# Byzantine Fault Tolerant Banking (BFTB)

SEC Project Stage 2 - 2022
Alameda - G05

Larissa Tomaz ist192506 Rodrigo Gomes ist192548 Marta Brites ist192520

## 1 - Introduction

The goal of this project was to develop a highly dependable banking system, with Byzantine Fault Tolerant (BFT) guarantees, named BFT Banking (BFTB). The BFTB system maintains a set of bank accounts and the clients can perform operations such as transfers between accounts and transactions history requests.
For this stage, we now should be able to tolerate Byzantine Servers and Clients.

## 2 - Assumptions

- There is a Public Key Infrastructure in place, although, for simplicity, the clients and the server use self-generated public/private keys and manual key distribution via shared directories
- There is a total of N = 3f +1 servers, and we can tolerate up to f byzantine servers, that can also be subject to crash failures from which they eventually recover
- The communication channels are not secure and an attacker can drop, manipulate and duplicate messages

## 3 - Proposed solution

Our system consists of the following components:

- Client: The module responsible to translate applications calls into requests to the servers.
- Servers: A set of replicated servers (3f + 1) that execute each client's requests
- Databases: Each replica has a different database used to store user's account and transaction information in a persistent way

For any given command by the user that respects the BTFB API, the client module sends the same request to all the different replicas. The servers execute the request and store/retrieve from their database whatever data needed and give a reply to the client. Once a byzantine quorum (2f +1) of responses (or exceptions) is reached the client proceeds to conclude the execution of the command accordingly.

## 4 - Sequence Diagrams

To show how reads and writes are performed, we'll use the case of the checkAccount and sendAmount requests. The correspondent sequence diagrams are available in the Attachments section.

When the Client wants to perform any read command (for example, the CheckAccount), it goes as follows:

1) It sends the request to all servers, and waits for a Byzantine Quorum (BQ) of replies(2f+1) or system's logic exceptions.

2) After reaching a final valid answer (by checking which valid register has the most updated sequence number associated with it) the Client proceeds to send it to all the servers as an attempt to keep all servers updated with the latest valid information and guarantee the linearizability property.

3) Finally, after reaching a BQ of ack replies, the client returns the valid values to the user.

When the Client wants to perform any write command (for example, the sendAmount), it goes as follows:

1) It sends a preliminary sendAmount request, with the flag isValidated as false, to obtain the values from each server that would be written according to them.

2) After receiving a BQ of responses and deciding the valid values with the highest corresponding sequence numbers to be written, the client signs them and performs a new sendAmount request, this time with the flag isValidated as true to indicate that the request contains effectively the values to be written

3) After receiving the second sendAmount request and before making the corresponding writes, each server will send an echoRequest correspondent to the request they received to all other servers. After receiving a BQ of echos (or one of readies) to a given request, that particular server sends, if it hasn't done so already (which can happen if it receives more than f ready messages) a ready Message to all the other servers and finally executes the request, giving then an ack reply to the client.

# 5 - Solution's Dependability Guarantees

## 5.1: Server Temporary Crash Faults:

Assuming that no more than f servers have crashed at a given time, our solution preserves the correctness of the system in the presence of such faults due to the guaranteed intersection of the read and write quorums, which assures that at least one replica always contains the most updated value. In the presence of more than f crashed servers at the same time, the client alerts the user of the situation and asks him to repeat the request.

## 5.2: Server Byzantine Faults:

When a write is executed, the client signs the register (sequence number + values) to be altered to "authenticate" the operation. The server then stores the register and its corresponding signature. When a read is executed, the client verifies the signatures for the returned registers and ignores those with invalid ones. This way, a Byzantine process can be prevented from returning arbitrary values with higher sequence numbers to the user. If more than f replies inside of the BQ reached have invalid signatures, the client ignores the original request and asks the user to repeat it.

