Final Report - BFT2f implementation

Benedetta Pacilli
benedetta.pacilli@studio.unibo.it

Valentina Pieri
valentina.pieri5@studio.unibo.com

November 13, 2024

This report describes the BFT2f [2] algorithm implementation as a Python library, building from the popular CastroLiskov PBFT [1]. The main focus is on investigating the system's behavior beyond f failures, covering the, often ignored, spectrum between full correctness and arbitrary failure. Similar to PBFT, BFT2f provides liveness and consistency when up to f replicas fail. In scenarios where f $<$ failures $\leq$ 2f, BFT2f confines malicious servers to specific consistency violations, providing a nuanced approach to system behavior. This report presents a practical implementation of the BFT2f algorithm, designed to enhance reliability and security in distributed systems. This report describes the design, implementation, testing, and validation that were performed in this project and provides a general view of the application-level facts on Byzantine Fault Tolerance in distributed environments.

1 Goals/requirements

The main goal is to fully understand and correctly implement the BFT2f algorithm. More specific objectives are described below:

- **Understanding of BFT2f Algorithm**: A preliminary study of the BFT2f algorithm to gain a profound understanding of its design principles and functionality.
- **Algorithm Implementation**: Development of a Python library to implement the BFT2f algorithm, adhering to the specifications outlined in the algorithm paper [2].
- **Functionality Demonstrations**: Illustrate the algorithm functioning in both failure and success scenarios to validate its correctness and effectiveness.
- **Automated Testing**: Implementation of automated tests to ensure the reliability and stability of the BFT2f algorithm under various conditions.

1.1 Scenarios

The BFT2f algorithm ensures system liveness and consistency under Byzantine failure conditions. The following scenarios sum up how the system is expected to behave:

- **Up to f replicas fail**: The system maintains full liveness and consistency, ensuring that all operations are processed correctly, and no arbitrary failure occurs.
- **The number of faulty replicas exceeds f but is less than or equal to $2f$**: The system restricts the impact of malicious nodes, confining them to limited consistency violations while preventing total failure.
- **More than $2f$ replicas fail**: The system's behavior may be compromised, requiring external intervention to restore normal operation, as BFT2f does not guarantee liveness or correctness beyond $2f$ failures.

1.2 Q/A

The following Q/A aims at proving a better and more detailed explanation of the algorithm performance and expected behavior.

- **Q: What happens when exactly f replicas fail simultaneously? A:** The system continues to operate normally, maintaining both liveness and consistency. All operations are processed as expected, and there are no inconsistencies.
- **Q: How does the system respond if the number of faulty replicas exceeds f but remains within $2f$?**

A: BFT2f limits the behavior of malicious replicas, ensuring that critical operations remain unaffected while preventing any unauthorized or unverified actions from being processed.

- **Q: Can the system recover from a scenario where faulty replicas exceed *2f*?**

  A: Beyond *2f* failures, BFT2f cannot guarantee recovery or consistency. The system may require manual intervention, such as shutting down the system, or external mechanisms to regain stability and restore proper functionality.

- **Q: How does BFT2f handle network partitions or split-brain scenarios?**

  A: In network partition scenarios, as long as the majority of nodes remain connected within any partition, the system continues to operate. In case there is no major partition, the system pauses new transactions until the network returns to a state where all nodes, or most of them, are able to communicate with each other again.

1.3 Expected Outcomes

- A fully operational Python library implementing the BFT2f algorithm.
- Verification of the algorithm's ability to maintain liveness and consistency up to f failures and its ability to constrain malicious behavior up to *2f* failures.
- Demonstration of the system's limitations in the presence of more than *2f* failures.
- Comprehensive test coverage to validate the system's reliability and robustness.

2 Background

The following subsections present and explain concepts necessary to understand the BFT2f algorithm.

2.1 Theoretical Aspects

First of all, the challenge of achieving consensus in a distributed system, particularly in the presence of Byzantine faults. The BFT2f algorithm is built on the foundations of *Byzantine Fault Tolerance (BFT)*, crucial in distributed systems theory. The algorithm extends the traditional PBFT algorithm by introducing mechanisms to tolerate more than f faulty replicas, ensuring that the system can handle up to $2f$ failures.

In traditional BFT systems, the system can maintain liveness and consistency while up to f replicas fail. However, BFT2f introduces a spectrum of fault tolerance between f and $2f$ faulty replicas. When the number of faulty replicas exceeds f but remains less than or equal to $2f$, the system limits the impact of these failures, while still preventing total system collapse. This allows the system to degrade gracefully, maintaining a certain degree of service availability even when it cannot guarantee full correctness.

In the context of the *CAP Theorem*, which states that a distributed system can only achieve two out of the three properties of *Consistency*, *Availability*, and *Partition Tolerance*, BFT2f allows for more flexible trade-offs between consistency and availability. When faulty replicas exceed f but remain under $2f$, the system maintains availability but may sacrifice some degree of consistency.

Replication and *consistency* are two other key concepts of this project, both important in ensuring that distributed systems remain operational under fault conditions. By replicating state across multiple replicas, BFT2f ensures that the system can recover from failures without losing critical information. This replication, combined with Byzantine fault-tolerant consensus, plays a crucial role in maintaining system integrity.

