UNIVERSITÉ Concordia UNIVERSITY

COMP-6231
Distributed Library Management System (CORBA)

Software Failure Tolerant or Highly Available

Winter 2019

Final Project Design Document

Submitted to: Dr. Rajagopalan Jayakumar

Help Taken: Pranav Bhatia, Kritika Sharma

Sr. No.	Name	Student ID
1	Arpit Malhotra	40088196
2	Shresthi Garg	40105918
3	Nikitha Jayant Bangera	40088393

Table of Contents

Project Description

The project demonstrates the implementation of a Distributed Library Management System(DLMS) using CORBA technology. There are three different hosts which are connected via Local Area Network to form a distributed system. There are three different library servers namely Concordia, McGill and Montreal. The DLMS provides some specific operations for the library managers and the library users. The manager's operations include adding books, reducing the quantity of books or removing books and list the availability of books in the library and the library user's operations include borrow, return, finding a specific book and exchanging an already borrowed book with an available book in the library.

The DLMS is made highly available using process replication, when one of the replica crashes and is also designed to detect the software bug by making the system software failure tolerant.

System Design

The system is built using the following main parts:

1. **Client**: The main functionality of the client is to send a request to the Front End using CORBA, based on the library user's or library manager's operations.

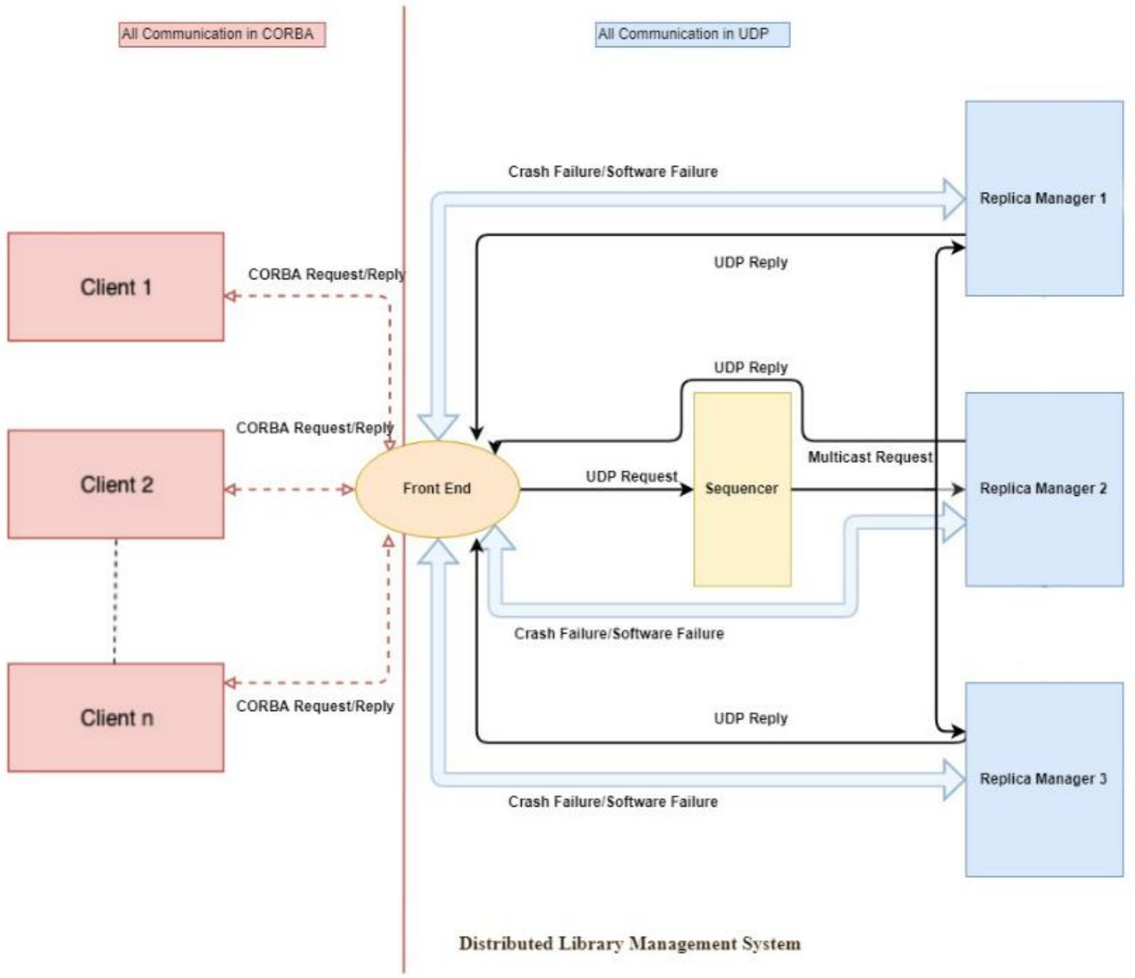
2. **Front End**: The main functionality of the front end is to forward the client's request to the sequencer and then send the reply received from the replicas back to the client. It also sends a message to the replicas in case of software failure and crash failure.

