Study and Implementation of the Sieve Protocol in BFT

Samuele De Tuglie
samuele.detuglie@studio.unibo.it

Emanuele Artegiani
emanuele.artegiani@studio.unibo.it

Pablo Sebastian Vargas Grateron
pablo.vargasgrateron@studio.unibo.it

March 20, 2024

This project aims to create a simulation of the Sieve protocol for Byzantine Fault Tolerance (BFT). Establishing a resilient and dependable replicated state machine (RSM) that can tolerate Byzantine faults in a distributed network was the main goal. During the simulation process, it is important to guarantee scalability and effective resource management. Furthermore, we offer a basic client that can communicate with the RSM by performing a small set of actions.

To evaluate the resilience of the protocol, Byzantine actors are included in the simulation, which comprises numerous nodes, each of which represents a participant in the RSM. Identifying and alleviating Byzantine faults by putting the Sieve protocol into practice will protect the distributed system's integrity and consistency even when there are malicious nodes present.

In order to prevent Byzantine failures, the Sieve protocol requires at least $3f + 1$ nodes, where f is the number of faulty ones.

The simulation's outcomes show how well the Sieve process works to achieve BFT. This project offers a basic, user-friendly tool for experimentation and analysis, advancing our understanding of and ability to apply Byzantine fault-tolerant protocols in distributed systems.

1 Goals/Requirements

1.1 Project Goals

The primary goals of this project include:

- **Implementation of Sieve Protocol**

  To improve the resilience of distributed systems, create a working simulation of the Sieve protocol with a focus on BFT.

- **Use of Consensus**

  Create an RSM, that acts as a server, wherein consensus is used to make decisions.

- **Concurrent Programming**

  Utilize numerous threads to run several routines concurrently.

- **Docker Integration**

  For containerization, use Docker to guarantee scalable deployment and effective resource management during the simulation process.

- **Graphical User Interface (GUI)**

  Provide a user-friendly graphical user interface (GUI) that offers centralized control over the simulation. Due to the project's simulation goal, it ought to have a *Admin* section and a *Client* section in the same GUI.

- **Scalability Analysis**

  Assess the system's scalability under varying conditions to understand the protocol's effectiveness in maintaining fault tolerance.

- **Automated Testing**

  Include automated tests to verify that the functionalities are correct.

1.2 Project requirements

The requirements of this project include:

- **Architectural Requirements**

  Internal communication among RSM nodes and between them and clients will be based on the UDP protocol.

- **Simulation Requirements**

  The Sieve protocol can manage f Byzantine nodes out a total of $3f + 1$. To simulate a Byzantine behavior we will set f faulty nodes artificially.

- **Functionalities**

  A basic key-value store database is the primary feature of the RSM. Clients have the ability to add, modify, and get values in the database. Since the project's main goal is to achieve the BFT via the Sieve protocol, just these bare minimum operations will be carried out.

1.3 Expected Outcomes

Upon completion, the project is expected to yield the following principal outcomes:

- **Functional Simulation**

  A fully functional simulation of the Sieve protocol with integrated BFT mechanisms and consensus.

- **Dockerized Deployment**

  Docker containers are configured for each node, facilitating easy deployment, scalability, and resource management.

- **Intuitive GUI**

  A user-friendly GUI lets users interact with the RSM.

1.4 Use Case Diagram

The use case diagram 1 illustrates the interactions and functionalities of the Sieve protocol simulation. These visual representations serve as valuable references for understanding the roles and responsibilities of each entity in the system.

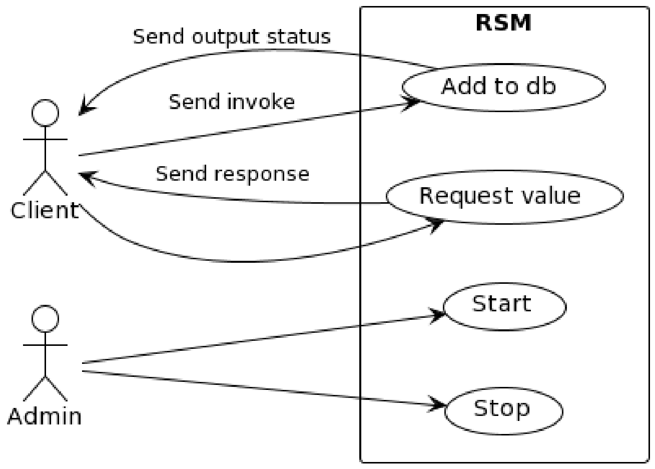


Figure 1: Use case diagram representing the interaction between admin/client and RSM.

1.5 Scenarios

Users are expected to interact with the project through the GUI, employing some functionalities to manage and interact with the Sieve protocol simulation. The following scenarios outline how and why users would engage with the system:

1.5.1 Admin

1. **Start and Stop Docker Containers**
   - *How:* Admins can initiate and terminate Docker containers through the GUI.
   - *Why:* This functionality provides admins with control over the simulation life-cycle, allowing them to start and stop the RSM.

1.5.2 Client

1. **Add items to database**
   - *How:* Users can input and send data through the GUI to be added to the RSM database.
   - *Why:* This scenario enables users to dynamically influence the data synchronized among the nodes, simulating real-world scenarios where key updates or additions occur.
2. **Request item from database**
   - *How:* Users can input the key of the item requested in the GUI.
   - *Why:* This scenario enables users to dynamically request the value associated with the key inserted, simulating real-world scenarios.
3. **Check Incoming Messages**
   - *How:* Users can use the GUI to monitor incoming messages to each node in real-time.
   - *Why:* To make it easier to spot unusual behavior and abnormalities, this feature enables users to view the distributed system's output messages as well as log messages that it generates.

These scenarios collectively empower users to interact with the project in a controlled, fostering experimentation, analysis, and a deeper understanding of Byzantine fault-tolerant protocols in distributed systems.

1.6 Self-assessment policy

- **Assessment of Software Quality**

  The quality of the produced software will be assessed through the following criteria:

  - *Automated Testing:* The software includes a comprehensive suite of automated tests implemented using the Python *unittest* framework. These tests cover various aspects of the codebase, ensuring functionality, reliability, and adherence to specifications.
  - *Code Maintainability:* The project code follows best practices for readability, maintainability, and documentation. The creation of modular code with thorough comments and documentation makes it easy to maintain and comprehend due to its breakdown into related components.
  - *Scalability and Performance:* The software undergoes testing for scalability and performance under various conditions, ensuring it meets the specified requirements and performs well in different scenarios.
  - *Reliability:* Measured in terms of fault tolerance.
  - *Communication Security:* All the communications are encrypted to enhance security.

