HDS Project Report Stage 2

Group 28

Eduardo Carvalho *IST 110828*
Rafael Alves *IST 99308*
Rui Pires *IST 95670*

1 Introduction

In this stage we will expand on the work completed in the first stage and improve HDS^2 dependability assurances and application semantics. In this stage, our goals are to:

- Develop a cryptocurrency application for client transfers.
- Enhance the system's Byzantine fault tolerance, including handling Byzantine [1] clients.
- Verify the implementation's correctness, performance, and resilience to attacks, ensuring a single malicious replica can't significantly slow down the system.

We'll maintain the simplifying assumptions from the first stage regarding static, pre-known participants and their details.

2 System Design

The system consists of a de-centralized ledger system, capable of executing transactions between different client accounts.

The system consists of a predetermined static set of x nodes denoted by $\mathcal{N} = \{n_1, n_2, \ldots, n_{\mathrm{x}}\}$ and a predetermined set of y clients denoted by $\mathcal{C} = \{c_{\mathrm{x}+1}, c_{\mathrm{x}+2}, \ldots, c_{\mathrm{x}+\mathrm{y}}\}$. Therefore, every node or client is a process $p_i \in \mathcal{P} = \mathcal{N} \cup \mathcal{C} : \; i \in [1, \mathrm{x}+\mathrm{y}]$

The algorithm is deterministic and relies on a leader-based approach. It achieves optimal resilience by tolerating up to f faulty nodes out of x, where $\mathrm{x} \geq 3f + 1$. It also is resilient against faulty clients, meaning that a client can only perform transactions it is authorized to.

The communication between components of the system occurs via UDP, enabling the implementation of communication channels as perfect links. This approach ensures that messages are eventually delivered, delivered only once, and delivered only if they were sent, thereby enhancing the reliability and integrity of the communication process

2.1 System Assumptions

Assumption 1: The system can support up to f nodes who don't follow the protocol in a correct way (Byzantine-behaving nodes).

Assumption 2: The system membership remains static throughout the entire lifetime of the system.

Assumption 3: The application can rely on its local instance of the client library.

Assumption 4: The network is unreliable. We can have message dropping, corruption, duplication or delay. The communication channels of the application lack security measures.

Assumption 5: Each process of the system has a pre-generated key Pair (public and private key). These keys are distributed in advance of the system's initiation. Each process has access to all system keys. However, we assume that a process p exclusively utilizes its own key pair and abstains from accessing other private keys, limiting its interaction to the public keys of other processes.

Assumption 6: The transfer operation exclusively deals with integer values for the transferred amount.

2.2 System Guarantees

Guarantee 1: Each account's balance is non-negative at all times.

Guarantee 2: The accounts current status is protected against unauthorized user alterations.

Guarantee 3: Non-repudiation is ensured for all operations performed on an account

3 Implementation Aspects

3.1 Communication

The implementation of authenticated messages is done with a wrapper class *AuthedMessage* that includes all the necessary tools to verify its integrity, non-repudiation and authenticity.

This class also takes care of not including the piggybacked messages while computing the hashing/signature. The Authenticated Perfect Link is achieved by using these authenticated messages on top of the existing perfect link.
With this, it is guaranteed that every message m transmitted from a process p_i to a process p_j is signed by p_i by having a signature Cipher($priv_i$, $Hash(m)$) appended.

3.2 Client

A conforming (non-faulty) client c_i broadcasts a request

$$\text{request}_i^k \in \{\text{<TRANSFER, } pub_{source}\text{, } pub_{dest}\text{, amount>,}$$
$$\text{<CHECK\_BALANCE, } pub_{account}\text{>}\}$$

to every $n_j \in \mathcal{N}$ and proceeds to wait for $f+1$ identical responses r_j^k different nodes, where amount$\in [0,\infty]$, *pub* represents a public key and k represents the request's id, used to distinguish between a client's requests.
Because the client stops listening for responses after accepting a result, there will be some nodes hanging on trying to respond to the client (due to the stubbornness of the perfect link). This is fine as these hanging responses will eventually be consumed on further requests (when the client starts listening to responses again) and will be ignored due to being outdated (by checking their request id).

