Sieve protocol: a modular approach to replicating non-deterministic applications in a Byzantine Fault-Tolerant system

Antonio Tagliente Davide del Vecchio

Abstract

This project considers a modular approach for dealing with non-determinism in replicated services, where some processes are subject to faults and arbitrary behaviour (so-called Byzantine faults), with an implementation of the protocol Sieve by IBM (Cachin et al., 2016). This approach doesn't require any changes to the potentially non-deterministic application (and neither access to its internal data) for filtering out non-deterministic operations. It ensures that all correct processes produce the same outputs and that their internal states do not diverge. The chosen programming language is Python and the system is runnable locally.

Goal/Requirements

The goal of the project is to develop a proof of concept of the Sieve protocol.

The implementation of the system is composed of replicated services able to operate in Byzantine faulty conditions through the adoption of the Sieve protocol. Tolerating Byzantine faults means that the clients receive correct outputs as long as a qualified majority of the processes is correct, even if the faulty processes behave in arbitrary and adversarial ways.

Requirements:

- the system must be able to guarantee the expected result if the condition $n > 2f + 1$ is met, with *n = number of processes and f = number of faulty processes*.
- there must be a single process able to reply to the client
- protocol specifications:
    - messages:
        - INVOKE
        - EXECUTE
        - APPROVE
        - ORDER (CONFIRM or ABORT)
        - NEW-SIEVE-CONFIG
    - events:
        - abv-broadcast(m): broadcasts a message *m* to all processes
        - abv-deliver(p, m): delivers a message *m* to process *p*
        - Ψ-start-epoch (e, p): indicates that the epoch with number *e* and leader *p* starts
        - Ψ-complain(e, p): expresses a complaint about the performance of leader process *p*
    - functions:
        - V(m): validates message *m*
        - rsm-execute(o): requests that the state machine executes the operation *o*
        - rsm-output(o,s,r): indicates that the state machine has executed an operation *o*, resulting in new state *s*, and producing response *r*.
        - verify_p (σ, m): trusts message *m*
        - sign_p(m): signs message *m*

Before moving forward here is a summary of the Sieve protocol:

It works as an immutable black box, it executes all operations speculatively and then compares the outputs across the processes. If the protocol detects a divergence among a

small number of processes, the diverging outputs are sieved. If a divergence among too many outputs occurs, the operation is sieved from the message set.

Scenarios

User can execute the system by command line interface to test the behaviour and to learn and get practice with the protocol. For more details look at the Usage Examples.

There are two possible scenarios/execution cases:

1. All the processes are correct processes or at least there's a scenario with $n > 2f + 1$ (with *n = number of processes and f = number of faulty processes*).
   In this case we get the response with the correct result.
2. All the processes are faulty processes or the $n > 2f + 1$ rule isn't met.
   There is no response and then the operation is discarded.

The project is focused on correct results only, that's why the response that contains the string "`result`" as value is considered a correct response, ignoring divergents occurrences.

Self-assessment policy

The quality assessment of the produced software consists in testing:

- correct execution of the protocol
- nodes connection
- sending/receiving messages

according to the protocol.

Requirements Analysis

In the Goal/Requirements part are described the explicit requirements of the project, below are the implicit requirements.

From the user point of view, the most important requirement, not formalised yet, is the user interface. The system interface will allow to:

- start the program
- set the run configuration (e.g. number of processes)
- watch and follow the execution
- visualise statistics

Another implicit requirement is the communication between processes, that represents the foundation for an effective and efficient interaction between them. Moreover the protocol cannot exist without the correct processes connection.

This is a simplified model of a distributed system, for this reason a single entry point to manage requests between client/user and the system is needed. In addition there's a necessity to have a load balancer to direct the requests to the processes.

The selected technologies to implement the requirements described so far are:

- Python language: flexible implementation and open to changes
- Object oriented: friendly system implementation
- MQTT: lightweight publish-subscribe communication protocol

The abstraction gap is present with regard to:

- communication: it's necessary to adapt the publish-subscribe communication to the generic communication described by Sieve paper
- processes: the protocol is meant to be used with multiple server replicas, instead for this project it's executed between different threads on the same machine (to simulate the replica independency).

Design

The project has been designed analysing each component and the interaction between them.

