DepChain: A Dependable Permissioned Blockchain System

Henrique Carrão 100313, João Maçãs 99970, and Duarte São José 103708

Instituto Superior Técnico, Lisbon, Portugal

Abstract. This project presents DepChain, a permissioned blockchain system designed with high dependability guarantees. The first stage of the project focuses on implementing the consensus layer using the Byzantine Read/Write Epoch Consensus algorithm. The system assumes a static membership model with a pre-designated leader and relies on a pre-established Public Key Infrastructure for authentication of the server nodes. The communication between blockchain members is built upon authenticated perfect links to ensure message delivery and it integrates cryptographic mechanisms for security guarantees. The client library is responsible for submitting requests to append data to the blockchain, while the blockchain service ensures correct and secure processing of transactions.

1 Introduction

Distributed systems are vital to modern infrastructure but face challenges from failures and attacks. Blockchain technology addresses these issues with its tamper-evident, append-only ledger. While public blockchains operate in open environments, permissioned blockchains offer controlled access and better performance, making them ideal for enterprises. DepChain is a permissioned blockchain designed to ensure dependability using Byzantine fault tolerance, which allows the system to operate reliably even amidst malicious or faulty nodes.

Our implementation focuses on the Byzantine Read/Write Epoch Consensus algorithm, providing a robust foundation for agreement among blockchain participants. The system operates with a static membership model and relies on a pre-established Public Key Infrastructure (PKI) for node authentication. We've built secure communication channels through authenticated perfect links, ensuring reliable message delivery despite network failures or adversarial behavior.

This report details our architectural decisions, implementation strategies, and the security measures employed to protect against both network-level threats and Byzantine behavior. We place particular emphasis on the dependability guarantees that our system provides, demonstrating how DepChain maintains both safety and liveness properties in challenging distributed environments.

2 System Design

2.1 Architecture Overview

Our implementation has two main actors, `Client` and `Node`. `Client` library can attempt to append a string to the blockchain using a TUI. `Node` on the other hand, represents the server nodes on the blockchain environment. It maintains a `ClientAplManager` for handling client communication and registration and a `ServerAplManager` to communicate with the other `Node`'s on the blockchain. Each `Node` which is constantly running a consensus thread deciding `AppendReq`'s that it or other`Node` has received and then a thread that handles all the decided values and append them onto the blockchain, sending a success feedback onto the `Client`.

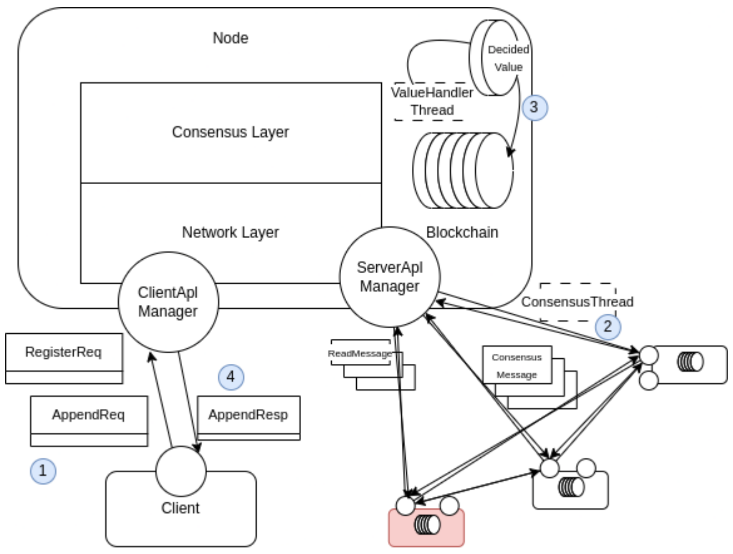

Fig. 1. Client append request process on DepChain

2.2 Network Layer - Authenticated Perfect Links

For the network layer, we implemented UDP sockets that inherently allow message loss, duplication, and reordering and built a message resending mechanism to approximate fair loss links behavior. From here, we built higher-level abstractions that enforce Stubborn Link and Authenticated Perfect Link (APL) guarantees.

The guarantees provided include:

- **No Creation**: No message is delivered unless it was sent.
- **No Duplication**: No message is delivered more than once.
- **Reliable Delivery**: If a sender continuously retransmits a message, it will eventually be delivered as long as the receiver is valid.
- **Authenticity**: Delivered messages are verified to have been sent by their claimed sender, ensuring identity and integrity.