3.3 Transfer

When a TRANSFER request is received by the nodes, a consensus procedure is instantiated following the IBFT [2] protocol. The procedure may traverse various rounds, starting at round 1. Given a certain round of a consensus instance, the a node n_j is considered the leader of the round if:

$$j = (\text{round MOD x}) + 1$$

The leader of a round r is responsible for proposing the requested transaction to the other nodes and following along the protocol to eventually decide on executing the transaction. For cases where the leader for a given round fails to conclude the consensus instance after a certain amount of time, each node will have a timer dedicated to each running consensus instance. Following a timeout, a procedure to advance to another round is triggered. The protocol ensures that, at the end of a consensus instance, every correct node will agree on the same order of transactions.
Upon deciding, the nodes reply to c_i with

$$\text{reply} \in \{\text{<TRANSFER, success>,<BALANCE, balance>}\}$$

where success $\in \{\text{TRUE, FALSE}\}$ and balance $\in [0,\infty] \cup \{-1\}$. A balance of value -1 means that the request failed.
Every consensus message transmitted between nodes includes the originally signed client message that triggered the consensus. This ensures that every node can check if the consensus messages are valid to an actual client request and can check if the client issuing the transaction is the owner of the source account.
A consensus instance $i \neq 1$ may not be decided until the instance $i-1$ is decided. This means that after all the different stages of the protocol have been completed, the processing of a waiting instance will stop at the commit stage until the previous one finishes.
The account is only checked for enough balance when a transaction is about to be decided and has been agreed by consensus. If the request is not possible or has been decided in a previous instance (by checking the request id k), an empty transaction object is appended to the ledger (symbolizing a discarded consensus instance).

3.4 Balance

When a CHECK_BALANCE request is received by the nodes, a consensus is triggered to make sure all correct nodes agree on an account state. A way to do this could be to have the nodes just replying with the balance on the local account, but there would be no guarantees that $f+1$ nodes would agree on the same value, as the individual states may not reflect the system state.

3.5 Generated Keys

The default path for generated keys is */tmp/test/keys/*. Each process gets a folder with a *private.key* and a *public.key*.

4 Evaluation - Behavior under attack

In this section we will analyze our Test Suite and understand how the HDS^2 system handles various scenarios.

We extensively validated the HDS^2 system through a comprehensive battery of tests, covering a wide range of scenarios. Our test suite is categorized into four distinct areas:

Test Category	**Description**
Communication Test	Evaluates the functionality of the AuthedMessage component.
HDS^2 System Test	Evaluates the core functionality of the HDS system under normal operating conditions.
Byzantine Nodes Tests	Tests the system's resilience when blockchain nodes exhibit Byzantine behavior.
Byzantine Clients Tests	Tests the system's resilience when clients exhibit Byzantine behavior.

Table 1: Test Suite Categories

4.1 Communication Test

Authed Message Verification: This category only has one test to verify the correctness of the creation of an authenticated message by checking its signature.

4.2 HDS^2 System Tests

This category is designed to evaluate the normal functioning of our system.
1. Client transfers a valid amount and then checks balance.
2. Client performs a transfer then checks balance.
3. Client performs various transfers with valid amounts.
4. Client tries to transfer more than available balance.
5. Client tries to transfer a negative amount.
6. Client tries to transfer to his own account.
7. Concurrent clients.

4.3 Byzantine Nodes Tests

In this section and in section 4.4, and because they are the byzantine sections, we will provide detailed explanations of how the system will respond to each test scenario.
This category will test our system in scenarios where blockchain nodes deviate from the protocol, exhibiting Byzantine behavior.

1. **Leader sleeps:** By forcing the leader to sleep and due to our timer implementation, when the timer eventually expires, the correct nodes will automatically broadcast a ROUND CHANGE message. This action forces the round to change, resulting in the election of a new leader.

2. **Leader starts consensus without client request:** By always verifying the messages being transmitted, every correct node will identify that the appended client signature is wrong.

3. **Leader ignores requests from specific client:** Even though the original leader ignores a client's request, the correct nodes will receive this request and eventually issue a round change when their timer expires.

4. **Leader alters transaction data:** Leader changes senderId of transaction but the original version is still applied thanks to the correct nodes.

5. **Leader overcharges fees:** Despite the leader's attempt to overcharge client fees, this will not have an effect on the overall system state due to the presence of at least $x - f$ nodes that agree on the correct balance for the client. Consequently, when the client executes the check balance operation, it will return the correct value, unaffected by the leader's attempted fee manipulation.

6. **Leader starts two consensus for one request:** If two consensus instances start for the same $request^k$, the second instance will fail due to the correct nodes verifying that k was already decided.

