Highly Dependable Systems Course - 2024/2025 DepChain Group 03

Guilherme Wind[1,2][ist1112176], Laura Cunha[1,3][ist1112269], and Rodrigo Correia[1,4][ist1112270]

[1] Institute Superior Técnico, University of Lisbon, Portugal
[2] guilherme.wind@tecnico.ulisboa.pt
[3] laura.r.velasco.cunha@tecnico.ulisboa.pt
[4] rodrigo.a.c.correia@tecnico.ulisboa.pt

1 Introduction

This report details **DepChain**, a project aimed at developing a simplified permissioned (closed membership) blockchain system with high dependability guarantees, implemented in Java. It assumes static membership and a pre-defined correct leader, eliminating the need for leader changes, while allowing all the blockchain members to exhibit arbitrary or Byzantine behavior.

In a brief way, our client sends an append request to all the processes, with all **communication** occurring over UDP, along with some abstraction upper layers to ensure that messages eventually arrive and that their integrity and authenticity are verified. Then, each blockchain member initiates an instance of **Byzantine Read/Write Epoch Consensus** and eventually either decides on a value or aborts.

As the clients have **accounts** in the system, they can perform **transactions** between them, which include transactions of a native coin called *DepCoin* or trigger transactions on smart contracts to exchange *ISTCoins* and manage the *Acess Control* to a *blacklist*, and the consensus algorithm will decide the order of the transactions that will be applied to all the members.

We also added a set of **tests** to assess various Byzantine behaviors, at the consensus, network abstraction layers, and transaction levels.

2 Design Explanation

In order to implement the *DepChain*, we made several design decisions regarding the architecture:

2.1 Network layers:

For the abstraction of the network layers, we implemented the following architecture:

- **Fair Loss Links (FLL):**
  A low-level abstraction of the UDP protocol. Since this protocol matches the specification of FLL, we simply initialize the UDP socket at this layer.

- **Stubborn Links (SL):**
  This layer is above the FLL, and every time it receives a message from FLL, it sets a timer that retransmits the message every 2 seconds indefinitely to the upper layer until it is signaled to stop.

- **Authenticated Perfect Links (APL):**
  This layer handles four types of authenticated messages: data messages, acknowledgment messages, handshake request messages, and handshake acknowledgment messages.
  Regarding the **handshake** between processes A and B, when any process creates an APL, it generates a Diffie-Hellman key pair. Then, A sends a handshake request through the APL, which delivers it to the lower network layers for B, containing A's signed Diffie-Hellman public key. When

B receives it, it verifies the signature using A's RSA public key and extracts the DH public key of A. If no symmetric key has been established yet, B generates one and always sends a handshake acknowledgment to A. A will keep retransmitting the handshake request via the stubborn layer until it receives an acknowledgment. If node A does not yet have the shared secret key and wants to send a message to B, the message will be buffered until the key is available, at which point all buffered messages are sent to avoid discarding them. Acknowledgments and messages contain a MAC and are always verified by the receiving process, using the symmetric key established through DH. This process is also performed symmetrically by B for A, ensuring that both share the same secret key according to the DH algorithm.

Regarding **messages that are not handshake-related**, they behave similarly. Since we already have the shared secret key from the handshake, when a message is sent, a MAC is calculated. Upon receiving a message, the APL verifies the MAC and then sends an acknowledgment with the same sequence number as the received message to confirm receipt. The message will continue to be retransmitted until an acknowledgment is received, and it signals to stop the retransmission at the stubborn layer.

These **decisions above** were useful since the **handshake** is needed to establish a symmetric key for the MACs. Furthermore, the **ACK system** is justified as it prevents resource waste, otherwise, the stubborn would keep retransmitting indefinitely without knowing when to stop.