Here is a brief description of the fundamental components:

- Process => SieveNode (Python Object)
- Message => MQTTmessage (Paho message)
- Event => MQTTmessage (Paho message)
- Ω => Omega (Python Object)
- Function => Python Object method

Structure

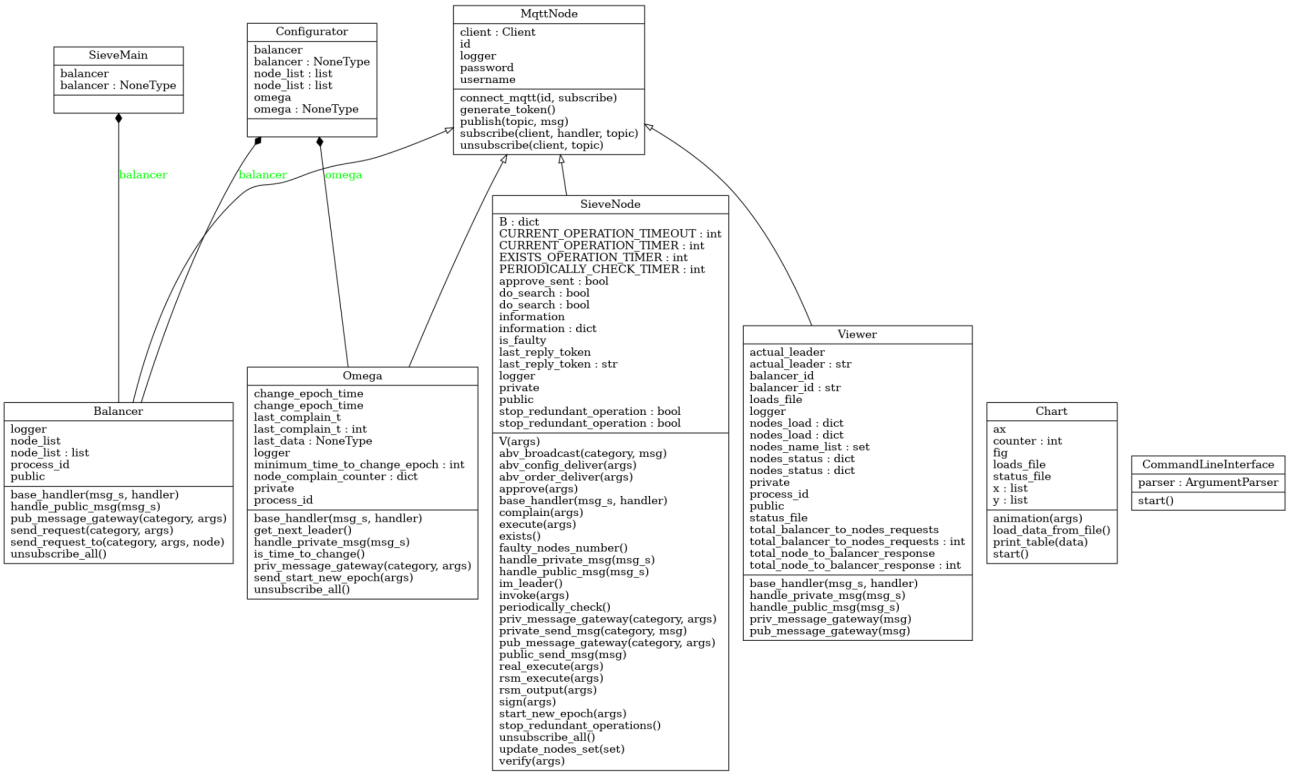
The core entities required, according to the protocol, are:

- SieveNode: encapsulates all Sieve protocol operations (message manipulation and data validation)
- Omega: manages Byzantine leader detector and Byzantine epoch-change operations.

Both entities operate as state machines, specifically SieveNode works as a Replicated state machine (the replicas running on multiple servers or threads in this case).

The other entities are:

- Balancer: directs client requests to SieveNodes. In this way it's possible to avoid process overhead.
- MqttNode: manages main MQTT functionalities
- Chart: shows SieveNodes loads (number of requests), in addition it shows on the CLI multiple system snapshots in tabular format
- Viewer: stores messages, SieveNodes requests and loads in specific files. This files are used by other project components
- CommandLineInterface
- Configurator: utility class to configure the system components such as:
    - SieveNode
    - Omega
    - Balancer



1. Class diagram

The design is very simple, there is only one hierarchy where the MqttNode is extended by SieveNode, Omega, Viewer. As stated above, the MqttNode implements a generic object able to use the MQTT protocol.

