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.

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, 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 between 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.). A MAC will be used to ensure the message's aforementioned security properties. - 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 a symmetric key that is used for 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 START_SESSION message. Then, the recipient responds with an ACK_SESSION message, containing a symmetric session key that is encrypted using the recipient's public key (thus ensuring confidentiality). 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 system's regular workflow would work as follows: 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 CLIENT_REQUEST message and send it to the leader process (known beforehand) using PerfectLink. Next, the library will wait for a CLIENT_REPLY. Since we assume a non-byzantine leader, the client does not need to wait for a quorum of replies.

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 as a list of strings. 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). It also

keeps a State object, which stores a sequence of TimestampValuePair objects, that represent the history of that member on a particular instance of consensus. 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 CLIENT_REQUEST is received by 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 keeps track of the current epoch number. The consensus instance is initiated by the leader (process 1) when it receives a CLIENT_REQUEST 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 epoch state, including the most recent write and the writeset (a list of all writes).

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 the quorum's 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 value is unbounded. In this case, the leader's most recent write is selected as the candidate value.

Write Phase Every process (members and leader), broadcasts a WRITE message to all members containing the previously determined value. It then waits for a quorum of WRITE messages where there must be at least f+1 messages with the same value it proposed. If this condition is not met, then the process aborts the consensus instance. Otherwise, it moves on to the next phase.

Accept Phase Every process then broadcasts an ACCEPT message to all members containing the value that was previously agreed upon. It then waits for a quorum of ACCEPT messages back containing the same value at least f+1 times. If this condition is not met, then the process aborts the consensus instance. Otherwise, it epoch-decides the value and writes it to the blockchain.

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 2f+1. The protocol ensures that all non-faulty members agree on the same value, even if f members behave maliciously, fail to respond or crash.

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 performance limitations of public key cryptography, the system derives a shared symmetric 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 should detect the misbehavior and abort the consensus instance.

3.3 Client communication

To simplify the system, the client only communicates with the leader process.

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 Ignore Messages

In this scenario, a faulty member ignores messages from the leader or other members, with the goal of disrupting the consensus process. The system is designed to handle this scenario by not needing to wait for all members to reply (only 2f+1 replies are required).

4.2 Byzantine State

In this scenario, a faulty member sends a STATE message containing an incorrect blockchain state, with the goal of affecting the members' choice of a value to write. Since the system uses a quorum of states to determine the value to write, the faulty state will be ignored, and the correct value will be chosen by all members.

4.3 Impersonate Member

In this scenario, a faulty member impersonates another member by sending a STATE message to the leader posing as the legitimate member. The system is designed to handle this scenario by using the session key to sign messages, ensuring that only the legitimate member can send messages on their behalf.

4.4 Spam Messages

In this scenario, a faulty member sends a large number of STATE messages to the leader, with the goal of overwhelming the leader and disrupting the consensus process. The system is designed to handle this scenario by storing the states received in a map and when more than one state is received from the same member, the system just overwrites its entry with the new state.

5 Conclusion

The first stage of the DepChain project has been successfully completed, resulting in a functional blockchain system that can tolerate Byzantine faults and operate reliably in an unstable network environment. The system leverages the Byzantine Read/Write Epoch Consensus algorithm to achieve consensus among non-faulty members, ensuring that all members agree on the state of the blockchain.

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