SOFTWARE FAILURE TOLERANT AND HIGHLY AVAILABLE CORBA DISTRIBUTED BANKING SYSTEM

SOEN 423

November 11th, 2017

Yasmine Chiter 27175299

Nahian Pathan 27105827

Amirali Shirkhodaei 26255906

Inna Taushanova-Atanasova 27876947

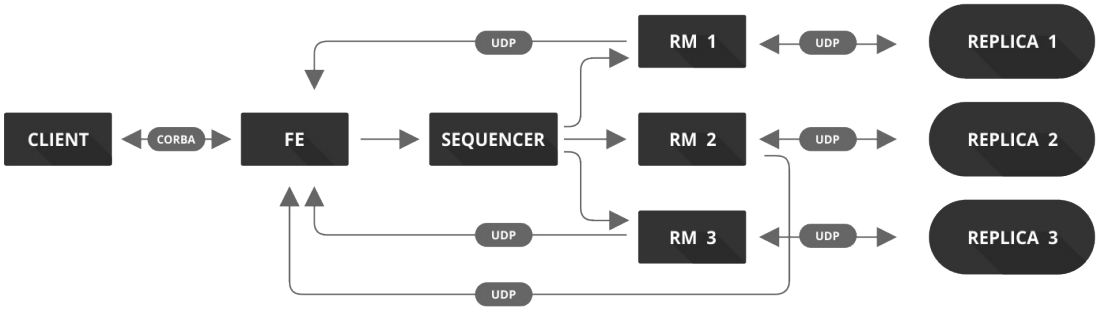
Introduction

The objective of this project is to develop a Banking system in which customers and managers could perform everyday banking operations. The system should be able to retrieve information for clients as well as allow clients to alter the current state of their account(s). Each phase of this system will be logging any operation being made to keep a history of all transactions performed on each account. All of this should be done in a transparent way meaning that the client should not be aware of what is happening behind the scene.

This project is based on the client-server model and it requires the implementation of UDP protocols to facilitate server-server communication as well as the implementation of CORBA for client method invocation.

This system should be able to handle crash failures as well as Byzantine failures. To accommodate to this requirement, we have designed an active replication scheme using process group replication and reliable group communication which will be discussed further in the upcoming sections.

Architecture



Active Replication

Active replication is a way to provide fault tolerance to distributed systems. Specifically, active replication can handle byzantine faults because the FE receives multiple responses in which it can compare them to see if a failure has occurred. In active replication, RM's are equivalent state machines which receive requests from the FE which forwards it to the Sequencer. The Sequencer then multicasts the requests and each RM executes the same request concurrently. In our case, each RM has a different implementation but should produce the exact results as the other RM's.

Active Replication Process:

1. Request: The front end (FE) receives the request and forwards it to the sequencer which assigns a unique id number to the request and multicasts it to the replica managers (RM).
2. Coordination: The RMs coordinate to execute the request consistently.
3. Execution: The RMs execute the request. The response will contain the requests unique id number.
4. Agreement: In our case, no agreement phase is necessary due to the multicast delivery semantics.
5. Response: Each RM sends its response to the FE. The FE receives all the responses and performs a check to see which responses have faults, then returns one correct value back to the client.

Request Ordering

For our project, we used FIFO ordering to handle the requests. When a request it made, it gets added to a queue and waits its turn to be handled. Similarly, when we receive responses, they get placed into an queue and sent to their respective clients based on the order in which they arrive.

Front End (FE)

The purpose of the front end is to connect the user to the replica. It acts as a request dispatcher, receiving user requests via CORBA and forwarding them to the sequencer for further handling. Once the FE receives the results, it will forward a single correct result back to the client. FE is also in charge of informing replica managers of any replica which produced an incorrect result.

Sequencer

A sequencer receives a client request from the FE, assigns a unique id number to the request and multicasts the request with the sequence id and FE information to all the RM's. It contains a Queue data structure to hold the requests in order.