3. **Sequencer:** The main purpose of the sequencer is to append a unique sequence number to the request message and multicast the request message to all the three replicas.

4. **Replica Manager and Replica:** Each replica consists of a Replica Manager(RM) which takes care of the total ordered execution of the requests based on the sequence number. Once the request if processed by each replica, the replicas then send the reply containing the result back to the Front End.

Project Architecture

Architectural Diagram



Data Flow in the modules of the project

The data flow in the modules of the project are as follows:

1. **Client**: The client receives the request from the end-user (library managers and library users) and once the end user selects the required function, client resolves the associated FE (Front-End) by the given name in the registry, makes a call to the remote method in the CORBA IDL interface which in turn calls the interface implementation to do the requisite function.

2. **FE (Front-End):** It consists of the method definition of all the methods that the client invokes upon user selection. It frames the UDP message with respect to each request it receives from the client and forwards the request message to the Sequencer via UDP communication.

Once the request has been processed by all the replicas, the Front End receives the reply from all the replicas, then compares the result of all the three replicas and forwards the majority of the result as a reply to the requested client using CORBA. If the Front end does not receive a reply from one of the replica within a specified duration and suspects that a replica has crashed. It informs that particular replica about the suspected crash by sending a UDP message to that replica. Similarly, if the Front End receives incorrect response from one of the replicas, it suspects a software failure and intimates that replica about the faulty bug.

3. **Sequencer**: The sequencer receives the UDP message containing the client's request from the Front-End. It then generates a unique sequence ID and appends the unique ID to the request message and multicasts the request message to all the Replicas.

4. **Replica Manager and Replica**: There will be three different hosts and each host consists of a Replica along with a Replica Manager. Each Replica holds the three library server (Concordia, McGill and Montreal) implementations. Each Replica Manager will manage a Priority Queue to hold the unique sequence numbers so that all the replicas will follow the same order to execute the client requests, thus fulfilling the total order of execution of requests.

   The replicas will process the client request and appends a "Success" or "Fail" keyword along with the replica manager ID (i.e. rm1 for Replica Manager 1, rm2 for Replica Manager 2 and rm3 for Replica Manager 3) to the reply message and sends the reply back to the Front End via UDP communication. The Front End then compares the results received from the replicas and forward a single result back to the client.

   In case of software failure where-in one of the replica sends an incorrect result, the Front End will the send a UDP message to the Replica Manager of that replica informing about the faulty bug present in the result. If the Front End receives incorrect result from the same replica for 3 times, it will inform Replica Manager each time an incorrect result is received, then that Replica Manager will replace the faulty replica with a working replica and sends the correct result back to the Front End.

   In case of crash failure, if the Front End does not receive a reply from one of the replicas within a specified duration of time, it suspects that a replica has crashed. The Front End informs the Replica Manager of that replica by sending a UDP message about a suspected replica crash. The Replica Manager will receive the UDP message from the Front End to replace the crashed replica with another working replica.

Operation of Different Components

This Section will display all the components and their behaviors and explains how they are combined together to work as a single distributed application which is fault tolerant and highly available.

1. Request from Front End to Sequencer:

All requests in the form of CORBA messages that are received at front end are forwarded to sequencer using reliable UDP Protocol. The Socket at which front End runs is defined as “11111”. Messages are sent to Sequencer address which is a separate component from Front End. There is no separate message ID attached as there are new instances created with every client request at FRONTEND. Therefore, it is also necessary to make this communication between front end and sequencer reliable using UDP only.

