HDS Project Report - Group 29

Gonçalo Guerreiro 95581 | João Ramalho 95599 | Vasco Sebastião 95686

1. Broadcast

The broadcast algorithm used to communicate in the consensus is Best-Effort, similar to what is used for the client to send the append request to the blockchain. The reason for the group to choose this algorithm is because we implemented Perfect Links in the lower layer of the application, which will be explained further in the report.

2. Consensus Algorithm

The consensus algorithm implemented is the The Istanbul Byzantine Fault Tolerant Consensus Algorithm, namely, for this stage we implemented Algorithms 1 and 2, which the pseudo-code is featured in the paper of this consensus algorithm.

3. Perfect Links

The communication between the processes is done using UDP with two layers of communication abstraction built on top: Stubborn Links and Perfect Links.

The Stubborn Links are implemented using Datagram Sockets (which approximate the behavior of Fair Loss Links). Each Stubborn Link maintains a list of all the messages that have been sent but have not yet received confirmation (ACK) from the receiver. When a Stubborn Link receives a message, it checks if it is an ACK and, if so, removes the corresponding message from the list. These links also have another thread running which, using an exponential backoff, re-sends all the messages in the list. This way, we guarantee that the messages keep being sent until it receives confirmation that they were delivered.

The Perfect Links are implemented using Stubborn Links. Each Perfect Link maintains a map with the highest sequence number received from each node in the system. When a Perfect Link receives a message, it checks if its sequence number is higher than the one saved in the map for that sender and, if so, it adds the message to the list. Repeated messages and ACKs are discarded, guaranteeing that only new requests are delivered. When a request is received, Perfect Links send an ACK back to the sender, confirming it was delivered.

4. Dependability and Security

Regarding this subject there are quite a few mechanisms implemented to ensure dependability of the system to the user.

Firstly, we made the library create a tuple that consists of the id of the client that sent the append request and the sequence number of the request. This tuple will be kept as a part of the structure of the message, even when being sent between members of the consensus, to ensure freshness to the protocol.

Then, for simplicity purposes we developed our solution using only 4 members for the consensus protocol, to ensure the minimum for a byzantine quorum is reached. And as mentioned we launch them from a configuration file. But, also, we created a directory holding all the asymmetric keys of each node, used for digital signatures which are then checked at the links layer. This provides integrity, authenticity and non-repudiation of the messages between nodes of the system.

To do so we employed mechanisms such as signing the messages between every entity of the program (even the ACKs), making sure that it is not possible for a node of the system to send a message on behalf of another one.

Also, when receiving messages related to quorums, each member also checks if the sender of that message had already sent a quorum message for that consensus instance, ensuring that all messages that form the quorum are sent from different members.

5. Client

The client can perform a set of operations that are then clustered in a block and appended to the blockchain. Clients can create an account, make transfers to other accounts in the system and also make read operations to check their balance with two different consistency types.

The client creates a thread that is consistently receiving responses from the servers, while also allowing it to send more operations at the same time, in an effort to avoid active waiting.

For write operations (create account and transfer), the client waits for *f + 1* equal responses to make sure they are all from non-byzantine members.

For strongly consistent reads, since they don't go through a consensus instance, it waits for a byzantine quorum of *2f + 1* equal responses. For each of these reads, the client creates a thread that checks if has reached this quorum, and if it is not able to so after a while, it times out and re-sends the operation. This process is repeated until it is able to achieve a quorum, since it might happen that, even if two members are non-byzantine, they may be in different stages of executing a block when they receive the read request, resulting in different responses to the client.

For weakly consistent reads, the client only waits for a response from one of the members, since they should work correctly even in low connectivity scenarios. The client creates a thread that waits for the first response received and then verifies all the signatures in the response, which will be discussed further on the report.

6. Token Exchange System

The token exchange system is essentially used by the client by sending its public key in the operations that are sent to the members. Then, on the server side, each replica has a data structure that holds all of the accounts of the system, namely a HashMap that uses the public keys of the clients as keys, which uniquely identify their account, and their balances as the values.

7. Operations

The operations allowed by the system are to create an account, transfer balance from one account to another and to check the balance of a client. These are then grouped in blocks of 5 operations and, after going through a consensus instance, are appended to the blockchain, creating a log of all the operations executed by the system.

8. Read-Only Operations

8.1 Strong Reads

Strong reads are essentially made with the Practical Byzantine Fault Tolerance (Castro and Liskov, 1999) algorithm, namely the part where it describes an optimization for read-only operations, where, after receiving a consistently strong read request, each replica sends the current balance of the requested account and then the client waits for a byzantine quorum of *2f + 1* equal responses.

8.2 Weak Reads

Weak reads are more carefully taken care of. To implement them, the replicas use snapshots, which essentially hold the current state of the accounts and are done every 3 instances of consensus, and each replica that takes a snapshot signs the key that identifies the account and its balance, for each account in the system. After the snapshot is taken, each replica broadcasts it to all other replicas. Afterwards, each replica waits for a quorum of *2f + 1* equal snapshots received, so that it can create a validated snapshot by aggregating all the signatures that form the quorum. Then, when a replica receives a weakly consistent read request, it just needs to send its latest validated snapshot to the client.

On the client side, the request is sent to all replicas and then it just waits for one response to take it as the result of the operation, but it also checks if the snapshot has at least *2f + 1* valid signatures. This is made so that a client can still receive a valid state of the system even when only one replica is up, in weak connectivity scenarios.