2.2 Consensus:

To implement the consensus, we made some modifications to the algorithm in the book [1]:

- **Conditional Collect (CC):**
  We implemented the conditional collect algorithm with signatures. The reason for this is that if the collected states were not signed, the leader would receive all the states and then forward them to all processes, and if the leader were Byzantine, it could forge the states of the processes and manipulate the values, thereby influencing the majority and ultimately compromising the decision made by the Byzantine consensus.

- **Byzantine Consensus (BC):**
  For the consensus, we followed the algorithm described in the book [1], but we slightly modified the predicate condition to prevent always choosing the value proposed by the leader. At the beginning, since we have no “history" from previous epochs (as we start in epoch 0), all pairs in the collected list and all write sets are initially set to null. As a result, both conditions must be met for a value to be decided by the majority. However, because the leader is only required to send $n + f/2$ values from the collected set and may not always reach a strict majority of $2f + 1$, the default value would end up being the one proposed by the leader. If the leader were Byzantine, this could lead to manipulation, so in order to mitigate this, the clients sign their requests and verify the signature.

  The timeout is set at the beginning of the consensus instance when it receives the client's request. If it reaches 1 minute and 30 seconds, it triggers a complaint, leading to an abort, which, if implemented, would trigger a new leader election.

  It is also important to mention that for f faults, we allow connecting a minimum of $3f + 1$ **nodes** to power up the system.

  We also added a **small alteration to the algorithm**, explained in subsection 2.4, to include a list of node signatures in the block.

Note: Only messages from the Conditional Collect and client requests/transactions are signed.

2.3 Client-node communication:

To implement the communication between the client and the node, every time a client wants to send a **request for a transaction**, they will choose through a terminal menu which kind of transaction they desire, that could be an external or a contract one. Then, sends a message with the **transaction signed** (in order to ensure the integrity, authenticity, and non-repudiation of it) to the blockchain, which is a list of blocks managed by various processes that are members of the blockchain. The request message is sent through a **point-to-point APL connection** established with each of the members of the blockchain.

We chose this approach because if the request were only sent to the leader, and if, by chance, this was Byzantine, then it could alter the values. That doesn't occur as we are ensuring that all nodes are aware of the value requested by the client, as mentioned in the Byzantine consensus topic above.

2.4 Blocks and Blockchain

The **blocks** of the blockchain are **persisted on disk in a JSON format**, so if the system goes down or if a node joins later, it can retrieve the overall state, and when a node connects, it reads the blocks from disk to obtain the system state.

Each **block** contains:

– **Own hash:** Used to identify the block, and it's generated from the hash of the block's transaction list.
– **Hash of the previous block:** To know the order of the block in the chain.
– **List of transactions:** That contains the transactions of the block. For testing purposes, we restrict each block to store a maximum of one transaction, but this value could be arbitrary.
– **List of signatures:** To validate the block, avoiding Byzantine nodes to submit "false" blocks.

So, when a node receives **transactions**, it waits until it has **accumulated a predetermined number** of them. Once this threshold is reached, it **creates a block, signs the two hashes and the list of transactions, adds the signature to the block, and sends** it to the consensus process.

Then, the block **validation** is performed by the **predicate** C of the consensus algorithm. As **blocks enter the consensus** process, the algorithm validates each block by first checking whether the transaction list hash is correct, followed by verifying the signatures of the transactions and the signatures of the nodes included in the block's list. If all are valid, the block is approved/decided, the **transactions** can be executed (all in the **same order for all the nodes**), and the block can be appended to the blockchain. When the **transactions are executed**, we update the system state, which is **persisted** in a JSON file.

It's important to note that, to perform the **signatures of the nodes**, we needed to slightly **modify the consensus algorithm**.

Now, as the blocks have **signatures** from the **nodes** that participated in the consensus to create the block, knowing that to validate it, it would only be necessary to verify if it contains at least $f + 1$ signatures, since a single correct node signing the block would be enough to ensure it is not Byzantine (although it will have $2f + 1$ signatures, as this comes from the WRITE phase, which has a threshold of that value).