To manage these communication properties, we introduced an abstraction layer called `AplManager`, which provides a peer-to-peer interface while utilizing a `single underlying socket`. This design enhances scalability by eliminating dependence on the number of available ports on the machine whilst maintaining a simple one-to-one interface that is easy to use by external components. The `AplManager` maintains a list of `APLImpl` instances, which handle message transmission and reception through a predefined `MessageHandler` interface.

2.3 Consensus Layer - BFT Read/Write Epoch

Algorithm Overview The consensus layer implements a Byzantine Fault Tolerant (BFT) Read/Write Epoch Consensus algorithm. This algorithm operates in epochs, with each epoch having a designated leader. The consensus process consists of two main phases:

- **Read Phase**: The leader collects state information from other nodes to understand the current system state.
- **Write Phase**: Based on the collected states, the leader proposes a value, which nodes then vote on to reach consensus.

The algorithm is designed to tolerate up to f Byzantine failures in a system with $3f + 1$ total nodes. This ensures that even if f nodes behave maliciously, the system can still reach consensus on transaction ordering.

Epoch Change Mechanism Our implementation includes an epoch change mechanism that allows the system to progress when the current epoch cannot reach consensus. The `EpochConsensus` class manages this process by:

– Detecting when an epoch should be aborted (using abort counts)
– Broadcasting abort messages to all nodes
– Incrementing the epoch number and resetting relevant state
– Starting a new epoch with a new leader (determined by `epochNumber % NUM_MEMBERS`)

This mechanism ensures liveness by allowing the system to recover from scenarios where consensus cannot be reached in the current epoch. The implementation carefully preserves safety properties during epoch transitions by maintaining consensus state information across epochs.

Implementation Details and Design Decisions Our implementation of the Byzantine Fault Tolerant (BFT) Read/Write Epoch Consensus algorithm is structured to ensure reliability, fault tolerance, and efficiency. It is implemented through several key classes:

– `BFTConsensus`: Orchestrates the overall consensus process
– `EpochConsensus`: Manages the consensus logic for a single epoch
– `ConsensusState`: Maintains the node's view of consensus state
– `EventCounter`: Tracks message events (writes, accepts, aborts) from different nodes

The `EpochConsensus` class implements the core algorithm logic, including:

– Leader election based on epoch number
– Collection and verification of state information from nodes
– Determination of the value to propose based on collected states
– Voting mechanism for reaching agreement on the proposed value
– Abort handling and epoch change procedures

Each node waits for a quorum of responses (at least $2f + 1$ out of $3f + 1$) before proceeding to the next step, ensuring safety despite Byzantine failures. Messages are cryptographically authenticated to prevent forgery and impersonation.

Epoch Consensus Management The consensus process operates in a loop, represented in the class `BFTConsensus`, with each iteration representing an epoch. In this loop:

– the main thread waits to be awake either by new consensus message from another member or by new transaction from clients (both situations mean that a consensus must occur)
– if it doesn't have any value from the client it joins consensus with a null value, knowing that it will never propose a value in this consensus (but it's assured that someone has a value to be proposed, or else it wouldn't have been awakened)
– starts a new epoch consensus instance with a value or null.
– after the consensus has been reached and a value is decided, it removes the transaction from the received transactions queue and push it to the decided transactions queue to be processed.

`NOTE:` All the transactions already decided are stored in a `Hashset` so we know if a late arrival transaction was already decided.

Each epoch follows a structured approach, incorporating:

– Leader election using the formula `epochNumber % NUM_MEMBERS`, ensuring a round-robin selection.
– State collection and validation from participating nodes before proposing a value. (This is only done once per epoch, an optimization that avoids the read phase if we maintain the leader from the previous consensus)
– Write and accept phases, where nodes vote and finalize the consensus decision.
– Abort and epoch change mechanisms to recover from unreachable consensus states in the presence of byzantine processes.

The `EpochConsensus` class encapsulates the logic for a single epoch and manages communication, validation, and state transitions. When an epoch starts, if it receives an `AbortedSignal` exception, it is confirmed first that `2*ALLOWED_FAILURES` members also broadcasted abort, before proceeding to a new epoch. A epoch is a "infinit" loop, that only stops when a value is decided or an `AbortedSignal` exception is thrown. More in detail, this loop:

– defines two paths:
  1. `LEADER PATH:` starts the read phase if it is needed, broadcasting `READ` and receiving `STATE` messages. After this, creates the `CollectedStates` structure, verifying if every state received is well signed (if not, it is not added). Then, `COLLECTED` messages are broadcasted.
  2. `NON LEADER PATH:` receives the `READ` messages from the leader and sends it the `STATE` message, after signing the state. After this, it receives the `COLLECTED` message, and proceeds to verify the signature for each state.
– the two paths converge. From the collected states, a value to be written is decided.
– the `WRITE` messages are broadcasted and it waits until it receives a quorum of writes for the same value. In case this doesn't happen, it broadcasts aborts and throws an `AbortedSignal`
– after waiting, it writes the value and timestamp in its state as the most recent quorum written value, and broadcast the `ACCEPT` messages
– finally, it waits until it receives a quorum of accepts for the same value and returns the value itself. In case this doesn't happen, it broadcasts aborts and throws an `AbortedSignal`

`NOTES:`

– We have made an optimization, where we store the epoch number between different consensus so the last leader of the last consensus is the starter leader of the next one.
– During early phases, when receiving messages, we always verify if the member also receives writes, accepts or newepochs, and count them, so that if a quorum of them is received, we jump to the respective phase.
– We ignore all messages from the past or some unsynchronized ones.
– We have two queue of messages for each pair of members, where the exchange messages between them are placed.

Message Handling and Processing The system relies on message-based coordination between nodes. Key message types include: `ReadMessage`, `WriteMessage`, `AcceptMessage`, `NewEpochMessage`

Message passing Managed by the `ConsensusMessageHandler` class, ensuring correct deserialization, storage, and processing of messages received from other nodes.

State Tracking and Verification Each node maintains a `ConsensusState` object, which holds:

– The most recent quorum-written value (i.e., the value confirmed by a majority of nodes).
– A write set tracking values proposed in the past.
– A cryptographic signature ensuring integrity and authenticity.

Before deciding over the collected states, nodes verify state signatures using the pre-established Public Key Infrastructure (PKI). This prevents Byzantine nodes from injecting false states.

Failure Recovery and Epoch Changes To handle situations where consensus cannot be reached in an epoch, an abort mechanism is implemented:

– Nodes track abort counts using an `EventCounter`.
– If aborts exceed a threshold (`ALLOWED_FAILURES`), a `NewEpochMessage` is broadcast.
– Nodes increment their epoch counter and attempt a new round of consensus when another threshold is met (`2*ALLOWED_FAILURES`)

This ensures liveness by preventing nodes from remaining in a state where consensus cannot be reached.

Security Enhancements Security is reinforced through cryptographic measures, including:

– Digital signatures: We chose to implement the `Signed Conditional Collect`. So we have digital signatures on some consensus messages to prevent tampering of the states of each member.

2.4 Client Library and Blockchain Service

The communication between clients and the blockchain service follows a well-defined protocol that ensures security and reliability. The protocol defines two primary interactions: **client registration** and **append operation**.

Client Registration - When a client wishes to interact with the blockchain for the first time, it must register its identity:

1. The client generates a key pair and sends a `RegisterReq` message containing its public key and a unique client identifier to the blockchain service
2. The blockchain service stores the client's public key.
3. After registration, the client can begin sending append requests to the service

Append Operation - To append data to the blockchain:

1. The client creates an `AppendReq` message containing:
   - The data to be appended
   - The client's unique identifier
   - A sequence number for freshness
   - A digital signature created with the client's private key
2. Upon receiving an `AppendReq`, blockchain members:
   - Verify the client's signature using the stored public key
   - Check the sequence number to prevent replay attacks
   - If valid, propose the request for consensus
3. Once consensus is reached (which was detailed on section **2.3**, the leader includes the data in the blockchain
4. The client receives an `AppendResp` message containing:
   - A success/failure indicator
   - A timestamp of when the data was appended
   - The same sequence number from the request for correlation

3 Threat Model and Protection

3.1 Threat Actors

The main threat actors include **byzantine blockchain members** and **malicious network entities**.

3.2 Threat Categories and Mitigations

Network Communication Threats

- **Message interception:** is mitigated through session key encryption.
- **Message tampering:** is prevented using HMAC verification.
- **Replay attacks:** are addressed with sequence numbers and timestamps.
- **Message injection:** is prevented through cryptographic validation.
- **Man-in-the-middle attacks:** are prevented by PKI and secure key exchange.

Consensus Protocol Threats

- **Byzantine behavior**: Handled by the BFT read/write epoch consensus algorithm.

Client-related Threats

- **Client impersonation** is prevented through registration with public keys.
- **Request forging** by Byzantine nodes is addressed with digital signatures.
- **Replay attacks** are mitigated using timestamps in client requests.

3.3 Trust Assumptions

The system assumes the leader process is always correct, the PKI infrastructure is trustworthy, client library instances are trusted, and a sufficient majority of blockchain members are honest.

