

**In Fulfillment**

**of the Requirements**

**of COMP 6231**

**Design Documentation**

**On**

**Highly Available CORBA Distributed Flight Reservation System**

**Submitted to:**

Instructor: Prof:Rajagopalan Jayakumar

Lab Instructor:Harpreet Singh Narula

Department of Computer Science & Software Engineering

Concordia University, Montreal, QC

**Submitted by:**

Varun Pattiah Sankaralingam - 27679009

Ravinder Pal Singh Rangia - 40021206

Shreyas Vasudevareddy Movva - 40027704

Master in Applied Computer Science

Concordia University

Table of Contents

1. Introduction……………………………………………………………………………………

[2. System Design 4](#_Toc454885773)

[a. Theory Description 4](#_Toc454885774)

[3. Use case Model 8](#_Toc454885775)

[4. System Architecture 10](#_Toc454885776)

[i. Clients 11](#_Toc454885777)

[ii. Front End 11](#_Toc454885778)

[iii. Request Handler (Leader) 12](#_Toc454885779)

[iv. FIFO Scheduler 12](#_Toc454885780)

[5. Sequence Diagram 13](#_Toc454885781)

[I. Assumptions 13](#_Toc454885782)

[II. Overview of Implementation 14](#_Toc454885783)

[III. Method Descriptions 14](#_Toc454885784)

[6. Test Case Scenarios 15](#_Toc454885785)

[7. Conclusion 17](#_Toc454885787)

[8. References 18](#_Toc454885788)

# 1. Introduction

In this project it contains the detailed description how High availability is achieved in implementation of Distributed Flight Reservation System (DFRS) Using CORBA.It has information on design of highly available DFRS system with total order implemented through FIFO.It contains the detailed explanation about various Replica, Front End, Sequencer and Replica Manager to achieve high availability.

It also discusses about the data structure used for implementing Distributed Flight reservation system and the problem faced while implementing the whole system. Also it contains detailed description of logs maintained for both Passenger and Manager. At the same time Logs at server also maintained to keep track of each action happened at server.

To implement a Distributed Flight Reservation System (DFRS): a distributed system used by Flight managers to manage information regarding the Passenger’s across different city’s using CORBA. It should have following methods:

* + bookFlight(firstName,lastName,address,phoneNumber,destination,dateOfFilight, classOfFlight)
  + getBookedFlightRecordCount(recordType)
  + editFlightRecord(recordID,fieldName,newValue)
  + transferReservation(passengerID,currentCity, otherCity)

The Goal of this project is implement a highly available CORBA Distributed Flight Reservation System, which tolerates process crashes only (no software bugs) using unreliable failure detection. Thus, there is a group of (at least 3) server processes (typically running on different hosts) providing the redundancy for high availability and periodically checking each other for failure detection. One of the processes in the group is the elected leader and receives requests from clients through a CORBA front end and sends responses back to them through the front end. The leader of the server group broadcasts the client request atomically to all the servers in the group using a reliable FIFO broadcast mechanism, receives the responses from them and sends a single response back to the client as soon as possible. Since the replicated servers are usually on a local area network, they communicate using the unreliable UDP protocol.

# 2. System Design

## a. Theory Description

Theory for protocols and algorithms used in our DFRS system as below:

**CORBA** is the acronym for **C**ommon **O**bject **R**equest **B**roker **A**rchitecture vendor-independent architecture and infrastructure that computer applications use to work together over networks. Using the standard protocol IIOP, a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network.

The four keys to object orientation are:

* Encapsulation
* Polymorphism
* Inheritance
* Instantiation

CORBA applications are composed of objects, individual units of running software that combine functionality and data, and that frequently (but not always) represent something in the real world. Typically, there are many instances of an object of a single type - for example, an e-commerce website would have many shopping cart object instances, all identical in functionality but differing in that each is assigned to a different customer, and contains data representing the merchandise that its particular customer has selected. For other types, there may be only one instance. When a legacy application, such as an accounting system, is wrapped in code with CORBA interfaces and opened up to clients on the network, there is usually only one instance.

For each object type, such as the shopping cart that we just mentioned, you define an interface in OMG IDL. The interface is the syntax part of the contract that the server object offers to the clients that invoke it. Any client that wants to invoke an operation on the object must use this IDL interface to specify the operation it wants to perform, and to marshal the arguments that it sends. When the invocation reaches the target object, the same interface definition is used there to unmarshal the arguments so that the object can perform the requested operation with them. The interface definition is then used to marshal the results for their trip back, and to unmarshal them when they reach their destination.