7. **f Nodes respond immediately to request (without doing consensus) - excluding leader:** Because the client waits for $f+1$ identical responses, and given that we have at most f faulty processes, there is a guarantee that the client will never accept a response value that has not been returned by at least one correct node. The consensus is still triggered by the remaining nodes and a correct response is eventually reached.

8. **f Nodes respond immediately to request (without doing consensus) - including leader:** The same as in item **6** but the correct nodes will trigger round changes due to no progress being made with the first leaders, eventually landing on a correct node as a leader and completing the consensus.

9. **f+1 Nodes respond immediately (without doing consensus) to request:** This test simply verifies that a client will accept a false value if (by definition impossible) $f+1$ nodes behave maliciously.

10. **Leader starts consensus then goes to sleep:** With no more progress being made with the leader, a round change is triggered and the consensus continues with another leader.

11. **f nodes don't respond on PREPARE:** Due to the protocol nature, everything still works even with f nodes not participating.

12. **Leader doesn't respond on PREPARE:** Because the leader fails to make progress, a round change is triggered and consensus continues with another leader.

13. **Leader doesn't respond on COMMIT:** Leader fails to make progress but a value has already been decided, a round change is triggered and consensus continues with another leader.

14. **f nodes send duplicate reply:** The client checks that a node i already responded to $request^k$ and ignores subsequent responses from that node.

4.4 Byzantine Clients Tests

This category will test our system in scenarios where clients deviate from the protocol, exhibiting Byzantine behavior.

1. **Client execute transactions pretending to be other clients:** This attempt will be unsuccessful because all

messages are protected by digital signatures. The authenticity of each transaction is confirmed through signature verification processes. Furthermore, every consensus message has the signed message, allowing each node within the system to validate if the signature truly matches the identity of the sender.

2. **Client attempts DoS attack with timestamp bypassing:** The HDS^2 System alone already discourages DoS attacks due to the fee that each client has to bear if they want to carry out a transaction, which makes most DoS attacks unfeasible. Besides this, we implemented a cooldown feature that only allows clients to generate new requests after a determined time interval (half a second). All solicitations during the timeout window are discarded.

3. **Client sends malformed requests:** Every time a node receives a message it verifies its integrity. In the event of a malformed message the correct nodes respond with failure.

4. **Client tries to send money from another account to his and leader node ignores the message verification:** The leader node starts a consensus with a bad request from the client but the correct nodes check the validity of the client message (which is appended to every consensus message) and ignore both the client request and the leader pre-prepare messages.

5. **Client sends different requests to each node:** The assigned leader for the first round will start a consensus with the request that it receives. As long as this process doesn't stall, that specific request is accepted. In the case that the timers expire and a round-change is triggered, another different request may end up being accepted, but this is still considered normal behaviour as it still is a client-made request after all.

6. **Client sends different requests to each node and the first leader sleeps:** As explained in the last test, with the first leader sleeping, it is the request sent to the leader of the second round that gets accepted. This is considered normal behaviour.

7. **Client sends a request only to the leader node:** The leader starts a consensus and execution follows as expected.

8. **Client sends a request only to a non-leader node:** The non-leader node is unable to issue a round-change and thus nothing happens. This specific scenario is assumed to never occur, as it is part of the system design that a client is aware that sending only a request to one node may not be enough.

With this test suite we can assure that the HDS^2 System is protected against a diverse range of popular attacks, as demonstrated in the table below.

Attack	**Protection Mechanism**
Non-Repudiation	Signed messages verification
DoS	Cooldown feature based on message timestamps
Message Tampering	Signed messages verification
Replay Attacks	Requests have an unique id

Table 2: Protection Mechanisms Against Popular Attacks

5 Miscellaneous

In this section, we will provide an overview of several components within our project.

- The start amount for each client account is 10000 and the fee value is 10.

- To streamline the creation of configurations for our various tests, and to ensure unique port assignments, we developed a Python script (*create_config.py*). This script generates configurations dynamically, incrementing ports based on a previous configuration.

- Accounts are created for the nodes of the blockchain system to facilitate the receipt of fees when they assume the role of leaders. These accounts serve the sole purpose of receiving these fees and are not utilized for any other function within the system.

- The timer used in Round changes expires after two (2) seconds.

References

[1] LESLIE LAMPORT, R. S., AND PEASE, M. The Byzantine Generals Problem.

[2] MONIZ, H. The Istanbul BFT Consensus Algorithm.