Computer Science and Software Engineering Départment

COMP 6231/1
DISTRIBUTED SYSTEMS DESIGN

Summer 2018 - Project Report

Team:

First Name	Last Name	Student ID	Email ID
Ezhilmani	Aakaash	40071067	aakaash07@gmail.com
Akhila	Chilukuri	40071241	akhilachilukuri1@gmail.com
Vigneswar	Mourouguessin	40057918	imvigneswar@gmail.com
Vaishnavi	Venkatraj	40049798	whyshu@gmail.com

TABLE OF CONTENTS

Objective

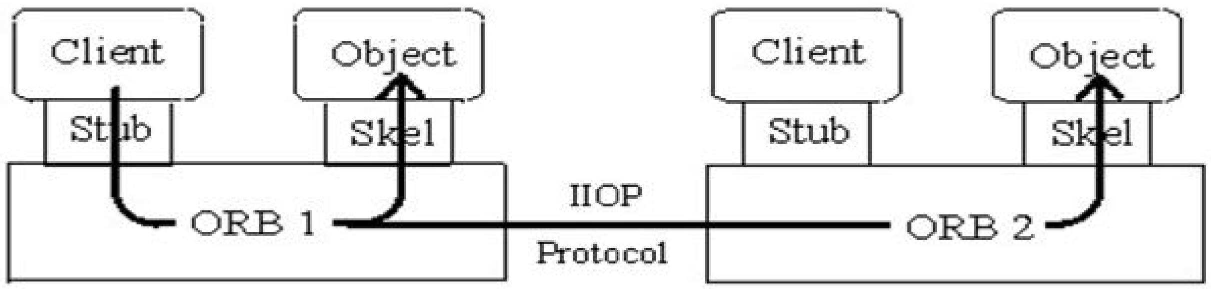
Design and develop a distributed system, which provides an infrastructure for Class Management System shared across three different data centers (MTL, DDO, and LVL). Expose CORBA APIs to the user by abstracting the underlying distributed nature of the platform and provide network transparency. Implement multi-threading with proper synchronization to handle concurrent requests safely and for efficient resource sharing. The key objectives of the Distributed Class Management are:

Fault Tolerant: Implement multiple replicas for the
primary server to maintain backups which could in turn serve as primary when
the primary server fails.

Highly Availability: Implement a failure detection subsystem in which the servers in the group periodically check each other for failed process and elect a new leader using distribution election algorithm when the leader process has failed.

