SEC-Report-2

Tomás Guerra 102178 / Diogo Branco 92453 / Beatriz Alves 86391

April 24, 2022

1 Design changes

In this stage, the server is replicated across N replicas. The number of replicas depends on the number of tolerated Byzantine faults f, such that $N > 3f+1$. We tested the server with 4 replicas, tolerating 1 fault.

Instead of connecting directly to the servers, the clients connect to a **BFTService**, which communicates with the replicas to manage the replication part of the protocol.

Each replica has its own separate database. We consider that each account in the database corresponds to a register. For each account, we store in a separate table the logical **timestamp** of its last update and the **signature** of the message that caused the update.

2 Protocol

The system implements a **(1, N) Byzantine Atomic Register** protocol, where every replica can perform read operations, but only the writer replica is responsible for write operations.

2.1 Writes

When the BFTService receives a *sendAmount* or a *receiveAmount* request from a client, it starts by forwarding the request to the writer replica. The writer replica performs the operation, increments the timestamps of the accounts affected by the operation, and returns their timestamps ***wts_source*** and ***wts_dest***.

The BFTService then adds the timestamps and its signature of *[wts, msg]* to the request and broadcast them to the other replicas asynchronously. If the timestamps are more recent than the ones currently in the replicas' registers, they also perform the operation and update the registers with the received timestamps and signature. They then return an ***ACK*** message to the server, which waits for a quorum of $> (N + f)/2$ before proceeding.

2.2 Reads

For *checkAccount* and *audit*, the BFTService broadcasts the request to all replicas along with an identifier ***rid*** to prevent replay attacks. When it receives a response, it should first check if the signature of the message was valid, to ensure it wasn't forged by a malicious replica. It waits for a quorum of responses and returns the value associated with the most recent timestamp.

To guarantee atomicity, the BFTService performs a **write-back** by broadcasting the result of a read operation to all replicas. The *checkAccount* and *audit* responses also return all transactions of a certain account, as well as its timestamp. During the write-back phase, if this timestamp is greater than the version in the replica's database, the replica checks if it has any missing transactions and inserts them into the database.

3 Protocol

3.1 Byzantine server tolerance

Before every register update, the BFTService should signs [wts, msg]. All replicas should store a copy of this signature, and return it on a read response

along with the respective values. If one of the replicas forged the timestamp or the values on a read response, the BFTService would verified that the signature wasn't valid, and ignore the response.

This part was not implemented.

3.2 Byzantine client tolerance

Malicious clients could attempt to corrupt the state of the register by sending requests to write different values to different servers, or by sending these requests only to a subset of servers. A possible solution to this problem is to implement a Byzantine Reliable Broadcast algorithm, such as the **Authenticated Double-Echo Broadcast**, which works as follows:

1. The client broadcasts a request to every replica
2. When a replica receives the request, it broadcasts an ***ECHO*** message, containing the request, to every replica
3. When a replica receives a byzantine quorum of $> (N + f)/2$ ***ECHO*** messages, it broadcasts a ***READY*** message to all replicas, indicating that it is ready to deliver the request
4. When a replica receives $2f + 1$ ***READY*** messages for the same request, it delivers the client's request
5. If a replica receives $f+1$ ***READY*** messages before receiving a quorum of ***ECHO*** messages, it sends a ***READY*** message itself. This enforces the totality property, which guarantees that if a message is delivered by a correct replica, every correct replica eventually delivers it as well.

As an example, if a malicious client sends a message only to one replica, that replica will never receive the quorum of *ECHO*s necessary to deliver it, and thus its state won't change.

On the other hand, if the client sends a message to only three replicas out of four, each of them receives a quorum of *ECHO*s and broadcasts *READY* messages. The replica that was left receives the required 3 *READY* messages with the original request, which means that all replicas receive the request and can deliver it.

3.3 DoS and Sybil attacks

When a client makes a request, it needs to attach a **Proof of Work** to the request, which consists in the solution of a SHA-256 hash of [message—solution] such that the first X bytes of the hash are zeroed. The BFTService then computes the same hash with the provided solution and, if the number of zeroes at the start isn't correct, it discards the request.

The client must dispend a large amount of computational power to find the solution to this problem. This way, creating a valid request takes so long that it prevents a malicious client from exhausting the servers' resources in a **DoS attack**. The same reason prevents users from creating a large number of accounts through a valid *openAccount* request in a **Sybil attack**. On the other hand, validating the hash is very fast, and doesn't constitute a problem for the BFTService.

For testing purposes, we set $X = 4$ so that it wouldn't take too long to compute, but this number should be increased to properly protect against these attacks.

3.4 Replay attacks

To prevent replay attacks, we implemented our proposed solution from the first stage using **nonces**. We considered that these were only necessary on requests that changed the state of the system.

The client first requests a nonce to the server. In the following request, the server verifies if the nonce corresponds to the one in the client's pending request, and if it hasn't yet been used, by checking a database table that keeps track of used nonces.

3.5 Non-repudiation

When a client sends a request for a state-changing operation, it must send the **signature** of that message along with the request. The server stores this signature associated with the transaction, which proves

with certainty that the operations were performed by the signer.