DDS-REPORT

Lorenzo Pecorari
Matteo Miletta
Martina Valentini

September 2024

1 Problem Explanation

This project focuses on the issue of causality detection between events in a distributed system, which is a fundamental component for managing and controlling distributed applications. The problem becomes particularly challenging in the presence of Byzantine processes, which are processes that can behave arbitrarily and unreliably, introducing errors into the system. The paper "Detecting Causality in the Presence of Byzantine Processes: The Synchronous Systems Case" demonstrates that causality detection in asynchronous distributed systems is unsolvable in the presence of Byzantine processes, regardless of the type of communication (unicast, multicast, or broadcast). In response to this limitation, the work shifts attention to synchronous systems, showing that in this context, it is possible to solve the problem of causality detection even with Byzantine processes. The project proposes an algorithm based on Replicated State Machines (RSMs) and vector clocks, which enables the detection of causality ensuring that events are correctly ordered despite faulty processes.

The problem formulation is done similar to the way in. An algorithm to solve the causality detection problem collects the execution history of each process in the system and derives causal relations from it. E_i is the actual execution history at pi. For any causality detection algorithm, let F_i be the execution history at p_i as perceived and collected by the algorithm and let $F = S_i\ F_i$. F thus denotes the execution history of the system as perceived and collected by the algorithm. Analogous to T(E), let T(F) denote the set of all events in F. The happens before relation can be defined on T(F) instead of on T(E). With a slight relaxation of notation, let $T(E_i)$ and $T(F_i)$ denote the set of all events in E_i and F_i, respectively. So to conclude we want to check that an event $(e^h)_h -> e^*_i$, which means the first event come before the second one. To demonstrate this we return $e^*_i > (e^h)_h$, that are the sequence number belonging to the own V matrix.

2 Technologies

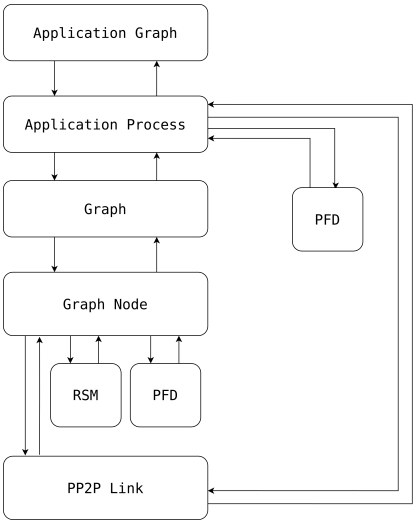
We've used python for this project with the following libraries:

1. *Networkx*: we've used this to handle with the graph creation and giving the correct topology to the system in order to simulate a network of processes.
2. *ZeroMq*: library to handle socket receiving and replaying for messages exchanging.
3. *matPlotLib*: library used for plotting results of experiments.

To sum up this project have also a cache problem on Windows, because linux handle an higher type of cache, instead Windows is more restrictive, so is suggested to erase cache and reinstall it when needed. In general, whatever UNIX-like architecture is strongly suggested for running, testing and modify the entire project.

3 Implementation

In a first place we've the diagram of our project:



Here an high-level explanation of every class we used into the system:

1. **Consensus**: We've used a **Commander - Liutant** pattern, which is an ad hoc implementation for application with Byzantines processes. How does it works: We take a Commander **A**, that gives "order" to the Liutant **B**, than all the Liutants should communicate between them to check what the Commander said to them. If some B says something different from what A said, that B will be removed by the agreement consensus, otherwise it will work like a normal Consensus algorithm.

2. **Perfect Failure Detector**: this module allows to detect faulty and maliscious processes/entities into the system; thanks to an internal (and increseable) timer, it can notify to whatever uses it the not-properly working objects.

3. **RSM**: Its implemented as a module used by Graph Node and it takes care about the LASKALSJ and V matrixes, events that happen during the execution and also handles an its own internal clock.

4. **Perfect Point To Point Link**: it schematically represents a bidirectional link that manages messages exchange between the entities that uses it. Its main working is based over ZeroMQ and the free TCP ports the operating system offers to the user.

5. **Application Graph**: it is the entity related to the administration of the entire system. It handles Application Graph objects and uses all the relative primitives allwoing to generate events among them and over the lower levels. Thanks to it, the user can interact with the system without using specific methods with higher complexity and lower understanding.

6. **Application Process**: a module capable to interface with the Perfect Point To Point Link module(s), its Graph and a Perfect Failure Detector; it is in charge to manage instruction asked by Application Graph like the send or the receive of a message other than specific operations like the consensus between RSMs of its lower levels.