Distributed Class Management System – Basic CORBA Architecture

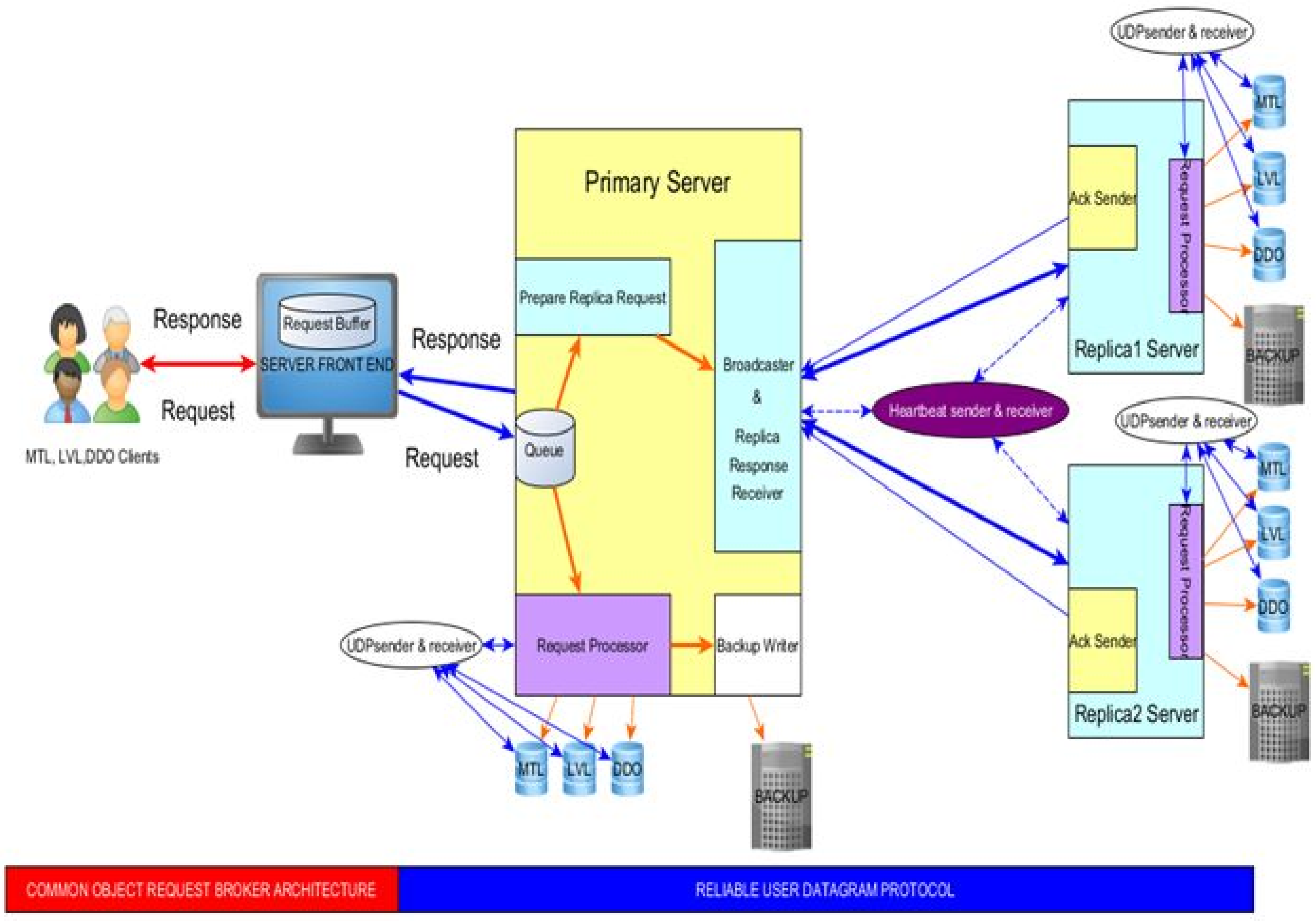
The CORBA specification dictates there shall be an ORB through which an application would interact with other objects. This is how it is implemented in practice.



The application simply initializes the ORB, and accesses an internal *Object Adapter*, which maintains things like reference counting, object (and reference) instantiation policies, and object lifetime policies.

The Object Adapter is used to register instances of the *generated code classes*. Generated code classes **(DcmsApp)** are the result of compiling the user IDL code **(Dcms.idl)**, which translates the high-level interface definition into an OS- and language-specific class base for use by the user application **(DcmsServerImplementation)**. This step is necessary in order to enforce CORBA semantics and provide a clean user process for interfacing with the CORBA infrastructure.

