System Design Document

Software Failure Tolerance or Highly Available CORBA Distributed Banking System

Prepared for: SOEN 423 - Distributed Systems
Version 2.0

Team Members

Radu Saghin	27667086
Mathieu Breault	27093969
Sylvain Czyzewski	27066333

“We certify that this submission is the original work of members of the group and meets the Faculty's Expectations of Originality”, 2017-12-03

Revision History

- Only 1 front-end is used, which dispatches the messages to the right branch group instead of multiple front-end.
    - Simplify the clients implementation and decouple it from the server architecture.
- Removed InterBranchBus as it's responsibilities were given to the front end.
- Deployment was modified to a single machine deployment setup as required by the tutor.
- Addition of a unique Reliable UDP stack used by all module of the system
    - Consistent communication method between the modules and reduction of duplicated workload.

Work Log

- Mathieu Breault
    - Reliable UDP Stack
    - Sequencer
    - Architectural Design
    - Documentation
    - Test Clients
- Radu Saghin
    - Front-End
    - Architectural Design
    - Documentation
- Sylvain Czyzewski
    - Replica Manager
    - Project Modularization + Deployment Scripts
    - Architectural Design
    - Documentation
    - Clients

Instructions

See the README.md file in the root of the project for instructions on how to start and use the system.

1 Introduction

1.1 Purpose

The purpose of this document is to describe the design decisions for the creation of a software failure tolerant or highly available distributed banking system using CORBA. The system's main requirements are found in the project description given in class. The design was created to maximize the following concerns:

1. Choice of highly available or failure tolerant mode
2. Maximize concurrency for client requests to the systems
3. Location Transparency
4. Access Transparency
5. Efficient server to server communication

2 Architecture Design

2.1 Overview

The architecture for the failure tolerant or highly available distributed banking system is built with the aim of having the client communication with the server transparent from the replication system. The client is using the CORBA middleware and system to communicate with multiple front-ends which abstract the underlying replication system of the architecture. The client uses the CORBA naming service to discover the front-end which forwards the query using Reliable UDP to the associated branch. The Front-End uses unique ids for each front-end to discover their listener's location.

The front-end handles transforming CORBA RMI into UDP messages to its appropriate server replication cluster. The replication cluster for each branch in the system is accessed through the Sequencer module, which handles the total ordering of the messages from the front-end and reliably multicasting these messages to each ReplicaManager. The ReplicaManager dispatches and receives messages to and from its associated replicas.

When a message is received back from each ReplicaManager to the FrontEnd it then sends the majority response to the client by a synchronous CORBA RMI reply. The FrontEnd will analyze if a message is not received and send a report to each of the ReplicaManager in the cluster which will then handle keeping track of faults.

The ReplicaManagers will use the fault information from the front-end to detect errors and communicate with the other ReplicaManagers in the cluster for the appropriate action to take.

Each module in the system will use a middleware like module for handling the UDP messaging and marshalling/unmarshalling. Each module, when going online, will register a unique ID composed of its role and its associated cluster if necessary <role><cluster>.

