HDS Serenity Ledger - A Highly Dependable Ledger

André Torres
andre.torres@tecnico.ulisboa.pt

Gonçalo Nunes
goncaloinunes@tecnico.ulisboa.pt

Pedro Lobo
pedro.lobo@tecnico.ulisboa.pt

1 Introduction

This project implements HDS Serenity, a simplified permissioned blockchain system with high dependability guarantees.

To achieve this, our solution extends the initial Java codebase provided by the faculty. We enhance the channels by adding authentication, develop a simple client library, implement a complete version of the Istanbul BFT Consensus Algorithm [1] and include a test suite that ensures our implementation is robust against Byzantine processes.

2 Implementation

2.1 Channels

The base code provides perfect links, which are not needed in non-byzantine models. However, in byzantine models, where nodes may fabricate messages and potentially jeopardize the **no creation** property of perfect/stubborn links, authenticated perfect links are needed.

To accomplish this, we modified the link so that each message is signed by the sending process. The receiving nodes verify this signature. If the signature fails validation, the message is disregarded.

2.2 Public Key Infrastructure

A Public Key Infrastructure (PKI) was needed to complete the implementation of the authenticated perfect links, as well as other cryptographic abstractions used in this project. The PKI provides an abstraction for generating key pairs, as well as signing and validating the authenticity of messages. This was implemented using the Java Crypto API.

2.3 Consensus

Consensus is achieved through the utilization of the Istanbul Byzantine Fault Tolerance (IBFT) algorithm [1]. This algorithm solves consensus in a partially synchronous communication model and tolerates f faulty processes out of n, where $n \geq 3f+1$.

The base code partially implemented IBFT, it was necessary for us to implement the *round change* and *message justification* protocols.

2.4 Application

The application consists of a client that reads commands from *stdin* and calls the library, which is responsible for translating the client's requests into consensus instances. The library sends the value to be appended to the ledger to $f+1$ nodes and wait for $2f+1$ responses.

We decided to only send append requests to $f+1$ nodes in order to reduce the number of messages transmitted by the client. However this introduces a new problem as a byzantine node can now propose a value that was not proposed by any client and this fake value might be decided. To solve this problem, the client now signs the value it wants to append to the ledger and during the various consensus rounds the correct nodes check if the value in the message is correctly signed by the client who proposed it. This prevents a byzantine node from getting a quorum to prepare a value not proposed by any client, preventing it from ever being decided.

Finally, the client waits for $2f+1$ responses. This guarantees that a byzantine quorum has received the value proposed. As any two byzantine quorums intersect in at least a correct node, it is guaranteed that the value will eventually be decided.

3 Test Suite

To test the system, we created various configurations files that define the (byzantine) behaviour of the nodes. Each of the tests parses a configuration file. The byzantine behaviours implemented include:

- **None**: The node is correct. Normal execution.

- **Drop**: The node drops all packets. It simulates a crashed node.
- **Fake Leader**: The node tries to act as the leader. When a new round starts, it tries to send a pre-prepare message.
- **Fake Value**: The node sends consensus messages with a value that was not proposed by any client.

The tests were implemented using JUnit testing framework. When running the tests, it is important to note that, sometimes, tests fail as the socket could not be bound to the port specified in the test configuration. This happens because the socket used in the previous test was not released before the start of the following test. To circumvent this problem, each of the tests can be run manually, one by one.

4 Possible Improvements

- Instead of returning $2f+1$ confirmations to the client, only $f+1$ could be used. However, these messages would now have to carry, each one, a quorum of signed messages confirming that each one of the processes in the quorum decided the proposed value. This would reduce the number of messages in the network, which could be beneficial in some scenarios.

References

[1] Henrique Moniz. The Istanbul BFT Consensus Algorithm. May 2020.