HDS Serenity Ledger

Daniel Pereira *99194*
Ricardo Toscanelli *99315*
Simão Gato *99328*

Abstract

This report describes HDS Serenity (HDS2), a permissioned blockchain designed for secure financial transactions. Section 1 details the system's core design principles. Section 2 explores the key implementation aspects that contribute to HDS2's dependability. Finally, Section 3 presents a experimental evaluation demonstrating the system's effectiveness in handling Byzantine faults.

1 System Design Overview

Our system follows a layered architecture designed for scalability and fault tolerance. The key components are:

1. **Client Application Interface:** This user-facing layer acts as the entry point for user interactions. It captures user requests and transmits them securely to the Client Service.
2. **Client Service:** This background service acts as a mediator between the Client Application and the server-side logic. It receives requests from the Client Application, performs signature validation (using cryptographic techniques), and broadcasts the message to the relevant server-side service.
3. **SerenityLedgerService:** This core server-side service receives messages broadcasted by the Client Service. It acts as the central coordinator, orchestrating the overall message processing flow. It prepares the data based on the received message and interacts with the Node Service to retrieve the consensus value.
4. **Node Service:** This specialized service encapsulates the logic for reaching consensus on a specific value. It interacts with SerenityLedgerService to receive the prepared data and leverages a defined consensus mechanism (Instambul Byzantine Fault Tolerance) to reach an agreement with other nodes. If consensus is achieved, the resulting ledger state is returned to the SerenityLedgerService.
5. **Communication Flow:** The communication primarily follows a client-server model. User interactions initiate at the Client Application, which transmits requests to the Client Service. The Client Service broadcasts the message to the designated SerenityLedgerService on the server side. SerenityLedgerService then interacts with the Node Service to reach consensus on a value. Finally, the agreed-upon ledger state is potentially relayed back to the Client Service for further processing or user notification.

This layered architecture promotes modularity and separation of concerns. Each layer has a well-defined responsibility, improving maintainability and promoting easier integration of future functionalities. The use of a dedicated Node Service for consensus allows for flexibility in exploring different consensus algorithms depending on the specific needs of the system.

2 Relevant Implementation Aspects

This section highlights some key implementation aspects of our system:

2.1 IBFT

We leverage the Istanbul BFT (IBFT) protocol for consensus, accordingly to the official IBFT paper [1], ensuring fault tolerance even in the presence of Byzantine failures (nodes exhibiting arbitrary behavior). The implementation has been thoroughly reviewed and corrected since the first stage of the project, having the Round Change mechanism totally implemented and a mechanism to synchronize nodes that are behind the others.

2.2 Triggered Consensus

The IBFT protocol execution is optimized to initiate only when a specific number (X) of requests are queued. This

number can be changed in the TransactionQueue class in the blockSize attribute (5 is the default size). This reduces unnecessary consensus rounds and improves system efficiency. During the commit phase upon reaching a "DECIDE" state, the participating nodes process the requests accumulated in the queue, by removing them from the queue and executing them.

2.3 Parallel Consensus

Our system enables concurrent execution of multiple IBFT consensus instances. However, to maintain data consistency, ongoing instance Y waits for the completion of instance X (where X precedes Y) if transactions within Y depend on the outcome of transactions in X. This parallel execution can introduce potential complexities if such dependencies exist between concurrently processed requests.

2.4 Ledger Design

The system maintains a ledger consisting of a blockchain structure where each block stores a predefined number (X) of transactions. Additionally, the ledger holds the current state of all accounts within the system.

2.5 Client-Side Verification

When a client queries their account balance, they must receive confirmation from 2f+1 nodes to ensure the retrieved value's accuracy. This requirement arises due to the presence of account states within the ledger. While using a system like UTXO could potentially offer advantages, time constraints during development prevented its implementation.

2.6 Meeting Application Requirements

Our system design effectively meets the following application requirements:

- **Non-Negative Account Balances:** The system enforces non-negative account balances by rejecting any transaction attempting to spend more money than the account's current balance. Clients attempting such transfers will receive an error message.
- **Account State Protection:** Unauthorized users cannot modify account states. Each transaction requires a digital signature, ensuring only the legitimate owner of an account can initiate transfers. This digital signature acts as an authorization mechanism, preventing unauthorized access and modifications.
- **Non-Repudiation of Operations:** We achieve non-repudiation through a combination of digital signatures and nonces. Each transaction is accompanied by a digital signature that binds the operation to the account owner. Additionally, we use a nonce, which is an incrementing number for each request sent by a client. The server verifies that the received nonce is greater than any previously received nonce for the same client. This prevents replay attacks and ensures that the client cannot repudiate a previously authorized transaction.