7. **Graph**: this module allows to handle its lower-levelled Graph Node; in a certain way is like a smaller and weaker version of Application Graph but it has to handle the entities representing the replicas of the process. Each of this objects has a number of Graph Nodes/RSMs equals to (3t+2), where t is the number of Byzantine processes tolerated.

8. **Graph Node**: this is the core of the entire system. Thanks to it, is possible to keep track of events relations through the RSM, the PFD and the PP2P Link. All of these modules allows it to communicate efficiently with all the others Graph Nodes (into the same ensamble/ Graph) and their relative RSM.

In the end there were not used the cryptographic primitives because of they would slow the execution of the entire system and they might also increase a lot the complexity and the understandability of the project.

4 Evaluation design and Dependability Evaluation

In a Byzantine fault-tolerant distributed system, dependability is crucial for ensuring the system continues to working correctly even in the presence of faulty or malicious nodes. To achieve dependability, we have implemented two key mechanisms:

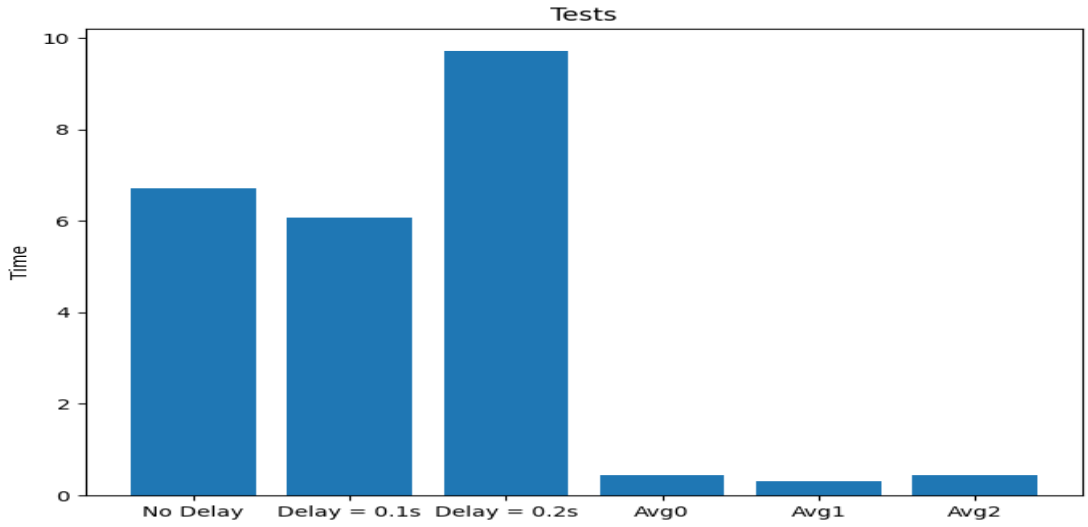
- **Consensus:** To ensure agreement among non-faulty nodes, even in the presence of Byzantine faults.
- **Replicated State Machine (RSM):** To maintain consistency across nodes by ensuring each replica processes the same sequence of operations.

Building on these mechanisms, the primary objective of this evaluation is to assess how well the system handles various workloads and maintains performance and reliability under different conditions. The key objectives of the dependability evaluation are:

- **Measure Latency:** Determine the average response time of the system under varying loads.
- **Analyze Throughput:** Assess the number of messages or requests processed per second.
- **Evaluate Scalability:** Examine the system's ability to scale with increasing nodes and workload.
- **Test Robustness:** Evaluate the system's resilience to Byzantine faults, including the ability to maintain consistency and availability in the presence of faulty nodes.

