SEC Report, Stage 2 - Group 21

Diogo Dias 90792

Gonçalo Velhinho 90718

Miguel Mota 90964

Communication

The communication between Server and Client is implemented using JSON strings and Java Sockets.

Java objects were converted to JSON using the GSON library. The messages are represented as JSON objects and then are digitally signed by the sender.

In the previous stage we used a challenge/response mechanism to ensure freshness. In this stage we use a logical timestamp for write requests and a request identifier for read requests. The timestamps/request ids are part of the signed message. Request ids are generated from Java's SecureRandom class.

This communication protocol ensures that only the people with the private key can sign messages, and it ensures integrity of the message since the whole message is signed. The signing of the timestamp/request id ensures freshness. This set of guarantees ensures non-repudiation.

Private and public key pairs are stored in a secure password-protected JCA KeyStore generated by each client using Java's keytool. Private keys are generated using 2048 bit RSA. We also assume prior distribution of the public keys, so every client already knows each other's and each server's public keys.

If an attacker drops, manipulates or duplicates messages the server rejects these messages, thus creating a potential denial of service. We do not fully address this problem. We however address some aspects of Denial of Service, by the server having absolutely no state for a client, which makes the use by a large number of clients possible. It does this by when executing commands it creates a new socket and closes it when the response from that command has been received. Another way our project deals with DDOS is addressed in the Byzantine Fault Tolerance section.

We do not implement perfect links but we use TCP. If a server cannot be reached we assume that that server is faulty. However, as described in the Byzantine Fault Tolerance section, if the server was only temporarily faulty then it will eventually receive the missing transfers (during the write-back part).

Persistence and Atomicity

The data is recorded in an SQLite database called SECx.db for each server, where x is the replica's number. Persistence is ensured since the data is written to a file, so in the event of a server crash the data is still safely stored. Data consistency is also guaranteed from the SQLite transactions, which make sure that in the event of a server crash only the committed data persists in the file or is rolled back.. The transactions also ensure atomicity during concurrent writes/reads. For testing purposes, servers have an argument that is called reset, which drops all the tables and their contents from the database and then recreates all tables.

Account's timestamps are persisted in the database. This guarantees that if the client crashes the servers still remember what the correct timestamp is and thus avoid replay attacks. After the client recovers, it queries the servers to find the highest timestamp. Another possible solution would be to update the client's timestamp whenever it fails a request because it has an unusable timestamp due to being too low. We chose the first option due to its simplicity.

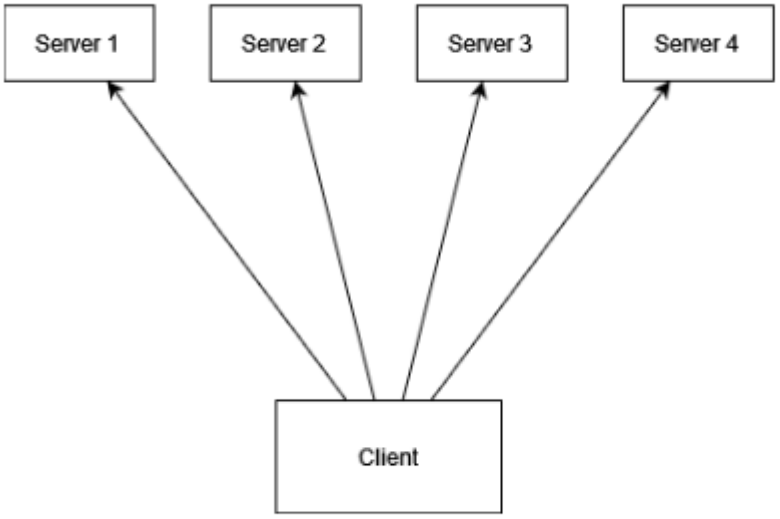
Byzantine Fault Tolerance

In this stage we implemented 3f + 1 replicated servers. We assumed that clients could only suffer crash faults but the servers could suffer byzantine faults. The algorithm implemented to achieve a (1,N) Byzantine Atomic Register was based on adding data authentication to a "Read-Impose Write-Majority" algorithm. Clients now broadcast their requests to all the replicas instead of only one server. It then waits for a byzantine quorum of responses before proceeding.

In the "Read-Impose Write-Majority" algorithm for the write messages we only send the new transfer. However when we read we receive the results of the operations, select the result with the highest timestamp and then use that result to write back what might be missing in other replicas. Therefore only on the write-back part of the reads, does the client send all transfers. Since this is a heavy operation, we have a system resembling a proof-of-work puzzle that the client has to solve before doing a read operation. This works as protection from DDOS attacks. Another possible DDOS protection would be a cache that would prevent spam by returning cached results for repeated operations.

In our implementation, we add a signature for each new transfer and new timestamp and a request Id to prevent a byzantine server from arbitrarily changing the contents of a transfer. Each signature, wts and request Id are persisted in the database as well. The client checks all of the signatures of the received messages in each read to verify that they are authenticated and legitimate.

We also assume that the accounts have been opened before any other operation is executed. If a replica doesn't have an account initiated, any operation that is related to that account can't be done.



Pros and Cons

- Using JSON as a message format has memory and performance issues but it's simple to use and also facilitates debugging since everything is a human-readable string.

- SQLite easily ensured two of the design requirements (persistence and atomicity) but compared to other databases it has worse performance. However SQLite is used alot by Server-Side Databases.

Considering the scope of the project and the time limit for submission, we decided to favour simplicity over performance for the items on the list. For this reason, the choices made above seemed reasonable.

Byzantine Clients

Even though we didn't implement it (due to time restraints), we would implement it using Byzantine Reliable Broadcast (BRB). Using BRB would ensure that all the servers would receive the same message, preventing a byzantine client from sending different correctly signed messages to different servers.

Corrections of the first project

We had a few problems on the first project. In this list below we address how we corrected them:

- Freshness: By using logical timestamps we no longer use challenge response, which decreases the number of messages exchanged making our system less vulnerable to man in the middle attacks.
- SQL Injection: We changed all our statements to prepared statements, which prevents SQL Injections.
- Security Problems: A client can no longer send negative money to other accounts. The server now verifies if the public key of the sender and the public key of the sent send request matches. The server now verifies if the public key of the receiver and the public key of the sent receive request matches. Both the above changes make it impossible for someone to send or receive currency as if they were somebody else.
- Storing Signatures: We now store signatures in the database guaranteeing full authentication.