BFT2f implementation

Achieving Byzantine Fault Tolerance with Enhanced Fault Capacity

Presented by:
Benedetta Pacilli
Valentina Pieri

01

Introduction

Introduction

Objective

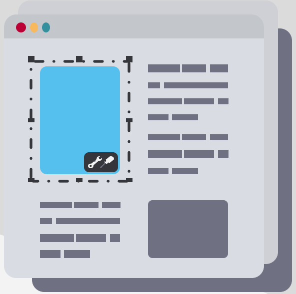
- **Implement** and **explore** BFT2F algorithm
- **Demonstrate** the algorithm's **abilities**

Context

- BFT is critical for the **reliability** of distributed systems
- Today increasing **reliance** on distributed systems

Overview

- Theoretical **Background**
- System **architecture** and implementation
- **Testing** and Results
- **Conclusions** and Future work



02

What is BFT?

What is BFT?

Definition

- Class of fault tolerance mechanisms
- Protect DSs from failures

Importance

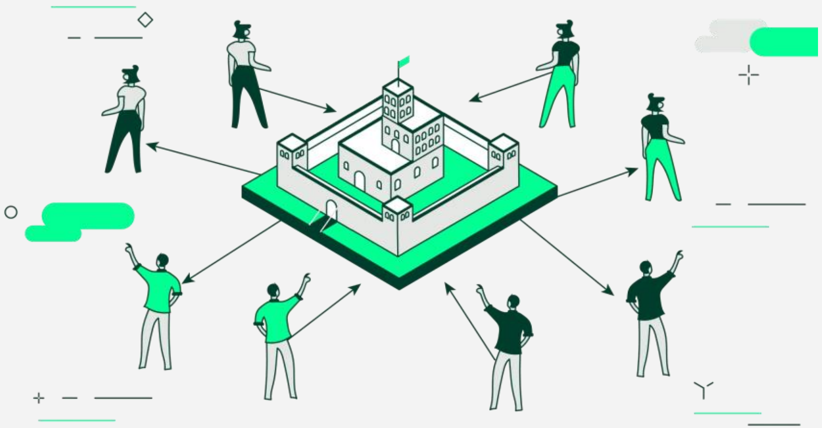
- Reliability and Security
- Arbitrary faults: sw errors, hw malfunctions, sabotage

Challenges Addressed

- Complex algorithms
- Ensure consensus
- Security-Performance trade-off

Relevance

- Fundamental in blockchain technologies
- Industries where data integrity and uptime are critical



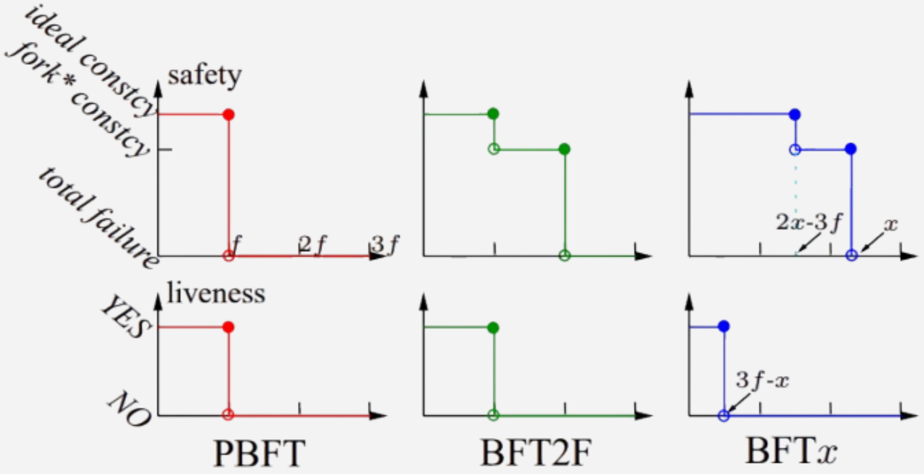
03

Why BFT2f?