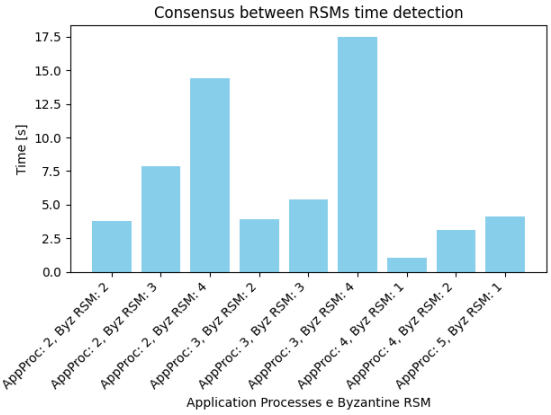
4.1 Latency and throughput

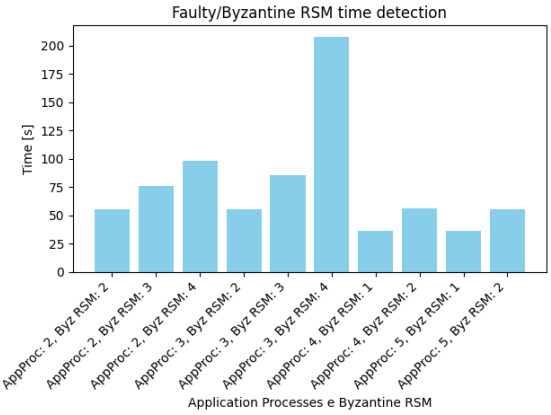
For latency, throughput and performances in general, there were done some experiments
The first one consists of the measurements of total time taken by the system for sending a set of 15 messages among the application processes of an application graph composed by 4 applications and each of them has 8 RSM (2 Byzantines). These measurements were replied 3 times and each of them counts a delay equals to 0, 0.1s and 0.2s . As the plots reports, it is interesting to observe that when the system receives instructions for sending messages each 0.1s, the average time needed to execute it is lower than the expectation and below the value obtained in the other two situations. By that, it can be evaluated that when the system has a load with that delay, is capable to reach a throughput of an average of 2.5 send/second (150 send/minute). Here the plots of this experiment:



4.2 Scalability evaluation

For this analysis, it was done a set of measurements of delays with respect to the variation of number of application processes and Byzantine replicas into the system with presence of a "real" one faulty RSM into each of the processes:





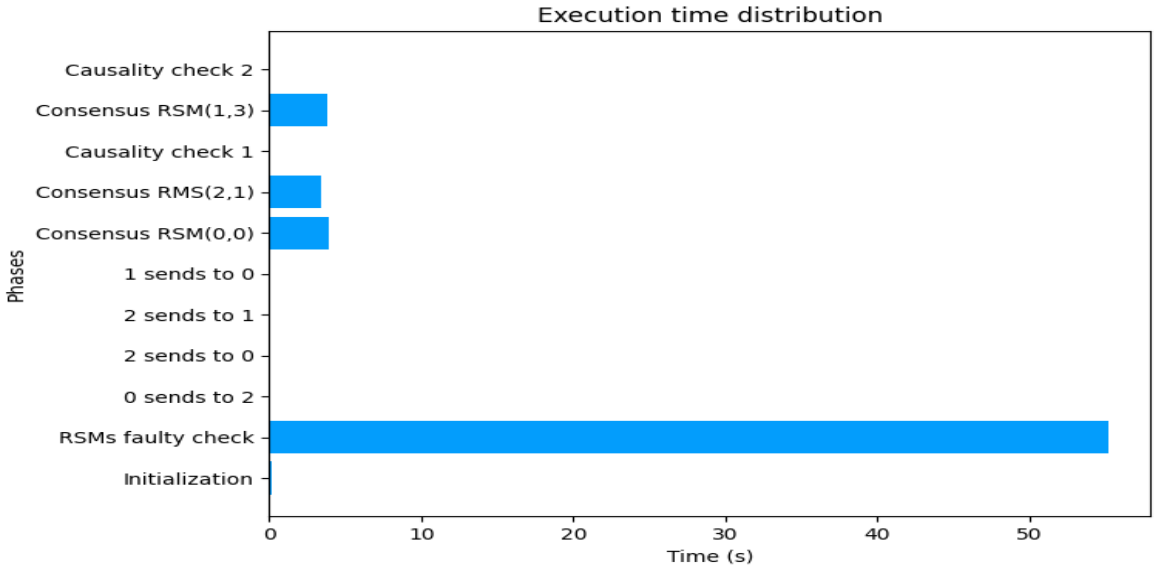
As it is possible to see, as it increases of Byzantine replicas into each application processes implies approximately an exponential growth of the time needed for reaching the consensus among the RSMs. In facts, this can be observed in particular for the situation where number of maliscious replicas goes from 2 to 4 for applications with 2 and 3 processes. For better results, it is suggested to adopt an architecture with 4 processes and 2 Byzantine replicas accepted for each of them in order to achieve a good tradeoff between performances and robustness against problems like these.

4.3 Robustness of the system

The project is based over communications without cryptographic primitives. As the paper suggests, it is needed to adopt a number of replicas equals to (3t + 1), since a consensus in presence of a Byzantine entity is guaranteed only with more than three times the number of maliscious users/objects. Also taking inspiration from the "Commander-Lieutant" consensus algorithm proposed by Leslie Lamport, the system starts with (3t+2) RSM for each of the n application process, implying that into a network there will be at least (3t+2)n replicas (plus n nodes) in the initial situation. After some evaluations, it emerged that, assuming one Byzantine replica for every process, the system is also capable to run properly without encountering issues.