Architecture - Fault Tolerant & High Available DCMS



Workflow

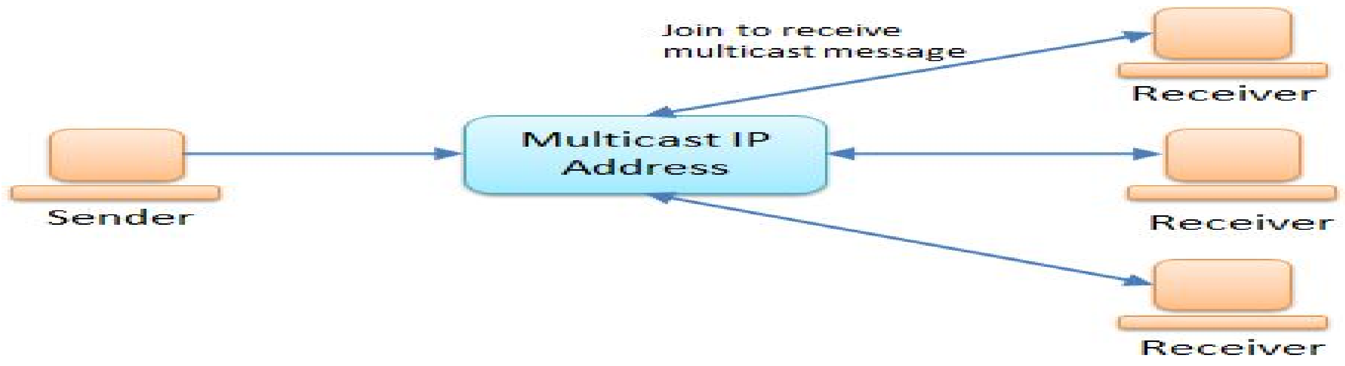
- Front End will register CORBA remote reference with Naming Service.
- Client will send request to Front End (FE) using CORBA remote invocation.
- The FE receives the request, puts it into the request buffer and forwards it to the Primary Server.
- Primary Server performs the following operation:
  - Prepares the replica request, multicasts it to other two replicas using Reliable UDP communication, which is a FIFO Broadcast implementation.
  - Writes the hash map to the primary server backup source.
  - Processes the received request in the FIFO order and sends the response back to the FE.
- Replica servers performs the following operation:
  - Receives the broadcasted request and sends the acknowledgement to the primary server.
  - Writes the hash map to the replica server backup source.
  - Processes the request and sends back the response to the primary server.
- Once FE receives the response from all the replicas, FE will do following:
  - All the responses from primary, as well as from the replicas will be logged in primary log and replicas log respectively.
  - The primary server's response will be returned back to the client.
  - In case of current primary server's crash, the FE sends the request again to the currently elected Primary server.

Fault Tolerance & High Availability

Maintaining multiple replicas for each server, in this case, the Primary server has two replicas with access to all three hashmaps MTL, LVL and DDO.

FIFO Broadcast Implementation

Once the Primary server receives the request from the Front end, it broadcasts the requests to the replicas in its group using Reliable UDP communication and FIFO queue implementation in the Primary Server.

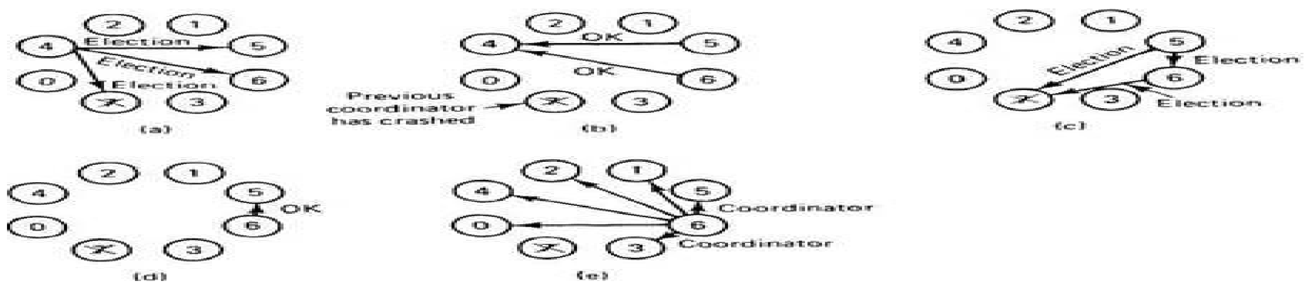


In this case, the sender is the Primary server and the receivers are the replicas in its group. The server also broadcasts the requests, in a First-In First-Out manner, using the Queue data structure.

