**Soen 423 Project :**

**Software Failure Tolerant and/or Highly Available Distributed Health Care Management System**

By

Kadeem Caines: 26343600

John Libera: 27728085

John Bakalis:

Jordan Cassivi:26787819

Table of Contents

1. Introduction……………………………………………………………………3
2. Design Description/Overall architecture………………………………………3
   1. Distribution of Tasks…………………………………………………..3
   2. Distributed Computing Paradigms…………………………………….3
   3. Packages……………………………………………………………….3
   4. Data Structures………………………………………………………...4
   5. Remote procedure calls (Corba)……………………………………….4
   6. Communication………………………………………………………..4
   7. Communication between components…………………………………5
   8. How we will simulate a Byzantine Failure…………………………….5
   9. Recovery from Failure…………………………………………………5
3. Components………………………………………………………………………..6
   1. Client…………………………………………………………………...6
   2. Front end……………………………………………………………….6
   3. Sequencer………………………………………………………………6
   4. Replica Managers………………………………………………………6
   5. Replicas…………………………………………………………………7
4. UML
5. Test Cases
6. Difficulties

**1.Introduction**

As a team of 4, we will be working on extending our Distributed Health Care Management System which consisted of Patient and Admin clients remotely executing methods in specific city hospital servers via CORBA. We will be testing it to tolerate either a single software (non-malicious Byzantine) failure or be highly available under a single process crash failure using active replication. This document will describe the architecture, classes, methods and implementations used in order to configure this system

**2.Design Description/Overall architecture**

**2.1 Distribution of Tasks**

|  |  |
| --- | --- |
| Tasks | Student |
| Front-End (Student 1) | Kadeem |
| Replica Manager (Student 2) | John Bakalis |
| Failure-free Sequencer (Student 3) | Jordan Cassivi |
| Test Cases + Client Program (Student 4) | John Libera |

**2.2 Distributed Computing Paradigms**

* Client-Server

**2.3 Packages**

* Client
* Replicas
* ReplicaManager
* UDP
* Front End
* Sequencer
* corbaFrontEnd (IDL generated files)

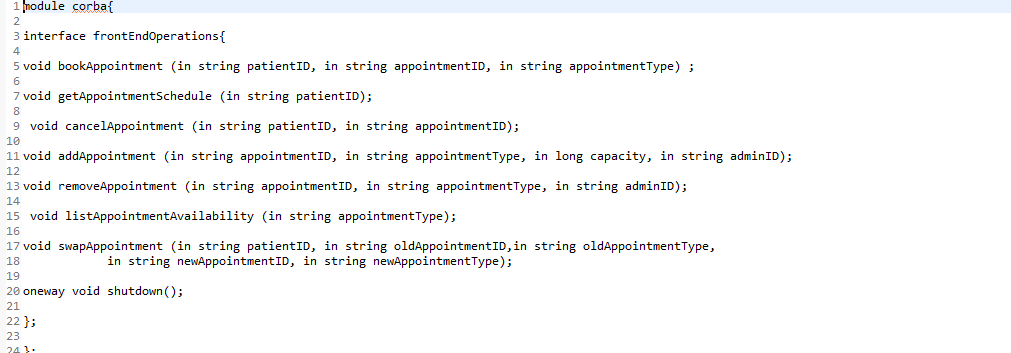
**2.4 Data Structures**

-Array List

-HashMaps (used in replicas in order to store client data)

**2.5 Remote procedure calls(Corba)**

We will be using CORBA for all RPC calls. Our methods are defined in the IDL following IDL file:



These methods will be implemented in the FrontEnd.java file and will be invoked by the Client. Java by the user when asked which operation they want to perform. All remaining Corba operations will be carried within the replicas themselves, as defined by assignment 2.

**2.6 Communication**

* UDP: communication between the Frontend, the sequencer, the RMs will all be done via UDP since it is faster than transferring messages since it does not have to establish a link with each component. UDP will be tested to the point where minimal packet loss is implemented. Even though this is not a real distributed system, packet loss in general is still possible.
* Reliable Multicast: The sequencer will oversee multicasting the string data that the front end has sent it to each replica manager.

