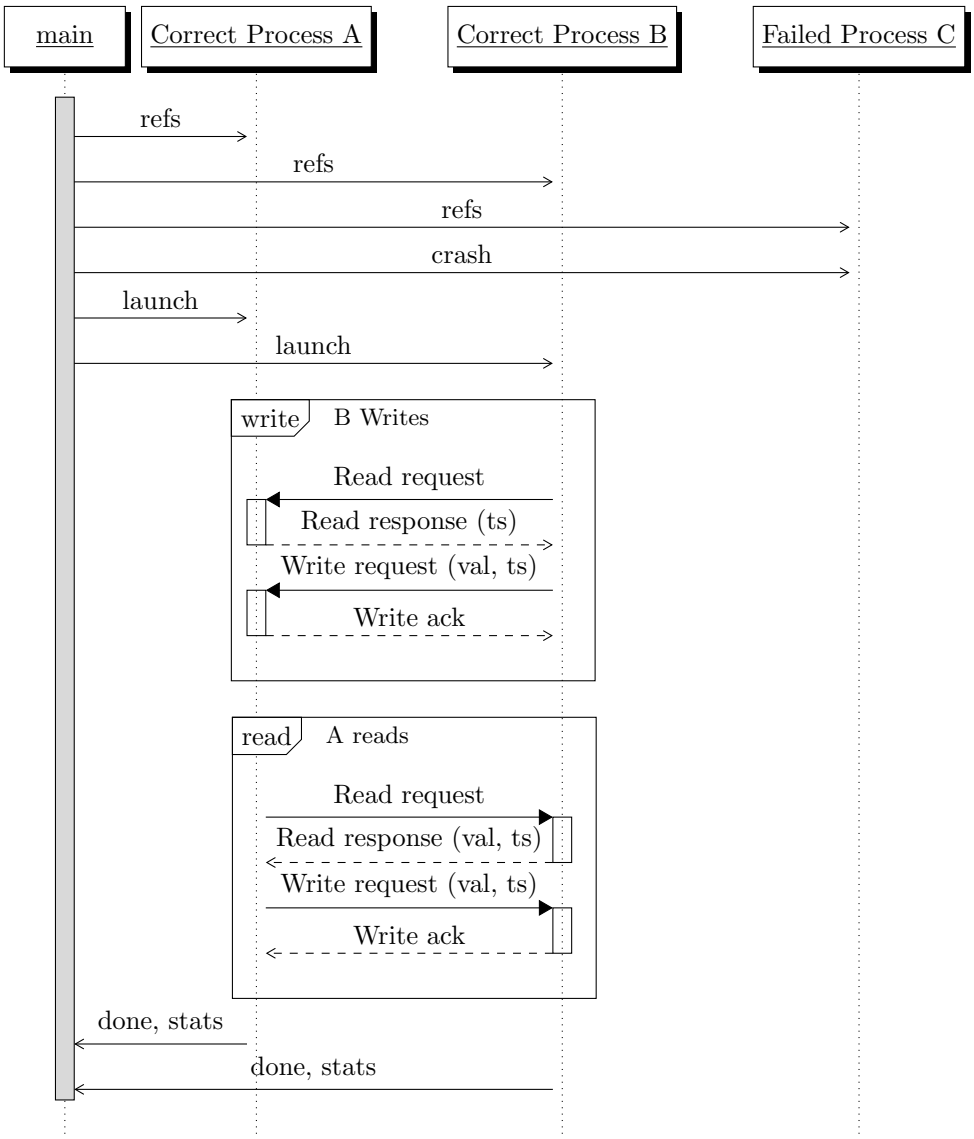


Figure 1: Sequence diagram of the Akka design

We decided not to use multiple threads in the `Process` class due to the high volume of processes being launched. It is more efficient to let the main logic thread of the Akka framework handle the processes. As a result, we implemented a state machine within the `Process` class to manage the different stages 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.

1.3 statistics computation

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

- The time taken for each process to complete a write operation.
- The time taken for each process to complete a read operation.
- The average time spend by a process to complete all the write operations.
- The average time spend by a process to complete all the read operations.
- The throughput of the system.