- **Assessment of Project Outcomes' Effectiveness**

  The effectiveness of the project outcomes will be evaluated based on the following criteria:

  - *Protocol Functionality:* The successful implementation and functionality of the Sieve protocol in Byzantine fault-tolerant scenarios will be a key factor in assessing the project's effectiveness.
  - *User Interaction:* The usability of the GUI and its ability to provide users with intuitive interaction with the RSM.
  - *Simulation Results:* The project's effectiveness will be gauged by the analysis of simulation results, including the detection and mitigation of Byzantine faults, scalability insights, and overall system performance.
  - *Documentation:* The clarity and completeness of project documentation, including user guides, code documentation, and system architecture, contribute to the overall assessment of project outcomes.

2 Background

To comprehend the essence and rationale of this project, it is imperative to explore its theoretical foundation. At the heart of this initiative lies the implementation of the Sieve protocol, a cutting-edge approach to achieving BFT expounded in the referenced paper [1].

State-machine replication enhances application resilience by executing it across multiple independent components, employing BFT to ensure consistency. BFT enables correct outputs with a majority of processes being correct, even in the presence of adversarial behavior. However, traditional replication assumes determinism, posing challenges for non-deterministic applications like those involving cryptography. The work introduces two protocols: *Sieve*, which detects and handles minor divergences among processes, and Mastercrypt, providing cryptographic security for random functions in master-slave replication. These protocols aim to address non-determinism in BFT replication, crucial for applications like permissioned blockchains.

When the application cannot be altered, modular replicated execution becomes necessary, often the practical scenario. Two integration approaches exist: *order-then-execute* orders operations before independent execution, communicating results via atomic broadcast, ensuring deterministic steps. Alternatively, *execute-then-order* executes operations speculatively first, then agrees on the output, possibly rolling back diverging results. Protocol Sieve, introduced here, utilizes speculative execution, adhering to execute-then-order. It's tailored for occasionally non-deterministic applications, offering a modular solution for replicating them within a BFT system.

The system comprises distributed processes[1] providing a fault-tolerant service via service replication. Processes communicate via broadcast, executing requests in the same order. Non-faulty processes are termed correct, while faulty processes may crash or exhibit Byzantine faults. Protocols are presented modularly, using event-based notation, with processes defined by interfaces and properties. Authenticated point-to-point communication ensures message integrity via shared secret keys. The system is partially synchronous, allowing unbounded message delays until eventual synchronization, representing real-world network scenarios.

In a broadcast primitive involving n processes, messages are broadcast and delivered upon agreement. The atomic broadcast also tackles the consensus problem, with a variant ensuring the delivery of messages satisfying an external validity condition. The definition of Byzantine atomic broadcast with external validity involves input and output events, governed by a validation predicate ensuring correct message delivery based on local computation by each process.

An RSM executes operations and maintains a consistent state across correct processes. Defined by input and output events, it guarantees agreement, correctness, and termination properties. The standard implementation relies on the atomic broadcast to disseminate requests, ensuring all processes start from the same initial state and execute operations in the delivered order. If operations are deterministic, correct processes'

[1]Processes and nodes in this paper are used equivalently and with the same meaning.

states remain consistent.

In the context of atomic broadcast implementations, synchrony assumptions or randomization are often necessary. An eventual leader-detector oracle is a common weak timing assumption, informing processes of a correct leader to facilitate protocol progress. For Byzantine processes, a leader detector can be implemented from a failure detector, albeit with more complexity due to arbitrary behavior. Processes employ a "trust, but verify" approach to monitor leader behavior, initiating leader switches upon detection of misbehavior. Byzantine leader detectors and epoch-change primitives ensure trust, agreement, and resistance to coups in leader elections, while also ensuring eventual leadership among correct processes. Cryptographic functionalities, such as hash functions and digital signature schemes, are modeled as ideal, deterministic functionalities implemented by distributed oracles, ensuring collision resistance and signature verification properties.

Protocol Sieve operates within a Byzantine atomic broadcast framework with weak external validity and employs a *Sieve-leader* for executing non-deterministic operations. The leader is elected using a Byzantine epoch-change abstraction, ensuring configurations progress with monotonically increasing epoch numbers. Each configuration has its Sieve-leader, to whom all operations are sent via an INVOKE message. Operations are executed speculatively, and the leader confirms them upon receiving 2f + 1 APPROVE messages from distinct processes. The confirmed operation is then broadcasted for commitment. Sieve ensures consensus on leader changes via atomic broadcast, preventing denial-of-service attacks and ensuring liveness. Sieve's design mitigates uncoordinated speculative request execution to prevent contention among self-proclaimed leaders, crucial for maintaining liveness in BFT protocols. While a faulty leader could disrupt liveness, this is consistent with other leader-based BFT protocols. The protocol's implementation details assume authenticated, unforgeable, FIFO-ordered point-to-point messages among correct processes, with unique invoked operations across all processes.

Optimizations Rollback and state transfer in Protocol Sieve ensure efficient recovery of application states post-operation. By utilizing rollback and state transfer primitives, processes can efficiently recover correct states from other clients, minimizing overhead. Additionally, synchronization with PBFT-based atomic broadcast further enhances efficiency, leveraging PBFT's leader and Byzantine epoch-change mechanisms for seamless integration with Sieve. These optimizations minimize message overhead while ensuring consistent state recovery and operation execution.

It should be noted that Sieve not only works with Byzantine atomic broadcast in the model of eventual synchrony but can equally well be run over randomized Byzantine consensus.

2.1 Used Frameworks / Models / Technologies

The development of this project is centered around the Python programming language, chosen for its versatility and extensive libraries that facilitate rapid application development. Specifically, the following frameworks, models, and technologies were employed:

- **Python**

  The entire project, including the Sieve protocol implementation, GUI development, and automated testing, is written in *Python*. Its readability, flexibility, and vast community support make it suitable for this endeavor.

- **PySimpleGUI**

  The GUI is constructed using *PySimpleGUI*, a Python GUI framework. Its simplicity and ease of use align with the project's goal of providing an intuitive interface for users to interact with the simulation.

- **unittest**

  Automated testing is facilitated through the use of the *unittest* framework in Python. This allows for the systematic testing of individual components, ensuring code reliability and functionality.

- **Docker**

  Docker is utilized for containerization, providing a lightweight and portable environment for the Sieve protocol simulation. Docker-Compose is employed to manage the orchestration of multiple containers, streamlining the deployment process and ensuring consistency across different environments.

The choice of these technologies is motivated by their compatibility, community support, and ability to enhance the overall development and deployment process of the Sieve protocol simulation.