Client

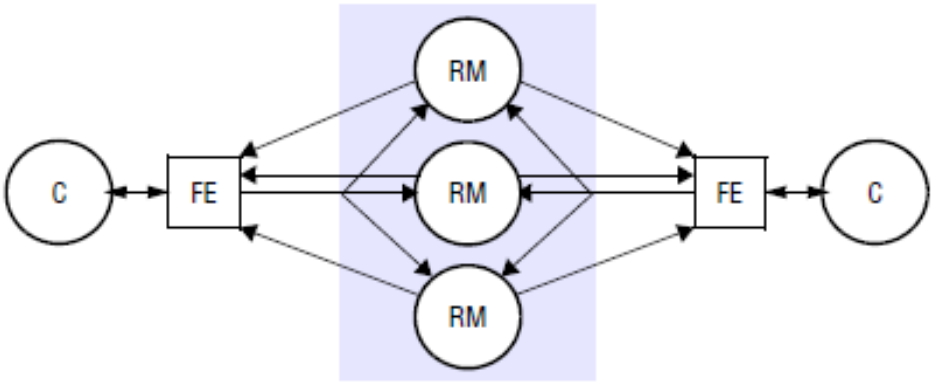
Clients can be of two types, customers and managers. Customers can only invoke specific methods where as managers can invoke all methods defined within the replicas. These invocations are done through CORBA.

Replica Manager

The replica manager (RM) is the middleware between the FE and replicas. It receives the request from the sequencer, processes the request and invokes its connected replica to return a result. The RM is also aware of the locations of the other RMs and replicas for when there's a need for a failure recovery. The RM acts as a server when connected to the sequencer and as a client when connected to other RMs. RMs also manage the life of replicas by sending a UDP message to the branch server proxy holding the incorrect replicated data.

Replica

A replica class contains all the banking methods that we are to use for our system. Not all of our replica classes are identical, rather they are based on each individual team member's implementation. However, all the methods invocations should generate the same results. Our implementation guarantees that a replicated process does not alter the state of another replica. The best-case scenario is that all replicas hold the same correct value. The worst-case scenario is that the returned value will be determined by a majority vote.



UDP

Client-side:

1. Client process requests the OS for a port number.
2. The client process gets assigned an ephemeral port number which gets destroyed when the process terminates
3. A client request is marshalled and placed into a datagram packet.
4. The process now sends packets by queuing them in the outgoing queue and providing the sources port number.

Server-side:

1. A datagram socket is created and assigned a port number.
2. Two-byte arrays are created, one for sending data and one for receiving data.
3. Server is constantly listening for any incoming messages.
4. Once a request is received, it gets stored in the data array.
5. The request is unmarshalled and depending on what was sent, the appropriate methods gets called.
6. Once the operation has been performed, the result is marshalled and put into the data array and is sent back to the client.

Queues on the server side stay alive as long as the server is running.

RMs must also be able to communicate with other RMs in case of failure. We do this through ping messaging.

1. FE notifies all the RMs of any replica which has produced an incorrect result.
2. All the RMs will have a variable to keep track of the number of incorrect results a replica produces.
3. If the replica produces an incorrect result 3 or more times, the RMs replace that replica with another correct one.
4. If the FE does not receive a response from a replica within twice the time taken for the slowest result (so far) it informs the RMs that there may have been a crash.
5. RMs then check the replica of concern and replace it with another working replica if they agree that the replica has crashed. This is done through the UDP communication process discussed above.

Byzantine scenarios

Our project must detect a single non-malicious Byzantine failure and correct it using active replication. The testing will be done by making one of the replicas send incorrect results. Since our project requires a single software (non-malicious Byzantine) failure, we can assume that only one replica will generate incorrect results.

The front end FE, which is keeping track of the each replica and the number of times they have consecutively sent false data, can then detect the single Byzantine failure by comparing two results that agree. This can be achieved by a hash table containing replica id and the number of consecutively sent false data.

CORBA

Users of CORBA can use the technique to remotely invoke methods without knowing about the implementation(transparency). One of the key features of CORBA is that it is platform and language independent. CORBA uses and file of type IDL, which makes it language independent. IDL file contains the all the method declarations which can be used across the system. The IDL file is compiled with the appropriate language (such as java and c++) directive to work with the chosen language's implementation. COBRA also uses an object of ORB, which is the main distributed object. This object is created and attached to the servant object (implementation class) and distributed across servers and clients. ORB object can act as server, client or both. One of the main implementation feature that helps with CORBA is the Naming Service. It generates a unique ID/ NAME for the CORBA object, which can be understood by any language/platform.

CORBA Implementation and Execution Requirements:

CORBA requires at least the following classes and interface to function (NOTE: * here would mean the name of the interface declared in the IDL file)

• IDL (interface declaration language) - This is where all the methods are declared to be used later by clients. The idl has its own set of type. It is the programmer's duty to see if the types correspond to the requirement of the software. E.g if the software requires a HashMap to be returned from a method, the programmer must define a HashMap in the IDL file as IDL doesn't have many predefined data structures like HashMap.