Failure Detection - Heartbeat Implementation

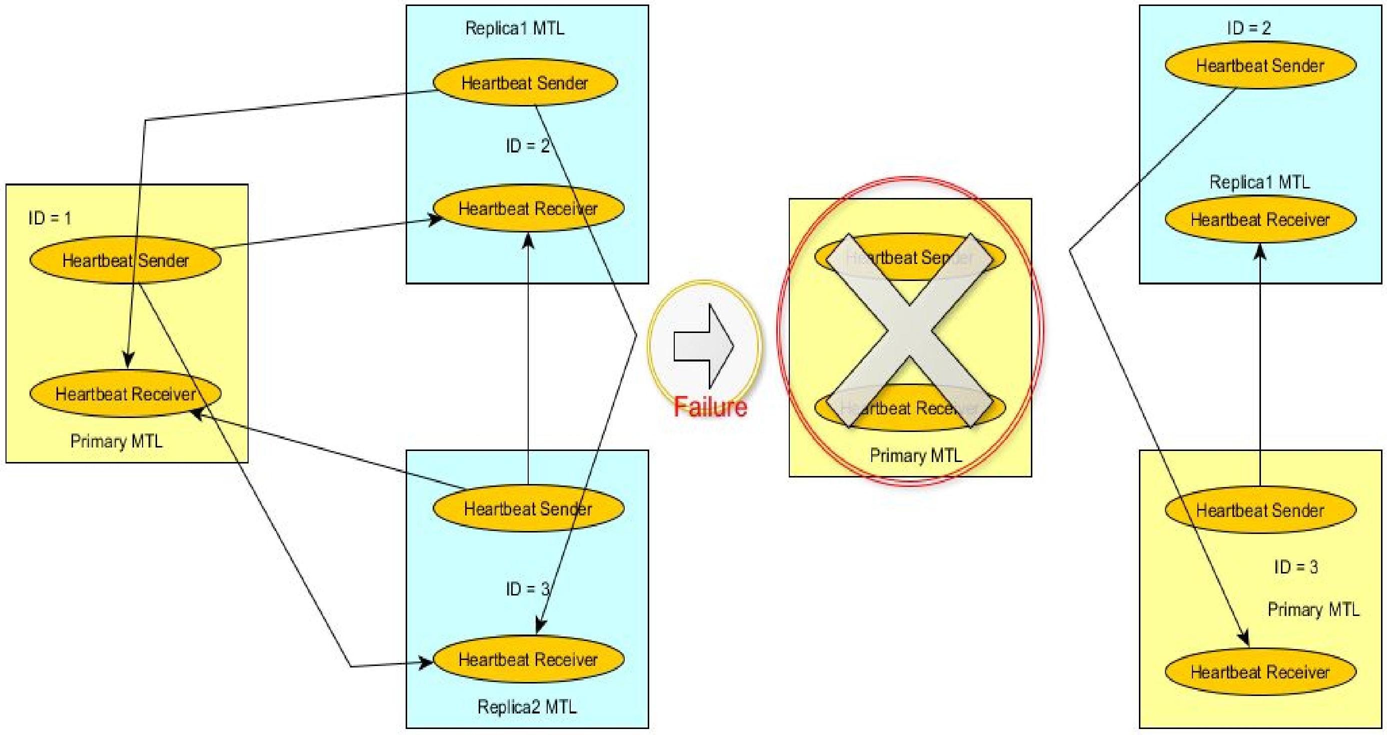
As the name suggests, heartbeat message is the functionality wherein one server process constantly communicates with the other server process to know the status of the servers in its group. This functionality is implemented in the 'Server Front End'.
If a server isn't active or has gone down for some reason then, the heartbeat implementation will immediately sense the situation and take the necessary actions to resolve it, which is in our case to initiates the **ELECTION Algorithm**.

Withstanding Failure - Election Algorithm Implementation



When the primary process fails, the process that senses the failure first sends the election message to all the remaining processes with **higher weight/priority**.
If a process with a higher weight/priority is available, that particular process begins the election again.
This goes on until the process with the highest priority elects itself as the leader and sends the coordinator message.

Heartbeat implementation among multiple servers , before and after failure.