The IDL interface definition is independent of programming language, but maps to all of the popular programming languages via OMG standards: OMG has standardized mappings from IDL to C, C++, C++11, Java, Ruby, COBOL, Smalltalk, Ada, Lisp, Python, and IDLscript.

This separation of interface from implementation, enabled by OMG IDL, is the essence of CORBA - how it enables interoperability, with all of the transparencies we've claimed. The interface to each object is defined very strictly. In contrast, the implementation of an object - its running code, and its data - is hidden from the rest of the system (that is, encapsulated) behind a boundary that the client may not cross. Clients access objects only through their advertised interface, invoking only those operations that that the object exposes through its IDL interface, with only those parameters (input and output) that are included in the invocation.

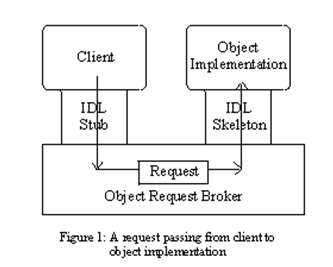
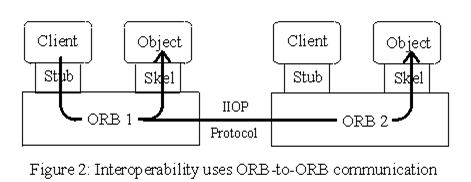


Figure 1 shows how everything fits together, at least within a single process: You compile your IDL into client stubs and object skeletons, and write your object (shown on the right) and a client for it (on the left). Stubs and skeletons serve as proxies for clients and servers, respectively. Because IDL defines interfaces so strictly, the stub on the client side has no trouble meshing perfectly with the skeleton on the server side, even if the two are compiled into different programming languages, or even running on different ORBs from different vendors.

In CORBA, every object instance has its own unique object reference, an identifying electronic token. Clients use the object references to direct their invocations, identifying to the ORB the exact instance they want to invoke (Ensuring, for example, that the books you select go into your own shopping cart, and not into your neighbor's.) The client acts as if it's invoking an operation on the object instance, but it's actually invoking on the IDL stub which acts as a proxy. Passing through the stub on the client side, the invocation continues through the ORB (Object Request Broker), and the skeleton on the implementation side, to get to the object where it is executed.

Figure 2 diagrams a remote invocation. In order to invoke the remote object instance, the client first obtains its object reference. (There are many ways to do this, but we won't detail any of them here. Easy ways include the Naming Service and the Trader Service.) To make the remote invocation, the client uses the same code that it used in the local invocation we just described, substituting the object reference for the remote instance. When the ORB examines the object reference and discovers that the target object is remote, it routes the invocation out over the network to the remote object's ORB. (Again we point out: for load balanced servers, this is an oversimplification.)



How does this work? OMG has standardized this process at two key levels: First, the client knows the type of object it's invoking (that it's a shopping cart object, for instance), and the client stub and object skeleton are generated from the same IDL. This means that the client knows exactly which operations it may invoke, what the input parameters are, and where they have to go in the invocation; when the invocation reaches the target, everything is there and in the right place. We've already seen how OMG IDL accomplishes this. Second, the client's ORB and object's ORB must agree on a common protocol - that is, a representation to specify the target object, operation, all parameters (input and output) of every type that they may use, and how all of this is represented over the wire. OMG has defined this also - it's the standard protocol IIOP. (ORBs may use other protocols besides IIOP, and many do for various reasons. But virtually all speak the standard protocol IIOP for reasons of interoperability, and because it's required by OMG for compliance.)

Although the ORB can tell from the object reference that the target object is remote, the client can not. (The user may know that this also, because of other knowledge - for instance, that all accounting objects run on the mainframe at the main office in Tulsa.) There is nothing in the object reference token that the client holds and uses at invocation time that identifies the location of the target object. This ensures location transparency - the CORBA principle that simplifies the design of distributed object computing applications.

In this project, the CORBA architecture is used between the Client and the Front End. The purpose of the CORBA methods in the implementation class on the Front End is to gather the details of the manager/records, perform marshaling in order to convert these details into external data representation and send a UDP request message to the Leading server. All these methods on FE would be declared in an interface which in turn is defined in the IDL.