3 Requirements Analysis

The functional requirements of the application encompass all the features and services provided by the system:

- **Start and Stop Docker Container**

  Initiate and terminate the Docker container seamlessly, ensuring a controlled environment for the Sieve protocol simulation.

- **Data Management**

  Enable the transmission of data to individual nodes, facilitating dynamic adjustments to their behaviors and configurations during the simulation.

- **Shared Data Retrieval**

  Facilitate the retrieval of shared data between nodes, essential for evaluating the consensus achieved by the Sieve protocol in the presence of Byzantine faults.

- **Log Inspection**

  Provide a mechanism to inspect the log of messages from the RSM, aiding in debugging, analysis, and a comprehensive understanding of the simulation's dynamics.

3.1 Implicit Requirements

Implicit requirements, not explicitly defined above but nevertheless fundamental, include:

- **Compatibility**

  The software is designed to function on every platform where Docker and Python are runnable, ensuring compatibility with almost every modern device.

- **Software Functionality Testing**

  Various tests are embedded in the program to ensure the correct functionality of the code.

3.2 Non-Functional Requirements

Non-functional requirements express the characteristics and properties that a system must meet. The following non-functional requirements are present in the project:

- **Scalability**

  Each node is represented by a Docker container, facilitating network expansion by simply adding new nodes to the Docker Compose file.

- **Security**

  As indicated in [1], the software utilizes symmetric cryptography to encrypt and decrypt messages, ensuring the confidentiality and integrity of communication.

- **Usability**

  The software is equipped with a user-friendly dashboard to interact with the nodes, enhancing the overall user experience.

- **Maintainability**

  The code has been crafted clearly, simply, and understandable, ensuring ease of maintenance and future enhancements.

3.3 Implicit Assumptions

Some implicit assumptions are noteworthy as they influence the design of the system:

- **Ports Not Occupied**

  Each Docker container runs and exposes a port. It is essential to check if the port to use is unoccupied.

- **Administrator Presence**

  It is assumed that there is a system administrator familiar with the IP addresses and ports made available by each node for the protocol implementation. In our simulation, this role is fulfilled using the *client_config.py* file.

- **Number of Nodes**

  Two potential issues may arise when a node experiences Byzantine faults. The first problem is the failure to send a message altogether. Specifically, within a distributed system to function correctly with N nodes, having f nodes suffering from Byzantine faults, consensus must be reached with $N - f$ messages. In other words, $N - f$ nodes are required for the quorum.

  The second problem arises when a Byzantine-faulty node maliciously sends different messages. In an extreme scenario, let's say that among the $N - f$ nodes that have achieved quorum, f have been sent by a Byzantine-faulty node. Even in this case, the system must operate normally, requiring $(N - f) - f$ messages to be greater than f messages sent by nodes experiencing Byzantine faults. To solve the two aforementioned problems, $(N - f) - f > f$. Therefore, in a BFT system, the total number of processes must be $3f + 1$ to tolerate f faulty nodes.

3.4 Models/Paradigms/Technologies

Among the models, paradigms, and technologies best suited to address the previously mentioned requirements are:

- **Client-Server Architectural Paradigm**

  The client communicates with the server using UDP protocol.

- **Master-Slave Model**

  Inside the server, a master-slave model is employed.

3.5 Structure / Domain Entities

To gain a comprehensive understanding of the entities requiring modeling, reference the UML class diagram depicted in Figure 2. Method and variable names defined in the reference paper [1] were retained as much as possible.

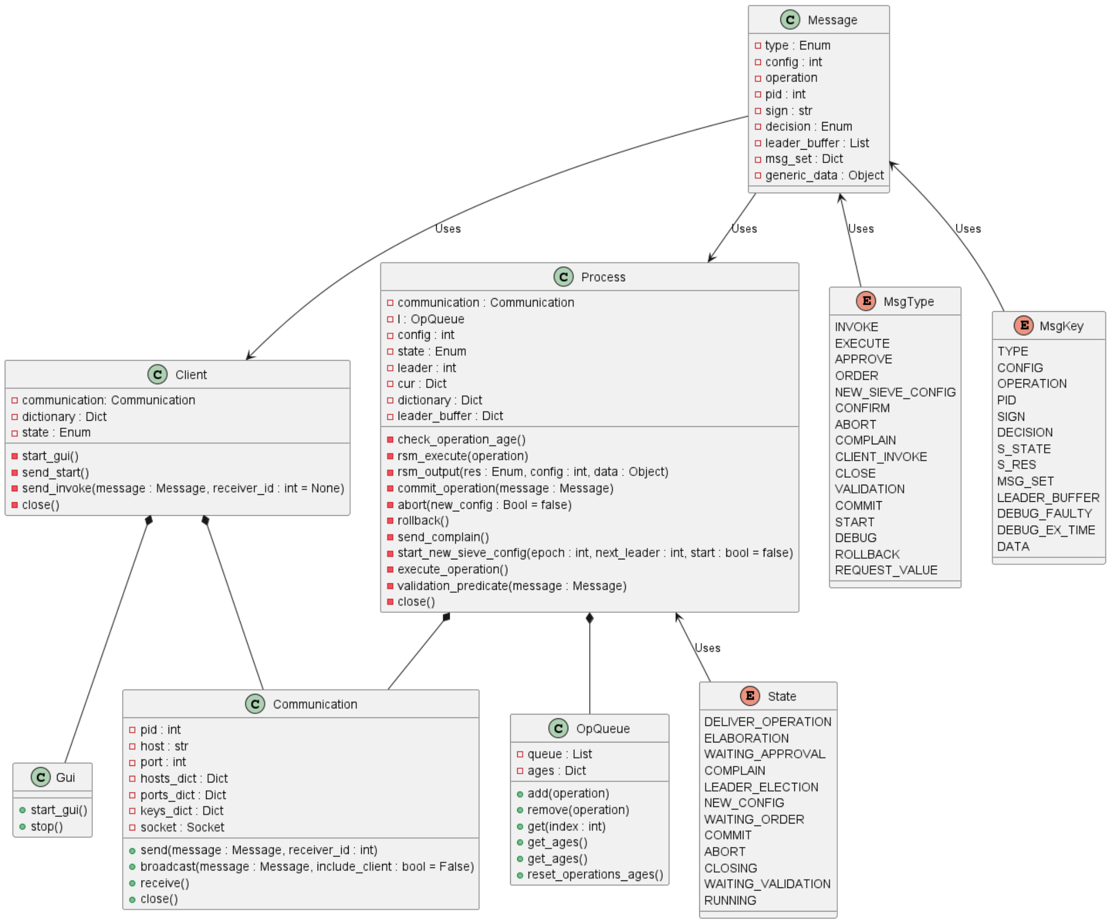


Figure 2: Class Diagram

This diagram represents the following domain entities:

- **Process:** A class representing the Sieve protocol node.
- **Message:** A data class that encapsulates the attributes of a message.
- **Client:** A class responsible for controlling interactions from the client to the server.
- **Communication:** A class designed to handle the exchange of messages.
- **Gui:** A class managing the user interface components.
- **OpQueue:** A class for queuing up operation requests initiated by the client.
- **State:** An enumeration representing the various states of the Sieve nodes.
- **MsgType:** An enumeration declaring different types of messages.

