TÉCNICO LISBOA

SEC Project Report

2022-2023

HDS Ledger (HDL)

Stage 1

MEIC-T, Tagus Park, Group 19

Afonso Bernardo (96834), César Reis (96849), Henrique Vinagre (96869)

I. System Description

The HDL system is a permissioned distributed ledger system that uses a *client-server* architecture, supporting an application which provides a service for clients to propose *strings* to this blockchain through an API.

Any client library process is external to the system, triggering the internal blockchain communication where all (**N**) participants are initially known by anyone and some of them (**F**) may eventually act as byzantine processes.

As a typical distributed system, the client transactions are collectively evaluated by the servers, using in this case *Istanbul BFT Consensus Algorithm (IBFT)* [1] if we have $\mathbf{N} \geq 3\mathbf{F} + 1$ processes.

II. System Design (with implementation)

The program is written in Java [2] with the Maven build tool [3], using HDLProcesses as an abstraction of real processes which can be one of the following:

- Client API single threaded;
- Server (being a blockchain participant) multi-threaded.

Our system resides on only the normal case of the stated consensus protocol (1 & 2) where we always have the same leader, which by being a correct process leads to no need for round changes.

For that purpose, we designed and implemented the following:

- Any participant is able to identify the static leader for each instance of consensus;
- 4 abstract link layers (fair-loss, stubborn, perfect and authenticated perfect links) with their respective guarantees.
- Regarding the stubborn links we optimized the *send* procedure by allowing the receiver to send an ack to the sender to notify him that he can stop retransmitting, keeping any repeated message with a “main” delivery thread in case the *ack* is lost. This thread separates *messages* and *acks* into two queues, which other threads access to get the message they need and send the corresponding *ack*.
- Best-Effort Broadcast, as we already have authenticated links, so any broadcast works for the IBFT algorithm, therefore we opted for the easiest to implement.
- The IBFT algorithm is implemented as described in the original paper [1], with each event (*upon* events) being handled in different threads allowing multi client requests at once. The START function is initiated once the leader of a certain instance has a pending request from a client. Each process in the IBFT system keeps a counter for the current instance and increments it once the current instance has been decided (when it receives a quorum of commits for that instance), thus only allowing for one instance of consensus at a time.
- Client API: this library makes new requests to the system by broadcasting the string to append to all instances of the closed membership. It then waits for $\mathbf{F} + 1$ equal responses (from different servers) to decide on the status of the request. This implies that at least one of these responses were produced by a correct process, allowing to derive the final status correctly.

III. Threat analysis

- Despite in our implementation the processes being able to see all keys (including private ones), in a real scenario this wouldn't be the case, as the "processes" would actually be different machines, making it physically impossible.
- The system ensures that any created message is different and no other mechanism can break its identity.
- To prevent byzantine behavior at an instance of consensus, we ensure that any participant only delivers a PRE-PREPARE from the leader and ignores any other IBFT message from someone that already sent something.
- Relating to the client API, we were aware of threats in both ways (API -> Server System and vice-versa): the client could use multiple threads to mess up with the API, so we assured that this API is thread-safe; and that a byzantine server could try to send multiple responses to fulfill the client's **F** + 1 demand on it's own, giving the wrong feedback to the client, so we used the help of authenticated links to assure that this wasn't a threat to our design.
- With the implementation of authenticated links, no messages can be forged above this layer, and with the help of perfect links, no replay attack can be performed, giving strong guarantees of integrity to the system.
- We didn't implement any form of encryption (which would give confidentiality properties to the data in the system), as that would be of no use because any byzantine server of the system could easily decrypt and leak this data.

IV. Dependability guarantees of the system

- As mentioned before our system provides integrity to blockchain with the guarantees of the used authenticated links.
- if client append string -> its string will eventually be added to blockchain
- if string added to blockchain -> string there forever
- if a correct participant decides a string proposed for some client -> all other correct participants eventually decide it in the same instance.
- Adding this version of consensus protocol (over the links guarantees) we still have problems when the leader is byzantine - need change of rounds (next stage) -> the value decided could be one not proposed by any client.

V. References

[1] Henrique Moniz. The IBFT Consensus Algorithm

[2] Java docs

[3] Maven build tool