The **bully algorithm** is a programming mechanism that applies a hierarchy to nodes on a system, making a process coordinator or slave. This is used as a method in distributed computing for dynamically electing a coordinator by process ID number. The process with the highest process ID number is selected as the coordinator.

**Bully Algorithm**

* Assumptions – Each process knows the ID and address of every other process – Communication is reliable
* A process initiates an election if it just recovered from failure or it notices that the coordinator has failed
* Three types of messages: Election, OK, Coordinator
* Several processes can initiate an election simultaneously – Need consistent result

**Bully Algorithm Details**

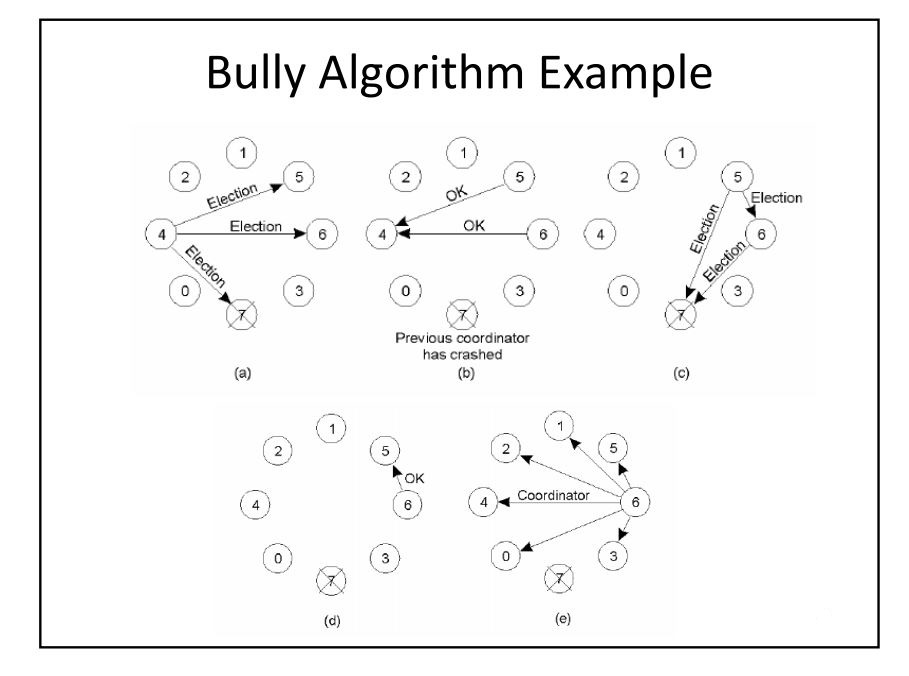
* Any process P can initiate an election
* P sends Election messages to all process with higher IDs and awaits OK messages – If no OK messages, P becomes coordinator and sends Coordinator messages to all processes with lower IDs – If it receives an OK, it drops out and waits for an Coordinator message
* If a process receives an Election message – Immediately sends Coordinator message if it is the process with highest ID – Otherwise, returns an OK and starts an election
* If a process receives a Coordinator message, it treats sender as the coordinator
* Assume n processes and one election in progress
* Bully algorithm