2.2 Used Frameworks/Models/Technologies

As previously mentioned, we developed a Python-based implementation of the BFT2f algorithm. This specific programming language was chosen for its simplicity and extensive support for network programming.

The project required the use of several key technologies and frameworks:

- **Socket Programming**: The system relies on socket communication for message exchange between replicas and clients. We used Python's built-in `socket` library to implement the low-level network communication required for sending and receiving messages in the consensus process.
- **Asynchronous Programming**: Since the system needed to handle multiple client requests concurrently, asynchronous programming has to be employed. Python's threading and locking mechanisms were used to manage concurrent operations,

ensuring that replicas could process multiple messages simultaneously without introducing race conditions.

- **Hash Chain for Operation Integrity**: To maintain the integrity of operations, we implemented a hash chain that tracks and verifies the sequence of operations executed by the system. By doing so, we ensure that the history of operations is verifiable and consistent across replicas, even under Byzantine faults.

- **Unit Testing**: Python's `unittest` framework was used to create automated tests for validating the correctness of the BFT2f implementation. Moreover, by simulating different fault scenarios, we were able to verify that the system behaves correctly under various conditions.

3 Requirement Analysis

The requirements of the BFT2f system revolve around achieving fault tolerance, maintaining consistency, and ensuring liveness in a distributed system, even in the presence of Byzantine faults.

3.1 Implicit Requirements and Assumptions

- **Message Integrity**: The system must ensure that messages exchanged between replicas and clients are secure and not tampered with. Although not explicitly mentioned, message integrity is crucial for the correct operation of the consensus protocol. Without it, malicious replicas could alter messages, leading to system-wide failures.
- **Fault Detection and Tolerance**: Another implicit mechanism is the ability to detect faulty replicas and tolerate their behavior. This requires mechanisms to differentiate between correct and faulty replicas during the consensus process.
- **Network Configuration**: The system implicitly requires the network configuration to be stable, with all clients and replicas reachable and able to communicate with each other.
- **Sufficient Number of Replicas**: The BFT2f algorithm assumes that the system will run with at least $3f + 1$ replicas to tolerate up to f Byzantine faults.

3.2 Non-Functional Requirements

- **Scalability**: The BFT2f system must scale efficiently with the number of replicas and clients. As the number of replicas increases, the system should be able to maintain its fault-tolerant properties without significant performance degradation.
- **Security**: Obtained through the hash chain that verifies the integrity of the operations performed by the system.
- **Maintainability**: The code structure should be easy to understand and extend. Given that BFT2f may need further optimizations or adjustments, the code should be modular and well-documented for future adjustments.

3.3 Models/Paradigms/Technologies

- **Client-Server Model**: The BFT2f system operates using a client-server architecture. Clients send requests to the primary replica, which coordinates the consensus process with other replicas, as visually explained by Figure 1, Figure 4 and Figure 5.
- **Replication Paradigm**: The core of BFT2f relies on state machine replication, where the system replicates states across multiple nodes to ensure fault tolerance.

Replicas execute the same operations in the same order, ensuring consistency even in the presence of faults.

- **Socket Programming**: Replica-Client and Replica-Replica communication is achieved through socket programming. Thanks to Python's *socket* library we could simulate a real-world distributed environment where nodes communicate over a network.
- **Hash Chains for Data Integrity**: The system uses a hash chain to track the integrity of operations across replicas. Similar to blockchain technology, each operation is chained to the previous one, ensuring the history of operations is verifiable and kept safe.

4 Design

This section describes on structure, behavior, and interaction of the algorithm.

4.1 Structure / Domain Entities

The BFT2f system is composed of several key entities, all shown in the UML class diagram of Figure 1:

- **Client**: Responsible for sending requests to the replicas and receiving replies.
- **Replica**: The core of the system, each replica maintains a log of operations and participates in the consensus algorithm to ensure fault tolerance.
- **Messages**: Various message types (PrePrepare, Prepare, Commit, etc.) facilitate communication between replicas during the consensus process.
- **Hash Chain Digest**: Tracks the history of committed operations to ensure data consistency.
- **Version Vector**: Manages the state of replicas to detect divergence and ensure consistency.

Each entity is modularized into separate Python classes, promoting a clean separation of concerns. For instance, the `Replica` class manages the state of each replica, while specific message types handle different phases of the consensus protocol.

4.2 Behavior

Each replica in the BFT2f system can either be a primary replica or a backup one:

- **Primary**: There is only one primary replica per system as it is the one responsible for initiating the consensus process and client request ordering. It receives client requests, assigns a sequence number to each request, and creates a PrePrepare message, which is subsequently multicast to all other replicas
- **Backup**: In each system, there is at least one backup replica. They participate in the consensus by validating the primary's proposal. Upon receiving the PrePrepare message from the primary, they generate a Prepare message and multicast it to all other replicas. They continue to participate in the consensus by collecting Prepare messages and generating Commit messages to finalize the agreement on the order of the client's request.

As hinted above, the message flow consists of three main phases handled differently: PrePrepare, Prepare, and Commit, as illustrated by Figure 2 and Figure 3. Each phase is critical for ensuring consistency and fault tolerance.

