Highly Dependable Systems

HDS Serenity Ledger

Group 41

Guilherme Pascoal 99079

Miguel Vale 99113

Tiago Caldas 99125

1. Introduction

This project aims to develop a simplified permissioned blockchain system with high dependability guarantees, called HDS Serenity (HDS2).

In this stage, our goal was to understand and then extend an initial and incomplete version of Istanbul BFT Consensus Algorithm. In order to guarantee the correctness of our implementation we also developed a set of tests.

2. Design Requirements

The design of the HDS2 system encompasses two essential components: a client library integrated with user applications and server-side logic responsible for maintaining system state and implementing the blockchain service.

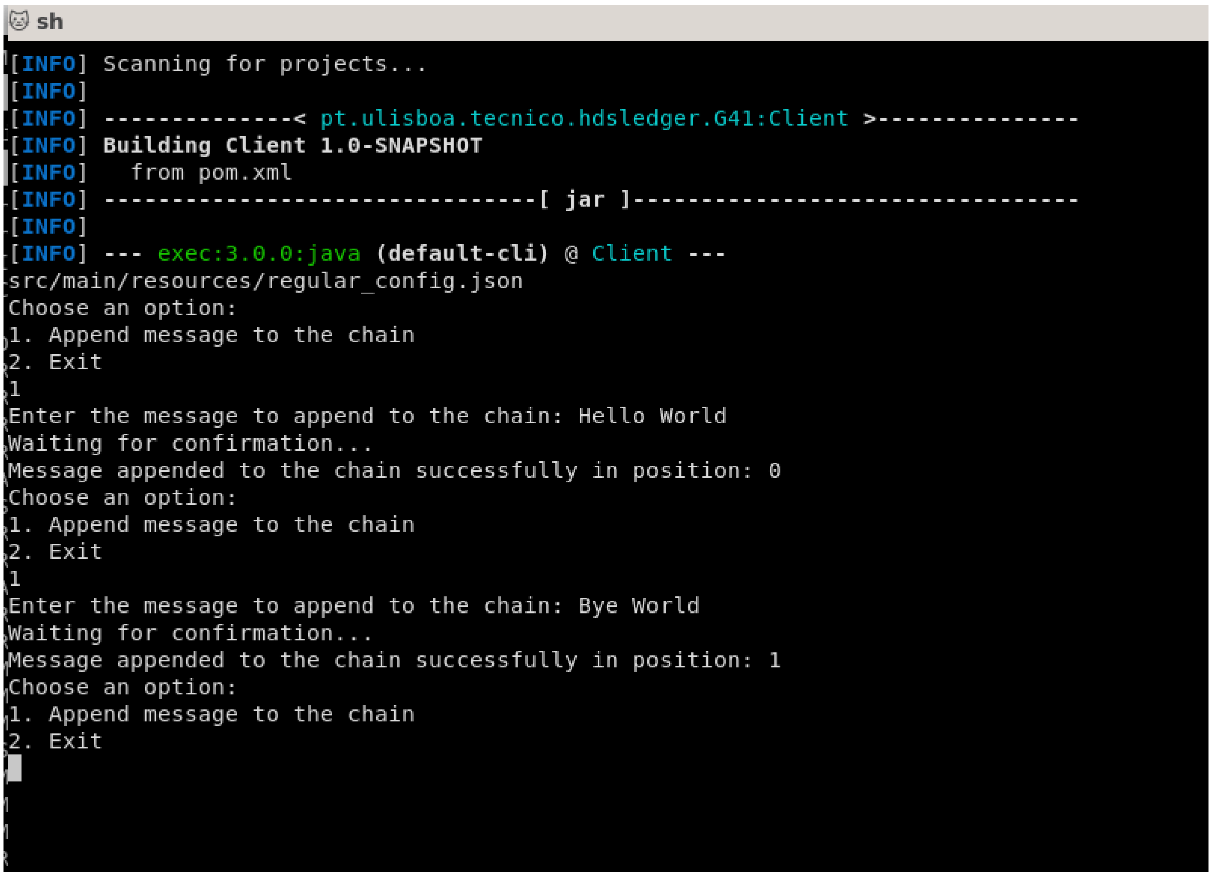
For this stage, we will use a simplified client that only needs to submit requests to the service and then waits for a reply confirming if and where in the sequence of blocks the request was executed.