- **MsgKey:** An enumeration is used to compose the messages.

3.6 Behaviour

3.6.1 Leader node

The leader node in the Sieve Protocol exhibits the following behavior (as expressed in the figure 3):

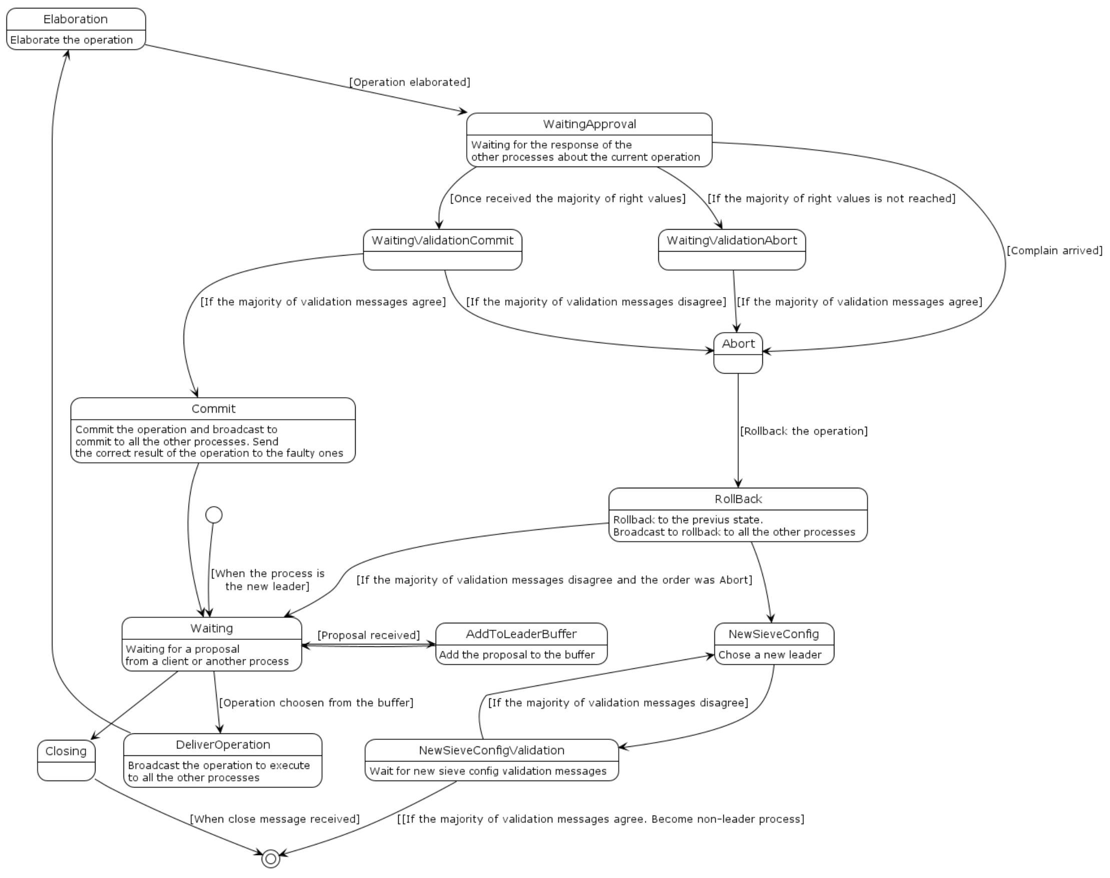

Elaboration
Elaborate the operation
[Operation elaborated]
WaitingApproval
Waiting for the response of the other processes about the current operation
[Once received the majority of right values]
[If the majority of right values is not reached]
[Complain arrived]
WaitingValidationCommit
WaitingValidationAbort
[If the majority of validation messages agree]
[If the majority of validation messages disagree]
[If the majority of validation messages agree]
Abort
Commit
Commit the operation and broadcast to commit to all the other processes. Send the correct result of the operation to the faulty ones
[Rollback the operation]
RollBack
Rollback to the previus state. Broadcast to rollback to all the other processes
[When the process is the new leader]
[If the majority of validation messages disagree and the order was Abort]
Waiting
Waiting for a proposal from a client or another process
[Proposal received]
AddToLeaderBuffer
Add the proposal to the buffer
NewSieveConfig
Chose a new leader
[Operation choosen from the buffer]
[If the majority of validation messages disagree]
Closing
DeliverOperation
Broadcast the operation to execute to all the other processes
NewSieveConfigValidation
Wait for new sieve config validation messages
[When close message received]
[[If the majority of validation messages agree. Become non-leader process]

Figure 3: State diagram of leader node

- **Waiting**

  This is the idle status, during which the leader awaits further operations to be requested from a client or another process.

- **AddToLeaderBuffer**

  During this state, the leader adds the requested operation to its buffer for processing.

- **DeliverOperation**

  When an operation is requested, the leader broadcasts the operation to be executed to all other processes in the network.

- **Elaboration**

  The leader processes the requested operation. If the leader receives a complaint message from any slave during this state, it transitions to the **Abort** state and then to **NewSieveConfig**.

- **WaitingApproval**

  After completing the operation, the leader broadcasts the local result and waits for confirmation from at least $2f + 1$ processes. Also in this state if the leader receives a complaint message from any slave during this state, it transitions to the **Abort** state and then to **NewSieveConfig**.

  - *WaitingValidationCommit*

    If the majority of processes validate the leader's result, the leader enters this state, awaiting validation of its local result from the majority of processes. If the majority validates the result, the state transitions to **Commit**; otherwise, it transitions to **Abort**.

  - *WaitingValidationAbort*

    If the majority of processes disagree with the leader's result, the leader considers aborting the operation and enters this state. It waits for validation of its local result from the majority of processes, transitioning to **Abort** upon validation.

- **Commit**

  The leader commits the operation and broadcasts the commit to all processes with the correct result and the state transitions to **Waiting**.

- **Abort**

  This state initiates the rollback operation.

- **Rollback**

  The leader reverts to the previous state and broadcasts the rollback operation to other processes. If the majority of received messages indicate disagreement with the abort, it transitions to the **Waiting** state; otherwise, it transitions to **NewSieveConfig**.