These features, combined with the Byzantine Fault Tolerance provided by the IBFT protocol, guarantee the integrity and security of the system's state, upholding the required application-level properties.

2.7 Transaction Fees and Incentive Design

This subsection addresses the introduction of transaction fees within our system, a common feature in modern blockchains. Transaction fees serve a dual purpose:

- **Discourage Spam:** A small fee per transaction discourages malicious actors from flooding the network with irrelevant transactions, thereby protecting system resources and maintaining network efficiency. This concept is similar to "gas" fees used in blockchains like Ethereum.
- **Incentivize Block Production:** Transaction fees provide an economic incentive for nodes to participate in the consensus process (e.g., block production in Proof-of-Work systems or leader election in IBFT). These fees reward nodes for the computational power and resources they contribute to secure the network.

Fee Structure Considerations

While the specific fee structure chosen for a system can be flexible, we decided to use a fee percentage of 5% because this is a small environment compared to the real blockchain systems. A common approach is to set the transaction fee as a percentage of the transferred value. A more balanced approach could be a fixed fee (e.g., 1 unit of the coin) for simple transactions and a variable fee based on transaction size or computational complexity for more resource-intensive operations. This aligns with the fee structure used in blockchains like Solana, where fees depend on the computational resources required for transaction processing.

Fee Distribution

The collected transaction fees can be distributed in various ways. One option is to allocate them entirely to the leader node responsible for processing the block. Alternatively, a portion could be distributed among participating nodes in the consensus process, further incentivizing their involvement. In our system implementation, no node receives the fee, so

the cash just "disappears". Would need to be corrected in the future.

Justification and Trade-offs

The chosen fee structure needs to balance several factors:

- Discouraging spam requires a sufficient fee to make it economically unviable for malicious actors.
- Keeping fees too low might lead to network congestion.
- Setting fees too high could hinder user adoption and limit system usability.

3 Behavior under attack

This section evaluates our system's dependability under Byzantine faults, where nodes (clients or servers) exhibit arbitrary and potentially malicious behavior.

3.1 Experimental Setup

We leveraged the configuration JSON file to introduce Byzantine behavior. Both clients and servers had a dedicated field within the JSON configuration specifying their Byzantine behavior mode.

3.2 Client-side Byzantine Faults

We explored various client-side Byzantine behaviors, including:

- **Arbitrary spending**: Clients attempted to spend more money than their account balance.
- **Double spending**: Clients tried to spend the same funds twice.
- **Self-payment**: Clients attempted to transfer funds from one account to itself disguised as another account.

For all these scenarios, our system successfully prevented the malicious client from manipulating the ledger, ensuring overall system correctness. Most client-side Byzantine fault tests were conducted with two clients, as we deemed this sufficient to cover various attack vectors.

3.3 Server-side Byzantine Faults

We investigated a broader range of Byzantine behaviors for servers/nodes. Most tests involved a system with four nodes, allowing for a maximum of one faulty node. The explored server-side Byzantine behaviors included:

- **Forged leadership**: A node falsely claimed to be the leader, attempting to disrupt the consensus process.
- **Value alteration**: Nodes tampered with the consensus value within messages.
- **Message spoofing**: Nodes impersonated other nodes to trick the system into accepting fabricated messages and achieve a quorum.
- **Incorrect value transmission**: Nodes deliberately sent wrong values to users.
- **Byzantine Broadcast**: A Byzantine leader sent different values to different nodes during the consensus process, attempting to create inconsistencies in the system state.

Our system successfully reached consensus in all server-side Byzantine fault tests. However, a significant performance difference was observed between scenarios requiring "Round Changes" and those that did not. Round Changes, a mechanism to maintain system liveness under faults, substantially increased the time required to reach consensus. While this ensures system progress even in challenging situations, it comes at the cost of reduced efficiency.

3.4 Discussion

For all test scenarios, our system successfully prevented malicious actors (both clients and servers) from manipulating the ledger, ensuring overall system correctness. We employed a mixed approach, running tests with various combinations of Byzantine client and server behaviors. The system consistently achieved consensus and maintained a valid system state regardless of the specific Byzantine behavior combination.

The experimental evaluation demonstrates the effectiveness of our system in handling various Byzantine faults from both clients and servers. The system successfully prevented malicious actors from manipulating the ledger and disrupting overall system operation. The observed performance impact of Round Changes highlights a trade-off between liveness and efficiency, which can be further optimized in future iterations.

References

[1] Moniz H. The istanbul bft consensus algorithm, 2020. https://arxiv.org/pdf/2002.03613.pdf.