3. Design Solution

We divided our solution into three main steps: developing the client library, extending the Istanbul BFT consensus algorithm, and creating tests for quality assurance.

3.1. Client library

To address the first requirement we created a new service for the client. This service is instantiated for each user interacting with the blockchain. To accommodate the interaction we created a simple CLI with the supported operations.



```
[INFO] Scanning for projects...
[INFO]
[INFO] --------------< pt.ulisboa.tecnico.hdsledger.G41:Client >---------------
[INFO] Building Client 1.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- exec:3.0.0:java (default-cli) @ Client ---
src/main/resources/regular_config.json
Choose an option:
1. Append message to the chain
2. Exit
1
Enter the message to append to the chain: Hello World
Waiting for confirmation...
Message appended to the chain successfully in position: 0
Choose an option:
1. Append message to the chain
2. Exit
1
Enter the message to append to the chain: Bye World
Waiting for confirmation...
Message appended to the chain successfully in position: 1
Choose an option:
1. Append message to the chain
2. Exit
```

3.1.1. Append operation

Users can add strings to the chain and will receive confirmation of their successful addition. When a user adds a string, the client service broadcasts this request to all nodes, initiating a new consensus round. We could have sent the append message to f+1 nodes, which would suffice to ensure that at least one process is correct but opted to broadcast to all nodes. Because links only manage messages, we introduced a new message type for append operations. Once the value is appended to the ledger, the leader sends a message to the client indicating the value's position in the ledger.

3.2. Implementation of Istanbul BFT

To implement the Istanbul BFT algorithm, we extended the code based on the specifications provided by the teachers [1].

3.2.1. Links

We determined that the previous connections were Perfect Links. For the algorithm's implementation, we transitioned to Authenticated Perfect Links. To achieve this, we devised a sign(byte[] data) method that generates a digital signature, and a validate(String nodeId, byte[] data, byte[] signature) method that allows receivers to confirm the data was indeed signed with the private key associated with the given public key. For each node and client, we generated a pair of public and private keys.

3.2.2. Round-change

The Round-Change mechanism is a critical element of the algorithm. To support this, we have created a RoundChangeMessage that is broadcast when a node suspects the leader is malfunctioning. Additionally, in the startConsensus function, we update the configurations of all nodes with information about the new leader.

3.2.3. Possible threats

When we have two or more clients whose input values differ, we may encounter a problem because we do not check the quorum for each individual value. This could lead to issues when determining the minimum round for each quorum.

3.3. Tests

We have developed a suite of tests to evaluate the implemented features and demonstrate our system's resilience to Byzantine faults. Test 1 examines scenarios in which the leader is Byzantine, ensuring the system successfully elects a new leader and commits the message. This test also checks whether the round and the leader are successfully reset during a new consensus instance. In Test 2, we investigate scenarios where a Byzantine fault occurs in a node that is not the leader. We also tested the normal case in the client terminal, where none of the nodes are Byzantine. When we run the tests, an exception appears saying java.lang.InterruptedException: sleep interrupted. This is intentional because the threads from previous tests remain active, and we did this to stop the previous threads.

4. Conclusion

To conclude, we believe that we have developed a robust system capable of tolerating Byzantine faults. However, we also think that there is room to develop additional tests, including those involving more clients.

5. References

[1] Moniz, Henrique. n.d. "Istanbul BFT Consensus Algorithm." *Istanbul BFT Consensus Algorithm*.