DepChain - Stage 1

Simão de Melo Rocha Frias Sanguinho[1[102082]], José Augusto Alves Pereira[1[103252]], and Guilherme Silvério de Carvalho Romeiro Leitão[1[99951]]

Instituto Superior Técnico, Lisboa, Portugal

Abstract. This project aims to develop a simplified permissioned blockchain system, named Dependable Chain (DepChain), with high dependability guarantees. The system is designed to be built iteratively, with the first stage focusing on the communication and consensus layer of a simple blockchain implementation, and the development of a client and a library that can interact with the blockchain system. The project leverages the Byzantine Read/Write Epoch Consensus algorithm, with simplifying assumptions such as static system membership, a predefined leader process, and a Public Key Infrastructure (PKI). For message communication, the implementation will use authenticated perfect links, with the assumption that the network is unreliable: it can drop, delay, duplicate, or corrupt messages, and communication channels are not secured. The implementation is structured in Java, utilizing the Java Crypto API for cryptographic functions, and is designed to handle malicious behavior from a subset of blockchain members while ensuring safety and liveness under the assumption of a correct leader. The final submission for stage 1 includes a self-contained zip archive with the source code, demo tests, and a concise report detailing the design, threats, and dependability guarantees of the system.

Keywords: Blockchain · Byzantine Fault Tolerance · Consensus Algorithm · Dependability · Java Implementation

1 Introduction

This report presents the design and implementation of a Byzantine Fault Tolerant (BFT) blockchain service, designed to withstand malicious behavior from a subset of blockchain members and operate reliably in an unstable network environment. The system is resilient to arbitrary (Byzantine) behavior from faulty nodes and can handle unreliable network conditions, including message drops, delays, duplication, and corruption, without relying on secure communication channels. To achieve consensus in such adversarial conditions, the project uses the Byzantine Read / Write Epoch Consensus algorithm, as described in the course book [1] (Algorithms 5.17 and 5.18). The report outlines the system architecture, which includes the network, client, library, blockchain, and consensus layers. It also describes how the system addresses various Byzantine attack scenarios, ensuring safety and liveness under the assumption of a correct leader. Finally, the report concludes with key findings, lessons learned, and potential areas for future improvement.

2 Architecture

2.1 Network Layer

The network layer is responsible for managing the communication any two processes (members or clients) in the system. By replicating Authenticated Perfect Links, the network layer ensures that messages are guaranteed to be eventually delivered to the intended recipient, with it's integrity and authenticity preserved. There are three main components in the network layer:

- Message, which encapsulates the message content and metadata. It contains the type field to identify the message type (e.g., READ, STATE, ACK, etc.). It's also this component that will be signed by the sender and verified by the receiver.

- PerfectLink, which implements the core communication logics, including sending, receiving and managing sessions (explained next). To simulate the unreliable network, we use UDP sockets.

- Session, which represents a communication session between two processes. The session contains information like the destination process ID, address, session key, and counters for tracking sent and acknowledged messages. The session key is used for encrypting and signing messages (because using the public key for every message would be too expensive).

Session Establishment Before any communication can occur, a session must be established between two processes. The session establishment process is as follows: A process initiates a session by sending a STARTSESSION message. Then, the recipient responds with a ACKSESSION message, containing an encrypted session key. Once the session is established, all subsequent messages are signed and verified using the session key to ensure authenticity and integrity.

2.2 Client and Library Layer

The client layer and library layer work together to enable clients to interact with the distributed system, specifically to append messages to the blockchain. The client layer handles user interaction and initialization, while the library layer manages the communication logic with the system's leader process.

Normal Workflow A user will issue the append <message> command via the client's CLI. The client layer will delegate the request to the library layer. Then, the library will construct a CLIENTREQUEST message and sends it to the leader process (known beforehand) using PerfectLink. Next, the library will wait for CLIENTREPLY from F+1 processes.

2.3 Blockchain Layer

The blockchain layer represents a node in the permissioned (closed membership) blockchain network. Each member is responsible for participating in the consensus protocol, maintaining a local copy of the blockchain.The system is designed to ensure that all members agree on the state of the blockchain, even in the presence of faulty nodes, by leveraging a Byzantine fault-tolerant consensus protocol (Byzantine Read/Write Epoch Consensus). The blockchain member maintains an in-memory representation of the blockchain, implemented as a State object. This State stores a sequence of TimestampValuePair objects, where each pair consists of a timestamp and a value representing a block in the blockchain. The blockchain is updated whenever a consensus decision is reached, ensuring that all non-faulty members have a consistent view of the blockchain.

A message handler loop continuously listens for incoming messages. If a CLIENTREQUEST is received and the member is the leader, it initiates a new consensus instance to process the request. For consensus-related messages, the member delegates the message to the current consensus instance for processing.

2.4 Consensus Layer

The consensus mechanism in this project is designed to ensure that all non-faulty members of the distributed system agree on the value to be appended to the blockchain, even in the presence of Byzantine faults. The consensus protocol is implemented in the ConsensusInstance class and involves multiple phases.

Consensus Instance Each consensus instance is identified by an epoch number, which also serves as the consensus instance ID. The consensus instance is initiated by the leader (process 1) when it receives a CLIENTREQUEST from a client. The leader is responsible for coordinating the consensus process, while other members participate by responding to messages and contributing their local states.