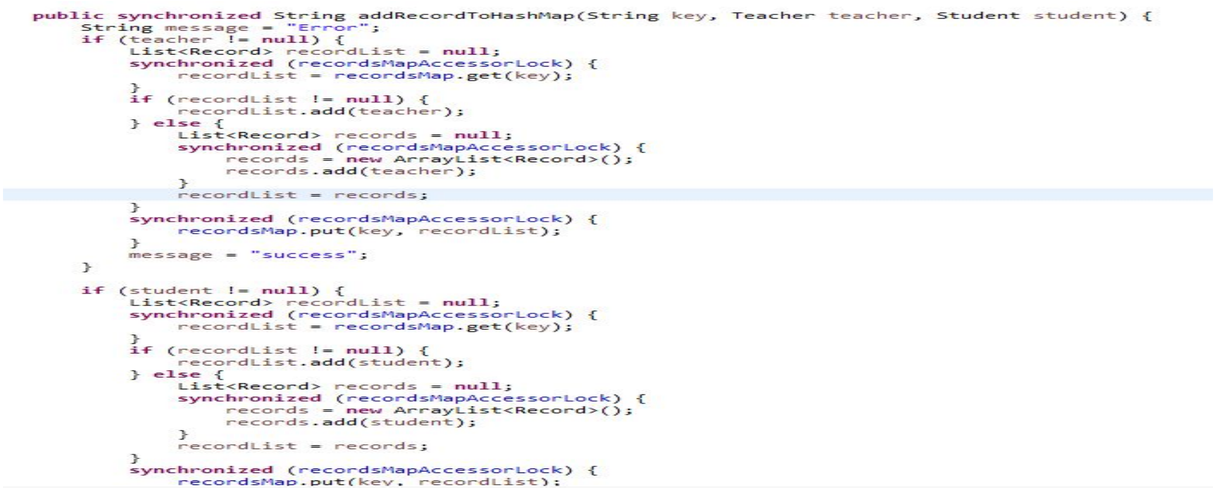


Synchronization & Concurrency Implementation

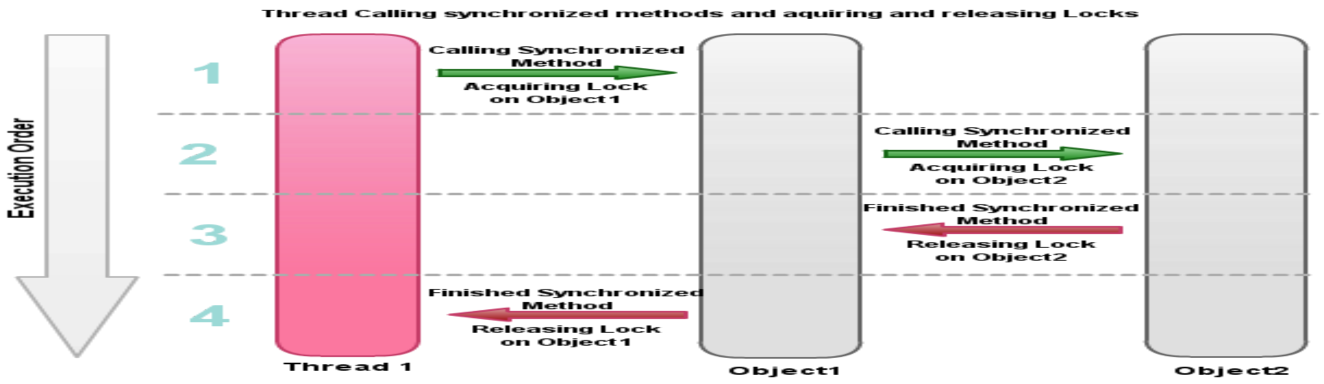
Concurrency factor plays a vital role when it comes to a distributed system with large amounts of records. This is because, many clients (managers in our case) can access the data concurrently which may lead to issues if not handled properly. Especially with many managers accessing at the same time from same/different locations, extra care must be put into concurrency implementation. Multiple methods which are accessed from the client in the server implementation access common methods, which should be synchronized for concurrent access by multiple clients. The DCMS implementation needs to be synchronized as multiple clients communicate with a server in the same location. Synchronization techniques implemented are as follows,

- The **data structures & statements** are synchronized at **lowest possible level** using the **Java's synchronization technique**.
- The **hashmap is synchronized**, also the **arraylist** within the hashmap is synchronized, everytime it is accessed.