<u>**Algorithm Used**</u>

```
Set Duration = 2000
Receive Request from Client
Format Request and send to Sequencer
Wait Until three response received(I<3).
    StartTime = Record current Time in milliseconds
    reply[i] = Receive response
    EndTime = Record current Time in milliseconds

If (Socket Timeout Occurs)
        Replica Crashed
        Send a message to Crashed Replica Manager
        Store in a List (EndTime-StartTime)
        Duration = Find Max Duration from List

If (reply[i] is not NULL)
    Message[i]=Collect Responses in variables
    Duration = Find Max Duration from List

If (All replies same)
    MajorityElement=Check for Majority Elements
Else (Two replies same and one different)
    Check for software failure
    Send UDP message to Replica Manager for software Bug
    MajorityElement=Check for Majority Elements

Send Response to Client
```

2. Request from Sequencer to ReplicaManager:

Sequencer is responsible for attaching a unique sequence number to each request. It stores all messages in the history buffer. It checks if history buffer is not full at any time otherwise it makes space in the history buffer to store new messages in the buffer.

Sequencer is also responsible to reliably multicast all the messages received from the front end to all the replica managers. To reliably multicast messages to replica managers, it multicast the same message multiple times to ensure that a packet is received at all the replica managers and is not lost in between.

Sequencer multicasts by multicasting message to multicast group running at multicast IP Address "234.1.1.1" so that all the servers who have joined multicast group on this address will receive this message. In our case, all Replica Managers are going to receive all multicast messages posted on this IP Address by sequencer.

Algorithm Used

```
sequenceNumber = 0;
While(true){
        receive(request)
        message = request
        If(History Buffer Full){
                Start Synchronization phase
         }
        Else{
                Assign Sequence Number to Message (message + sequenceNumber)
                Update history Buffer  //Store message with sequence no in buffer
                Multicast(message, "234,1,1,1")//Multicast message 1st time
                Multicast(message, "234,1,1,1");//Multicast same message 2nd time
                sequenceNumber++;//Increment Sequence Number to 1
        }
}
```

3. Receiving messages at RMs

RM's receive all the message send by sequencer and front End. Messages sent by sequencer to the multicast group are received by all the RM's whereas message sent by Front End are received only by some specific RM to which it is sent.

Replicas are crash failure tolerant which means that If a replica crashes at a point due to some reason and is no longer functioning, front end informs replica manager about a possible crash. Replica Manager, in turn checks for the crash. If a replica is crashed at a point, replica manager recovers replica from that crash by restarting it and executes all the messages received so far in a sequential manner based on sequence number.

Thus, at the end of crash recovery, replica 3 is restored to the same status as it was before the crash.

4. If Sequencer Fails

At the point when the message reaches at the RM it first checks that the sequence number of the message is the one that it is expecting and increments the LastSeqReceived. If not, it is more likely than not that RM has missed one or more messages and request the sequencer for retransmission. Out of Sequence messages are kept in the Hold Back Queue with requested message being passed to the Delivery Queue in which it is handled in FIFO request. On the chance that the RMs does not get the missing message in different retries, at that point it accepts that the sequencer has failed and RM's decide between themselves to play the job of the sequencer.

Algorithm Used

```
AcceptMulticast(Messages);

//SequenceNum is the Sequence number received from the sequencer.

If(SequenceNum =SequenceNum(Last Sequence Number)+1)
{
        Correct order Message Received;
        Process Message;
        SequenceNum++;
}
Else if (SequenceNum= SequenceNum(Last Sequence Number)
{
        Duplicate Message Received
        Drop Message
}
Else if (SequenceNum< SequenceNum(Last Sequence Number))
{
        Missed Message
        Send (Re-transmission Request, Last Sequence Number)
        SetTimer (); (Check If  Sequencer is up)
}
```

5. Sequencer doesn't fail

In our Replica Manager, we have assumed that sequencer is a very reliable and it will never fail so all the messages will eventually be received at replica manger. But those messages may be out of order so to execute them in order, we have maintained a priority queue wherein all the messages will be stored sequentially on the basis of the sequence number. Replica Manager ignores duplicate messages and polls the head of the queue and executes that message.

<u>Crash Failure Algorithm:</u>
Below is the algorithm for crash failure

```
PriorityQueue queue;            // A queue to store all the message
While(true){
    Receive(request)
    If(Duplicate message){
        Drop Message
        Continue;
    }
    Else{
        Add message in Queue
    }

    If(Queue is not Empty) {
        Extract First message from Queue;
        Set failureType from Message;
    }

    If (failureType = “faultyCrash” && crashCounter=0){
        Close all the sockets
        Increment crashCounter;
    }
    Else{
        Start all the Servers again
        While(Queue is not empty){
            Extract HEAD message
            Execute message
    }
}
```

Test cases

This section lists the single functionality test cases covered as part of unit testing. Attached below is the test cases sheet:

Test Case.xlsx