Each node, when sending the WRITE value, signs the block along with its fields/content. Before sending an ACCEPT message, all **signatures are merged into the block**, thereby finalizing a block with signatures from multiple nodes.

2.5 State

The **state** contains all the information of the accounts and is loaded from the **genesis** file, and when it is updated, the **storage**, which contains the values of contract variables, is also updated. This last is only present in **contract accounts** as they have code, balance, and storage, whereas **external accounts** only have a balance.

As the **state** contains the **storage**, both are also persisted in a **JSON file** that is **continuously updated**. We decided that there is no need to generate a new file for each state, instead, we keep updating the same JSON file, as a node starting from the *Genesis* initial state, it is able to execute all the transactions from the blocks and **reconstruct the state at any point in the timeline** up to the current state.

2.6 Accounts

The **account model** follows *Ethereum*'s model, meaning there are **two** types of accounts:

– **External Accounts:** These are client accounts that interact with the blockchain and hold a balance.

– **Contract Accounts:** These are smart contract accounts that have associated storage, code, and a balance.

2.7 Transactions

There are two types of transactions in the system:

– **External Transactions:** Allows the clients to transfer some balance of *Depcoin*, and send the transaction through the network to transfer the local currency.
– **Contract Accounts:** When the client wants some transaction from a contract, it parses the function name and parameters into hexadecimal using the *Keccak* hash (as used by the EVM), and sends it through the network.

Each node also maintains a list of account addresses with associated **nonces** to prevent clients from submitting the same request twice, ensuring **freshness**.

2.8 Smart Contracts

We implemented two independent smart contracts in **Solidity**, executed using **Besu** [2], which provides an implementation of an EVM:

– **ISTCoin Contract:** This contract extends **ERC-20** [3] and is based on an OpenZeppelin [4] implementation. It allows the clients to manage the **ISTCoin** currency, so we decided to allow the client to execute functions for the name, symbol, decimals, total supply, balance, transfer, transfer from, approve, and allowance. It's also important to refer that both transfers can only be executed if the sender is not on the **blacklist**.

– **AccessControl Contract:** Allows clients to add, remove, and verify whether a user is on the **blacklist**. The first two functionalities can only be performed by **administrators**, who are defined statically in the genesis, determining whether they can perform ISTCoin transactions.

Each transaction, whether from the smart contract or external sources, allows the **client to see the reason when the transaction fails.**

Note: The contracts were **implemented separately without using the simplification** of putting them together into a single file.

2.9 Static Information Script

We implemented another Java project to generate **static information**, as the **ECC keys for the clients, the server, and the genesis.**

The **genesis** contains the first block, which includes an initialization with all account addresses, their balances, and smart contract addresses, including their bytecode compiled for the EVM and initial key-value storage values.

We chose **ECC** over RSA keys for simplicity, as it allows us to sign client transactions and later recover the public key from the signature to verify it, avoiding the need for a mapping between account addresses and public keys.

Here, we also initialize a single account with 100 million units of ISTCoin, which will later be distributed across multiple wallets as transactions occur.

3 Threat Analysis and Protection Mechanisms

Having in mind possible threats to the system, we implemented several protection mechanisms to address them:

- **Signed Conditional Collect:** To prevent the leader from forging the triplets of the collected values when sending the `COLLECTED` during the read phase of the consensus to the blockchain members, as it could modify the values provided by others to influence the final decision, we used signatures, to ensure authenticity and integrity, so any tampering by the leader would be detected.

- **Authenticated Perfect Links with MACs:** To prevent nodes from impersonating others, we add a Message Authentication Code (MAC) that relies on a shared symmetric key established via Diffie-Hellman, with this, we ensure both the authenticity and integrity of the data and its origin.

- **Signed Handshake of APL:** To prevent a node from impersonating another during the handshake phase when establishing the secret key, the exchanged Diffie-Hellman values are signed, being protected against man-in-the-middle attacks.