Regarding how the system can maintain Liveness and Consistency with Faulty Replicas Between f and $2f$, this is due to several different factors:

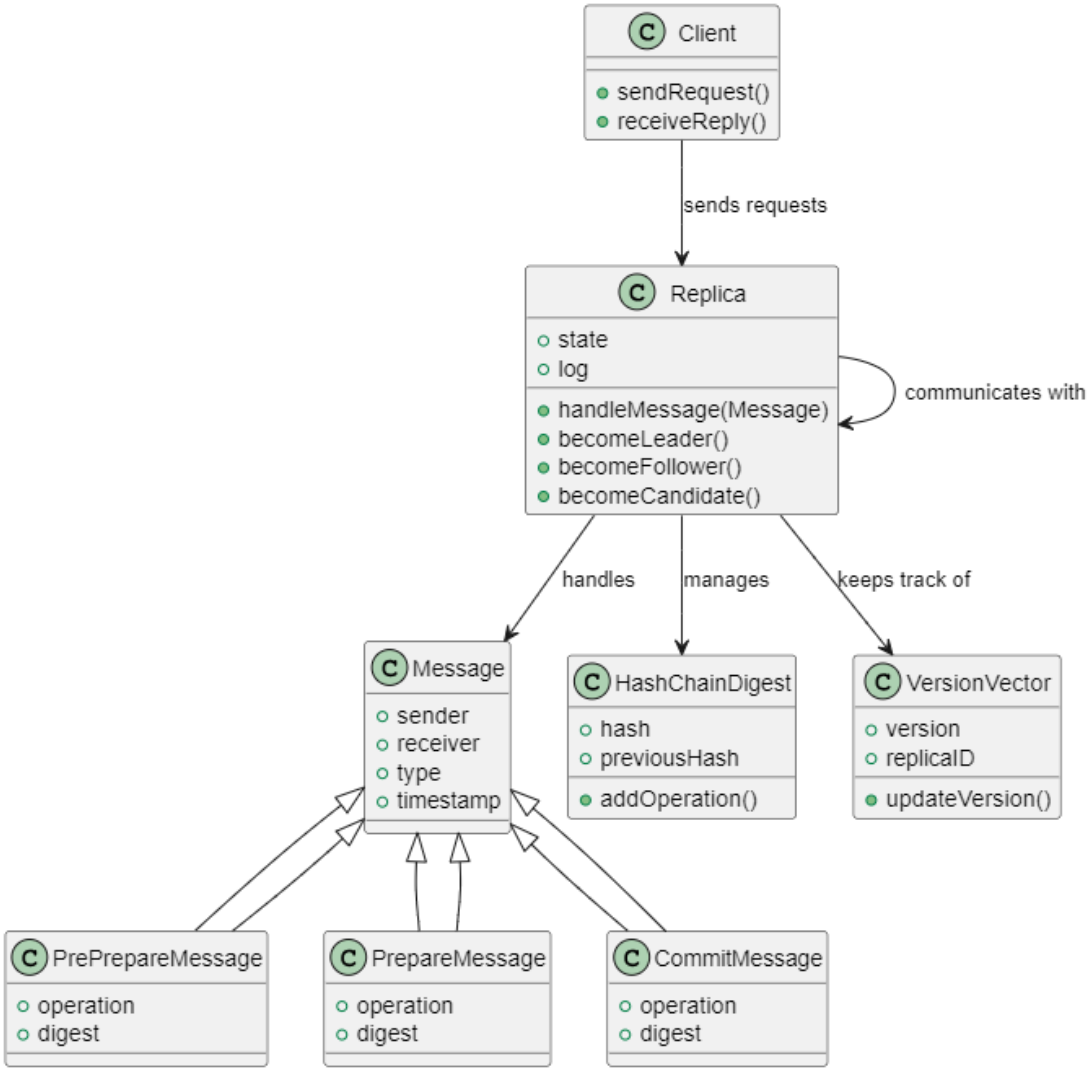

Figure 1: UML Class Diagram

- **Liveliness:**
  - **View Change Protocol:** The view change mechanism ensures that if the primary is faulty, a new primary is elected, allowing the system to continue making progress. The algorithm collects $2f + 1$ view-change messages, ensuring a majority of non-faulty replicas participate, allowing the system to recover from the failure of the original primary.
  - **Quorum-based Decisions:** The algorithm relies on a quorum of at least $2f + 1$ replicas to make decisions during the consensus process. Even when up to $2f$ replicas are faulty, the system can still gather enough valid messages to form a quorum, allowing the replicas to continue processing requests.
- **Consistency:**
  - **Commit Phases:** During the commit phase, replicas multicast commit messages and require at least $2f + 1$ matching commit messages to agree on the order and validity of operations. This ensures that faulty replicas cannot cause divergent states among the non-faulty ones.

4.3 Interaction

Interaction in the BFT2f system occurs primarily between the client and replicas, and among replicas themselves, following a series of phases that ensure fault tolerance and consensus.