Read Phase The consensus process begins with the read phase, where the leader broadcasts a READ message to all members. Each member responds with a STATE message, which contains its local blockchain state, including the most recent write and the writeset (a list of all writes). The leader collects these STATE messages and uses them to determine the most recent value that has been agreed upon by a quorum of members.

Collected Phase Once the leader has received STATE messages from a quorum of members, it broadcasts a COLLECTED message to all members. The COLLECTED message contains the collected states from all members, allowing each member to independently determine the value to be written. Members use the collected states to select the most recent value that appears in the writeset of more than f members (where f is the maximum number of Byzantine faults

tolerated). If no such value exists, the leader's most recent write is selected as the candidate value.

Write Phase The leader broadcasts a WRITE message containing the candidate value to all members. Upon receiving the WRITE message, each member updates its local state and sends an ACCEPT message back to the leader. The leader waits for ACCEPT messages from a quorum of members before proceeding to the decision phase.

Decision Phase Once the leader has received ACCEPT messages from a quorum of members, it broadcasts a DECIDED message to all members. The DECIDED message contains the final value that has been agreed upon by the consensus instance. Upon receiving the DECIDED message, each member appends the agreed-upon value to its local blockchain and updates its state accordingly.

Fault Tolerance and Quorum The consensus protocol is designed to tolerate up to f Byzantine faults, where f is the maximum number of faulty members allowed in the network. A quorum is defined as the minimum number of members required to reach agreement, calculated as floor((N + f) / 2), where N is the total number of members. The protocol ensures that all non-faulty members agree on the same value, even if some members behave maliciously or fail to respond.

3 Implementation details

3.1 PKI and Cryptography

The system relies on a Public Key Infrastructure (PKI) to ensure the identity of members. Due to the perfomance limitations of public key cryptography, the system derives a shared simmetric key for each session, which is used to sign the messages. Since confidentiality is not a requirement, the system does not encrypt the messages.

3.2 Byzantine leader

Due to the scope of the project, the system assumes a correct leader process (process 1) that initiates consensus instances and coordinates the consensus protocol. However, the system is designed to handle Byzantine behavior, and so, if the leader behaves maliciously and deviates from the protocol, the other members will detect the misbehavior and abort the consensus instance.

3.3 Members replies

Whenever a client issues a request, it will not block waiting for the reply, since the consensus protocol may take some time to reach an agreement. Instead, the client will asynchronously wait for the rely. As a client, it will wait for F+1 replies, where F is the number of Byzantine faults the system can tolerate.

4 Possible threats and corresponding protection mechanisms

To demonstrate the system's resilience against various Byzantine scenarios, the implementation was tested under multiple configurations, each representing a different type of attack. The following sections provide a detailed explanation of how each attack is executed and how the system effectively mitigates it.

4.1 Threat XYZ

Cras quis bibendum erat. Aliquam massa tortor, euismod a venenatis eget, convallis ut odio. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

4.2 Threat XYZ

Cras quis bibendum erat. Aliquam massa tortor, euismod a venenatis eget, convallis ut odio. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

4.3 Threat XYZ

Cras quis bibendum erat. Aliquam massa tortor, euismod a venenatis eget, convallis ut odio. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

4.4 Threat XYZ

Cras quis bibendum erat. Aliquam massa tortor, euismod a venenatis eget, convallis ut odio. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

4.5 Threat XYZ

Cras quis bibendum erat. Aliquam massa tortor, euismod a venenatis eget, convallis ut odio. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

4.6 Threat XYZ

Cras quis bibendum erat. Aliquam massa tortor, euismod a venenatis eget, convallis ut odio. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

4.7 Threat XYZ

Cras quis bibendum erat. Aliquam massa tortor, euismod a venenatis eget, convallis ut odio. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

4.8 Threat XYZ

Cras quis bibendum erat. Aliquam massa tortor, euismod a venenatis eget, convallis ut odio. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

4.9 Threat XYZ

Cras quis bibendum erat. Aliquam massa tortor, euismod a venenatis eget, convallis ut odio. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

4.10 Threat XYZ

Cras quis bibendum erat. Aliquam massa tortor, euismod a venenatis eget, convallis ut odio. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

4.11 Threat XYZ

Cras quis bibendum erat. Aliquam massa tortor, euismod a venenatis eget, convallis ut odio. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

5 Conclusion

Donec vulputate ultrices dignissim. Donec vel elit a massa pellentesque euismod eget at velit. Aliquam eget est et urna auctor sodales. Aenean vulputate finibus libero ac pretium. Etiam lacinia ultrices odio, vitae bibendum metus accumsan sed. Etiam sit amet condimentum eros, pulvinar pharetra nisi. Phasellus ac purus a libero euismod ultrices. Duis feugiat, mi id facilisis blandit, magna magna commodo ante, ut porta lacus justo ut neque. Donec eget nisl feugiat, pretium libero eu, mollis dolor.

References

1. Christian Cachin, Rachid Guerraoui, Luís Rodrigues: Introduction to Reliable and Secure Distributed Programming. 2nd edn. Springer, 2011
2. Java Crypto API, https://docs.oracle.com/javase/8/docs/api/javax/crypto/package-summary.html
3. Theoretical Classes Slides, HDS2425 https://fenix.tecnico.ulisboa.pt/disciplinas/SDTF236/2024-2025/2-semestre/theoretical-classes