  - *NewSieveConfig*

    This operation selects a new leader.

  - *NewSieveConfigValidation*

    The leader awaits validation messages for the new configuration. If the majority of processes agree, the current leader (this process) becomes a slave and the new leader chosen becomes the effective leader; otherwise, it returns to **NewSieveConfig** to choose a new leader again.

- **Closing**

  Upon receiving a close message, the process shuts down.

3.6.2 Non-leader node

The non-leader node in the Sieve Protocol exhibits the following behavior (as expressed in the figure 4):

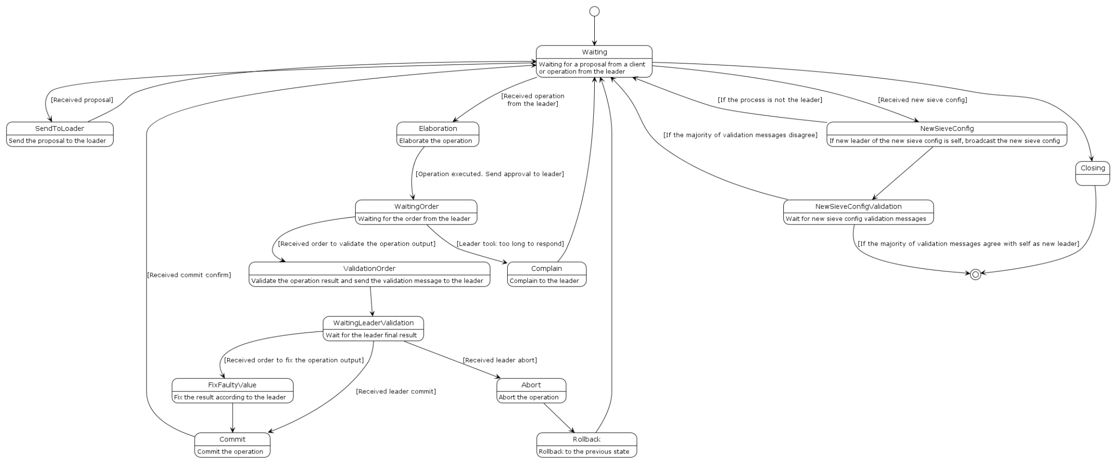

Waiting
Waiting for a proposal from a client or operation from the leader
[Received proposal]
SendToLoader
Send the proposal to the loader
[Received operation from the leader]
Elaboration
Elaborate the operation
[Operation executed. Send approval to leader]
WaitingOrder
Waiting for the order from the leader
[Received order to validate the operation output]
[Leader took too long to respond]
ValidationOrder
Validate the operation result and send the validation message to the leader
Complain
Complain to the leader
[Received commit confirm]
WaitingLeaderValidation
Wait for the leader final result
[Received order to fix the operation output]
[Received leader abort]
[Received leader commit]
FixFaultyValue
Fix the result according to the leader
Abort
Abort the operation
Commit
Commit the operation
Rollback
Rollback to the previous state
[If the process is not the leader]
[Received new sieve config]
[If the majority of validation messages disagree]
NewSieveConfig
If new leader of the new sieve config is self, broadcast the new sieve config
Closing
NewSieveConfigValidation
Wait for new sieve config validation messages
[If the majority of validation messages agree with self as new leader]

Figure 4: State diagram of non-leaders nodes

- **Waiting**

  In this state, the slave waits for proposals from a client or operations from the leader.

- **SendToLeader**

  If the process receives a proposal from a client, it forwards the proposal to the leader.

- **Elaboration**

  Upon receiving an execute operation message from the leader, the process elaborates on the operation sent by the leader.

- **WaitingOrder**

  After completing the operation, the process sends the result to the leader and awaits the next instruction. If there's a delay in the leader's response, the process transitions to the **Complain** state.

  - *Complain*

    The process sends a complaint to the leader and transitions back to the **Waiting** state.

- **ValidationOrder**

  Upon receiving the leader's results, the process validates the result, using a validation predicate, and sends a validation message.

- **WaitingLeaderValidation**

  The process waits for the final message from the leader.

  - *Commit*

    If the process receives a commit message, it commits the operation and transitions back to the **Waiting** state.

  - *FixFaultyValue*

    Upon receiving a correction message, the process updates the result and transitions to the **Commit** state.

  - *Abort*

    If the process receives an abort message, it aborts the current operation and transitions to the **Rollback** state.

- **Rollback**

  The process rolls back to the previous correct state.

- **NewSieveConfig**

  Upon receiving the new sieve configuration, if the process is the new leader, it broadcasts the new configuration to other processes; otherwise, it returns to the **Waiting** state.

- **NewSieveConfigValidation**

  The process waits for validation messages regarding the new sieve configuration. If the majority agrees, this process becomes a leader.

- **Closing**

  Upon receiving a close message, the process shuts down.