- **Client-Replica Interaction** (shown by the UML sequence diagram in Figure 4): The client sends a `RequestMessage` to the primary replica, which is responsible for initiating the consensus process. Upon receiving the request, the primary assigns a sequence number and multicasts a `PrePrepare` message to all replicas. Here the Replica-Replica interaction begins. The Replica-Client interaction ends once the client receives sufficient matching `ReplyMessages` (at least 2f+1).
- **Replica-Replica Interaction** (shown by the UML sequence diagram in Figure 5): The replicas communicate with each other to reach consensus on the client's request. This process involves the exchange of several types of messages:
  - PrePrepare: The primary sends this message to all replicas, initiating the agreement process by proposing a specific sequence number for the client's request.
  - Prepare: Upon receiving the `PrePrepare` message, replicas send `Prepare` messages to each other, confirming that they have received and accepted the primary's proposal.
  - Commit: After receiving enough `Prepare` messages (from at least $2f + 1$ replicas), each replica sends a `Commit` message to all other replicas, finalizing the agreement on the sequence number and client request.

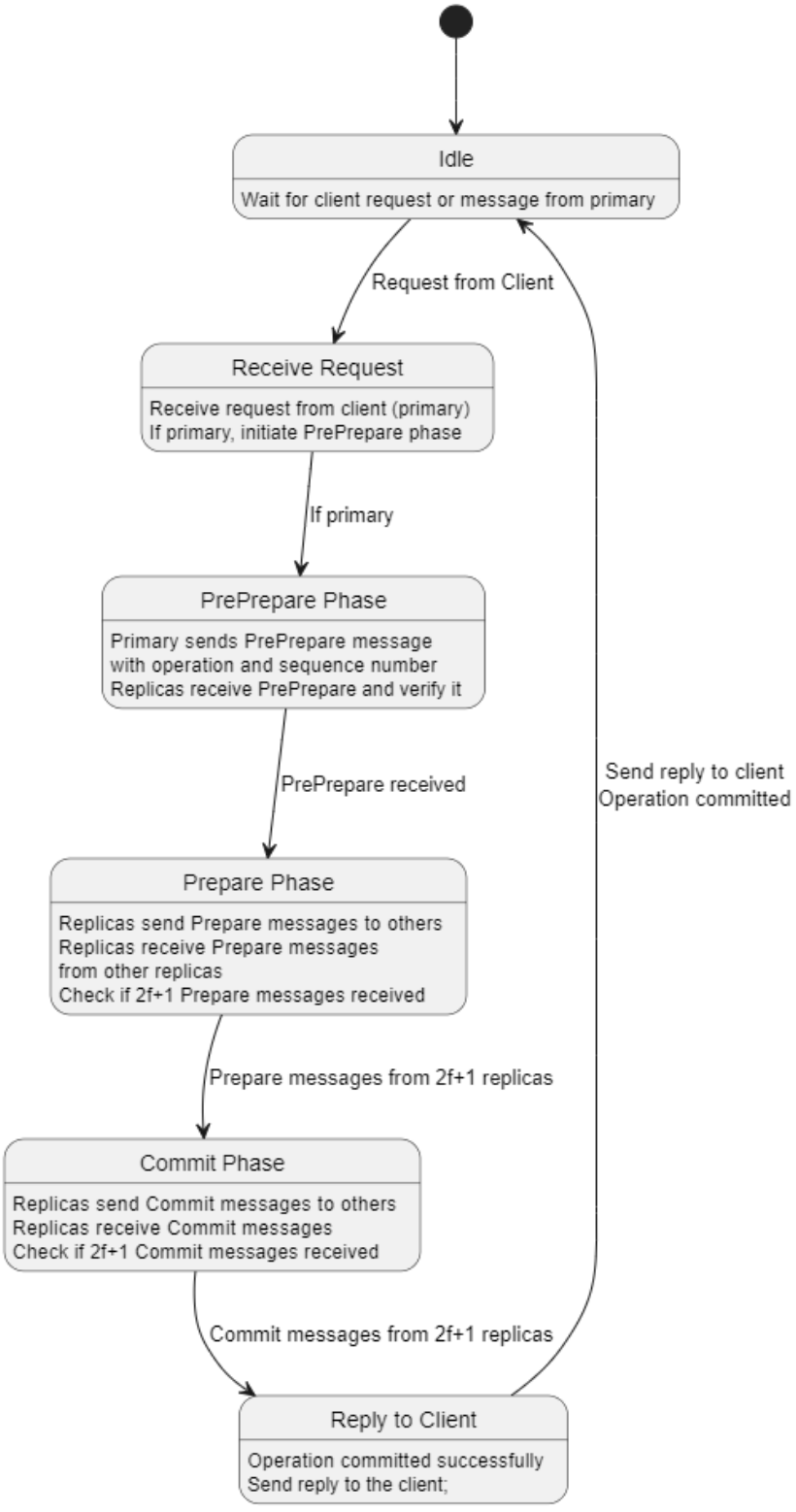

Figure 2: State Diagram of the General Functioning

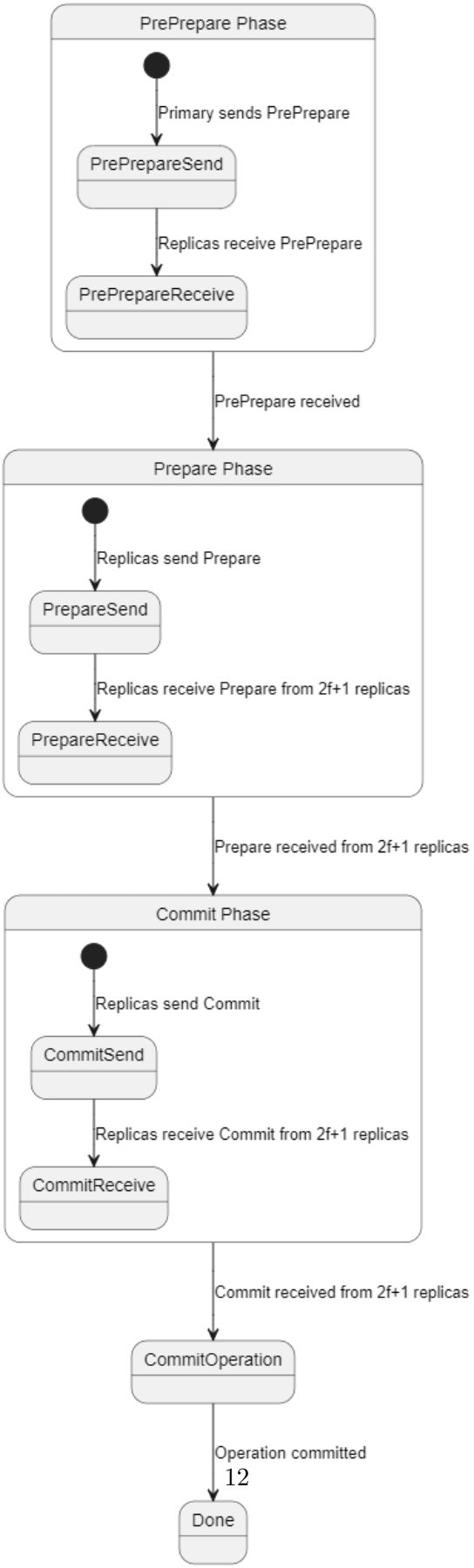

Figure 3: State Diagram of Phases Handling

