Byzantine Fault Tolerant Banking
Stage 1

Highly Dependable Systems

Group 34

79730 João Silva
92513 Mafalda Ferreira
92451 Diogo Barbosa

Master Degree in Computer Science and Engineering

Instituto Superior Técnico

Alameda

2021/2022

Introduction

The goal of this project is to develop a Byzantine Fault Tolerant Banking, known as BFT Banking.

Design

The system is divided into a client and a server.
The server domain holds an **Account**, a **Transaction,** and a **Bank**.

- An **Account** contains the public key provided by the user, the current balance, and a list of completed transactions. It also contains two hashmaps, one for the pending withdrawals and another for pending credits, both associated with their transaction id.

- A **Transaction** contains the source and destination public keys of both involved entities, their amount, id, and timestamp of creation time.

- The **Bank** stores all bank accounts and a synchronized transaction count that serves as an id for the next transaction.

The server logic takes into account the different bank operations:

- **OpenAccount**: the system starts by incrementing the number of bank accounts. Then, it creates a new bank account associated with the current id and its balance is set to 50 euros.

- **SendAmount**: the current transaction count is incremented and the system creates a new transaction, associated with an id. This transaction is added to the pending withdrawal of the source account and pending credit hashmaps of the destination account, both along with the transaction id. The source account balance is also decremented by the amount required.

- **ReceiveAmount:** the system ensures the source account already contains a pending withdrawal associated with the current transaction. If so, the transaction is marked as completed and removed from the pending withdrawal and credit hashmaps, and added to the transaction list of both source and destination accounts. The balance of the destination account is then incremented with the respective amount.

- **CheckAccount:** obtains the current balance associated with the public key and returns all incoming transfers in the pending credits hashmap.

- **Audit:** returns all transactions saved into the transaction list of the bank account.

A user can perform bank operations through the client app, which supports the management of the user’s keys. The user needs to register in the system with a username and password, in order to keep them inaccessible to other users. These values are hashed (SHA-256). The password is used to create an AES key, which will be later used to encrypt and save the generated private key, using CBC with constant IV and Salt values. Public keys are stored without encryption since everyone already had access to them.

The client operates through the client API, an underlying library accountable for receiving the required parameters and sending requests to the server.

Security Approach

The first phase concerns the implementation of security mechanisms:

- **Authenticity**: every message containing the requested data and the user's public key is signed with the user's private key. The server then verifies the signature, using the user's public key, hence proving its identity.

- **Integrity**. Signatures are done on top of a message digest and sent to the server along with the original message. The server then computes the hash of the original message and decyphers the signature with the public key. Ultimately, both digests should be a match. Otherwise, the server detects the message was tampered with, known as a **man-in-the-middle attack**.

- **Nonces**: prior to an operation, the client requests a new nonce, which is later sent along with the operation request. The server only accepts requests associated with the last generated nonce and, after completing the operation, the saved nonce is removed. Therefore, any **replay attack** that meant using the same nonce would be invalid.