3.6.3 Network

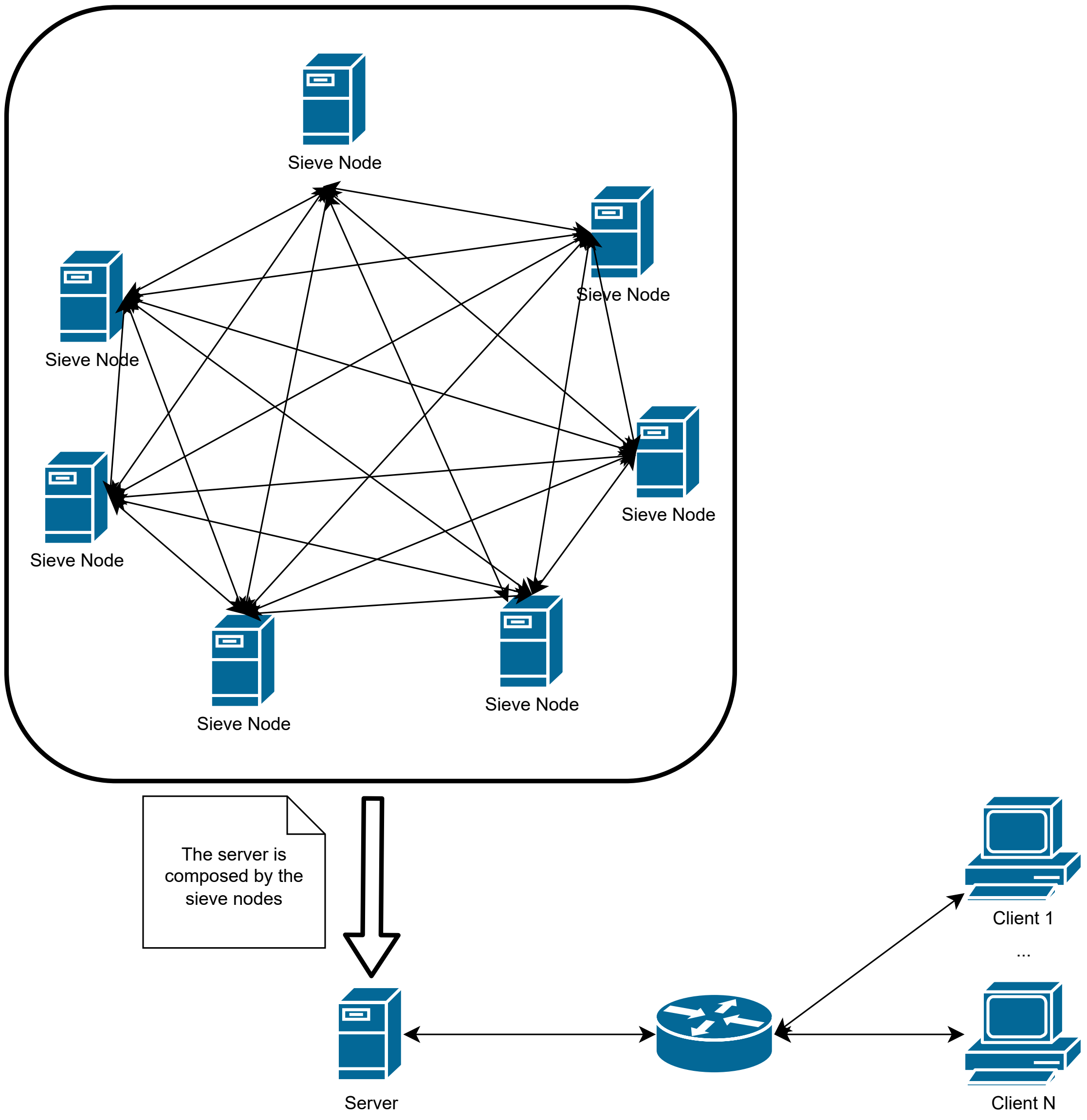


Figure 5: Logic representation of the network used in our simulation.

The behavior of all nodes in the network, while the number of nodes is indefinite, manifests as a single unified server responsible for receiving, processing requests, and sending responses to the client. This is achieved through the centralized management depicted in figures 3 and 4. Most queries in this configuration go through the leader node. Because of the RSM's nature, any slave process that receives a request is redirected to the current leader to be handled, with the exception of requests for values given keys, which can be handled by any node.

4 Implementation Details

In the development of the Sieve protocol simulation, several noteworthy implementation details have been incorporated to enhance the robustness and effectiveness of the software. The documentation for this project is crafted using *mkdocs*, providing a structured and accessible resource for understanding the intricacies of the implemented solution.

Here are some salient implementation details of the project:

- To emulate real-world scenarios and Byzantine faults, the simulation incorporates dynamic node behavior. Nodes can exhibit various forms of Byzantine behavior, such as sending conflicting messages or intentionally delaying message propagation. This dynamic simulation enriches the testing environment and ensures a more comprehensive evaluation of the Sieve protocol's effectiveness.
- Cryptographic techniques play a pivotal role in the Sieve protocol's functionality. The implementation includes the integration of cryptographic libraries to secure communication channels and validate the integrity of messages exchanged between nodes. This cryptographic layer enhances the overall security posture of the system.
- The Sieve protocol's consensus algorithm has been adapted to accommodate the dynamic nature of Byzantine faults. The algorithm dynamically adjusts its parameters to respond to changes in the network's behavior, allowing for a more adaptive and resilient consensus-building process, the total number of the nodes must be at least 3f + 1, where f is the number of faulty nodes.
- The simulation logs events and interactions at various stages of the protocol execution. This logging serves as a valuable resource for simulation analysis, enabling the identification of patterns, bottlenecks, and potential improvements in the Sieve protocol's performance.

Aiming to simplify our implementation, it was also decided:

- not to allow changes to the cluster (addition/removal of processes) while the RSM is already in execution;
- not to maintain the execution data and logs after the RSM shutdown. This implies that there are always nodes with base configurations at restart, without memory of the previous states and events.

Given the complexity of the Sieve protocol and the nuanced challenges posed by Byzantine faults, these implementation details collectively contribute to a simulation environment. The accompanying mkdocs documentation further provides a structured guide for users and developers, facilitating a smooth understanding of the software artifacts.

5 Self-assessment / Validation

It was necessary to conduct a variety of tests with varying degrees of detail and emphasis on various areas to confirm that the software developed was accurate. The following subsection contains an account of the parts that were tested using autonomous testing. Additionally, manual testing was done, in particular to evaluate the GUI and particular client-server interaction.

5.1 Sieve protocol testing

The **unittest** package, a Python framework for test-driven development and code correctness verification, was utilized to confirm in greater detail the correct functionality of the Sieve components (tests results can be found in Figure 6).

All tests created use one or more instances of the client to verify the correctness of various features. The client instance is necessary to send operations requests, as well as checking the responses and log messages received.

A system of "debug messages" was developed in order to dynamically configure internal parameters of the Sieve processes, ensuring test reproducibility.

In particular, the following tests were performed, which can be found within the file *src/test.py*:

- **Base functionalities**

  To verify that a Sieve process's primary capabilities were correct, the following tests were conducted:

  - **test_commit**: tests if the commit section is executed
  - **test_abort**: tests if the abort section is executed
  - **test_rollback**: tests if the rollback section is executed
  - **test_complain**: tests if a complain occurs
  - **test_new_sieve_config_after_abort**: tests if a new sieve config starts after an abort
  - **test_new_sieve_config_after_complain**: tests if a new sieve config starts after a complaint
  - **test_close**: tests if the Sieve processes stop when reaching the close message

- **Functionalities combinations**

  At this point are listed the tests for compound and serialized operations:

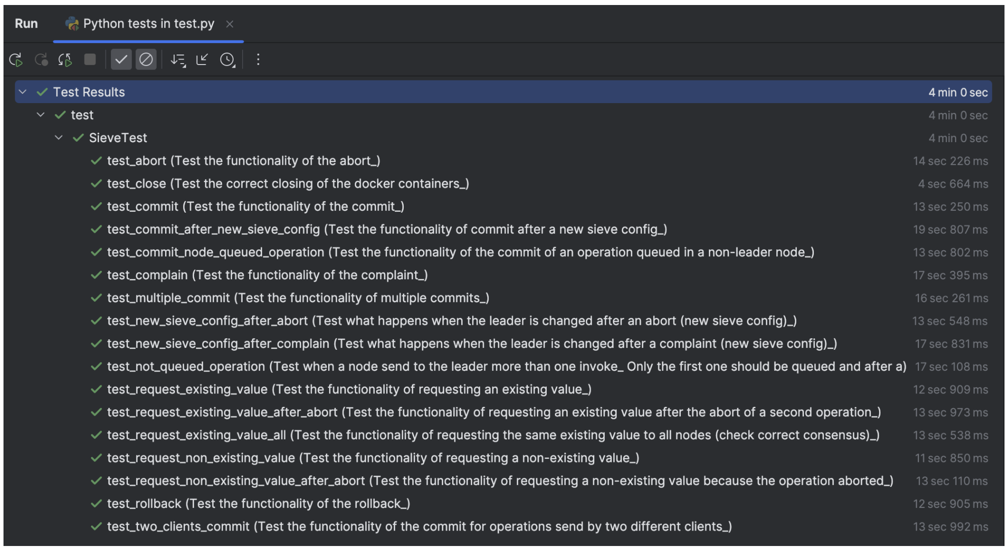
  - **test_multiple_commit**: tests if multiple commits are executed correctly
  - **test_commit_after_new_sieve_config**: tests if the commit section is executed after a new sieve config, so if a new leader executes it
  - **test_not_queued_operation**: tests if, after a new sieve config, the new leader has an empty operation queue if the previous leader has it in turn

  - **test_commit_node_queued_operation**: tests if, after a new sieve config, the new leader has an operation queue with the operations not executed by the previous leader
  - **test_two_clients_commit**: tests the commit of operations sent by more than a single client

- **Values requests**

  The tests below were used to check if the functionalities offered to the client are correct:

  - **test_request_non_existing_value**: tests the case when a client requests a value associated with a non-existing key
  - **test_request_existing_value**: tests the case when a client requests a value associated with an existing key
  - **test_request_non_existing_value_after_abort**: tests the case when a client requests a value associated with a non-existing key because the abortion of the relative operation which should have added the key-value
  - **test_request_existing_value_after_abort**: tests the case when a client requests a value associated with an existing key, which must exist also if another operation aborts
  - **test_request_existing_value_all**: tests the proper functioning of consensus



```
✓ Test Results                                                                                                                         4 min 0 sec
  ✓ test                                                                                                                               4 min 0 sec
    ✓ SieveTest                                                                                                                        4 min 0 sec
      ✓ test_abort (Test the functionality of the abort_)                                                                              14 sec 226 ms
      ✓ test_close (Test the correct closing of the docker containers_)                                                                4 sec 664 ms
      ✓ test_commit (Test the functionality of the commit_)                                                                            13 sec 250 ms
      ✓ test_commit_after_new_sieve_config (Test the functionality of commit after a new sieve config_)                                19 sec 807 ms
      ✓ test_commit_node_queued_operation (Test the functionality of the commit of an operation queued in a non-leader node_)          13 sec 802 ms
      ✓ test_complain (Test the functionality of the complaint_)                                                                       17 sec 395 ms
      ✓ test_multiple_commit (Test the functionality of multiple commits_)                                                             16 sec 261 ms
      ✓ test_new_sieve_config_after_abort (Test what happens when the leader is changed after an abort (new sieve config)_)            13 sec 548 ms
      ✓ test_new_sieve_config_after_complain (Test what happens when the leader is changed after a complaint (new sieve config)_)      17 sec 831 ms
      ✓ test_not_queued_operation (Test when a node send to the leader more than one invoke_ Only the first one should be queued and after a) 17 sec 108 ms
      ✓ test_request_existing_value (Test the functionality of requesting an existing value_)                                          12 sec 909 ms
      ✓ test_request_existing_value_after_abort (Test the functionality of requesting an existing value after the abort of a second operation_) 13 sec 973 ms
      ✓ test_request_existing_value_all (Test the functionality of requesting the same existing value to all nodes (check correct consensus)_) 13 sec 538 ms
      ✓ test_request_non_existing_value (Test the functionality of requesting a non-existing value_)                                   11 sec 850 ms
      ✓ test_request_non_existing_value_after_abort (Test the functionality of requesting a non-existing value because the operation aborted_) 13 sec 110 ms
      ✓ test_rollback (Test the functionality of the rollback_)                                                                        12 sec 905 ms
      ✓ test_two_clients_commit (Test the functionality of the commit for operations send by two different clients_)                  13 sec 992 ms
```

Figure 6: Screenshot of executed tests results.

6 Deployment Instructions

To ensure a smooth deployment of the software in a clean environment, follow these steps:

6.1 Prerequisites

Ensure that the following prerequisites are installed on the target system:

- **Python 3.10**: Install Python 3.10 on the system. You can download the target version from the official Python website.
- **pip**: Ensure that the Python package installer, pip, is available. It is typically installed automatically with Python. If not, install it using the appropriate package manager for your system.
- **Docker**: Install Docker to enable containerization. Follow the instructions for your operating system on the official Docker website.

6.2 Deployment Steps

Follow these steps to deploy and launch the software:

1. **Clone the Repository**

   Clone the project repository to the target system using Git:

   ```
   git clone https://dvcs.apice.unibo.it/pika-lab/courses/ds/projects/
   ds-project-detuglie-artegiani-grateron-ay2324.git

   cd ds-project-detuglie-artegiani-grateron-ay2324
   ```

2. **Install Python Dependencies**

   Install the required Python dependencies listed in the *requirements.txt* file:

   ```
   pip install -r requirements.txt
   ```

3. **Build Docker Image**

   Build the Docker image for the project. Ensure that Docker is running, then execute the following command from the project root:

   ```
   docker build -t sieve-process .
   ```

4. **Run The Application**

   Run the application with the following command:

   ```
   python3 ./src/client.py
   ```

These steps assume a clean environment and provide the necessary instructions for setting up and deploying the software with minimal configuration. These details are also available in the project's *README*.