For the throughput, we have used the formula :

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

2 Correctness

2.1 Liveness Proof:

The only time a process might not finish is when it is waiting for a majority of processes (lines 9, 13, 18, 21). Since the connections are assumed to be reliable, all messages sent are received in finite time by the other processes. Let p be a correct process that has sent a message to the other processes and is waiting for a majority of responses. At most f processes may fail, so there are at least $N - f - 1$ other processes that will function and respond. Since there are $N - 1$ other processes, the majority is $(N - 1)/2$ processes. However, since $f < N/2$, it follows that:

$$N - 1 - f > (N/2) - 1.$$

If N is even, since both $N - 1 - f$ and $(N/2) - 1$ are integers, then:

$$N - 1 - f \geq N/2 > (N - 1)/2.$$

If N is odd, since $N - 1 - f$ is an integer, we have:

$$N - 1 - f \geq (N/2) - 1 + 1/2 = (N - 1)/2.$$

In all cases, p will receive a response from a majority of processes and can complete its wait. Hence, liveness is ensured.

2.2 Safety Proof:

First, let's show that when a write occurs, it always uses the highest available timestamp.
When a write operation starts, it contacts a majority of processes to learn the highest timestamp currently in use. If a write has completed before this one, the new write will be aware of the timestamp from that previous write operation because it has already written to a majority of processes.
Let's consider two writes from two different processes in parallel. If one write writes its timestamp and the second write notices it, the second write will have a higher timestamp. We can then easily linearize the operations by ordering them according to their timestamps. If the writes have the same timestamp, we can linearize them by placing the linearization point for the write with the smaller value before the write with the larger value.

Now, consider a read operation. A read will first contact a majority of processes to learn the most recent values, which will correspond to the highest timestamps. Because each write updates at least a majority of processes, any write that completes before the read will have written its value to at least one of the processes contacted. Therefore, the read operation will always know about the most recent write.
Let's consider a scenario where a write occurs in parallel with a read. There are two possibilities: Either the read operation has already seen the value from the write: In this case, we follow the reasoning above, and the linearization point for the read will be placed just after the write. Or the read operation does not know about the write: If the read has not yet seen the value from the write, the linearization point will be placed just after that write.

Thus, the entire history of operations is linearizable. This argument shows that by maintaining the highest timestamps for each write and ensuring that reads always return the most recent value, we can create a linearizable execution history, where the order of operations reflects a valid sequential execution.

3 Performance analysis

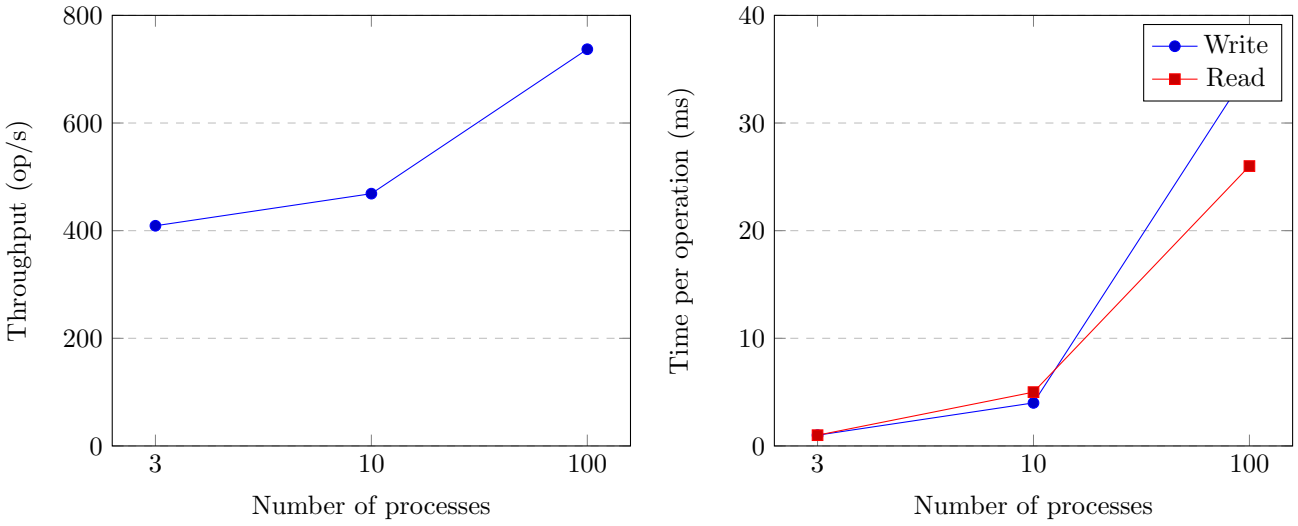
The simulation has different parameters that can be changed to test the performance of the system. The main parameters are:

- N The number of processes.
- M The number of operations per process.
- f The number of failures.

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

3.1 Results for a fix number of operations

Let's first analyze the results for a fix number of operations and different values of N and f.