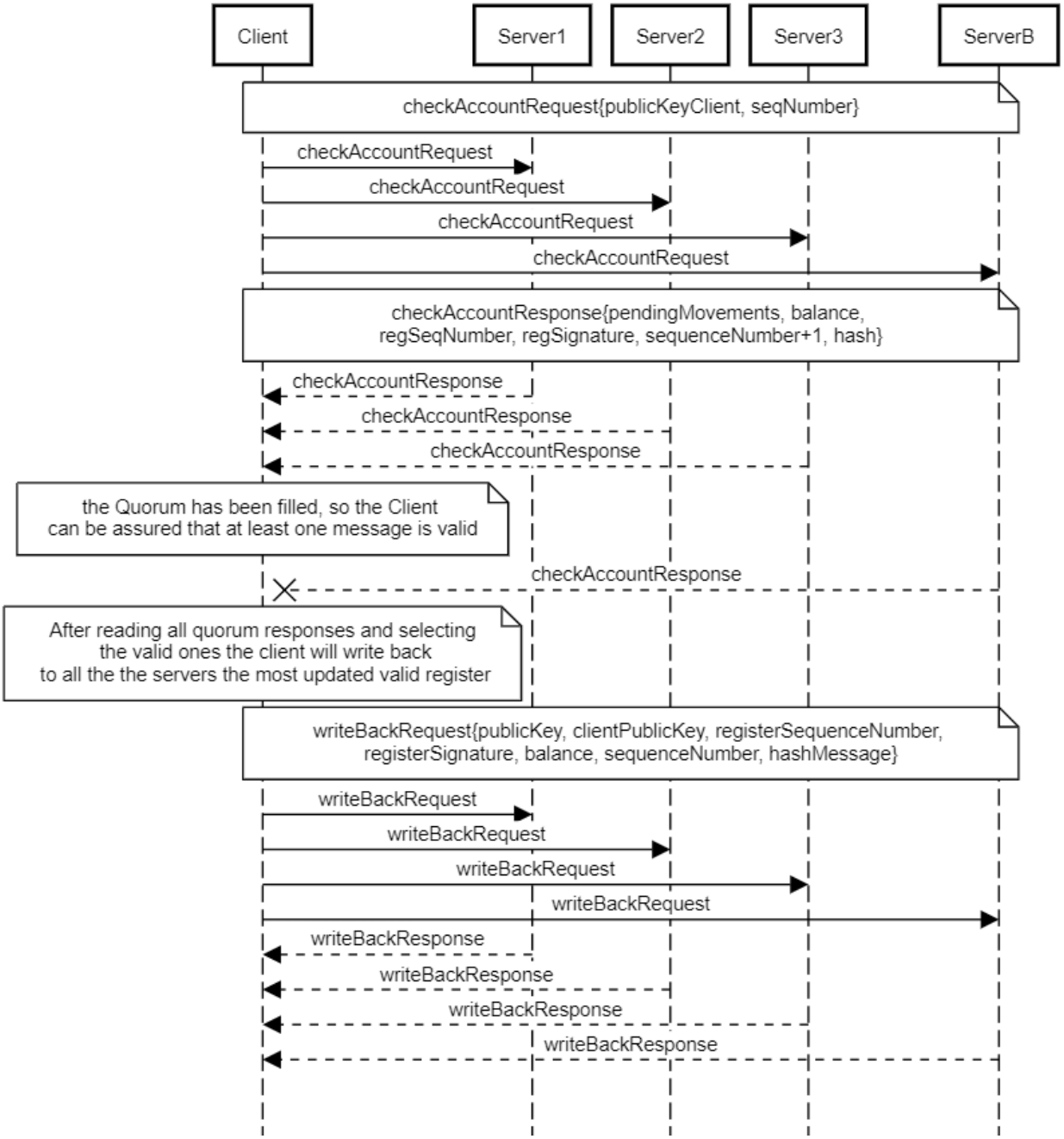
## 5.3: Client Byzantine Faults:

Our solution, through the use of the Byzantine Reliable Broadcast protocol, prevents the specific attack of a Byzantine Client sending different write requests to the servers. Waiting for a BQ of ECHO (or READY) messages for a given write request gives the assurance that at least that same amount of other replicas received the same request and so the server can execute it. If a BQ of such messages for a given request is not received after a given timeout, the request isn't executed by any of the correct replicas and the user is duly informed.