- IDL must be complied using Java's directive ◊ idlj -fall [file name]

- If done correctly, it should produce all the java files needed for CORBA connection (The folder containing all the files should be in a single folder named after 'module' in IDL)

• Class to define methods declared in the interface, also known as the Servant class - This is the class where the methods, initially declared in the interface (IDL interface and its *Operation.java file) are defined. The class should extend *POA, one of the many classes created after running idlj -fall command. Apart from defining the methods in this, this class should also have an attribute of the class ORB and setter for it. With the help of the ORB object, this enables the methods to be used in stub by the client and skeleton by the Server.

• Server class - This class initializes an object of servant class along with an ORB object to go with it. With the help of the Name service, the new Implementation can now be distributed along the network.

• Client class - This is where another ORB is created and using naming services, the server connected to the ORB is located. The client can now invoke methods defined using java in the servant class.

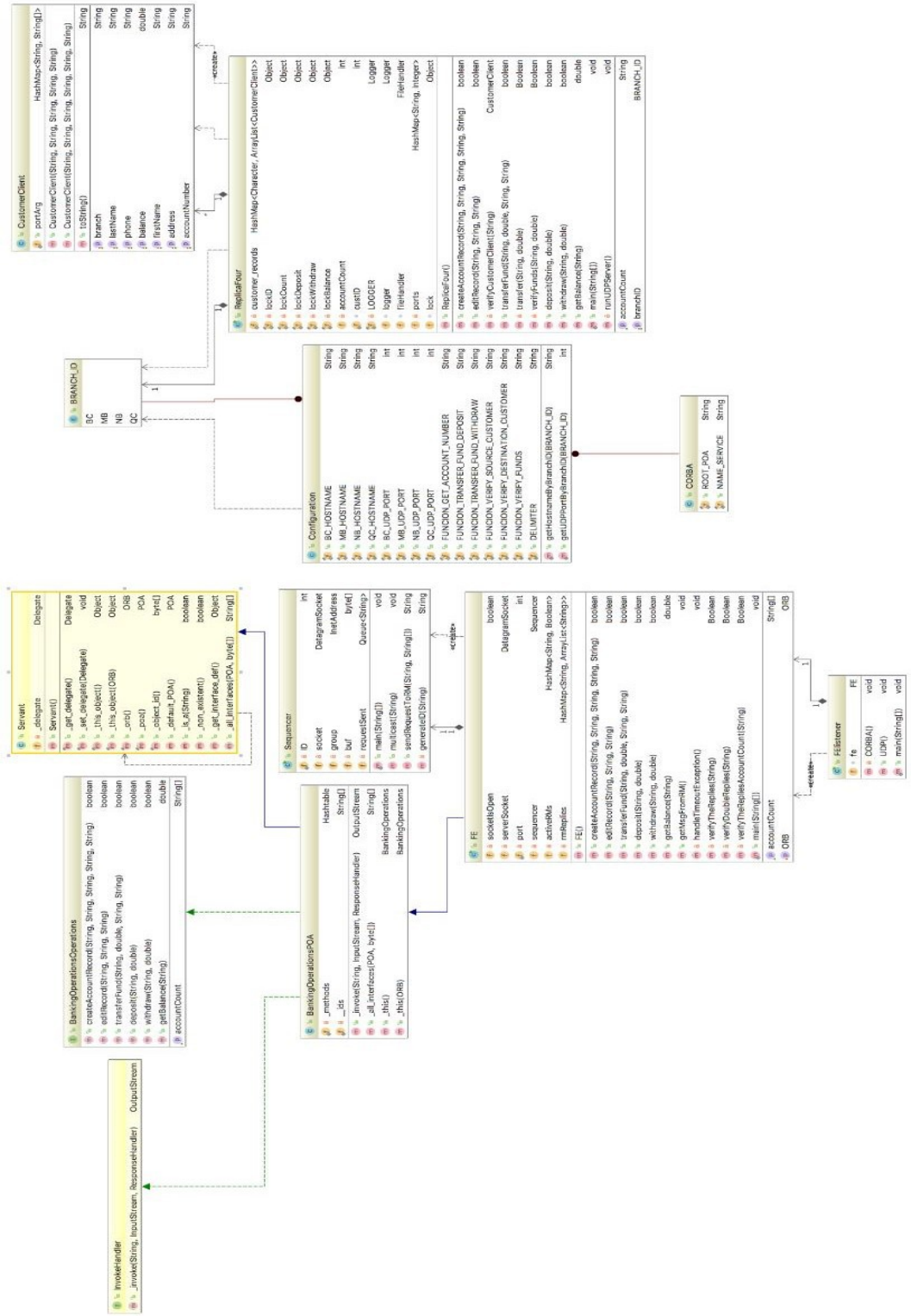
Steps to setup CORBA:

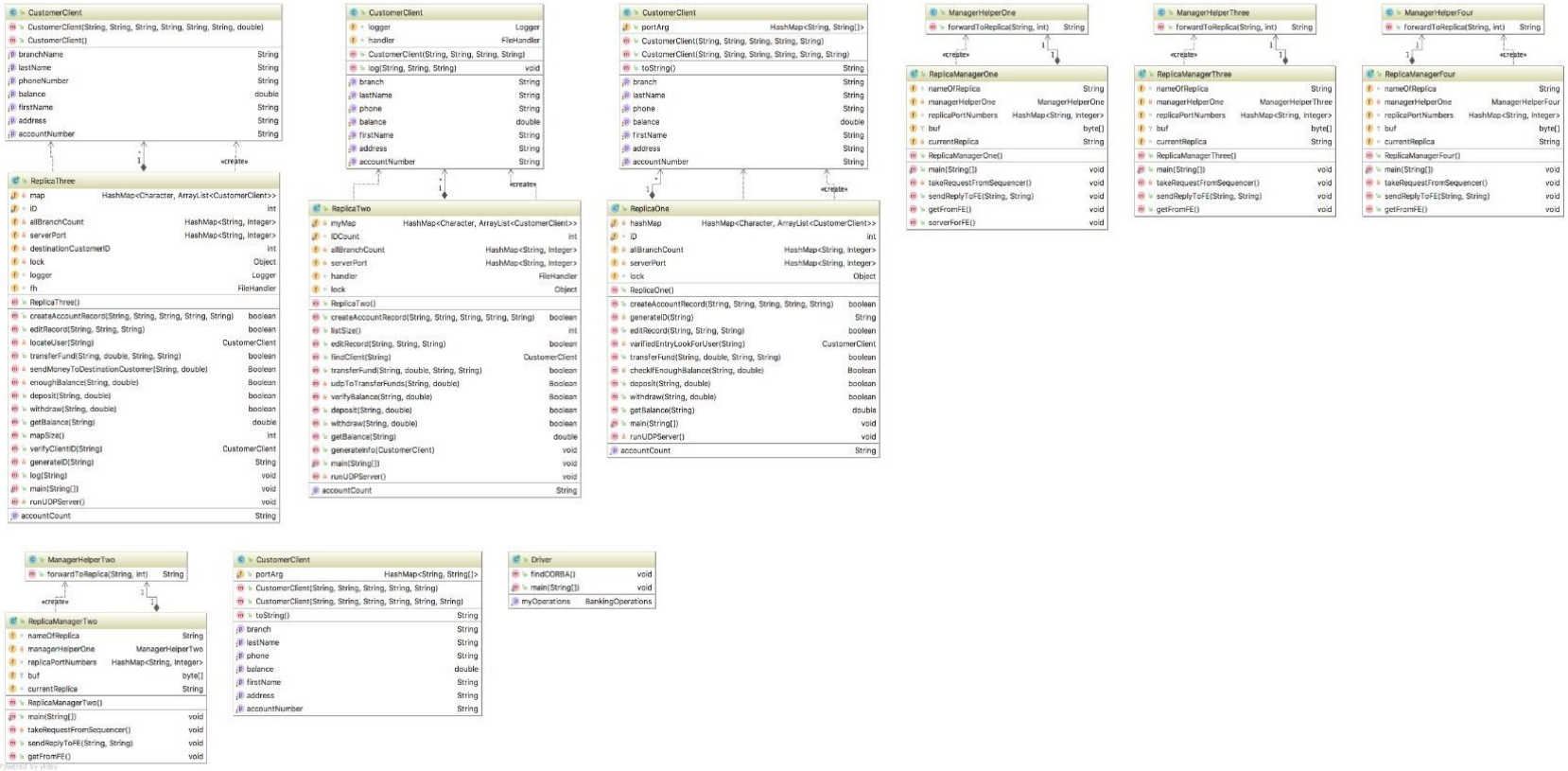
1. Define an IDL Interface. All the methods to be invoked are declared in this interface.
2. A POA class is generated from the idlj compiler and each of the server implementation classes extend this POA class.
3. Create and initialize an ORB.
4. Get a reference to the rootpoa and activate the POAManager.
5. Create the servant, register it with the ORB and get the object reference from the servant.
6. Cast this reference to a CORBA reference.
7. Get the root naming context and bind the object reference.
8. Invoke the orb run() method and wait for invocations from the client.
9. A client must get a reference to the remote object and initialize an instance of the ORB.
10. From there, a client is able to invoke remote methods

HashMap

Each replica class contains 1 hash map's where customer client objects get stored. A Hash Map is a data structure consisting of Key-Value pairs. The keys for the HashMap are Characters(letters) and the values are ArrayLists. Each ArrayList contains customer clients all whose names begin with the same character as the key.

