BFTB Report – 2^{nd} *Delivery*

Group Number:22

Lab Shift: SDTF2L04

Students:

- **Francisco Bento, 93581**
- **João P. Lopes, 93588**
- **João A. Lopes, 93584**

1.Design changes introduced in this stage

1.1. Include signature of the client in server's response to guarantee freshness

In our discussion, it was made a point that our system didn't have a way for the clients to know if the answer that they got was really meant to them. As a solution, we decided to append the signature of the client's request to the server's answer so it can be verified by the client. This change was made in both ServerImpl.java (in every answer from the server) and Library.java. This is where after we verify the server, we also verify the signature by comparing the one received from server with the one received as a parameter from the library's function.

1.2. Locking access to signature's HashMap to solve concurrency issues

During our conversation, it was also mentioned that we experienced concurrency issues when multiple clients were accessing hash maps in the main functions of the ServerImpl.

To solve it, we synchronized the critical sections by using Re-entrant Locks. This allows to protect the critical sections from multiple client's accessing it at the same time, by locking the remaining clients. This was done inside the function isReplayAttack, which is called in every type of request, guaranteeing that clients will offset themselves from others at any time.

2.Implementation steps from stage 2

2.1. Replicating the server and connecting the client to each replica

Each client now connects himself to 4 replicas (due to use of (1, N) Byzantine Regular Register which requires N>3f replicas, with f=1) using 4 different sockets, from port 5000 to 5003. This means that each client Library now handles a list of sockets, each connected to a different replica and client requests are propagated one replica at a time.

2.2. Implementing (1, N) Byzantine Regular Register

As for this delivery we need to consider that the system may be subject to Byzantine faults, affecting both client and server processes. To make them resilient to Byzantine faults, we started with by implementing a (1, N) Byzantine Regular Register as suggested by our guide, taken from the course book (Introduction to Reliable and Secure Distributed Programming,2nd Edition). Considering this type of register, we built the Authenticated-Data Byzantine Quorum by implementing the following steps:

- A client must now store two new variables, *rID* and *wTS.* These will be incremented by one every time a reading request and successful writing request are done, respectively.

- When a client signs his message, it also signs it with the respective incremented ID.

- The client's library handles with *rID, wTs, ackList and readList*. The two latter will serve to recognize which replicas are answering to the correct number *rID* or *wTs* as way to build a quorum later.

- For a Writer (client who does a write request), the replicas will update their timestamp and value (for us it means storing the new transaction with two new fields, the original request and his sign) if the criteria of the new timestamp being bigger than the old one is met. It's also worth mentioning that if the new request is approved, each replica will update its view (what would be the return of functions audit and checkAccount).

- After finding an answer for the new writing request, all replicas will reply to the client, one by one. If a replica replies with the same timestamp as the one present in the client's library, it will update the acknowledge list replica position to the answered timestamp. If at a certain time a majority of replicas acknowledge correctly (that being bigger than (N+f)/2, i.e, 3), then the library will send back the answer to the client according to the quorum.

- For a Reader (client who does a read request), the replicas will retrieve the latest view they have updated according to the type requested (audit or checkAccount).

  For an audit, the replicas return the following list of components: *rID*, timestamp of account being audited and the view of all transactions done so far by that client. This view includes all information useful to a client to consult, but also the original request and its signature so it can be later verified as a valid operation done by that client and not an arbitrary timestamp and value.

  For a checkAccount, the replicas return the following list of components: *rID*, timestamp of the account, the account balance, the view of pending transactions to accept and the view of all transactions done so far by that client as audit does. This is done so each transaction is verified like audit and the balance can be calculated manually to later be compared with the one sent by a replica, to prevent arbitrary values.

- After verifying each transaction, each replica updates the *readList* with her most updated timestamp and respective views, so the quorum evaluates what is the highest timestamp and retrieve that to the client.

Our implementation guarantees that the system has Correctness, due to having 4 – 1 = 3 correct processes, making the readers and writers receive N-f> (N+f)/2 replies and complete their operations correctly.

On the validity side, we can guarantee that either the replicas will return the concurrent value being written or the previous write saved on the replicas, depending if at least one replica returns the concurrent written value as it corresponds to the highest timestamp.

2.3. From a (1, N) Byzantine Regular Register to an (1,N) Byzantine Atomic Register

Although our group couldn't update our register, we took time to understand what would be needed to make the transformation (by merging the concepts with a Read-Impose Write Majority Atomic Register).

In the first place, clients would need a boolean to allow distinguish if they are reading or not. This variable is updated to *TRUE* whenever a read operation is requested from a client.

After a reply is sent by every replica, the quorum will now extract what is the current highest timestamp and its value. Instead of this value being sent to the client, alternatively it will make a Write to other replicas with this timestamp and value. This change allows an user to always read a value being written concurrently while reading instead of having a chance of reading that value or the value previously written.

Lastly, the writing quorum handle part on the client's Library will judge the value of the boolean that distinguishes if a client is reading or writing and according to that return the proper value to the client.

3.Security Threats Avoided

3.1. Denial of Service (DOS)

These types of attacks are prevented through the use of a proof of work created by the replicas side after a client sends his first request. Each replica challenges the client to a feasible amount of effort to avoid malicious clients from spamming requests.

3.2. Sybil attacks

Like the first stage, Sybil attacks are still avoided because a client cannot open different accounts on different replicas, which would be the only way a malicious client would forge multiple entities to attack our system.

Besides that, now that we have 4 replicas, each gets a different pair of keys assigned to it, to prevent all 4 replicas from merging into one entity.