– ViewChange: If the primary replica is suspected to be faulty (e.g., fails to send a `PrePrepare` message), replicas exchange `ViewChange` messages to initiate a new round of consensus, selecting a new primary.

The replica-replica interaction ensures the system remains fault-tolerant and consistent, even when up to $2f$ replicas are faulty. Each communication phase is crucial to guarantee that no single replica can dominate or corrupt the decision-making process.

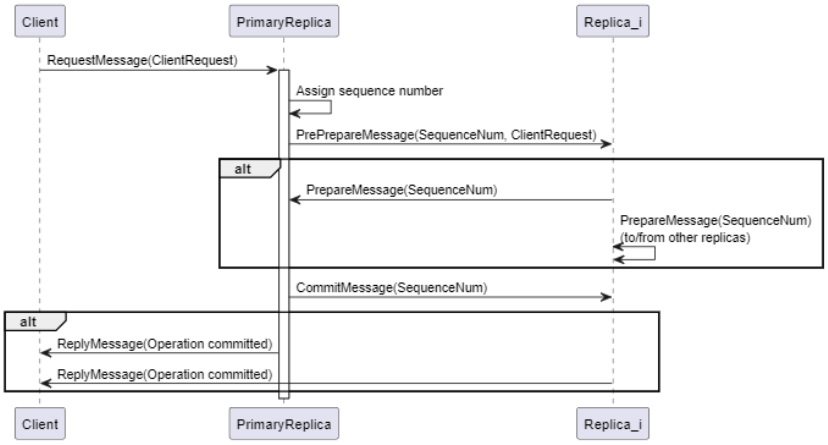

Figure 4: Sequence Diagram of Client-Replica Interaction

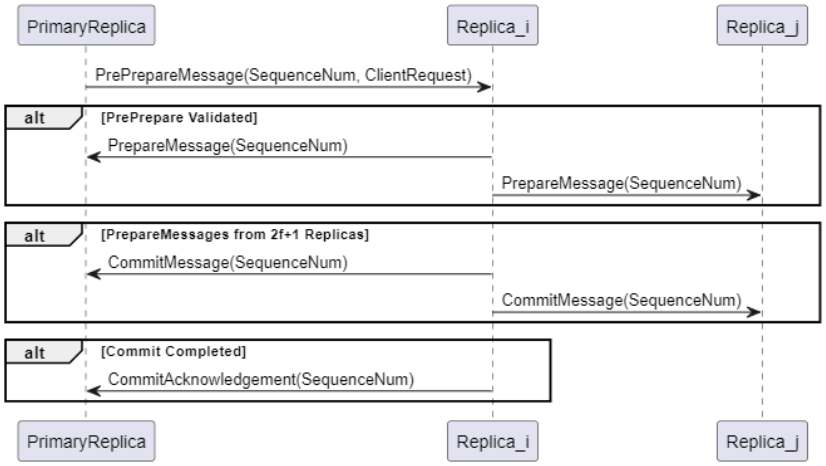

Figure 5: Sequence Diagram of Replica-Replica Interaction

4.4 View Change Mechanism

The **view change mechanism** is part of the Replica-Replica interaction and is crucial for maintaining the system's availability and resilience, triggered when replicas detect

that the primary replica is not functioning correctly. The view change process follows several key steps, as shown in Figure 6:

1. **Triggering the View Change:** Each replica that suspects the primary of being faulty broadcasts a `ViewChange` message to all other replicas. This occurs after a timeout period, during which no PrePrepare messages were received from the primary.

2. **Collecting View Change Messages:** Once a replica collects `ViewChange` messages from at least $2f + 1$ replicas (including itself), it prepares to initiate a new view. This ensures that enough replicas have agreed that the primary is not functioning correctly.