– Worst case: initiator is node with lowest ID

* Triggers n‐2 elections at higher ranked nodes: O(n2) messages

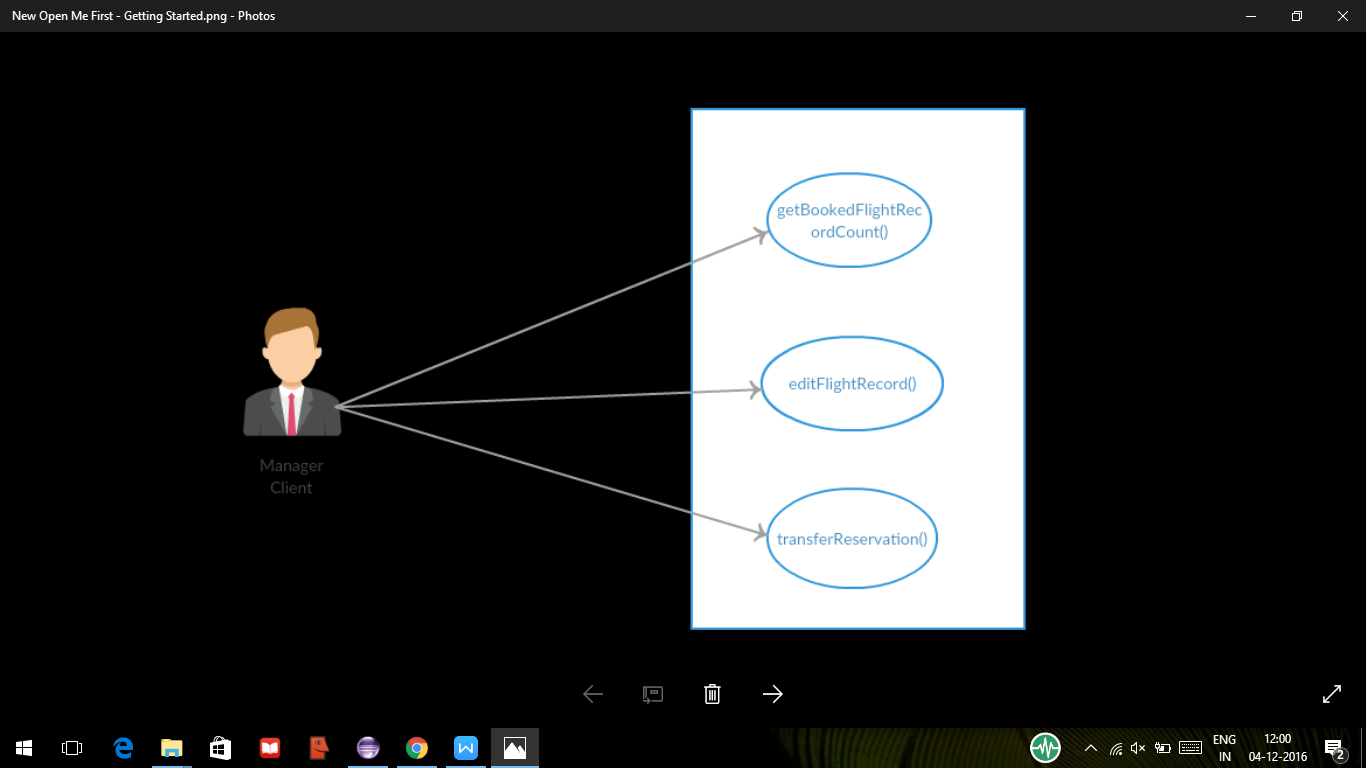
– Best case: initiator is node with highest ID