3.4 Security and Cryptographic Measures

DepChain's security architecture employs multiple cryptographic mechanisms to ensure confidentiality, integrity, authentication, and non-repudiation across both the network and consensus layers.

Network Layer Security At the network layer, we implemented a secure communication protocol built on top of UDP with the following security features:

- **Session Key Establishment**: When an Authenticated Perfect Link (APL) initiates communication with a new node, it establishes a secure session by generating a symmetric key, encrypting it with the recipient's public key (leveraging the pre-established PKI), and sending it to the destination node.
- **Message Integrity and Authentication**: All messages transmitted through an established link include an HMAC generated using the session key. This guarantees that messages cannot be tampered with during transmission without detection.
- **Freshness Guarantee**: Each message includes a sequence number that protects against replay attacks by ensuring that messages are processed in the correct order.
- **Receiving Mechanism**: Upon receiving a message, the APL verifies the HMAC and sequence number before delivering the message to the message handler. Then it sends an acknowledgment to confirm the successful receipt and verification.

Consensus Layer Security The consensus layer builds upon the secure network layer and adds additional security measures to ensure correct behavior of the blockchain system:

- **Client Registration**: Before submitting blockchain operations, clients must register with the system by sending a registration request containing their public key. This establishes a trust relationship between clients and the blockchain service.
- **Client Non-repudiation**: After registration, all client requests include a digital signature created with the client's private key. This provides non-repudiation, allowing blockchain members to verify the authenticity of client requests and preventing Byzantine nodes from forging requests.
- **Timestamp-based Freshness**: All application layer requests include timestamps to prevent replay attacks, ensuring that old requests cannot be resubmitted to the system.

4 Dependability Guarantees

4.1 Safety and Liveness

Our dependability guarantees are founded on the fundamental properties of distributed systems: safety ("nothing bad happens") and liveness ("something good eventually happens").

Safety Guarantees The safety properties of DepChain ensure that the blockchain maintains a consistent state despite Byzantine faults. The agreement property ensures that if two correct processes decide on values in the same epoch, they decide on the same value. This is achieved through our quorum-based voting mechanism requiring at least $2f + 1$ nodes to agree.

The validity property ensures that if all correct processes propose the same value v, then no value different from v can be decided. Our read phase enforces this by ensuring previously proposed values are considered before new ones are written.

Integrity is maintained by tracking decided values to prevent duplicate processing, while data authenticity is guaranteed through cryptographic signatures that prevent tampering.

These safety properties hold even with up to f Byzantine nodes in a system of $N > 3f$ nodes, thanks to our authenticated message passing and consensus implementation.

Liveness Guarantees DepChain's liveness properties ensure the blockchain progresses despite faults. The termination property guarantees every correct process eventually decides some value through our epoch change mechanism, which transitions to a new epoch with a different leader if consensus stalls.

Our system continues processing transactions even with up to f failed nodes, as long as $2f + 1$ correct nodes remain operational. This non-blocking progress is essential for maintaining availability in the presence of faults.

It's worth noting that our liveness guarantees rely on the eventual synchrony assumption, meaning the system will make progress as long as message delays eventually fall below an unknown but finite bound.

5 Conclusion

`DepChain` demonstrates the feasibility of building highly dependable permissioned blockchain systems that can operate correctly even in the presence of Byzantine faults. By implementing the Byzantine Read/Write Epoch Consensus algorithm with strong cryptographic protections, we've created a system that provides robust safety and liveness guarantees.

The layered architecture of our implementation—from the authenticated perfect links at the network layer to the consensus protocol and blockchain service—provides a solid foundation for dependable distributed applications. Our careful attention to security considerations, including threat modeling and corresponding mitigations, helps protect against both network-level attacks and Byzantine behavior among blockchain participants.

The system successfully handles epoch transitions when consensus cannot be reached, ensuring continued progress despite potential Byzantine leaders or network disruptions. By requiring agreement from a quorum of nodes for all decisions, `DepChain` maintains consistency while tolerating up to f Byzantine nodes in a system of $3f + 1$ total nodes.

`DepChain` serves as a practical example of how traditional distributed systems principles can be applied to blockchain technologies, resulting in systems that combine the benefits of distributed ledgers with strong dependability guarantees. This approach is particularly valuable for enterprise applications where both security and reliability are paramount concerns.

References

1. Anderson, R.J.: Security Engineering: A Guide to Building Dependable Distributed Systems. 3rd edn. Wiley (2020)
2. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to Reliable and Secure Distributed Programming. 2nd edn. Springer (2011)