3. **Selecting a New Primary:**
   A new primary is deterministically selected based on the current view number and replica IDs. The new primary collects the state of the system from the `ViewChange` messages and generates a `NewView` message, which is broadcast to all replicas. This message contains the necessary information to continue the consensus process under the new primary.

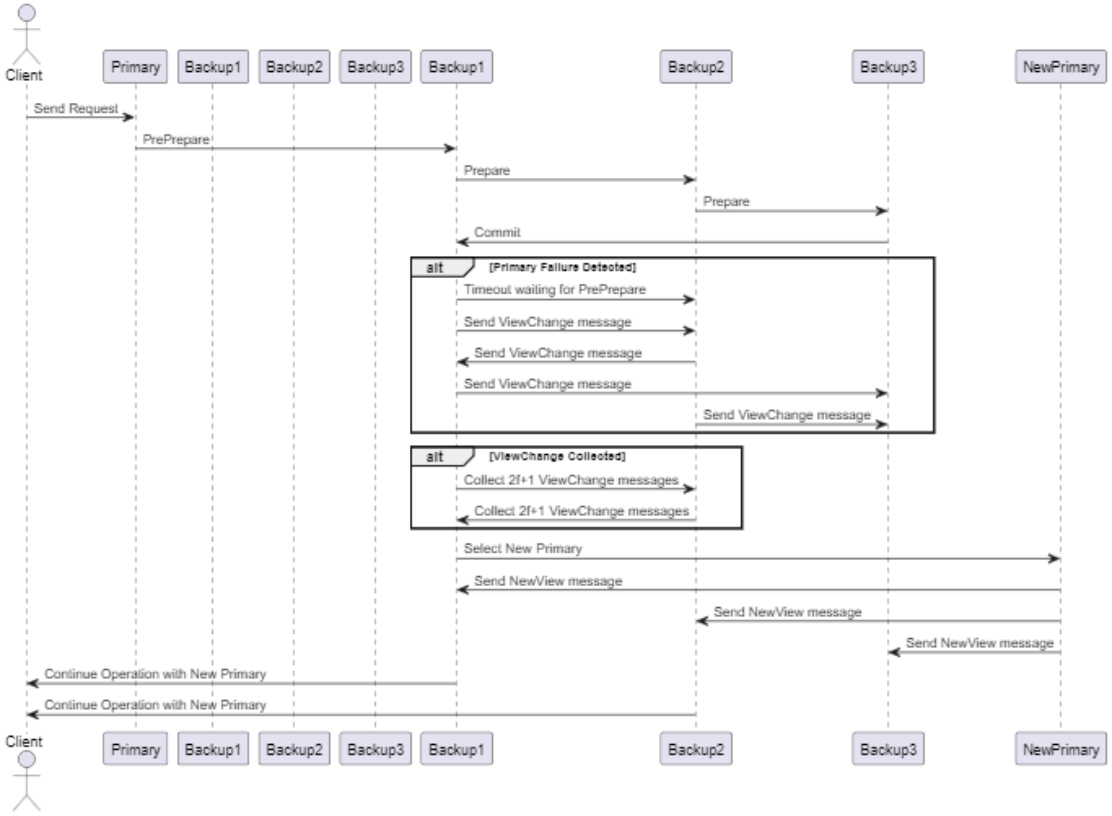

Figure 6: State Diagram of the View-Change mechanism

5 Implementation Details

The code implementation follows a modular structure where each function and class is documented with Python *docstrings*, ensuring clarity for each part of the consensus algorithm.

Non-trivial or non-obvious implementation details are documented directly within the code for easier access. The docstrings include explanations of input arguments, return values, and exceptions for each function.

6 Self-assessment / Validation

The BFT2f algorithm was implemented following a test-driven development (TDD) approach, with validation focused on both the system's functional requirements and fault tolerance under different conditions.

Evaluation criteria used to validate the produced software are explained in the following subsection:

6.1 Evaluation Criteria

- **Correctness under no-fault scenarios**: The system must be able to process client requests and reach consensus with full liveness and consistency when no replicas are faulty.
- **Tolerance to Byzantine failures**: The system must be able to tolerate up to f faulty replicas without breaking liveness or consistency.
- **Graceful degradation with more than *f* but less than or equal to *2f* faults**: The system should restrict the impact of faulty replicas in this scenario, maintaining liveness while allowing limited consistency violations.
- **Failure with more than *2f* faulty replicas**: The system is expected to break both liveness and consistency when the number of faulty replicas exceeds *2f*.
- **Resilience to incorrect or malicious messages**: The client and replicas must gracefully handle invalid or incorrect messages to prevent crashes, ensuring that the system continues operating even when Byzantine replicas send conflicting or corrupted data.

Two types of tests were written: Component Testing and Scenario-Based Testing.

Component tests focus on testing single isolated units of the implementation to ensure correctness in the behavior of the individual classes before integrating them into the broader system. On the other hand, scenario-based tests were developed to validate the system's behavior in both normal and faulty conditions.