Behaviour

This section will describe the system and its components behaviour.

Below is a simplified explanation of the Sieve protocol behaviour, for an in-depth description look at the original study.

First thing first, the Leader is elected. The Leader is a SieveNode elected randomically (or fixed) at first. It can change in particular cases (described further below).

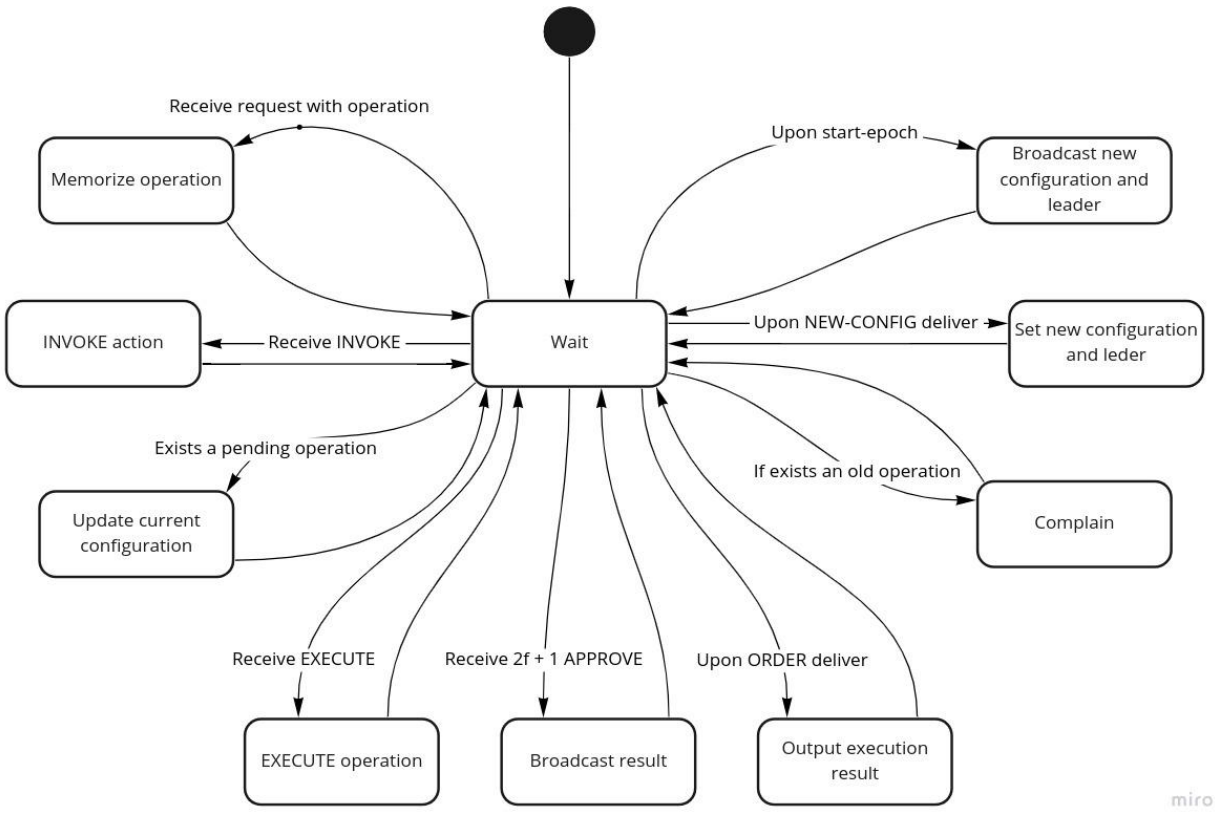
The processes send (INVOKE) all operations received by the client/Balancer to the Leader. The Leader then initiates that all the processes execute the operation speculatively; subsequently the processes agree on an output from the operation and thereby commit the operation.

To do so, the Leader sends a command (EXECUTE) to the processes with the operation. Every process executes the received operation speculatively and sends back to the leader a message (APPROVE) with the response, signed and hashed.

The Leader receives 2f + 1 messages (*f = number of faulty processes*). If the Leader observes at least f + 1 approvals from the same output, then it confirms the operation. Otherwise the Leader concludes that the operation is aborted because of diverging output.