- **Signed State Response:** To prevent nodes from impersonating others when sending results to the client, the responses are signed. Without this, a malicious node could send multiple responses to manipulate the quorum and decide the outcome arbitrarily, so with signatures, we ensure both integrity and authenticity.

  List of node signatures in the block: As a Byzantine node can submit a “fake” block even with a correct hash of the transaction list and valid transaction signatures, we add a list of node signatures to the block. Now, we can prevent these Byzantine blocks, as this list is also checked in predicate C.

- **Timeout for Consensus Nodes:** To prevent response omission by the leader, if a node does not receive a reply within a predefined timeout, it triggers a timeout complaint and subsequently aborts the process.

- **Replay attacks:** To prevent replay attacks, the node also maintains a map of account addresses with the associated nonces for each request made by that account, ensuring that a client cannot send the same request twice, maintaining freshness.

- **Transaction requests signed:** To prevent client impersonation and ensure non-repudiation, authenticity, and integrity of transactions/ requests, all transactions are signed. Additionally, since the signature includes all transaction parameters, including the nonce, any attempt to manipulate it will result in verification failure.

- **Accounts authorization:** Only authorized users can perform transactions with the *ISTCoin* currency, as it checks if the sender of the request is not in a blacklist. Additionally, only users with an account in the system, and consequently the corresponding private key, can perform operations in the system.

- **Avoid negative balance:** To prevent an account from having a negative balance, we always check whether there are sufficient funds before processing a transaction.

4 Dependability Guarantees

With our implementation, we address some dependability guarantees:

- **Fair Loss Links:**
  - This layer guarantees, through UDP, that even though messages may be lost, if a message is sent infinitely, it will eventually be delivered, as there is no infinite packet loss.
  - A message may be delivered more than once, but only a finite number of times.

  - No message will be created spontaneously, being that only the sent messages can be delivered.

- **Stubborn Links:**
  - In this layer, if a correct process sends a message to another correct process, the latter will keep retransmitting it indefinitely over the FLL.
  - We add an optimization so that it stops when it receives a signal, to avoid wasting resources. Additionally, no message is received unless it has been sent, as defined by FLL.

- **Authenticated Perfect Link:**
  - In this layer, any message sent between two correct processes, as long as it is sent by one, will eventually be received by the other, as it is built over SL as described above, where the message keeps retransmitting indefinitely until it eventually reaches the destination.
  - No message is sent to a process more than once, which we ensure through the message sequence number, that is stored in a set that checks if the message has already been sent; if so, it only processes the acknowledgments, otherwise, it sends the message.
  - When a message is received from a correct process, sent by another correct process, it guarantees that it is from the intended sender, this is ensured by the MAC, which has a handshake with a symmetric key known only between these two nodes.

- **Byzantine Consensus and Signed Conditional Collect:**
  As we implemented the algorithm following the book's implementation, with the added **variant** that here there are **clients proposing values instead of the leader**, so we achieve the same guarantees as the ones shown by the book [1] and the lectures:

  Note: The value is a block with a bunch of transactions.

  - If all processes are correct, a process in the current epoch decides a value that was proposed by the leader of a previous or current consensus epoch.
  - Two processes in the same epoch do not decide different values.
  - If a correct process has already decided a value in a previous epoch, a process in the current epoch cannot decide a different value.
  - If the leader is correct and proposes a value, and no process aborts the epoch, every correct process will eventually decide on a value.
  - The value must have been written by at least a quorum of f+1 processes, which ensures that it was written by at least one correct process, ensuring overlap with the majority of correct processes.
  - The decided value is the most recent one among those that were written.
  - All processes independently check the collect to verify if a value has already been decided in a previous epoch.
  - The value chosen in the collect has not been tampered with by any process due to the conditional collect signatures.
  - If there is no progress in the current epoch, it triggers a timeout, processes will start a new epoch, and if changing epoch would be implemented.