If the client cannot communicate with the containers it can be required to add them in */etc/hosts* file using the following sintax:

```
127.0.0.1 process1
...
127.0.0.1 processN
```

6.3 Build documentation

Follow these steps to build the documentation as a web page:

1. **Install MkDocs**

   Install MkDocs and his dependency using the following command:

   ```
   pip install -r ./mkdocs/requirements.txt
   ```

2. **Build the documentation**

   Generate web page:

   ```
   mkdocs build
   ```

3. **Consult documentation**

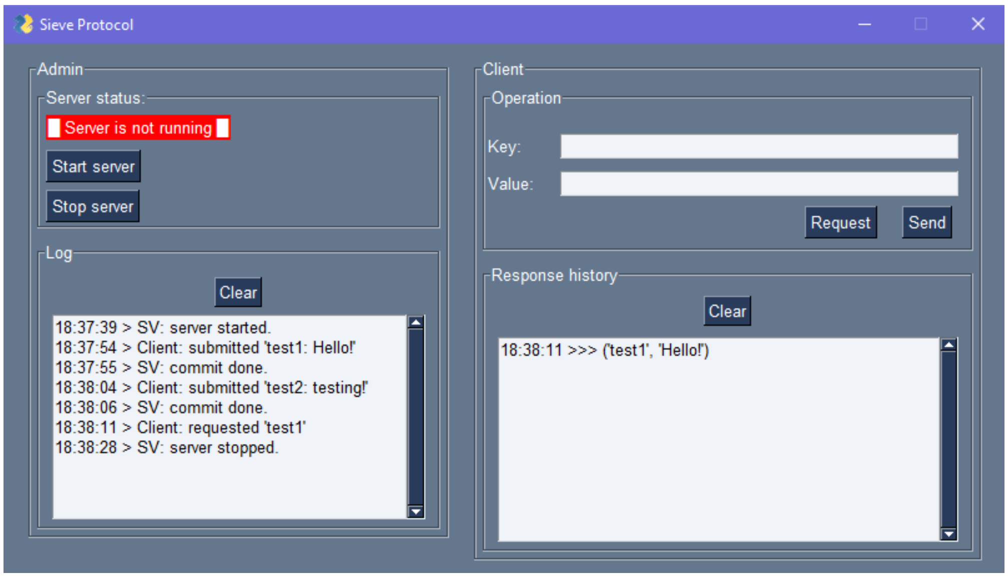
   It is possible to self-host and view the documentation with the following command:

   ```
   mkdocs serve
   ```

For more information check the MkDocs documentation.

7 Usage Examples

Once the *src/client.py* script is executed, it will launch the following GUI, as illustrated in Figure 7.



```
18:37:39 > SV: server started.
18:37:54 > Client: submitted 'test1: Hello!'
18:37:55 > SV: commit done.
18:38:04 > Client: submitted 'test2: testing!'
18:38:06 > SV: commit done.
18:38:11 > Client: requested 'test1'
18:38:28 > SV: server stopped.
```

```
18:38:11 >>> ('test1', 'Hello!')
```

Figure 7: Interface for controlling the application

To begin, click the "Start Server" button to initiate the server. The "Log" field provides real-time updates on the status of the operations.

After the server has been started, operations, to update the shared dictionary, can be sent for execution by entering a key-value pair and clicking the "Send" button. Alternatively, clicking the "Request" button will query the value associated with the entered key. The responses to these actions are displayed in the "Response history" section. The server can be stopped at any time by clicking the "Stop Server" button.

This interface provides a straightforward means of interacting with the application, allowing users to manage operations and monitor responses conveniently.

8 Conclusions

In conclusion, this project not only achieved the successful implementation and simulation of the Sieve protocol but also made us practice the theoretical aspects we learned during the course. Summarizing the activities we did, we started studying the Sieve protocol from the paper [1] and the connected documentation, and then we analyzed and extrapolated the implementation aspects we used for the design phase. After the design phase, we focused on the implementation and deployment. We opted for the use of *Docker* to create a cluster of Sieve processes and the use of *Python DGRAM sockets* for the communication. The group tried as much as possible to comply with the test-driven development approach using *unittest* framework. In the first part testing was concentrated on base functionalities, then on routines and simulation of some more complex scenarios.

8.1 Future Works

In this work, we have developed a representation of the Sieve Protocol in Python, focusing on its behavior and interaction within a distributed network. However, several aspects have not been addressed or could be further improved:

- **Greater Fault Tolerance**

  Although the protocol includes mechanisms for leader election and failure recovery, further analysis and testing are needed to assess its robustness in various failure scenarios, such as denial-of-service attacks.

- **Database Integration**

  Currently, our implementation relies on a shared dictionary for storing data. In future work, we aim to integrate a more robust data storage solution, such as a distributed database or a blockchain, to enhance data persistence and reliability.

- **Performance improvement**

  Currently, each Sieve process does its tasks using three threads without any special optimizations. Further research could benefit from using additional thread-specific data types and instructions to improve performance.

- **User Interface Enhancements**

  While we have implemented a basic client interface for interacting with the network, there is room for improvement in usability and functionality. Future work could involve refining the user interface to provide users with more intuitive controls and informative feedback.

- **Real-scenario implementation**

  For the time being, this project only offers a localhost simulation. However, future work could expand this implementation of the Sieve protocol to a real-world setting, i.e., a distributed system that is available via the internet.

Addressing these areas of improvement would not only enhance the functionality and performance of the Sieve Protocol implementation but also contribute to the broader understanding of distributed consensus protocols.

8.2 What Did We Learn

Throughout the development of this project, several key learnings were acquired:

- **Docker Utilization**

  The project provided hands-on experience in using Docker for containerization. Understanding how to create, deploy, and manage Docker containers proved to be crucial for achieving a scalable and consistent simulation environment.

- **Inter-Container Communication**

  The project involved making Docker containers communicate with each other using Python sockets. This learning extended to managing communication channels between distributed nodes, a valuable skill in the context of distributed systems.

- **PySimpleGUI for GUI Development**

  Integration of PySimpleGUI for creating the GUI enhanced the understanding of developing user-friendly interfaces in Python. This knowledge is transferable to various application domains requiring interactive user interactions.

- **Automated Testing with unittest**

  The use of the unittest framework in Python for automated testing contributed to the development of robust and reliable code. Implementing test cases and ensuring code coverage became integral to maintaining software quality.

- **Processes Synchronization**

  It was required to learn how to synchronize the states and actions of individual processes in order to synchronize the various stages of the Sieve protocol.

- **Consistency and Consensus**

  Understanding consistency and consensus is necessary since putting the Sieve protocol into practice.

- **Byzantine Fault Tolerance Concepts**

  Working on the Sieve protocol simulation deepened the understanding of Byzantine fault tolerance concepts in distributed systems. Practical implementation and experimentation illuminated the challenges and solutions associated with mitigating malicious behaviors.

These learnings collectively contribute to a well-rounded skill set, encompassing containerization, inter-container communication, GUI development, automated testing, and distributed systems concepts. The insights gained are not only applicable to this project but also lay a foundation for addressing challenges in diverse software development scenarios.

References

[1] Marko Vukolic´ Christian Cachin, Simon Schubert. Non-determinism in byzantine fault-tolerant replication. *IBM Research - Zurich*, 03 2016.