The Leader then broadcasts a message (ORDER) containing the operation, the output, the decision confirmed or aborted and for validation the set of received messages that justify the decision to confirm or abort. At this moment the validity check is executed to verify the justification. If the operation is confirmed, the process adopts the output decided by the Leader.
In case of a broadcast with an abort decision, the current operation is discarded.

The non-Leader processes are constantly monitoring the Leader performance. If the Leader doesn't achieve the desired goal after some time or there are complaints, a switch to a new Leader is initiated.



2. SieveNode State Machine

The implementation of the logic described above is splitted between SieveNode, Omega and Balancer.

The Balancer as already mentioned sends the operation to one SieveNode, which in turn sends the message to the Leader to start the algorithm described above. The SieveNode

encapsulates all the behaviour of the protocol mentioned, with the exclusion of one part, included in Omega.

Omega is in charge of managing the COMPLAIN received by the SieveNode and executing the Leader change when needed.

When the Balancer sends the request/operation to the SieveNode and the algorithm starts, the Viewer also registered on the messages topic, starts to log the data to the files, data read and shown to the user by Chart component.

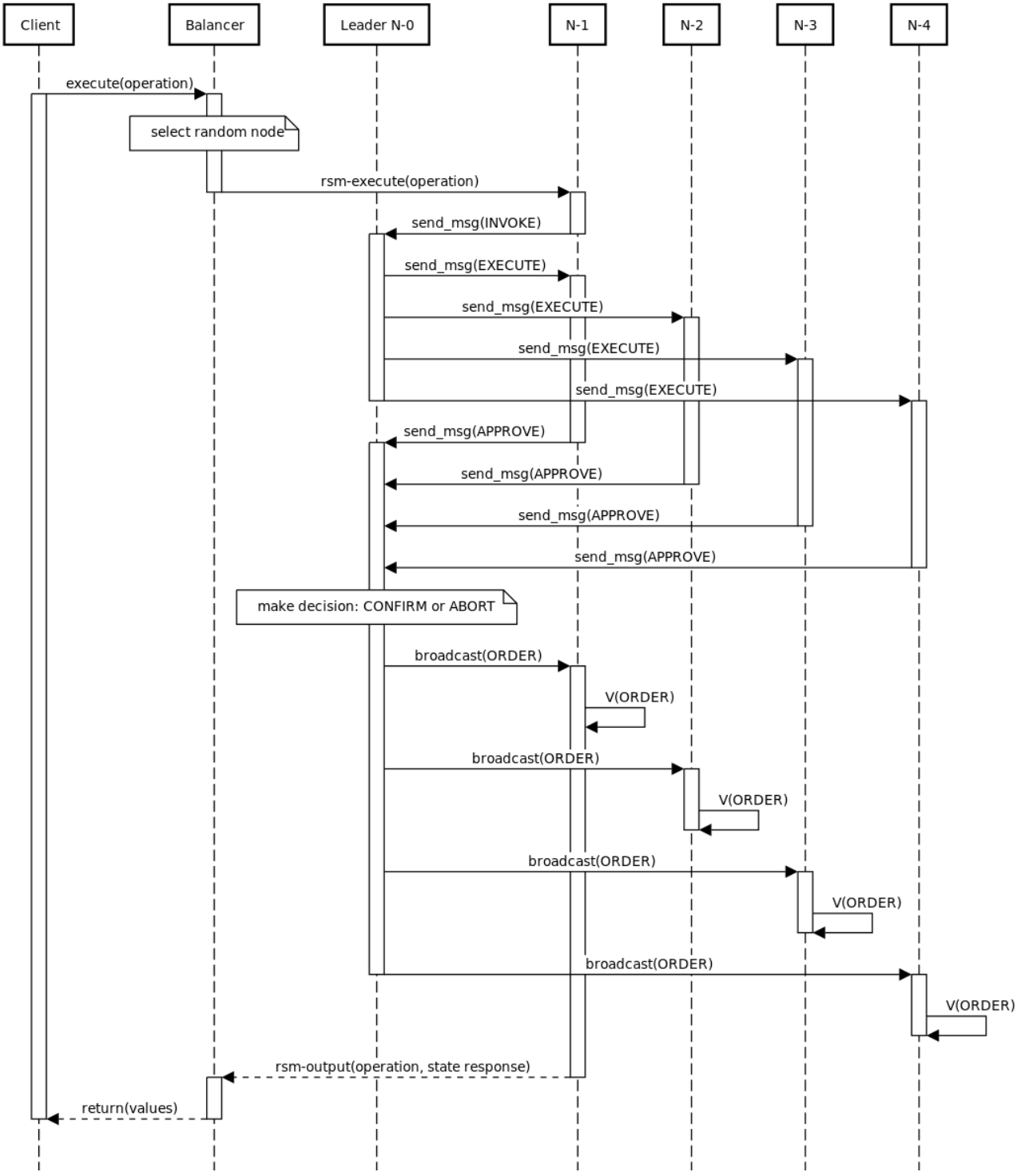
Interaction

The interaction between clients and servers is emulated via method calls through the Balancer; this kind of interaction can be found in SieveMain.

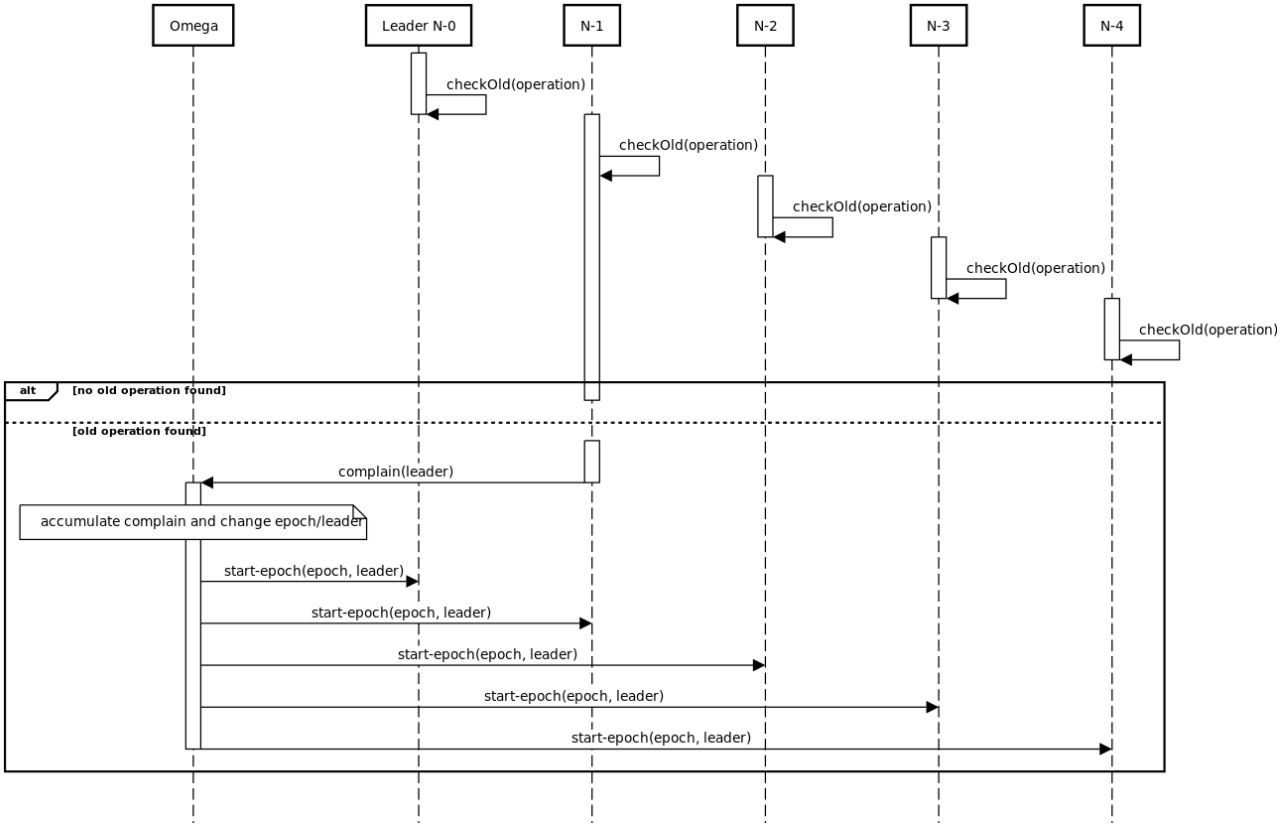
Beyond the Balancer every communication and interaction is done via MQTT.