* Immediate election: n‐1 messages



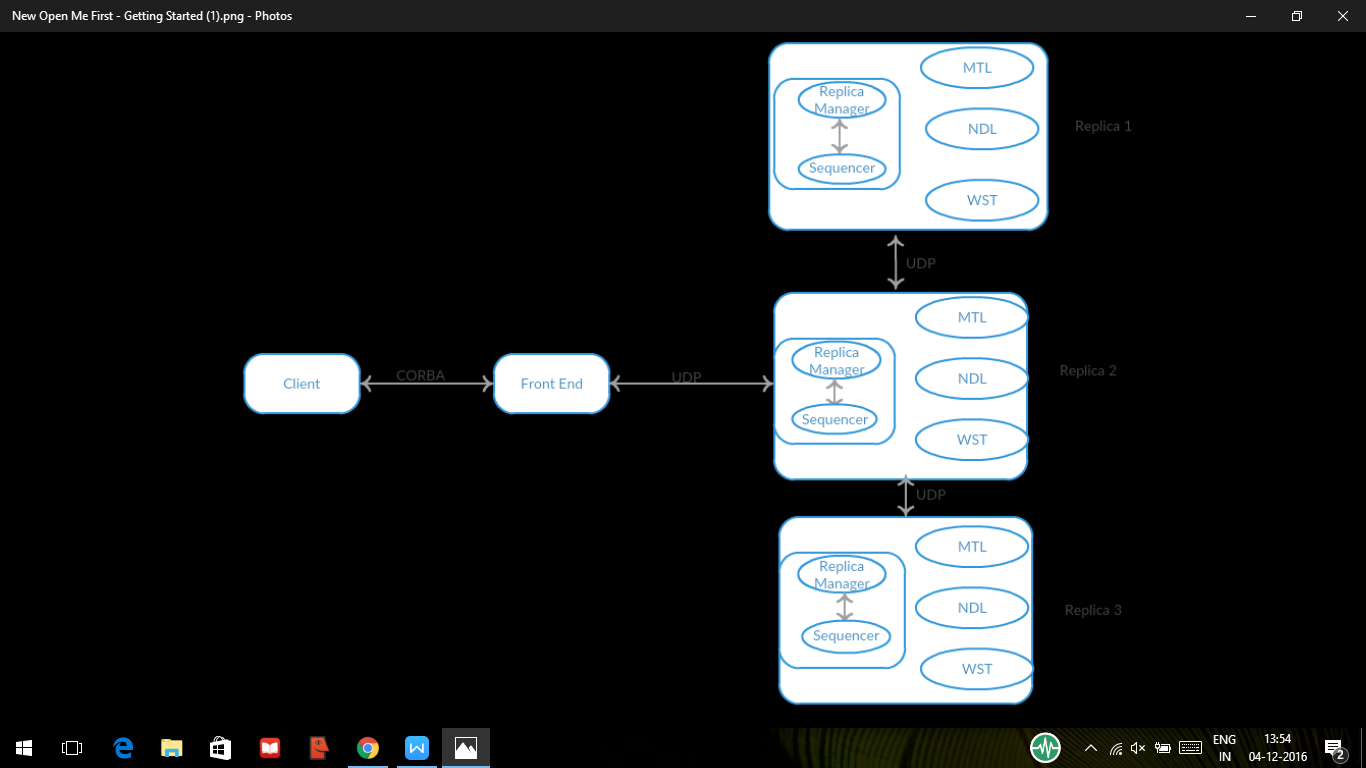
# 3. Use case Model

The system for Flight Reservation has Manager could perform functionalities like getBookedFlightCount, editFlightRecord and transfer Record from one city to other.The Diagram of use case for manager is given below.



**Fig : Use-case model**

# 4. System Architecture



**Fig: DFRS system architecture**

In order to design the Highly Available CORBA Distributed Flight Reservation System, the previous assignment, the CORBA IDL is utilized by modifying the original work keeping the design flexible, simple, and comprehensible. To access for each DFRS of all group members same services, all three DFRS should have the same interface. The three systems are ready to publish these services using the CORBA architecture. This functionality is also already tested as part of previous delivered works. These three systems are converted to the three Replicas of the DFRS required to build up the highly available DFRS. Their services would still be accessed using the CORBA architecture. Compared to the direct access between the system clients and the DFRS Replicas, in this system, the client request would be managed by a set of components playing between the clients and the Replicas. These components coordinate the Replicas work in order to support fault tolerance.

The system clients communicate with the system Front End (FE) by using the CORBA architecture. The FE is responsible for broadcasting the request with using the User Datagram Protocol, and that request is broadcasted from the FE to the Leader. The Leader processes these requests iteratively by using FIFO mechanism. In order to process each request, the Leader multicasts each request to the other two Replicas which process in their local servers and send the reply back to the Leader. After then, the Leader compares the results from all the Replicas and sends the correct result to the FE.

Since the replicated servers are usually on a local area network, they communicate using the unreliable UDP protocol. However, the communication among them should be reliable and FIFO. At a same time, a low-level protocol is also possible since the developers know the communication protocol and data representation strategy about all the details concerning the module implementation. Using a low-level protocol allows alienating from the overhead linked to the generalizations required for the higher-level protocols and to design an ad-hoc communication.

## Clients

The DFRS System include Passenger and Manager Request. Passenger can bookFlight and Manager could perform functionalities like getBookedFlightCount, editFlightRecord and transfer Record from one city to other.

## Front End

The Front End (FE) is the connection between the clients and the Replicas. When a client needs a service from the DFRS, it sends a request over the CORBA to the FE. The FE then sends the request to the Lead Replica Manager using a UDP connection. The Request Handler implemented in the Leader Replica communicates with other Replicas and sends the request to them. Each Replica computes the result and sends it back to the Lead. The Request Handler receives three results including the result of the Leader Replica from each Replica and compares them. If the results are equal then it is sent to the FE. Assuming that only one Replica may produce a wrong result at a time, to tolerate this failure, the FE takes what it considers as the most returned result and sends it back to the client. The FE guarantees the transparency of clients by using the CORBA architecture. In addition, multithreading is used as a way for Replicas and replies to synchronize with each other.