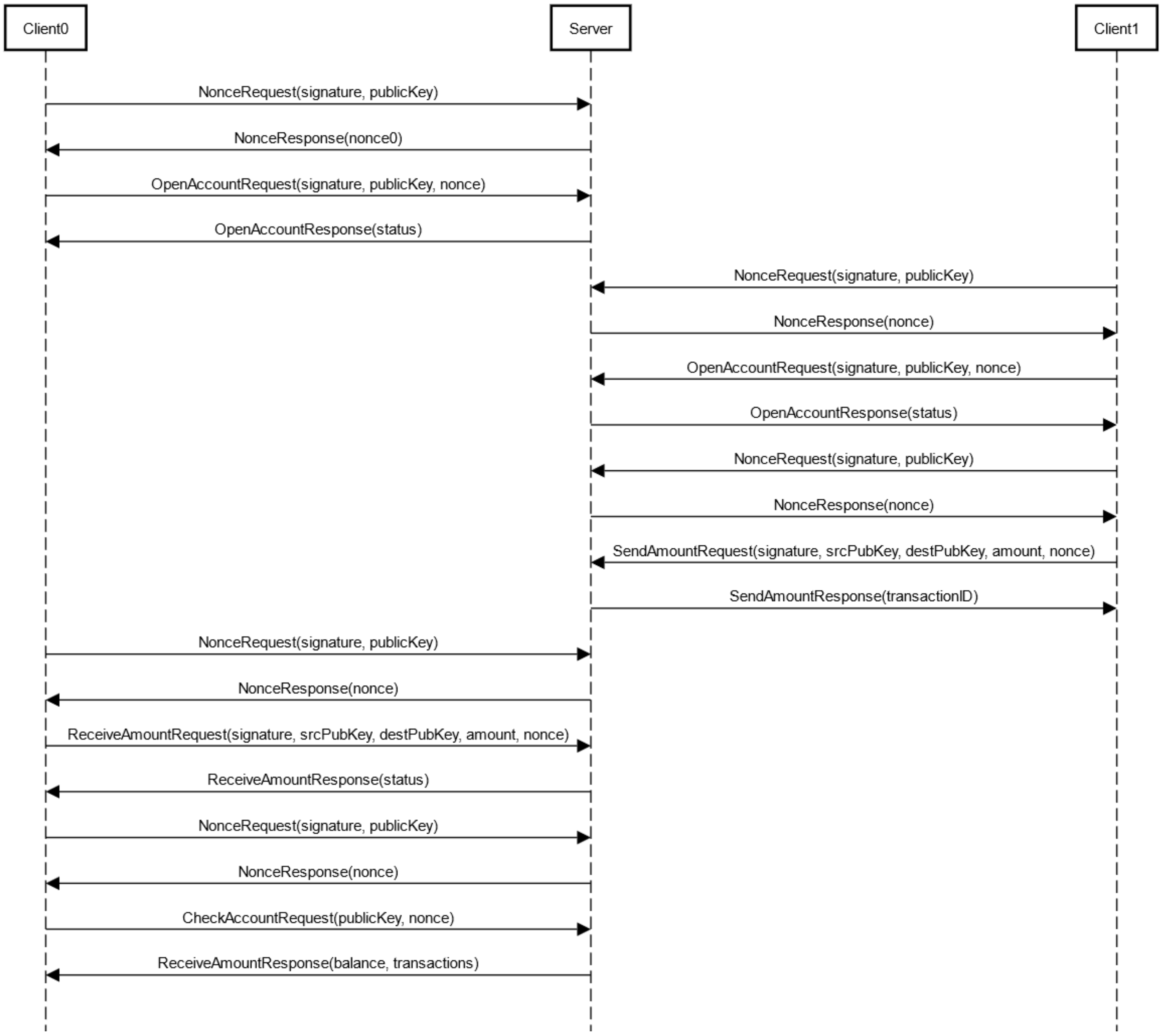
Dependability Approach

The second phase concerns the dependability of the system:

- **Server Atomic Persistence**: the system is able to recover from its previous state. Before every bank operation, the server performs a backup, which is ensured to be correct and noncorrupted. Only then, the previous stable backup is deleted, and the most recent one is renamed as the main backup.
- **Client Persistence**: generated keys are always persistent since the client app manages to save them on the disk.

Figure 1 - Example of a secure client-server communication

Client-Server Communication


Client0
Server
Client1
NonceRequest(signature, publicKey)
NonceResponse(nonce0)
OpenAccountRequest(signature, publicKey, nonce)
OpenAccountResponse(status)
NonceRequest(signature, publicKey)
NonceResponse(nonce)
OpenAccountRequest(signature, publicKey, nonce)
OpenAccountResponse(status)
NonceRequest(signature, publicKey)
NonceResponse(nonce)
SendAmountRequest(signature, srcPubKey, destPubKey, amount, nonce)
SendAmountResponse(transactionID)
NonceRequest(signature, publicKey)
NonceResponse(nonce)
ReceiveAmountRequest(signature, srcPubKey, destPubKey, amount, nonce)
ReceiveAmountResponse(status)
NonceRequest(signature, publicKey)
NonceResponse(nonce)
CheckAccountRequest(publicKey, nonce)
ReceiveAmountResponse(balance, transactions)