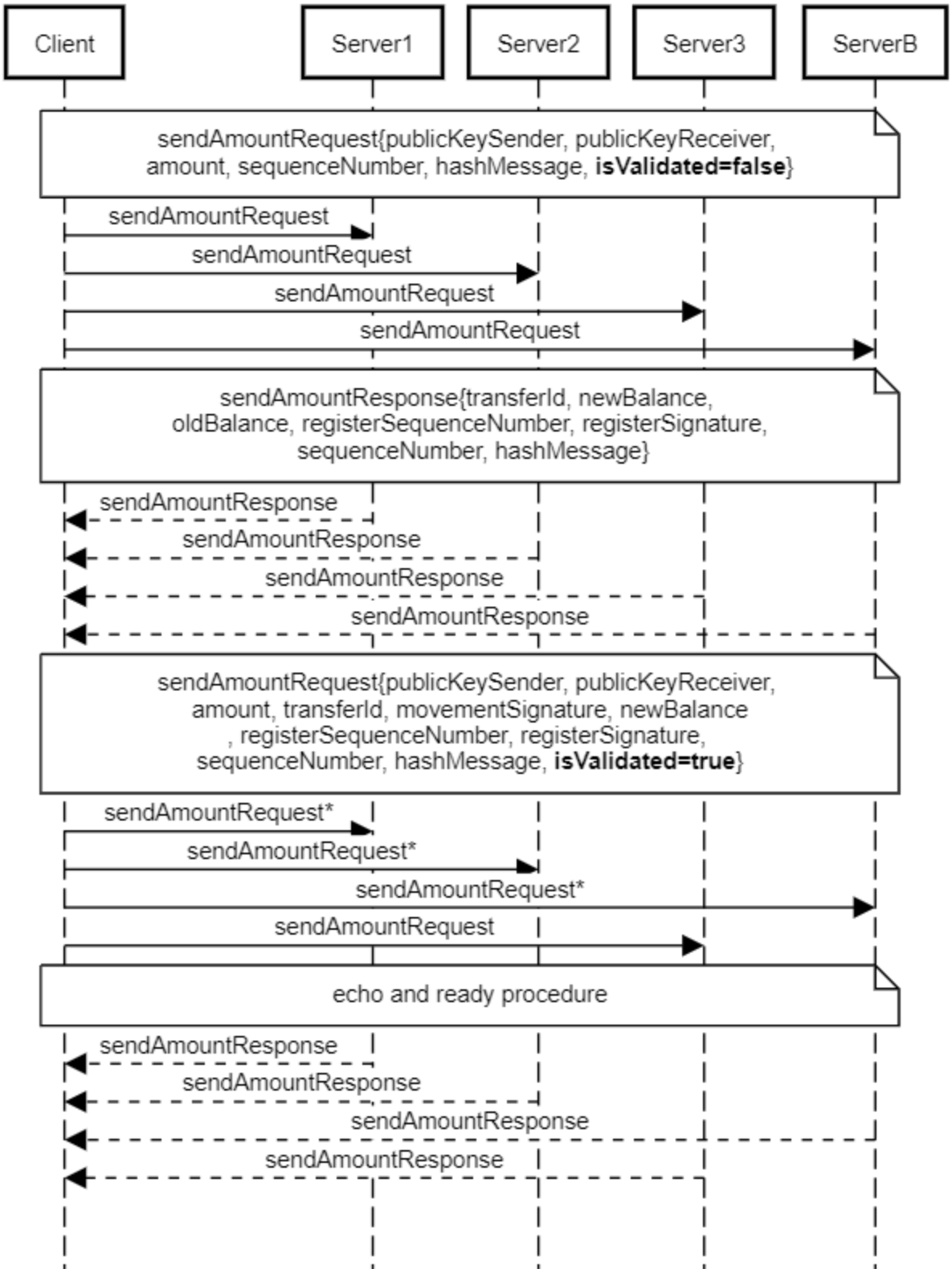
## 6 - Solution's Security Guarantees

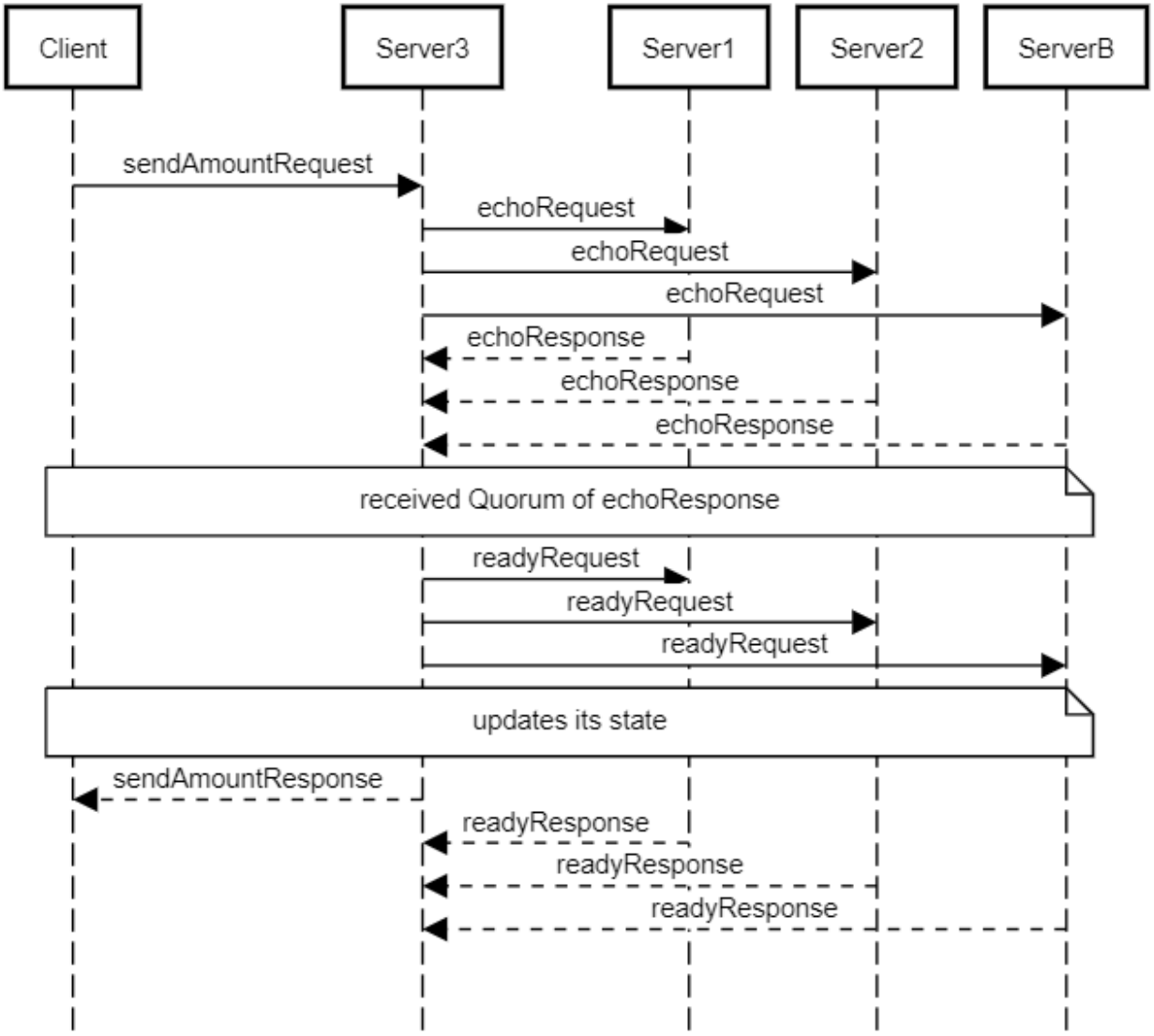
| Property | Attack | How the property is guaranteed |
|---|---|---|
| Message Integrity | Message Tampering | When sending a message, the sender also sends its hash in one of the message's fields. Before any further operation, the receiver confirms if the sent hash is equivalent to the hash of the content sent. |
| Freshness | Replay Attack | The requests exchanged between the server and the client include a nonce (and the replies include a specified operation on that nonce), also used in the calculation of the message's hash. This integrity-protected nonce ensures that duplicated messages are not accepted, thus preventing replay attacks. |
| Non-Repudiation | Man-in-the-Middle (active) / Message Forgery | All the hashes of the exchanged messages are signed with the sender's private key. This ensures that no one without such key can change the content of one of the message's fields and then re-calculate and update the message's hash, as they wouldn't be able to sign it again and would therefore be detected. |
| Authenticity | All the above | Guaranteed by the assurance of fulfillment of the above properties. |

# Read


Client
Server1
Server2
Server3
ServerB
checkAccountRequest{publicKeyClient, seqNumber}
checkAccountRequest
checkAccountRequest
checkAccountRequest
checkAccountRequest
checkAccountResponse{pendingMovements, balance,
regSeqNumber, regSignature, sequenceNumber+1, hash}
checkAccountResponse
checkAccountResponse
checkAccountResponse