As the FE is implemented as a CORBA object and managed by the CORBA engine, the server is automatically multithreaded to communicate with several clients in parallel. The FE may process several client requests in parallel and broadcast multiple requests simultaneously. In order to send multiple requests, a sequence number is generated by the FE. This number is then attached to the request so as to keep track of it and deal with concurrency issues.

## Request Handler (Leader)

The Request Handler is the component that translates and manages the request broadcasted from the FE. Request Handler module is implemented in the same host where the Leader is. The module receives the requests via UDP/IP from the FE and uses FIFO technique to process them iteratively. As several of these requests may have been sent in a random order from the FE, the Request Handler is responsible for arranging the received requests.

The received requests are processed and ordered based sequence number assigned to it by the FE before sending. The Leader will then multicast each request to the other two Replicas and also processes the request locally. After getting the results from other Replicas, the Leader will compare all three results and send the most returned result to the FE. This is done for each request received iteratively. Moreover, if any of the Replicas generates a wrong result, the Request Handler will inform the RM.

## FIFO Scheduler

To ensure a correct response (response for the correct method invoked) is sent back to client, scheduling request at the leader is important and this task is performed by scheduling module at the leader processes, which schedules client request in FIFO order for the requests and sends the replies to the clients in the same order.

# 5. Sequence Diagram



**Fig:Sequence Diagram**

## Assumptions

* The Replicas in this server subsystem are free from crash failure (software bug excluded).
* Only one of 3 Replicas (except Leader) can produce an incorrect result.
* No failures could occur during the recovery of a faulty Replica.

## Overview of Implementation

* Implementing Leader:
  + Manage FIFO for all request coming from FE
  + Multicast requests to replicas
  + Process locally and compare with the replies of other two replicas
  + Reply the maximum of two result to the Front End
  + If any replica gives faulty result, then combine all results send correct result .
* Implementing Front End:
  + Generate corba remote object
  + Request any particular operation from client side via corba object
  + For each client request, create a new thread & forward the request to the Leader by attaching the sequence number
  + Accept the replies from the Leader.

## Method Descriptions

**bookFlight Method:**

This method is used by passenger to book a flight to different cities,where he need to mention the destination and present city,also personal details to book a ticket.

**editFlightRecord** **Method:**

This method is used to edit the values of stored records. The required record is searched for in the database and modified .Where the manager a authority to change the flight details and also Passenger booking.

**getBookedFlightRecordCount Method:**

This provides the client with the total number of records stored in each location i.e. MTL, NDL and WST.

**transferReservation** **Method:**

This method is used to transfer a record stored at a particular location to another location. For example, a Passenger record stored at MTL can be moved to NDL.

# 6. Test Case Scenarios

Expected Results are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **Method Name** | **Test Cases** | **Description** | **Expected Output** |
| 1 | bookFlight | booking | Check if the specified flight is booked | Input – (firstName,lastName,address,phoneNumber,destination,dateOfFilight, classOfFlight)  Output – True  [Flight booked Successfully] |
| 2 | editFlightRecord | Record Existence | Check if the required record exists in database | Input – (recordID,fieldName,newValue)  Output – False  [Record Not Found] |
|  |  | Input Validity | Check that the location of flight record is either MTL,WST or NDL. | Input – (recordID,fieldName,newValue)  Output – True  [Record Edited Successfully] |
| 3 | getBookedFlightRecordCount | Calculate local number of records and request other locations to send their total number of records | Check that a connection to remote servers is possible | Input – (MTL1111)  Output – MTL-1, WST-1, NDL-2  [Local and Remote Counts available] |
| 4 | transferReservation | Input Validity | Check that the required record exist on the source location | Input – (passengerID,currentCity, otherCity)  Output – True  [Record Transferred successfully]  Input – (passengerID,currentCity, otherCity)  Output – False  [Record could not be found] |

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Test Cases** | **Description** | **Expected Results** |
| 1 | Failure Test | If the replicas find out that the leader has not responded, they assume that the leader has failed. They hold election to select a new leader. | A new Leader is elected/Failed replica is restarted |
| 2 | Concurrency Test | Automatic test case that creates multiple requests accessing various functions. | Requests are executed in parallel at the leader and sequentially at the replicas and majority of the results are returned to the client through Front End. |

# 7. Conclusion

This design document successfully demonstrates how the DFRS will use CORBA for communication between remote client and server and implement a Highly Available Flight Management System.

# 8. References