Class Diagrams




CustomerClient
CustomerClient(String, String, String, String, String, String, double)
CustomerClient()
branchName String
lastName String
phoneNumber String
balance double
firstName String
address String
accountNumber String
«create»
ReplicaThree
map HashMap<Character, ArrayList<CustomerClient>>
ID int
allBranchCount HashMap<String, Integer>
serverPort HashMap<String, Integer>
destinationCustomerID int
lock Object
logger Logger
fh FileHandler
ReplicaThree()
createAccountRecord(String, String, String, String, String) boolean
editRecord(String, String, String) boolean
locateUser(String) CustomerClient
transferFund(String, double, String, String) boolean
sendMoneyToDestinationCustomer(String, double) Boolean
enoughBalance(String, double) Boolean
deposit(String, double) boolean
withdraw(String, double) boolean
getBalance(String) double
mapSize() int
verifyClientID(String) CustomerClient
generateID(String) String
log(String) void
main(String[]) void
runUDPServer() void
accountCount String
CustomerClient
logger Logger
handler FileHandler
CustomerClient(String, String, String, String)
log(String, String, String) void
branch String
lastName String
phone String
balance double
firstName String
address String
accountNumber String
«create»
ReplicaTwo
myMap HashMap<Character, ArrayList<CustomerClient>>
IDCount int
allBranchCount HashMap<String, Integer>
serverPort HashMap<String, Integer>
handler FileHandler
lock Object
ReplicaTwo()
createAccountRecord(String, String, String, String, String) boolean
listSize() int
editRecord(String, String, String) boolean
findClient(String) CustomerClient
transferFund(String, double, String, String) boolean
udpToTransferFunds(String, double) Boolean
verifyBalance(String, double) Boolean
deposit(String, double) boolean
withdraw(String, double) boolean
getBalance(String) double
generateInfo(CustomerClient) void
main(String[]) void
runUDPServer() void
accountCount String
CustomerClient
portArg HashMap<String, String[]>
CustomerClient(String, String, String, String, String)
CustomerClient(String, String, String, String, String, String)
toString() String
branch String
lastName String
phone String
balance double
firstName String
address String
accountNumber String
«create»
ReplicaOne
hashMap HashMap<Character, ArrayList<CustomerClient>>
ID int
allBranchCount HashMap<String, Integer>
serverPort HashMap<String, Integer>
lock Object
ReplicaOne()
createAccountRecord(String, String, String, String, String) boolean
generateID(String) String
editRecord(String, String, String) boolean
varifiedEntryLookForUser(String) CustomerClient
transferFund(String, double, String, String) boolean
checkIfEnoughBalance(String, double) Boolean
deposit(String, double) boolean
withdraw(String, double) boolean
getBalance(String) double
main(String[]) void
runUDPServer() void
accountCount String
ManagerHelperOne
forwardToReplica(String, int) String
«create»
ReplicaManagerOne
nameOfReplica String
managerHelperOne ManagerHelperOne
replicaPortNumbers HashMap<String, Integer>
buf byte[]
currentReplica String
ReplicaManagerOne()
main(String[]) void
takeRequestFromSequencer() void
sendReplyToFE(String, String) void
getFromFE() void
serverForFE() void
ManagerHelperThree
forwardToReplica(String, int) String
«create»
ReplicaManagerThree
nameOfReplica String
managerHelperOne ManagerHelperThree
replicaPortNumbers HashMap<String, Integer>
buf byte[]
currentReplica String
ReplicaManagerThree()
main(String[]) void
takeRequestFromSequencer() void
sendReplyToFE(String, String) void
getFromFE() void
ManagerHelperFour
forwardToReplica(String, int) String
«create»
ReplicaManagerFour
nameOfReplica String
managerHelperOne ManagerHelperFour
replicaPortNumbers HashMap<String, Integer>
buf byte[]
currentReplica String
ReplicaManagerFour()
main(String[]) void
takeRequestFromSequencer() void
sendReplyToFE(String, String) void
getFromFE() void
ManagerHelperTwo
forwardToReplica(String, int) String
«create»
ReplicaManagerTwo
nameOfReplica String
managerHelperOne ManagerHelperTwo
replicaPortNumbers HashMap<String, Integer>
buf byte[]
currentReplica String
ReplicaManagerTwo()
main(String[]) void
takeRequestFromSequencer() void
sendReplyToFE(String, String) void
getFromFE() void
CustomerClient
portArg HashMap<String, String[]>
CustomerClient(String, String, String, String, String)
CustomerClient(String, String, String, String, String, String)
toString() String
branch String
lastName String
phone String
balance double
firstName String
address String
accountNumber String
Driver
findCORBA() void
main(String[]) void
myOperations BankingOperations