the Quorum has been filled, so the Client
can be assured that at least one message is valid
checkAccountResponse
After reading all quorum responses and selecting
the valid ones the client will write back
to all the the servers the most updated valid register
writeBackRequest{publicKey, clientPublicKey, registerSequenceNumber,
registerSignature, balance, sequenceNumber, hashMessage}
writeBackRequest
writeBackRequest
writeBackRequest
writeBackRequest
writeBackResponse
writeBackResponse
writeBackResponse
writeBackResponse

# Write


Client
Server1
Server2
Server3
ServerB
sendAmountRequest{publicKeySender, publicKeyReceiver, amount, sequenceNumber, hashMessage, **isValidated=false**}
sendAmountRequest
sendAmountRequest
sendAmountRequest
sendAmountRequest
sendAmountResponse{transferId, newBalance, oldBalance, registerSequenceNumber, registerSignature, sequenceNumber, hashMessage}
sendAmountResponse
sendAmountResponse
sendAmountResponse
sendAmountResponse
sendAmountRequest{publicKeySender, publicKeyReceiver, amount, transferId, movementSignature, newBalance , registerSequenceNumber, registerSignature, sequenceNumber, hashMessage, **isValidated=true**}
sendAmountRequest*
sendAmountRequest*
sendAmountRequest*
sendAmountRequest
echo and ready procedure
sendAmountResponse
sendAmountResponse
sendAmountResponse
sendAmountResponse


Client
Server3
Server1
Server2
ServerB
sendAmountRequest
echoRequest
echoRequest
echoRequest
echoResponse
echoResponse
echoResponse
received Quorum of echoResponse
readyRequest
readyRequest
readyRequest
updates its state
sendAmountResponse
readyResponse
readyResponse
readyResponse