5 Achieving the goal of the algorithm

All the previous description and analysis of the projects lead to what the algorithm tries to reach: consensus in presence of byzantine replicas. Using the methods and everything the system offers, we reached our purpose using an application that has 4 processes and each of them is capable to resist at most at 2 Byzantine replicas. Initially, we asked the system to check for every processes for faulty or Byzantine replicas through the method `check_faulty_rsms()` that, for every correct process into the system, invokes the PFD for all of their replicas. Once that, a set of send were executed in order to populate the V and LASKALSJ matrices and generate some events for the processes. Before checking for the causality of two couples of events, was asked to the interested processes to check for consensus among their RSMs; in the end the equivalent version of the "test" function called by the paper was executed. Following the upper instances, we had the results of our analysis plotted in this way



After this set of the instructions, it is possible to check, using the matrices V, the correctness of the results obtained for reaching causality of the four events.
The following table shows the send and receive events:

Action	Sending process	Receiver process
Send from process 0 to 2	0	2
Send from process 2 to 0	2	0
Send from process 2 to 1	2	1
Send from process 1 to 0	1	0

Action	Receiver process	Sender process
Receive to process 2 from 0	2	0
Receive to process 0 from 2	0	2
Receive to process 1 from 2	1	2
Receive to process 0 from 1	0	1

In relationship with these tables, can be represented the matrices tracking the vector clock of each process once it register an event:

V0	Proc 0	Proc 1	Proc 2	Proc 3
e1	1	0	0	0
e2	2	0	2	0
e3	3	2	2	0

V1	Proc 0	Proc 1	Proc 2	Proc 3
e1	0	1	3	0
e2	0	2	3	0

V2	Proc 0	Proc 1	Proc 2	Proc 3
e1	1	0	1	0
e2	1	0	2	0
e3	1	0	3	0

In general, the algorithm asks to check if e_h^x happens before e_i^*; the first step is to verify that e_h^x is into the set of event registered by h denoted by F_h and that $*$ is below or equal to the ID of the latest event registered by the process i . After these checks, if the value into $V[*, h]$ of i is greater or equals of x is returned True, False otherwise.

Into our schedule of instructions, the first check we made, after being sure that both of ensemble of RSM represented by the nodes involved into this operation, is about the event e_0^1 and e_2^1 (event number 1 of process 0 and event number 1 of process 2 respectively). So, into the code was needed to do the check if into the list of events of the process 0 there exists an event with sequence number equals to 1. The same was done for the process 2 and its event number 1. Since these checks returned a positive result, the system proceed to check if $V[1, 0]$ of 2 contained a value greater or equals to 1. Since that cell stored 1, the return of the algorithm was `True` . Thinking more about that without knowing what the code returned, it is simply to verify that e_0^1 causes e_2^1 .

The second and last check executed was about e_1^2 and e_2^1 and the same procedure were done as before. This time, it logically appears that the second event of process 1 could not have generated the first event of process 2. In facts, the code processes $V[1, 1]$ of 2 contained 0 that is not greater or equals of 2.

The algorithm states that should be a RSM to make the send and che causality check but, for complexity reasons, we made the consensus starting from a specific replica before executing the check from the above application process level.

6 Architecture Limitation

Python problem:

1. Stack limitation: Python have a fixed size for its stack, blocking all the recursive call. To handle this problem we've augmented this size, bringing it to 2147483647 (max int C possible)

2. Thread limitation: Python have also a Thread limitation, due the poor optimization under this point of view, in fact the higher are them, the hotter became the CPU. So we have incremented also this to the value we needed: 2^{10}

3. Ports conflict: Sometimes happens that the same port a process or one of its replicas tries to use is already occupied; for this reason, the system makes, thorugh ZeroMQ, a multiple (5) attempts of connection or bind to that port since it should be available for the user to be used.

Architecture problem:

1. Cache issue: Exists a different generation of cache between Linux and windows, in fact exists two version of our code to let it run better on the selected machine.

2. CPU issue: To achieve this code a shorter time you need to have a multi-core CPU with lots of thread and core. The quite good results we've obtained are mesured using an **Intel i7-1165G7** with **Ubuntu 22.40-LTS** installed, so a CPU of 11th generation with a UNIX-based OS.

3. Processes limit: We've used mainly a network configuration of 4 Processes and 2 byzantine for the best tradeoff it offers on our machines, also that going over this limit can represent a limit for a machine due the algorithm nature.

7 How To Run the code

To run the Main instances of this code having **n** macronode and 3t + 2 internal node (with t the byzantine processes):

- The [Windows] command is: >`python app_main.py`
- The [Linux] command is: >`python3 app_main.py`

Some libraries as Networkx or ZeroMQ may needed to be installed through the pip tool of Python.