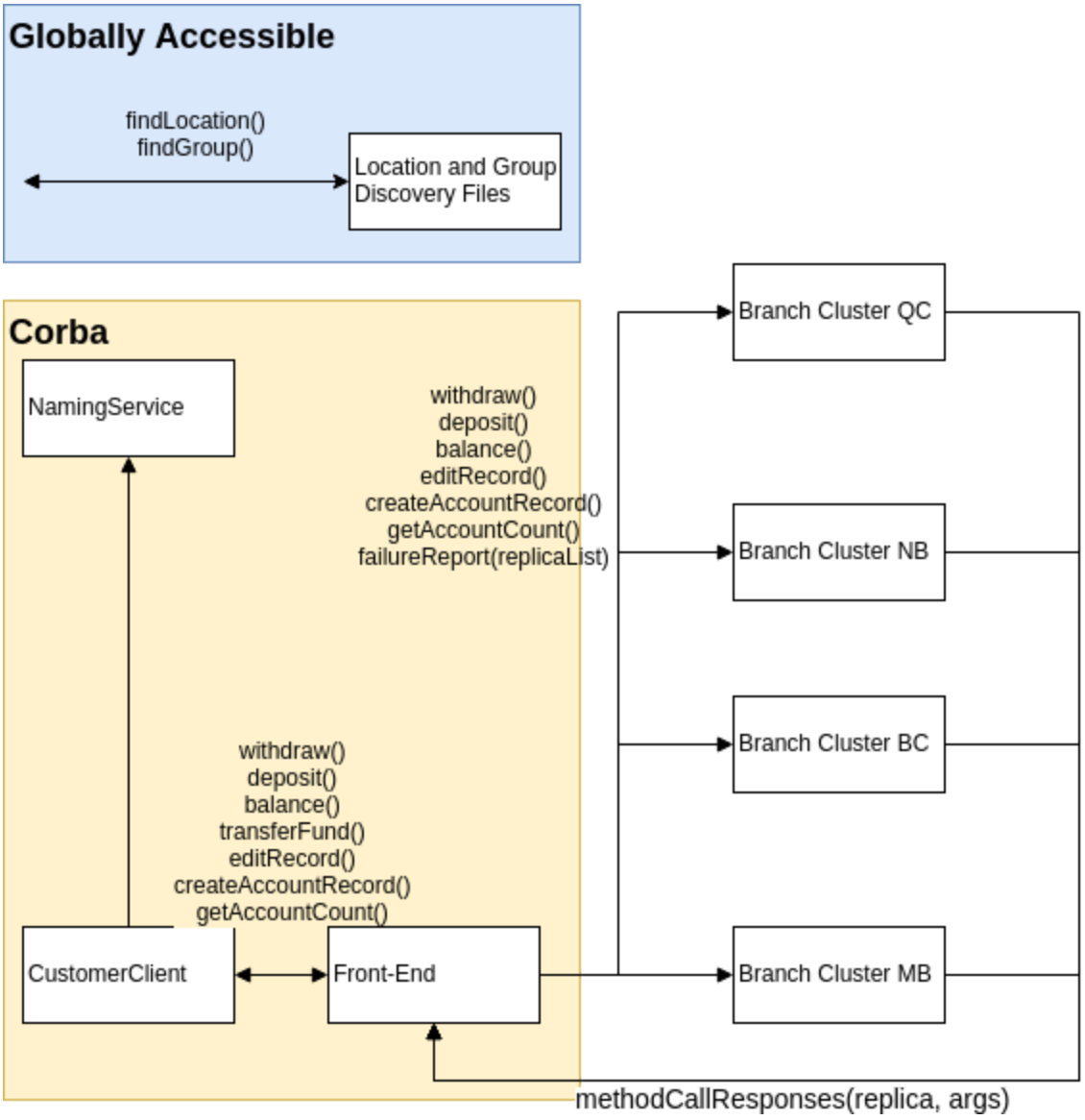
Role Names:

- FrontEnd : FE
- ReplicaManager : RM<num><BRANCH>
- Sequencer : SEQ<BRANCH>

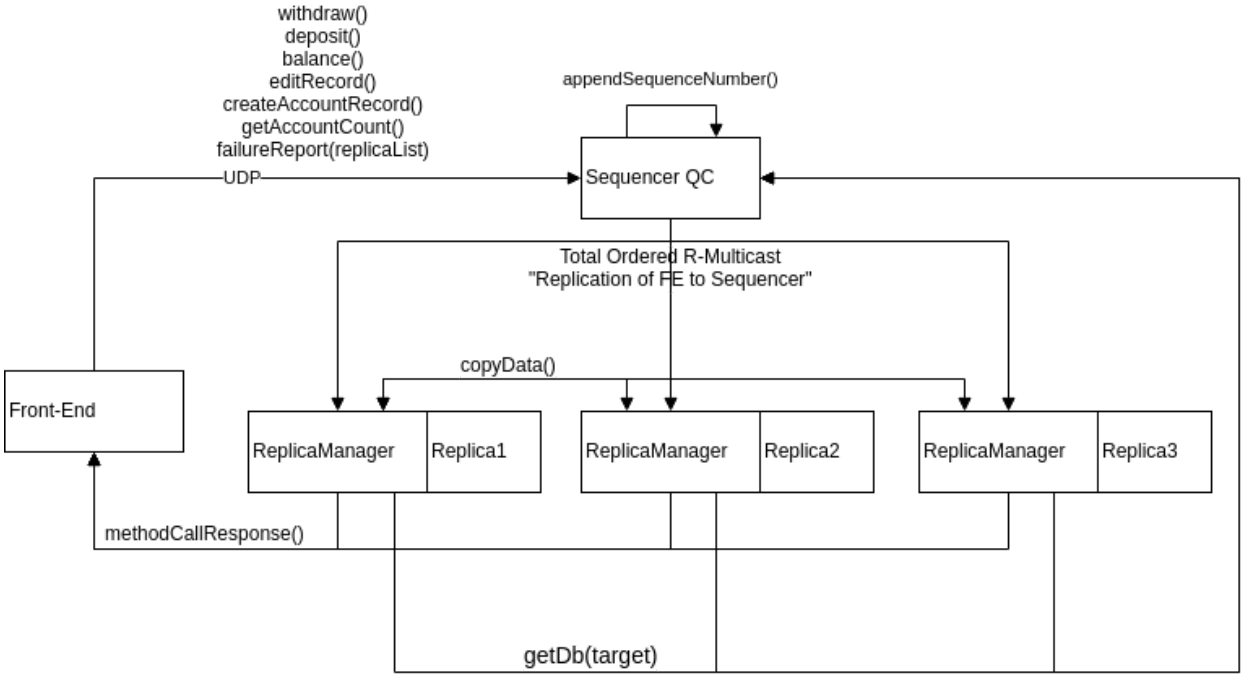
Example:

- Front-End: FE
- ReplicaManager1 for QC: RM1QC
- ReplicaManager2 for QC: RM2QC
- ReplicaManager3 for QC: RM3QC
- Sequencer: SEQQC

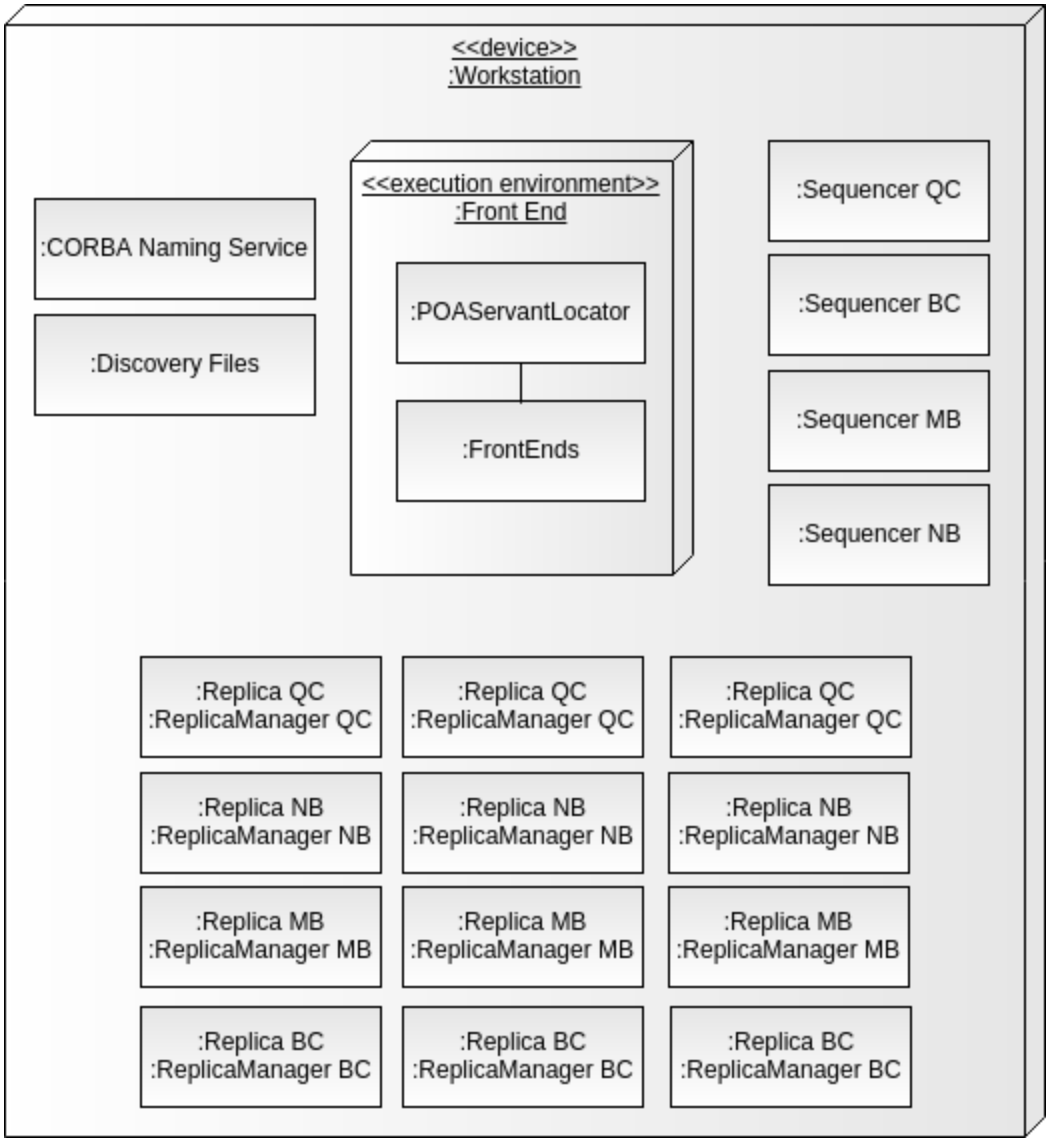
2.2 High-level Architecture



2.3 Branch Cluster



2.4 Deployment



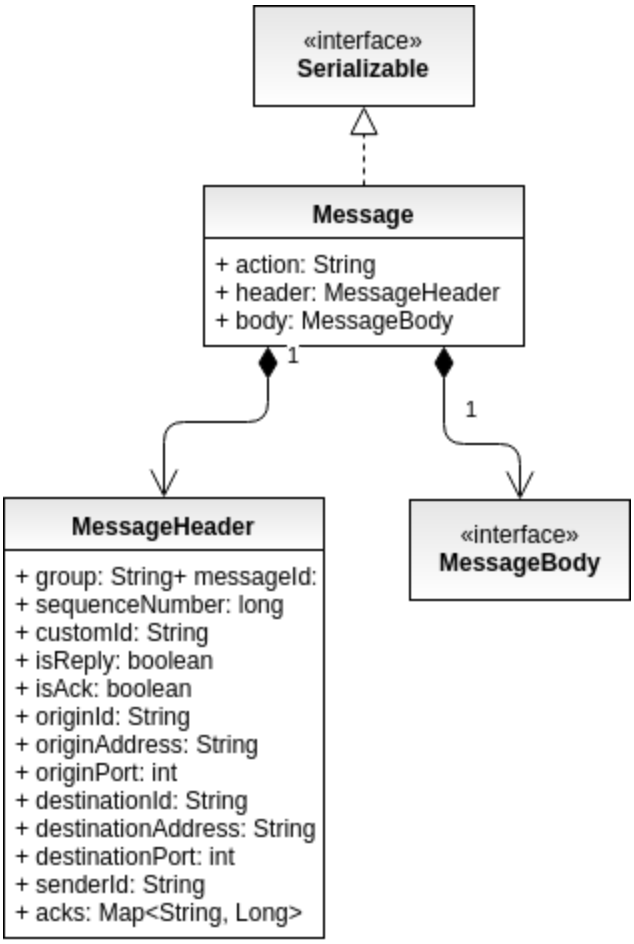
2.5 UDP Interface

2.5.1 Overview

To homogenize the communication between servers using the UDP protocol, a standardized Reliable UDP stack and message format using serialized objects is used as a form of IPC. This interface is based around standard object composed of an action, header and body object. Each communicating entity in the system will handle transforming the message into the appropriate method call in the module.

The message format is loosely based on the SOAP xml model.

2.5.2 Message Format



3 Detailed Design

3.1 Reliable UDP Stack

3.1.1 Design Overview

All components which communicates through reliable UDP unicast and multicast uses a common module. All client and server requests are handled by the module and sent/delivered through a facade ReliableUDP. All received messages are routed to the same message buffer, but any ordered message are added to the buffer in the correct order to guarantee total ordering and simplify the application interface.

Reliable Unicast

The Reliable Unicast is implemented using a request, reply, ack protocol to guarantee reliability. Each message sent is individually reliable and the protocol uses a unique system messageId for this purpose. Meaningful replies are sent as a new message, since they come from different location than the destination of the original message. The application uses the customId value to associate request and replies.

Reliable Multicast

The Reliable Multicast is implemented using the TO R-Multicast algorithm found in the book. A sequence number is associated with each message sent to the group by a certain host. The UDP client uses this sequence number and an hold-back queue to find any missing message and request them from the sender using a negative acknowledgement. This sequence number is also used to guarantee the total ordering delivery in the client coming from a single host.