Why BFT2f?

BFT limitations

- Up to ***f*** faults
- Struggles with **scalability** and **overhead**



Instead BFT2f...

- **Extends** fault tolerance capacity
- Systems with a **high number of nodes**

- Manages both Byzantine and crash faults
- Adjusts **dynamically**

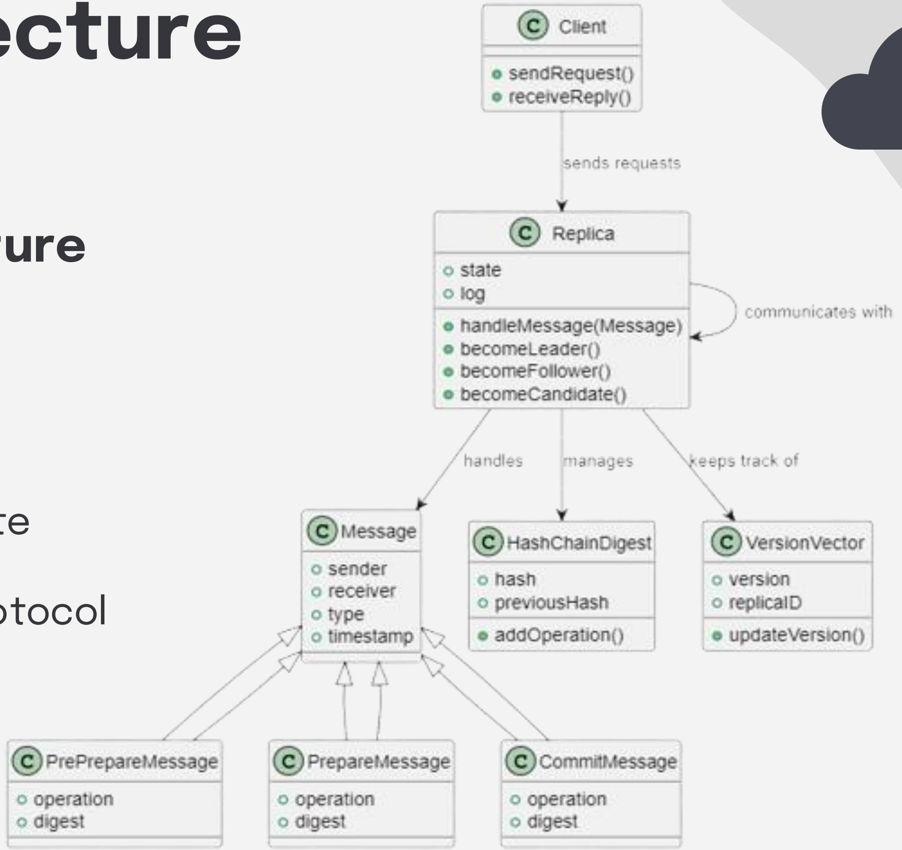
04

System Architecture

System Architecture

Client - Server Architecture

- **Client**
  - **Interface** for requests
  - **Handling** replies
- **Replica**
  - **Maintaining** system state
  - **Processing** messages
  - Manages **consensus** protocol