(a) Throughput comparison

(b) Operation execution time comparison

Figure 2: Performance analysis for $M = 3$

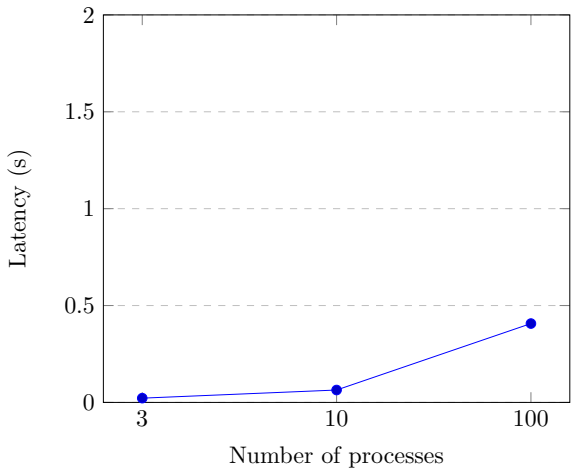


Figure 3: Latency comparison for $M = 3$

We can observe that throughput increases with the number of processes. However, the growth factor is not linear. Adding a significantly larger number of processes does not lead to a substantial increase in throughput, as there are not many operations per process to execute.

Similarly, the time required to complete an operation increases with the number of processes. This is because processes must wait for a majority of responses to proceed. As the number of processes grows, it takes more time to gather a majority of responses.

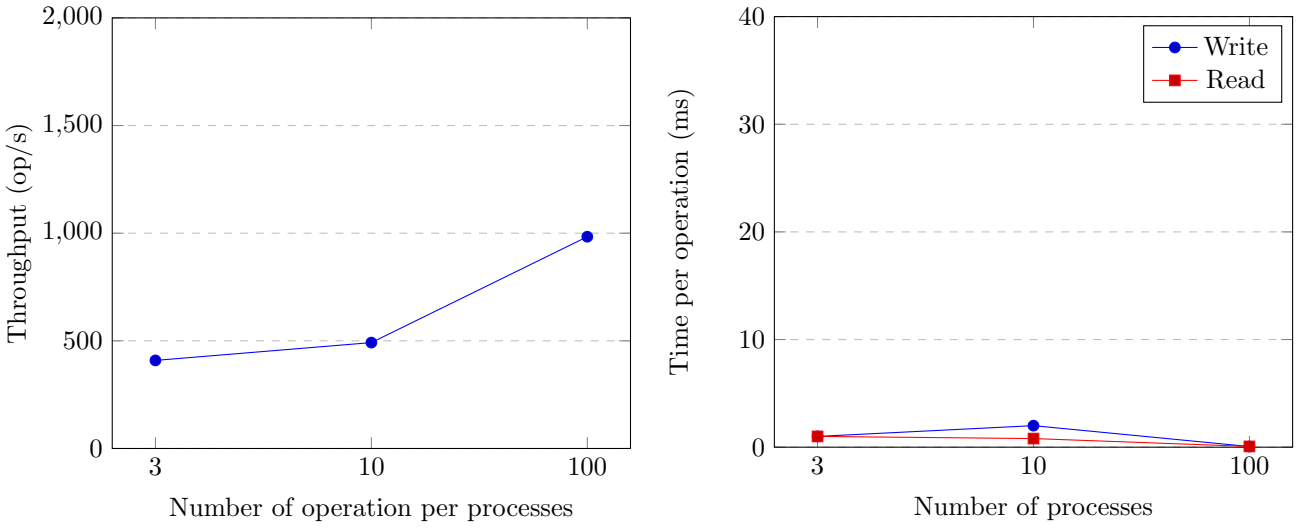
One reason the time taken to complete a write operation is longer than for a read operation is that, during a read, some processes may have already completed their operations. The network might also be less congested, allowing processes to receive a majority of responses more quickly.

The results are consistent for the other values of M: both throughput and the time taken to complete an operation increase with the number of processes. However, the more operations there are per process, the better the performance of the system with 100 processes running, though some latency remains.

3.2 Results for a fix number of processes

As we did in the previous section, we will analyze the results for a fixed number of processes and different values of M. This will allow us to identify which configuration is most efficient in each scenario.