**2.7 Communication between components**

Client→ Frontend (via Corba RPC)

Frontend→ Sequencer (via UDP unicast)

Sequencer→ Replica Managers (via UDP multicast)

Replica Manager→ Replicas (via UDP multicast)

Replicas→ Main Server (via Corba RPC and UDP for listavailability and swapAppointment methods)

Replicas→ Frontend (via UDP unicast)

Frontend→ Client (print statements based on FrontEnd method success )

**2.8 How we will simulate a Byzantine Failure**

Since this is but a simulation of a distributed system, we are not expecting a hardware type of byzantine failure for any of the replicas. We will be focusing on a software type of failure; one where incorrect data is sent back to the client. If one replica out of the 4 does not have data that matches the other 3, then the system will continue as normal. We will manually create a byzantine failure by sending incorrect data.

**2.9 Recovery from Failure**

* Software Byzantine Failure:

If any of the replicas produce an incorrect result, the frontend will inform the replica manager about it so that replica can be restarted.

* Single Process crash failure:

Single process failure will lead to the replica giving incorrect data or timing out with the frontend. Leading the frontend asking the associated replica manager to restart the replica that is failing.

* Error recovery:

The sequencer will keep an in-memory log of all the sent messages. After a replica is restarted, its replica manager will ask the sequencer to replay the log to its replica in order to catch up with the state of the other replicas in the system.

**3. Components**

**3.1 Client:**

The client will ultimately be responsible to speak with the frontend using CORBA and invoke the methods needed in order to pass the data to the sequencer and so forth. Firstly, it greets the user and asks them what city they are from. Once selected, the system now asks if the user is an admin or patient. It then assigns them a random ID based on their chosen identity and city. Afterwards, the user is offered different choices to choose from and must enter information depending on the option. The system will look up the frontend remote object and invoke the Corba methods. These methods will take the inputted information from the user and send it to the sequencer.

**3.2 Front end**

The purpose of the frontend is to manage the communication between the client and sequencer. It gives the clients a failure free interface to branch the servers to allow them to perform patient as well as administrative operations. The frontend will register itself as the regional server as a remote object with CORBA and will wait on requests from the clients. It contains all the Corba methods of Corba IDL file. When requests are received, they are then forwarded to the sequencer. Frontend component then waits for the replicas to send their responses, where it can decide which response is correct and send it back to waiting client. We will have if statement conditions in order to determine which response is from which replica. If the replica detects a fault, the frontend will send a message to the replica managers to inform them of that fault.

**3.3 Sequencer**

The sequencers job is to receive requests from the frontend and provide them with a sequence ID. This identifier will need to auto-incremented and will need to act as a way to ensure that all replicas execute the operations in the correct order. The moment each request has an ID, it multicasts to all the replicas. Locations inside the replicas should receive the request as determined by the request manager ID. The message sent should also be stored in a history log in which is can be played back in the event of a replica failure. The sequencer will constantly be listening for datagram packets from the front end.

**3.4 Replica Manager**

The purpose of the replica manager is to manage a single replica. In other words, it’s primarily responsible for restarting its replica if the frontend thinks it is misbehaving. It will also need to know how to plant need data into a restarted replica so that it would be in the same state as the other replicas.

**3.5 Replicas**

Each of our replicas from assignment 2 will be running on different machines. These replicas (or server processes) will have their own RMs and will send back confirmation messages (via UDP) to the Front end. The replicas will each contain our implementations of the patient/admin and hospital server DHCM and will have their own UDP system for specific methods (listing available appointments and swapping appointments). The overall purpose of multiple replicas is to simulate the distributed system, where if one of the replicas fail, the system can still function and be available.

**4.UML**

**5. Test cases**

-Run each individual method to see if they all return a result

**6.Difficulties**

In general, our biggest issue was figuring out how to code in java each segment and

have each one communicate with one another properly. Building the client was straightforward, since it’s just asking for user input.