FElistener class

Name	Return Type	Parameter	Description
CORBA	None	None	• Creates a local ORB object; • Get reference to RootPOA and get POAManager; • Creates a servant instance and register it with the ORB; • Gets a CORBA object reference for a naming context in which to register the new CORBA object; • Bind the object reference to the Naming Context; • Runs the server.

FrontEnd class

Name	Return Type	Parameter	Description
getMsgFromRM	None	None	• The method receives the replies from the Replica Managers; • Keeps track of the active Replica Managers; • Sets the time for receiving the reply from the RM to twice the time taken for the slowest result so far;
handleTimeoutException	None	None	• Handles the case when Timeout exception occurs
verifyTheReplies verifyDoubleReplies verifyTheRepliesAccountCount	Boolean	String idRequest	• The FronEnd returns a single correct result back to the client as soon as three identical (correct) results are received from the replicas.

Sequencer

Name	Return Type	Parameter	Description
multicast	None	String multicastMessage	• Multicast the message to Replica Managers
sendRequestToRM	String	String nameOfTheMethod, String [] args	• Generates a String containing the name of the method and the arguments separated by "-" • Calls the generateID method and passes as parameter the generated string • Invokes the multicast method
generateID	String	String req	• Generates a unique requestID and attaches it to the request that is passed as parameter.

Replica Manager

Name	Return Type	Parameter	Description
takeRequestFromSequencer	None	None	• Receives the request from the Sequencer • Splits the message and extracts the method name and requestID • Generates a new request containing only the method name and the parameters being passed • Send the request to the Replica • Receives the reply from the Replica • Invokes sendReplyToFE method
sendReplyToFE	None	String requestID, String replyFromReplicaString	• Generates a message containing the requestID and the reply from the Replica • Sends the message back to the FrontEnd
getFromFE	None	None	• Joins the multicast group • Receives multicast message

WORKFLOW

1. The client (Driver class) communicates with the FrontEnd through CORBA.

2. The FrontEnd receives the request and invokes an object of the Sequencer class which first assigns a unique request ID, attaches it to the method name and parameters, and then multicasts the message to the Replica Managers. The ID of the request is stored as a key in a HashMap, whose value is an ArrayList holding the replies from the Replicas. The FElistener class is a helper class for establishing the CORBA connections.

3. Replica Manager class receives the multicast request from the Sequencer, extracts the request ID and generates a new request that is sent to the Replica through UDP.

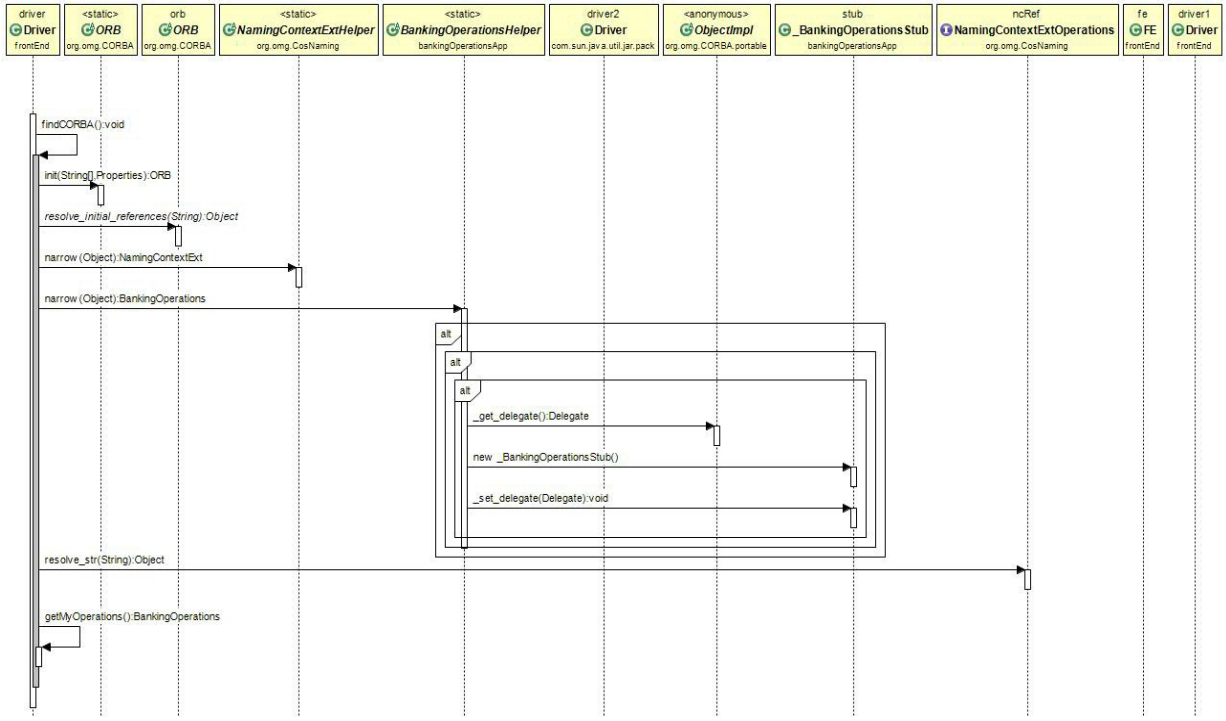
4. Replica class executes the request and sends back the reply to the ReplicaManager.