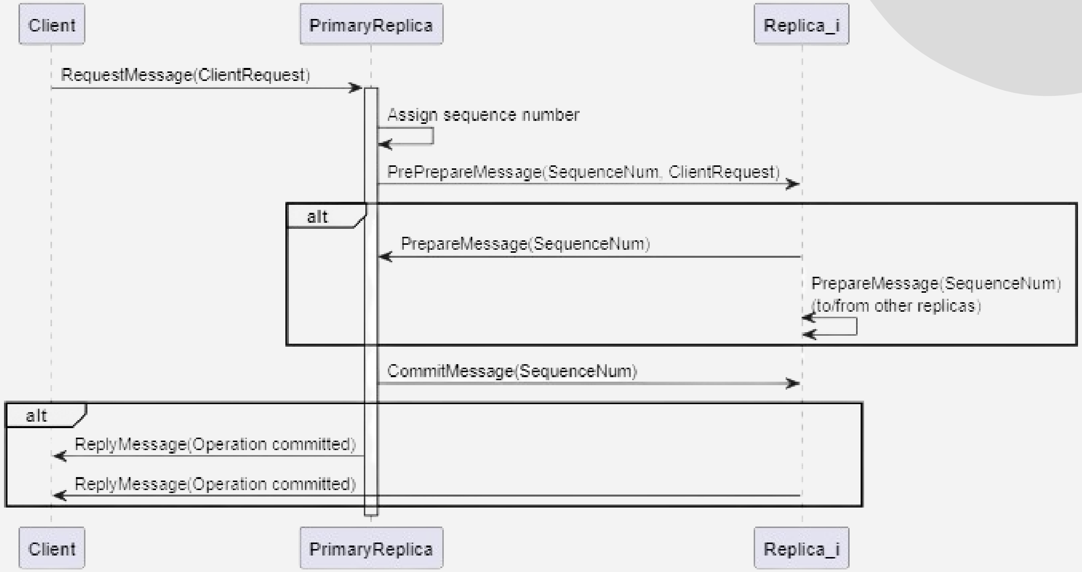


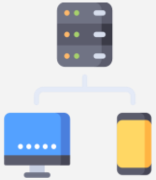
05

Algorithm Functioning

Client – Replica Interaction

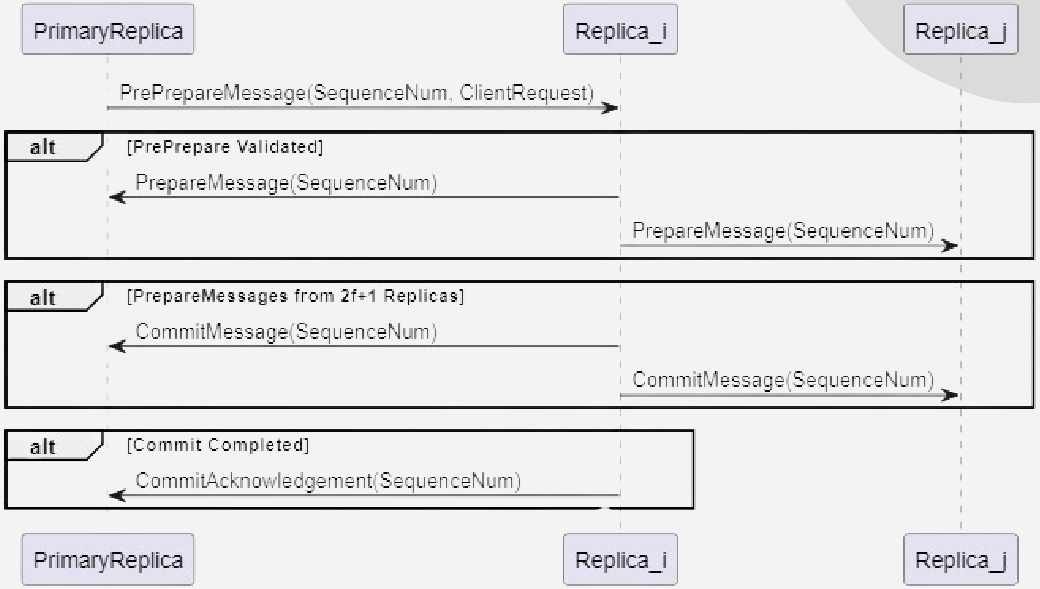
- Clients send operation **requests** to **primary** replica
- Primary assigns sequence numbers and sends pre-prepare messages
- Beginning of **Replica-Replica** Interaction
- Replicas send **reply** to Client once operation is executed