The MQTT messages are exchanged on two different topics, a public and a private one. The former is used to communicate outside the algorithm, the latter strictly used between internal processes. This kind of communication has been designed to simulate and anticipate a future containerized implementation.

The interaction can be better visualised with the following diagrams, the first shows the steps and the message exchange for the execution of the protocol from start to finish, regarding the execution of the operation received from the client; the other diagram shows the interaction between all SieveNodes and Omega.



3. Sequence diagram showing the operation flow from client to system and back to client (back to client only if the consensus is obtained, otherwise is discarded and a timeout is produced)



4. Sequence diagram showing the interaction between Omega and SieveNodes. An operation is 'old' when the delta between now and the receiving time is greater than a predefined value.

Implementation Details

One detail that immediately stands out is the use of `args` as method parameter for most classes. This choice has been made to make the most of Python flexibility and its dictionary data structure. In this way the core fields/parameters needed by the protocol are used and sent via message, but can be expanded in the future without any difficulty.

In the MQTT messages a token is included. The token has been introduced to discriminate equals calls coming from different clients. For example client1 and client2 send “SELECT * FROM table” and they expect two separate responses.

Speaking of MQTT, an online broker has been choosed to avoid the local installation of MQTT server.

Self-assessment

The main project goal is to implement the Sieve protocol based on the study by IBM.

To this end we are satisfied with the implemented software, because even if the project isn't running on multiple servers (as discussed in the Requirement Analysis part), but in different threads, by following the protocol constraints we obtain the expected results even with different executions and simulations.

To attest this, we have also introduced unit tests to easily verify the correct execution.

Deployment Instructions

Software needed:

- Python interpreter (release 3 or more)
- pip, the package installer for Python

With that installed, we need to:

1. enter in the root directory of the project.
2. create the Virtual Environment *venv* in '`sieve_mqtt`' with '`python3 -m venv ./`'
3. start the venv with '`source venv/bin/activate`' on POSIX or '`venv\Scripts\activate.bat`' on Windows.
4. install all the project dependencies via pip with '`python3 -m pip install -r requirements.txt`'

After that last step, everything is installed and ready to use.

Usage Examples

The software usage is very straightforward once that we are in the venv, in the project root folder.

We need to execute the following commands:

- '`python3 src/cli.py -f 0 -c 5 -r 1`'
  to execute the Sieve protocol; cli.py takes three arguments: `-f` number of faulty nodes, `-c` number of correct nodes, `-r` number of requests to send to nodes
- '`python3 src/chart.py`'
  to show the system state and the processes load of requests

The cli.py starts all the system processes and executes the operation.
The execution can be followed with the prints in the command line in a simple but comprehensive table.

Each table row shows the single Sieve process state, reporting:

- The process ID (surrounded by `>process_id<` if the current process is leader)
- If the process is a faulty process
- The operation requested by the client, whit 'RESPONSE' as prefix when the algorithm is capable to reply with the solution
- The communication token
- The type of message currently managed by the process (INVOKE, EXECUTE, APPROVE, ORDER, NEW-SIEVE-CONFIG)

Additionally is possible to execute unit tests from the same position using

'`python3 -m pytest`' (avg execute time 5')

Conclusion

The project goal was to develop a working example of software exploiting the Sieve protocol to manage Byzantine faulty cases.

For this purpose we have built a multi-threaded software system in Python language with the MQTT protocol for information exchange and charts for data visualisation.

At the end of the process, the implemented software is capable of managing faulty “replicas” as expected, following the protocol directives.

Future works

With this solid foundation, there is the possibility to extend the project.

There are multiple options:

- Deploy the system on real distributed infrastructure using technologies such as Docker
- Add a web interface
- Introduce REST protocol to manage web calls

What did we learned

Working on the project, in the first place we gained further experience dealing with research papers in depth; working with non-exhaustive information and researching related works as books or other papers.

We also had the opportunity to deeply explore and work “hands on” with a Byzantine faulty system, building and then trying to break it.

References

Cachin, C., Schubert, S., & Vukolić, M. (2016, March 23). *Non-determinism in Byzantine Fault-Tolerant Replication*. arXiv. https://arxiv.org/abs/1603.07351