5. The Replica Manager generates a reply which is sent back to the FrontEnd through UDP. The reply consists of the requestID, Replica Manager's name, the current time in milliseconds and the reply from the Replica.

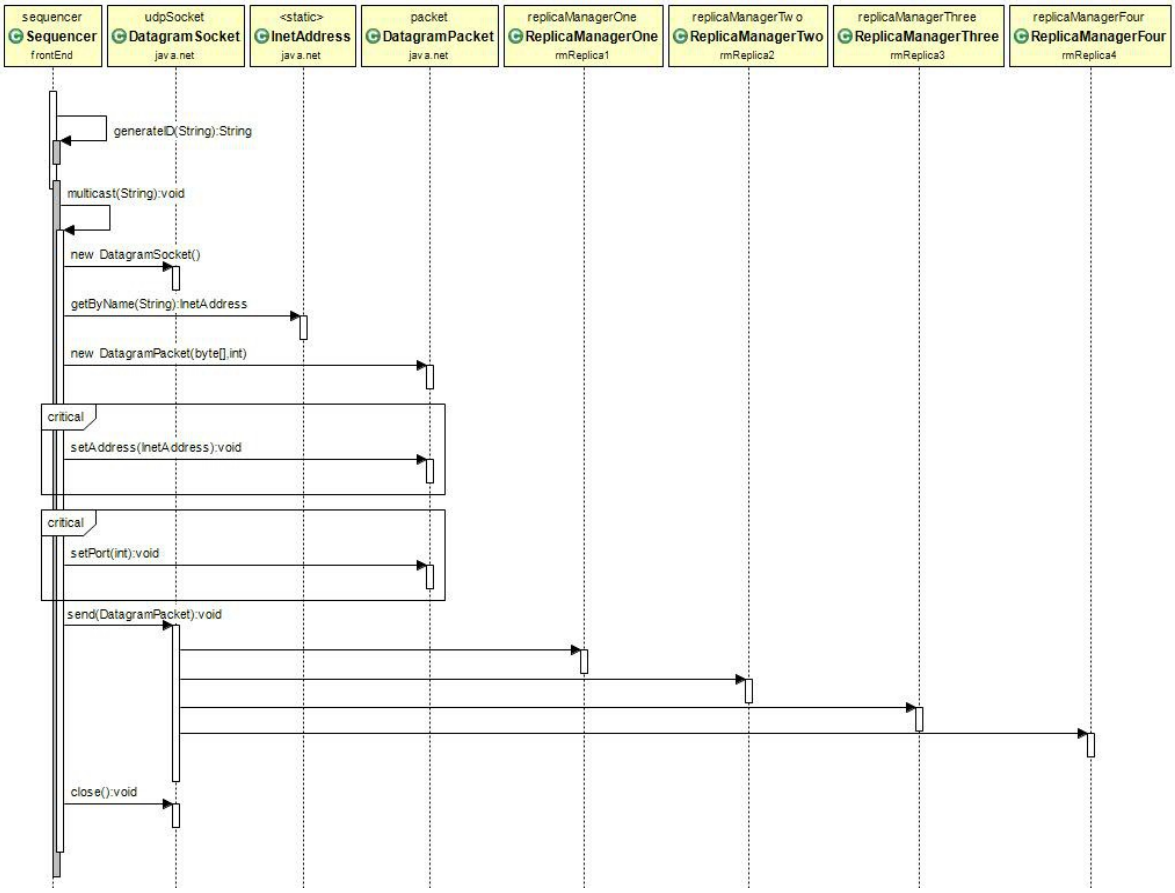
6. The FrontEnd receives the replies from all the Replica Manages, stores them in the ArrayList associated with the requestID and then performs a verification. The FrontEnd returns a single correct result back to the client as soon as three identical (correct) results are received from the replicas.