- **Smart Contracts:**
  - Only admins can add and remove clients from the blacklist.
  - Only clients who are not on the blacklist can perform ISTCoin transfers.

- **Transactions:**
  - Transactions cannot be repudiated.
  - Transactions cannot be replayed.
  - Transactions resulting in a negative value are not executed.
  - All nodes execute transactions in the same order.
  - If a block is not validated, none of its transactions are executed; all correct nodes either execute the same transactions or none at all.

  - A block can only be appended if it is validated with the correct hash and transaction signatures.

– **Persistence:**
  - If the system goes down, block persistence ensures availability and allows recovery.
  - If a node joins, it retrieves the current state.

– **System:**
  Our system guarantees safety by always deciding the same value, and liveness by ensuring that something happens, even if it's just a timeout.

5 Tests

In order to test our system, we implemented the following tests that address some Byzantine behaviors:

– **No Response:** In this test, the node is configured not to respond to messages, it will accept messages (sending ACKs) but will not send any further replies to make progress in the consensus algorithm.

– **Crash:** The node starts and after 1 minute crashes. It behaves the same as the previous test, but does not send ACKs.

– **Different Write:** This node is configured to use a different write value for Byzantine behavior. So, the algorithm needs to deal with different writings and check if the quorum is still being achieved.

– **Different Accept:** This node is configured to use a different accept value for Byzantine behavior. It behaves the same as the above, but for the accepted value.

– **Different Client Value:** This node is configured to submit different client requests as Byzantine behavior. It will use a different string than the one requested in the append by the client, which will lead to different results, similar to the last two tests above.

– **Replay Attack:** The client has Byzantine behaviour, resending the last sent transaction request.

6 Conclusion

Finally, we conclude that our implementation meets all the required specifications, as we successfully implemented static membership and Byzantine behavior under an unreliable UDP-based network, allowing the clients to perform transactions of currencies.

The layered network abstractions, with Fair Loss Links, Stubborn Links, and Authenticated Perfect Links, provide a modular and resilient communication framework, ensuring reliable message delivery despite potential losses. Additionally, the client can perform external transactions using a native coin or execute remote code on an EVM through smart contracts. The Byzantine consensus algorithm guarantees that transactions are executed in the same order across all nodes, as the blocks have a determinate number of transactions, ensuring a consistent global state in the system. This state (that includes the storage) and the blocks are persisted on disk, providing availability.

Additionally, cryptographic enhancements, including signed conditional collects, the transactions messages, and the content of the block, MAC-verified messages in the APL layer, and secure handshakes, strengthen the system against threats such as forgery, impersonation/ man-in-the-middle attacks, and replay attacks, as detailed in our threat analysis.

Regarding the testing and validating the effectiveness of these mechanisms, we conclude that Byzantine leader and client forgery were thwarted by signatures, network unreliability was mitigated by stubborn retransmissions, client-node communication integrity was maintained through the point-to-point connections between the nodes that were used in a way to broadcast the append requests, and replay attacks were mitigates through nonce freshness in the transactions.

Dependability guarantees, including safety, as there's no conflicting decisions among correct processes, and liveness address as eventual message delivery between correct processes, are upheld, contingent on the assumption of a pre-defined correct leader. However, the lack of epoch transitions limits full liveness in failure scenarios, highlighting a key area for improvement in future work.

In summary, *DepChain* establishes a currency transaction system based on a dependable consensus layer that balances security, integrity, and performance, laying a solid foundation for future enhancements.

References

1. Introduction to Reliable and Secure Distributed Programming. 2nd Edition.
2. Besu Ethereum Client. https://github.com/hyperledger/besu/tree/main
3. ERC-20 Token Standard. https://ethereum.org/en/developers/docs/standards/tokens/erc-20
4. OpenZeppelin – ERC20 Implementation.
   https://github.com/OpenZeppelin/openzeppelincontracts/blob/master/contracts/token/ERC20