3.2.1 Results for $N = 3$ and $f = 1$



(a) Throughput comparison

(b) Operation execution time comparison

Figure 4: Performance analysis for $N = 3$

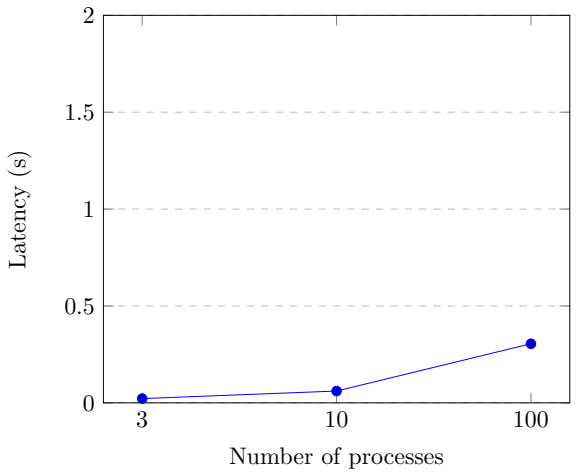
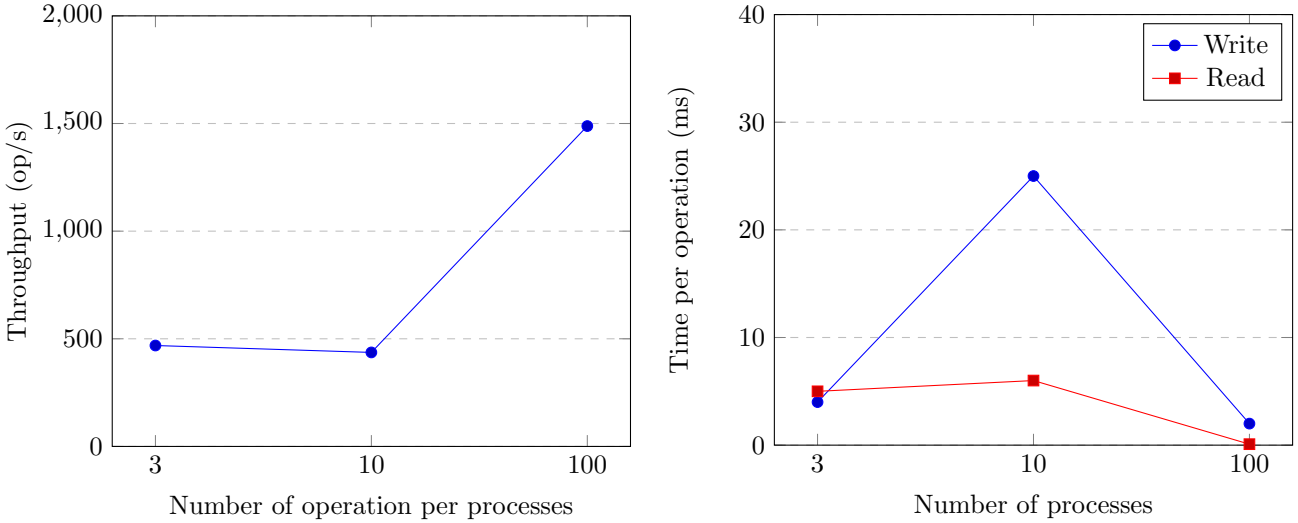


Figure 5: Latency comparison for $N = 3$

We can observe that throughput increases with the number of operations per process, while the time taken to complete an operation remains almost constant.

Although we cannot compute many operations in parallel, the throughput remains high because the quorum is reached almost instantaneously.