6.2 Component Testing

- **Hash Chain Digest (HCD)**: Tests were conducted to ensure that the hash chain correctly adds blocks and maintains the sequence of operations. This component is critical for ensuring the integrity of the replicated system, as the HCD is used to verify the history of operations across replicas.
- **Version Vector**: Tests were designed to validate the *VersionVector* class, which tracks the state of each replica. The version vector was tested for proper updates, retrieval of entries, and correct representation of the system's state at a given time.

- **Message Classes (Request, Commit, Checkpoint, New View)**: A set of unit tests were written for different message types to ensure they are correctly initialized and serialized. These tests validate that the message objects are consistent and can be used for communication between replicas in the consensus process.

Unit tests were conducted using the *unittest* framework in Python. Each component passed all isolated tests, ensuring the correctness of the core building blocks of the BFT2f system.

6.3 Scenario-based Testing

- **No-fault scenario**: A test case with a client and multiple non-faulty replicas was created. The system successfully processed requests, and the client received matching replies from all replicas, ensuring full consensus and correctness.

- **Faulty replica scenario**: A replica was intentionally designed to simulate real-world faulty behavior in this scenario. This implementation mimics how a faulty node might act in a distributed system, sending both valid and invalid messages, and affecting consensus among replicas. The faulty replica was programmed to exhibit randomized behavior, enabling it to send a mix of valid or invalid messages or to skip sending messages entirely. This randomness allows for the testing of different outcomes based on the behavior of the faulty replica, reflecting how real-world systems can encounter unpredictable node behavior due to various reasons such as software bugs, network issues, or malicious actions.

These scenario-based tests are flexible in terms of the number of replicas and the fault tolerance parameter f. Parameters can be adjusted to simulate different network sizes and failure conditions.

By adjusting the parameters of the faulty replica, such as the number of total replicas and the fault tolerance level f, we can simulate a situation where the number of faulty replicas is up to f. In this case, the system is expected to continue functioning correctly, maintaining liveness and consistency.

Or, by configuring the test to include more than $2f$ faulty replicas, we can simulate a scenario where the system cannot achieve consensus. The randomness in message sending allows the faulty replica to create varying levels of disruption.

6.4 Test Results

In total, the following tests were conducted:

- **Component-specific tests**:
  - Total number of *unittest* tests: 13 tests
  - Passing tests: 13/13 (100% of the tests passed).

  - Test coverage: The tests cover the core components of the BFT2f algorithm, including isolated component tests for *HashChainDigest*, *VersionVector*, and different *Message* classes

- **Scenario-based tests**: Two main scenarios adjustable through parameters and the number of Replica/Client class instances created. In a normal no-fault scenario and an up-tp-2f faulty replicas scenarios, "Consensus reached" is printed on the terminal; this does not happen when the number of faulty replicas exceeds 2f.

7 Deployment Instructions

This section provides detailed steps for setting up and running the BFT2f library in a clean environment.

7.1 Prerequisites

Ensure that the following software is installed:

- **Python 3.8+**: The BFT2f algorithm requires Python version 3.8 or higher. Download and install Python from `https://www.python.org/downloads/`.
- **Git** (Optional): If cloning the repository from a remote source, ensure that Git is installed. Otherwise, you can download the source code as a ZIP file. Download and Install Git from `https://git-scm.com/downloads`

7.2 Setting Up and Running

1. **Clone the repository** (if applicable):

   ```
   git clone https://dvcs.apice.unibo.it/pika-lab/courses
                        /ds/projects/ds-project-pacilli-pieri-ay2324
   cd ds-project-pacilli-pieri-ay2324
   ```

2. **Install dependencies**: No external dependencies are necessary since all currently required libraries are part of Python's standard library. However, if future dependencies are added, the project includes a `requirements.txt` file.

   Use the following command to install dependencies (if any):

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

3. **Set up the library path**: To use the BFT2f library, you need to configure your environment to recognize the library location. Follow the steps below depending on your environment:

   - **For VSCode**: Add the following configuration to your workspace settings (`.vscode/settings.json`):

     ```
     {
         "python.analysis.extraPaths": [
             "path_to/BFT2F_library"
         ]
     }
     ```

- **For PyCharm**: Go to `File > Settings > Project: <project_name> > Project Structure` and mark the `BFT2f_library` folder as a source.
- **From the Command Line (Linux/Mac)**: Add the library path to the `PYTHONPATH` environment variable:

```
export PYTHONPATH=$PYTHONPATH:/path_to/BFT2F_library
```

- **For scripts or other environments**: If you're working in a different environment, add the following snippet at the start of your Python script:

```
import sys
sys.path.append('path_to/BFT2F_library')
```

4. Running the Software: To use the library in your own Python project, the required modules are Replica and Client:

```
from BFT2F_library.replica import Replica
from BFT2F_library.client import Client
```

Multiple clients and replicas can be set up; each must be initialized with specific parameters and in a network where they can communicate, as shown in the example in Listing 1. The `Client` class is instantiated with parameters like the host (e.g., `"localhost"`), the port it listens on (e.g., `6000`), and the fault tolerance level `f`. The `Replica` class is instantiated similarly with host and port information. Both classes have additional optional attributes, thoroughly explained in the documentation.

Listing 1: Client and Replica initialization example

```
f = 0
client = Client(host="localhost", port=6000, f=f)
replica = Replica(host="localhost", port=5000, f=f)
```

After all the desired classes and replicas are set up, the main algorithm can be triggered by issuing a request from a client, as illustrated in Listing 2. A potential user has to call the *make_request()* function and supply it with the operation to be performed, the implemented system will take care of the rest.