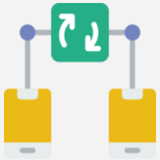
Replica – Replica Interaction

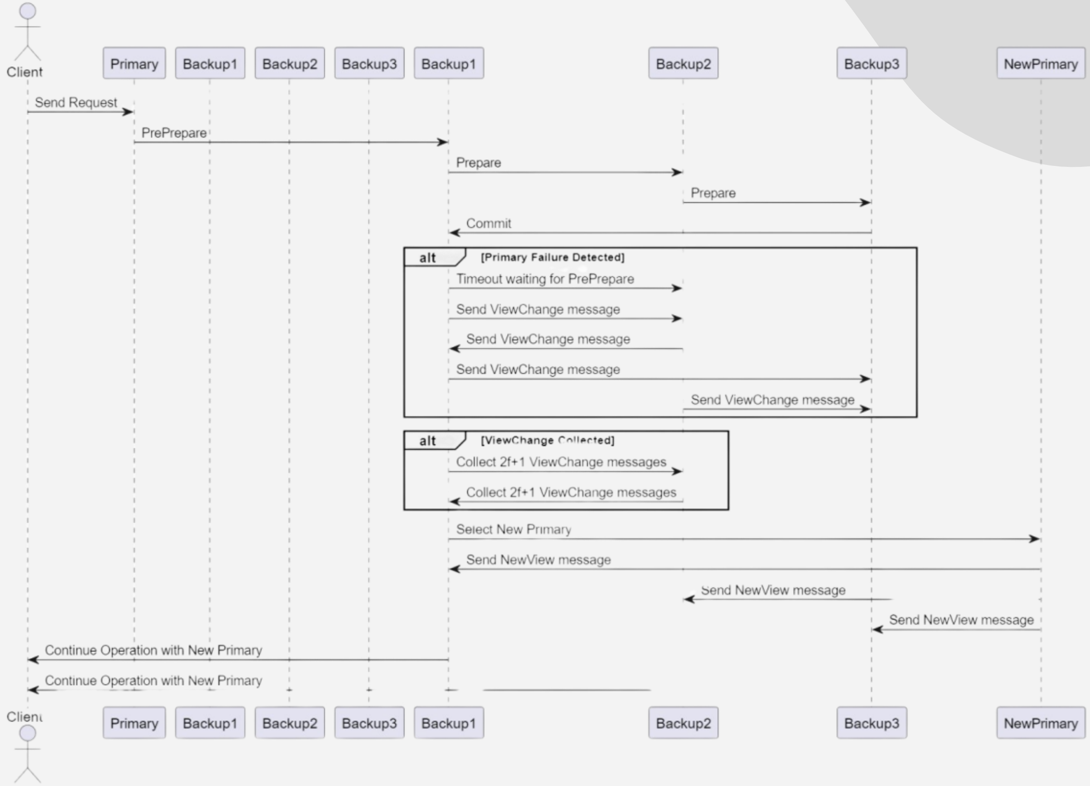
- 3 phases:
  - PrePrepare
  - Prepare
  - Commit
- ***f*** faulty replicas -> ***2f+1*** honest replicas to reach consensus
- Successful commit phase → operation is **executed**
- Result is **permanently** recorded



View – Change Mechanism

- Triggered by primary **failures** or **unresponsiveness**
- Election of **new primary**
- New **synchronization** of the state among replicas





Fork Conditions

What are fork conditions?