Sequence Diagrams

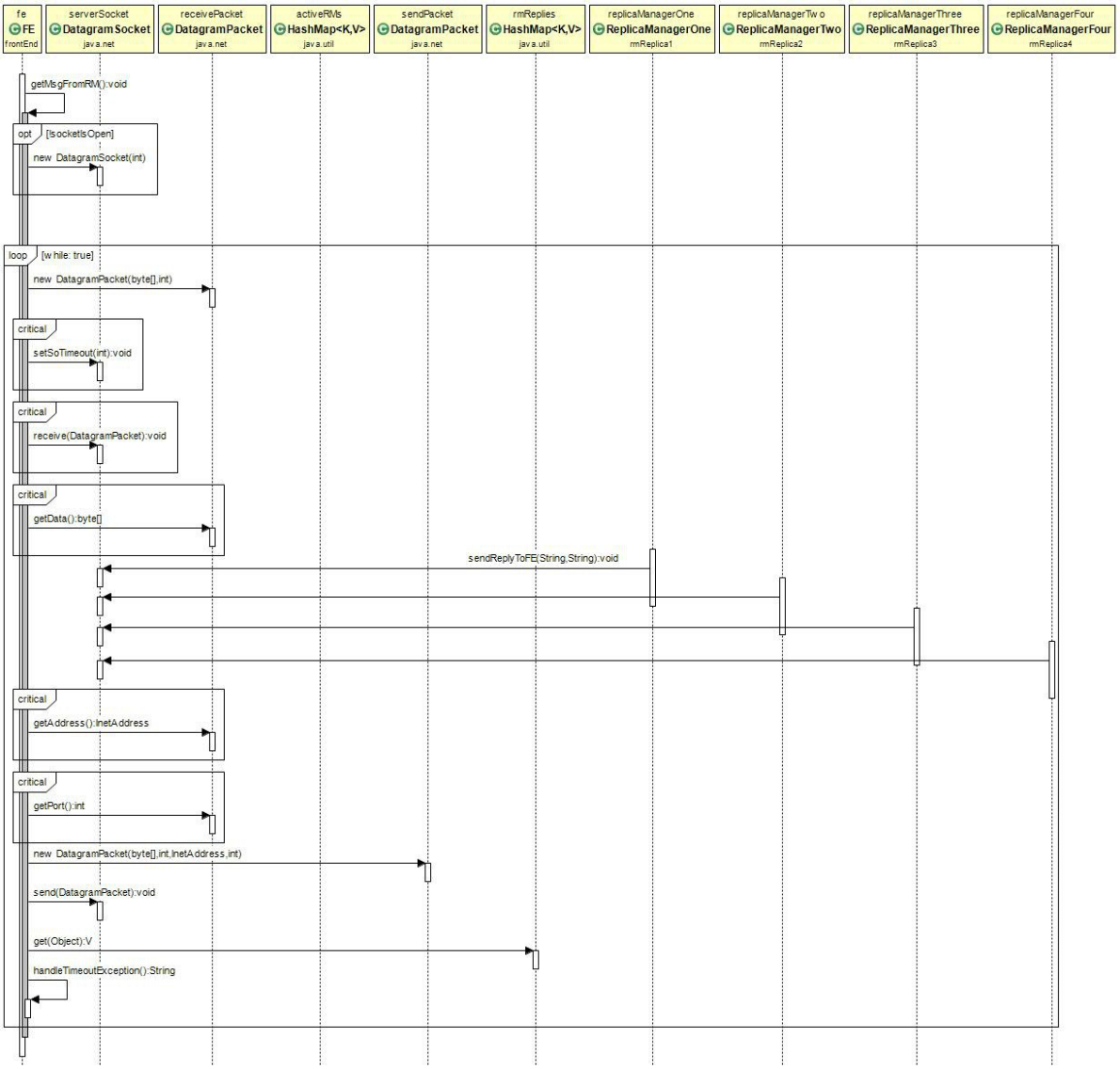
1. Corba



2. Send a request to Replica Managers



3. Get response from the Replica Managers



4. Operations