Highly Dependable Systems - Stage 2: Group 29

João Furtado *99095*

Guilherme Carabalone *99078*

João Bettencourt *96880*

Abstract

This report outlines our final version of HDS Serenity, a cryptocurrency application that allows clients to perform cryptocurrency transfers between them, by Group 29. We explain the architecture of the system, detail crucial aspects of our implementation of the Istanbul BFT consensus algorithm and finally analyze its behaviour under the attack from either byzantine clients or server nodes.

1 Introduction

This stage of the project addresses some issues regarding the IBFT algorithm of the first stage, namely justification and timers, while also building a cryptocurrency application that allows clients to make balance and transfer calls by an API. The application is now resistant to byzantine clients.

2 Overview of the System Design

Most of the client side remains the same as in stage 1, meaning that the client still broadcasts their request to all server nodes. These requests are handled by the BlockchainService and adds them to the BlockPool. If the BlockPool size meets a certain size threshold, consensus is started and responses to the client are sent. The initial balance is set in configuration files for the client. For reads we now wait for f + 1 responses, to guarantee at least one response from a correct node. We talk more about how to identify the correct response later in the report.

In the node side, a lot has changed. Clients are represented as accounts, to ease transactions. The ledger now is a class that holds a list of blocks, the accounts, and signatures for each consensus instance.

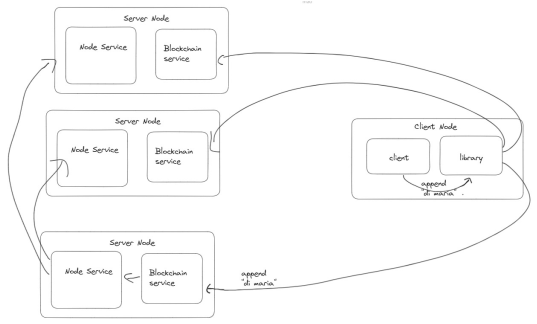

Figure 1: System Design

3 Implementation Details

3.1 Client Library

The major change of this stage regarding the client library was the operations that the client has access. Instead of only appending a string to the ledger, all of the clients have a starting balance of 1000 and they can make transfer operations between themselves, and can also check the balance of everyone.

In the update operations, i.e transfer, the client pays a fee to the block producer (leader). We had two options: make the client pay a fixed fee or a variable fee. A variable fee is optimal in cases where the client pays less for smaller transactions or when network congestion is low, while a fixed fee is optimal when we want to emphasize predictability and fairness. Since in this project all of the nodes and clients are static, which means that no process can join or leave in the middle of the system execution, and the system only has one update operation, meaning that transaction size will not change significantly, we decided to opt for a fixed fee.

There is only one read operation, balance, that returns the balance of a node. Notably, two distinct methodologies exist for querying the balance: weak reads and strong reads. By omission, the system defaults to conducting a weak read. However, for explicit strong read requests, the syntax "balance strong <id>" is used.

The library keeps a Balance Cache in its state initialized with the values present in the configuration files, it serves as fallback if a balance request to the nodes is unsuccessful.

The library waits for f + 1 responses from the nodes to return a value, guaranteeing that at least one correct process will respond. The system knows the correct response by selecting the one response that has a quorum (2f + 1) of signatures of the block decided in that consensus instance. In the case that there are no valid responses, the cache is used.

3.2 Read Operations

3.2.1 Weak Reads

Weak reads are denominated weak because they do not require a fully fledged instance of consensus to return a value. Instead, each node returns the balance present in its local copy of the account in the ledger, accompanied by a list of signatures from this consensus instance. These signatures are a digital signature of the block decided by the IBFT algorithm along with the current balances of all accounts during that particular instance of consensus. Each participating node in the network generates a signature for the block and the corresponding balances at the moment it commits a value. These signatures serve as a form of digital authentication, ensuring the integrity and correctness of the information contained within the block. When verifying the validity of a response received from the network, the system checks whether there exists a quorum of signatures that have signed the same block and associated balances. If this check passes, it indicates the response is correct and thus we can trust its balance value.

3.2.2 Strong Reads

Strong reads build up on weak reads by starting a consensus instance with an block containing only a balance operation. This does not modify the state and is used only to make sure that the nodes are not changing values in the middle of the consensus. So then we can make a weak read operation and return the values.

3.3 Justifications

Justifications are a way to guarantee that the round changes will not overwrite old proposed but not decided values. To accomplish this, both pre-prepare and round-change messages carry a quorum of round-change messages and a quorum of prepare messages, which are promply tested for justification in the JustifyPrePrepare and JustifyRoundChange functions stated in the IBFT paper.

3.4 Blockpool

In short terms, the Blockpool is a queue of LedgerRequests with a fixed size defined at launch time. When the BlockchainService receives a client request, they add it to the queue and check if the queue has already reached the defined size. If this is true, a consensus instance is started by converting the queue into a Block.

The system only has one instance of the Blockpool that is shared between all server nodes, and all of them can read and update it. One option was to make only the leader modify the pool and the rest of the nodes could only read it, but this is not ideal since the leader can become faulty or be byzantine, and since all of the nodes receive the client request, it makes sense that all of them should be able to modify it.

3.5 Future improvements

• Timer in the Blockpool: If we had more time, a nice change to improve user experience would be to add a timer in the Blockpool. At the moment, a consensus instance is only triggered when a Block reaches a certain size that is defined in the puppetmaster. The optimal way would be to trigger it also if the timer of the Blockpool expires, this way the client avoids waiting for other requests to fill the pool.

• Client making multiple requests: Another change would be a client making multiple requests at the same time. Currently, a client makes one request and waits for a consensus for it. The solution is simple but due to time constraints we could not make it. Ideally, we would have a different thread for every client request, and the client waits for them separately in that thread.

• One consensus instance at the time: currently our service can make multiple consensus instances at the time. Ideally we would make only one, but again due to time limitations we could not meet this requirement. This solution is also simple, we just needed to have a variable that checks if the last block was already committed, and only start a consensus instance for the current block if this verification returned true.

• Further testing: As mentioned in the test section, our service has protection against multiple attacks, but ideally we would make even more. Some examples include; test against a server node injecting money to himself; a node modifying the Blockpool; a block producer not charging the correct fee for an update transaction. In theory, our current implementation does not protect against these. Now for the tests that we did not make but in theory our implementation should be resistant: Sending pre-prepare / prepare messages with random values; Server node sends messages with a forged balance.

4 Behaviour Under Attack

These tests can be run by choosing the corresponding configuration file in puppetmaster.

4.1 Byzantine Server Nodes

• Dictator leader: The server node always acts as a leader and does not broadcast round changes.

• Silent leader: Send no messages to start a consensus instance.

• Fake leader: Send messages in the name of the leader.

• Drop packets: Leader starts consensus and drops all subsequent packets.

• Bad consensus: Attempt to start consensus, to test if other nodes only accept messages from the actual leader.

• Message delay: Simulates network congestion on the leader to trigger a round and a leader change.

4.2 Byzantine Clients

• Greedy client: Sets the destinationId as the sourceId, meaning that the client will try to transfer money to himself.

References

[1] Henrique Moniz, "The Istanbul BFT Consensus Algorithm", https://arxiv.org/pdf/2002.03613.pdf.

[2] Ethereum, "Quorum Byzantine Fault Tolerance Specification", https://entethalliance.github.io/client-spec/qbft_spec.html.