3.2.2 Results for $N = 10$ and $f = 4$



(a) Throughput comparison

(b) Operation execution time comparison

Figure 6: Performance analysis for $N = 10$

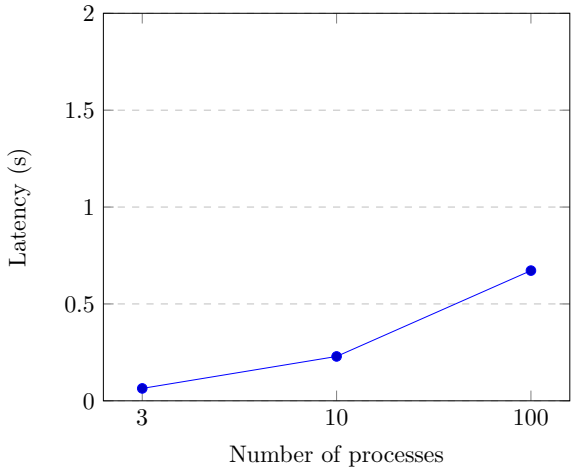


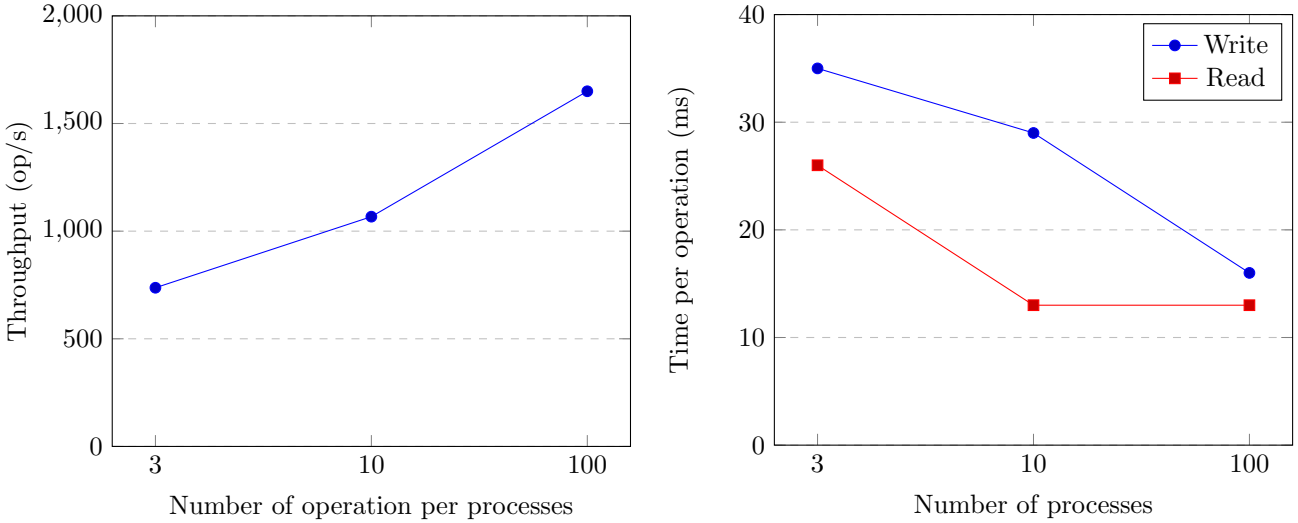
Figure 7: Latency comparison for $N = 10$

First, we observe that with a low number of operations, this configuration is less efficient than the previous one—about 12% less efficient. It experiences higher latency, and throughput is lower. However, for higher values of M, this configuration becomes more efficient because it can process several operations concurrently.

If we consider each operation as a request from a user, and latency is a critical factor, this configuration is better suited. Fewer users will have to wait for processes to handle their requests.

The operation execution time may seem unusual, but except for the case with 10 processes, the time taken to complete an operation remains almost constant. This indicates that the processes are waiting for each other more in this configuration compared to the previous one.

3.2.3 Results for $N = 100$ and $f = 49$



(a) Throughput comparison

(b) Operation execution time comparison

Figure 8: Performance analysis for $N = 100$

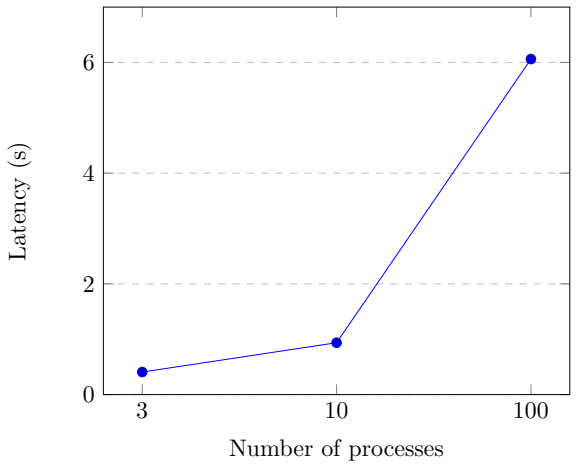


Figure 9: Latency comparison for $N = 100$

Once again, the throughput follows a logarithmic trend with the number of operations per process. This configuration is less efficient than the previous ones for smaller values of M.

For instance, with 300 operations, the configuration with 3 processes is 26% more efficient than this one. Similarly, for 1000 operations, the configuration with 10 processes is 13% more efficient. However, as the number of operations increases, this configuration becomes more efficient because it can process several operations simultaneously.

4 Conclusion

In conclusion, we can say that the configuration with 3 processes is more efficient for a small number of operations, while the configuration with 100 processes is more efficient for a large number of operations. The configuration with 10 processes lies in between.

If we were to implement this system in a real-life scenario, we would need to consider the number of operations to be processed and the acceptable latency. Server characteristics and network capacity may also impose constraints that influence the choice of the optimal configuration. The configuration with 10 processes could be a good compromise between the other two in many situations, as it is sufficiently efficient for handling both small and large numbers of requests.