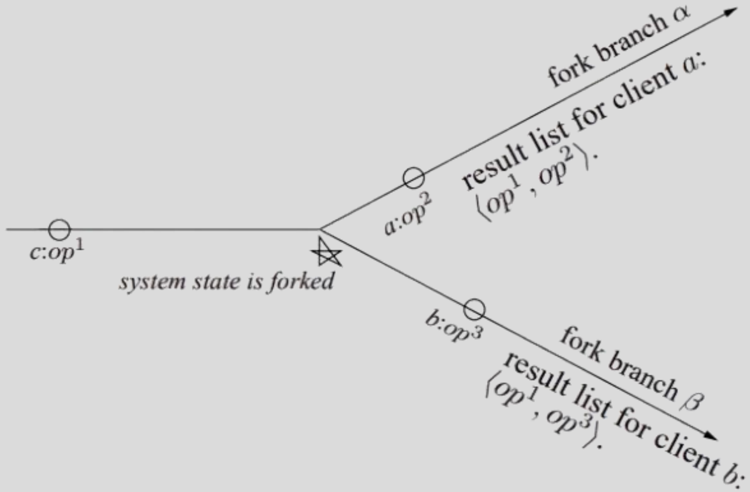
- System may experience **split behaviors**
- **Conflicting** information from replicas
- f < faulty replicas < 2f

Consistency

- BFT2f requires ***2f+1*** **matching responses** before committing any state change
- Mitigating the impact of up to 2f faulty replicas

Liveliness

- Maintained through **view-change** protocol and **quorum**-based decisions
- Ensuring system **continuously** processes requests



06

Testing

Scenarios

No faults

- **All replicas** participate in consensus
- Requests processed **smoothly**
- **Consistent** results across replicas
- Consensus is **reached**

Between f and 2f

- **Inconsistencies** and potential **delays**
- The system should maintain **overall consistency and availability**
- Potentially isolating faulty replicas

More than 2f

- System may **fail** to process requests correctly

How we tested

Setup

- On a **single** machine
- Python’s **socket** lib
- Multiple instances of classes
- **unittest** framework

FaultyReplica Class

- Designed to simulate various types of faults
  - **conflicting** messages
  - **delaying** responses
  - **not** responding
- Can randomly choose a behavior at runtime
- Instances and *f* are **parameterized**

Components tests

- 13 tests
- For **individual** components
  - HashChainDigest
  - VersionVector
  - Message classes
- Ensure correct functionality in **isolation** and when **integrated**
- 100% of the tests passed

07

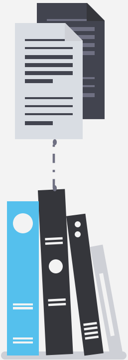
What did we learn?

What we learned

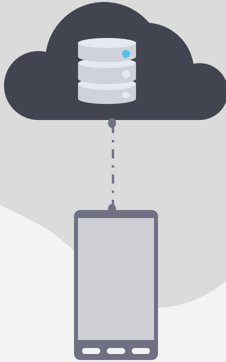
CAP theorem	Trade-off fundamental for system functioning
The problem of **Consensus**	Crucial for implementing message phases, how consensus underpins the reliability and integrity of distributed systems
Replication & **Consistency**	How replication can enhance system's availability and reliability
Communication Design	Ensuring smooth data flow between parts
Distributed Ledger Technology	Hash Chain to track operations' integrity, security boost
Asynchronous Programming	Effective, real-time and concurrent communication among parts

08

For the future



Future Improvements



Real DS

- Currently operating on a **single machine**
- → deploying/testing **across multiple** machines

Real Operations

- Requests operations are limited to **strings**
- → **actual operations**
  - Data processing
  - Database transactions

UI & Integration

- Interaction through **scripts**
  - Importing modules
  - Invoking functions
- → user-friendly **interface**
- → **integrating** into larger systems

09

To conclude

Conclusions

Practical Application of Theory

- Theoretical concepts in **practical setting**
- Experienced **relevance** and **utility** in DSs

Successful Implementation

- **Fully functional** BFT2F library
- **Effectively handling** Byzantine faults

Testing Framework

- Validated system **performance** and **components**
- Various **scenarios**

Thank you!

Benedetta Pacilli – Valentina Pieri

benedetta.pacilli@studio.unibo.it – valentina.pieri5@studio.unibo.it

Resources

[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 Mazi`eres. 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.