Listing 2: Client issuing request

```
client.make_request('log in')
```

5. Running the Tests: The tests can be run using Python's built-in `unittest` framework. All unit and scenario-based tests are located in separate Python files and can be executed from the command line. To run the component tests, navigate to the project directory `ds-project-pacilli-pieri-ay2324`, then each *unittest* file can be executed with the following command:

```
python -m unittest <test_filename>.py
```

The `unittest` framework will output the results, including passing and failing tests.

The scenario-based tests can be executed from the name directory, through the command:

```
python <filename>.py
```

8 Usage Examples

In each scenario, the usage process remains unvaried: users will create clients, initialize replicas, and issue requests from clients. However, the system's behavior will vary depending on the number of faulty replicas.

8.1 Scenario 1: Up to f Replicas Fail

In this scenario, the system can tolerate the failure of up to f replicas while maintaining full liveness and consistency.

- **Expected Behavior:** All operations are processed correctly, and the system continues to function as intended. The clients' requests are executed, and the results are returned without any errors.
- **User Expectation:** Users should expect that their requests are handled efficiently and correctly, with no interruptions or inconsistencies in the data returned.

8.2 Scenario 2: Faulty Replicas Exceed f but Less Than or Equal to 2f

In this scenario, the system is designed to restrict the impact of malicious nodes while still functioning correctly under certain conditions.

- **Expected Behavior:** While some operations may experience limited consistency violations, the system continues to operate without total failure. The algorithm effectively mitigates the influence of faulty replicas, allowing clients to perform actions, even if with potential discrepancies in the results, as explained in subsection 4.2.
- **User Expectation:** Users may notice inconsistencies in the outcomes of their requests or delayed responses, but the system will still process operations. Users should remain vigilant for potential issues but can generally expect the system to remain operational.

8.3 Scenario 3: More than 2f Replicas Fail

In this critical scenario, where the number of faulty replicas exceeds *2f*, the system's behavior is compromised.

- **Expected Behavior:** The BFT2f algorithm cannot guarantee liveness or correctness beyond this threshold. The system may halt or provide erroneous results, leading to potential data integrity issues.
- **User Expectation:** Users should expect failures in processing requests and may need to intervene manually to restore normal operation. Users must monitor system performance and be prepared for scenarios requiring external corrective actions, such as restarting the system.

9 Conclusions

This project involved the implementation of the BFT2f algorithm, a key Byzantine Fault Tolerance (BFT) approach that enhances the reliability and security of distributed systems.

9.1 Application of Theoretical Concepts

The theoretical concepts encountered during the project are here illustrated:

1. **The Problem of Consensus in Distributed Systems**: The BFT2f algorithm addresses the fundamental problem of achieving consensus among replicas in the presence of faults. Understanding consensus algorithms was crucial for implementing the message-passing phases (PrePrepare, Prepare, Commit) effectively, and designing a robust mechanism to ensure that replicas can agree on the system's state, even when facing malicious actors.

2. **Replication and Consistency in Distributed Systems**: By employing a well-defined replication strategy, we ensured that the system maintained consistency despite failures, understanding how replication can enhance a system's availability and reliability.

3. **The CAP Theorem**: Understanding the CAP theorem was one of the fundamental blocks of this project to understand the trade-offs inherent in designing a distributed system. For example, in BFT2f, when the number of faulty replicas is between f and $2f$ the system keeps operating but users may experience delay or inconsistencies in results.

4. **Distributed Ledger Technology**: A hash chain was created to track the integrity of operations within the BFT2f algorithm.

5. **Sockets and Asynchronous Programming**: To implement effective and concurrent communication between clients and replicas.

9.2 Future Work

At its current state, we believe the following are the most important and interesting improvements for the future:

- **Deployment as a Real Distributed System**: The current implementation operates on a single machine, which limits the demonstration of its true distributed capabilities. Future work could focus on deploying the system across multiple machines, enabling actual distributed operations.

- **Real operations in Request Messages**: At present, the operations specified in the issued request messages are limited to *strings*, without performing any actual operations. A significant enhancement would be to implement concrete operational

logic that the replicas can execute upon receiving requests. This could include operations such as data processing, state modifications, or database transactions.

- **User Interface and Integration**: Currently, potential users interact with the library by creating Python scripts, importing necessary modules, and manually invoking functions. To improve usability, future work could involve developing a more user-friendly interface or integrating the BFT2f implementation into larger systems, such as web applications or microservices architectures.

9.3 What did we learn

Throughout this project, we gained a deep understanding of the theoretical and practical aspects of implementing distributed systems, in more detail, the BFT2f algorithm:

- **Distributed System development and management**
- **Consensus and Fault Tolerance**
- **The CAP Theorem in Practice**
- **Replication and Consistency**
- **Communication design between the system parts**
- **Ledger Technology implementation and importance**

References

[1] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA, 1999. USENIX Association.

[2] Jinyuan Li and David Mazières. Beyond one-third faulty replicas in byzantine fault tolerant systems. In Hari Balakrishnan and Peter Druschel, editors, *4th Symposium on Networked Systems Design and Implementation (NSDI 2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings*. USENIX, 2007.