3.2 Discovery Service

3.2.1 Design Overview

The Discovery service is used to register each UDP component's location for access transparency. It is composed of a hashMap of ID to location and port and a simple search algorithm. This service is backed by two text files, location.csv and groups.csv, which identify the UDP ip:port combinations and the group membership respectively.

3.3 Clients

3.3.1 Design Overview

The Clients will use the CORBA Naming Service to retrieve a remote reference to a Front-End. The clients will be use to perform user facing operations on the server via a console command line. All server requests will be serviced the the Front-End component, returning a value to the user upon interacting with the distributed banking system.

3.4 Front-End

3.4.1 Design Overview

Each client is connected to only one Front End, or FE. A new servant instance of the FE is used to service requests. The FE is able to execute requests to and from every branch cluster, therefore only a single FE is started, however a new instance is used for each request to guarantee proper concurrency. The communication between client and FE is over TCP, as it is done via CORBA mechanisms.

The FE handles the request by sending it to the sequencer. In normal cases, the FE receives 3 responses from the Replica Managers and sends only one correct answer to the client. In the case where the FE receives a result that is not the same as the other 2, then it will take the majority result and send that to the client. It will then inform the RM that sent the incorrect result so that it can deal with it if it happens again.

3.5 Sequencer

3.5.1 Design Overview

A total order sequencer is used by the sequencer component to reliably broadcast messages to ReplicaManagers with a predefined and agreed upon sequence in order to avoid server side inconsistencies for client operations. By coordinating a consensus via multicast with all the ReplicaManagers, each request will be guaranteed a unique sequence number, allowing the maintenance of a consistent state among all replicas.

The Sequencer uses the UDP stack to guarantee the total ordering.

3.6 Replica Manager

3.6.1 Design Overview

There should be 3 RM's initialized per branch. Each RM creates and handles its own replica, out of which there are 3 replica implementations to choose from (Sylvain, Radu, Mathieu). RM's receive in total order the request from the sequencer. It then asks its corresponding replica to handle the request. Once it receives an answer from the replica, it will forward it directly to the FE. The RM handles a potential replica crash and byzantine errors by re-initializing a new replica with an implementation different than the one it was initialized with. It will ask other RM's for their database so that the newly initialized replica is up to date. The RM's communicate between each other inside a server over UDP for synchronization operations, and through the sequencer to communicate with RM's from other branches.

3.6.2 Algorithms

Ordering Algorithm

The ordering algorithm used in this project is total ordering, which is a standard algorithm that can be found in the slides. We assume we don't need to use causal ordering even though it would be ideal to have it.

3.7 Replica

3.7.1 Design Overview

Replicas are the individual instances of the server software. Since there are three distinct CORBA implementations of the server software that will be integrated into the distributed system, access transparency will be be achieved through a set of interfaces that ensure all the separate server implementations are able to accept and return data in the same format. High availability is ensured by having 3 separate and distinct implementations of the replicas. This means that if any particular replica returns an incorrect result several times, or crashes, it will be replaced with a different implementation.

4 Testing Procedures

4.1 General Functional Tests

1. Multiple CustomerClient sending withdrawals and deposits concurrently to the same account.
    a. Validate the balance to be consistent
    b. Testing for synchronization
2. Multiple CustomerClient sending withdrawals and deposits to multiple branch at the same time and some concurrent call to the same accounts.
    a. Validate the balance to be consistent
3. Multiple ManagerClient sending concurrent calls on the same branch and to the same account details.
    a. Validate the Manager Interface
    b. Testing for synchronization of call at the branch level
    c. Testing for multiple managerial call to branches and the dbs system.
4. General tests for all methods in the system by both the CustomerClient and ManagerClient.
    a. Testing that all methods behave correctly when being used concurrently by many users.

4.2 Failure Tolerance Test

1. Purposely trigger failure of a replica via inbuilt UDP failure mechanism.
    a. Ensure a new replica with a different implementation is successfully spawned by the associated replica manager.
    b. Ensure new replica's state is properly synchronized
        i. Customers and managers replicated

4.3 High Availability Test

1. Purposely trigger invalid return data for a withdrawal operation in a single replica.
    a. Ensure the majority data returned by the other replicas is selected to be shown to the user, instead of the invalid data.
    b. Ensure that after 3 instances of sending invalid data, the replica manager for the faulty branch restarts the replica with a different implementation and re-synchronizes its state.