- The **methods in the class** that are accessed by multiple threads are synchronized and synchronized blocks are used to obtain locks on different objects/variables/data structures wherever concurrent access is possible.
- **GetRecordCount and transferRecord** are served as UDP requests, wherein every request to getrecordcount/transferrecord is **served as a separate thread** by UDPRequestProvider and UDPRequestServer classes, which allows faster and more reliable communication among different locations.
- And moreover these two functionalities require synchronization among the three server locations, as the requests are handled not only between client-server, but also between the 3 server locations (MTL,LVL & DDO), **for which all the methods and data structures access within the threads are synchronized.**

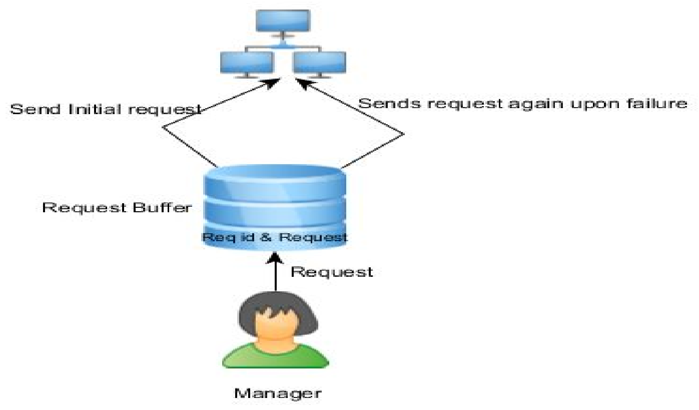


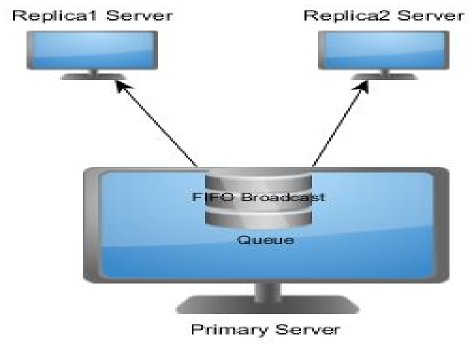
```
public synchronized String addRecordToHashMap(String key, Teacher teacher, Student student) {
    String message = "Error";
    if (teacher != null) {
        List<Record> recordList = null;
        synchronized (recordsMapAccessorLock) {
            recordList = recordsMap.get(key);
        }
        if (recordList != null) {
            recordList.add(teacher);
        } else {
            List<Record> records = null;
            synchronized (recordsMapAccessorLock) {
                records = new ArrayList<Record>();
                records.add(teacher);
            }
            recordList = records;
        }
        synchronized (recordsMapAccessorLock) {
            recordsMap.put(key, recordList);
        }
        message = "success";
    }

    if (student != null) {
        List<Record> recordList = null;
        synchronized (recordsMapAccessorLock) {
            recordList = recordsMap.get(key);
        }
        if (recordList != null) {
            recordList.add(student);
        } else {
            List<Record> records = null;
            synchronized (recordsMapAccessorLock) {
                records = new ArrayList<Record>();
                records.add(student);
            }
            recordList = records;
        }
        synchronized (recordsMapAccessorLock) {
            recordsMap.put(key, recordList);
```


Thread Calling synchronized methods and aquiring and releasing Locks
Execution Order
1
Calling Synchronized Method
Acquiring Lock on Object1
2
Calling Synchronized Method
Acquiring Lock on Object2
3
Finished Synchronized Method
Releasing Lock on Object2
4
Finished Synchronized Method
Releasing Lock on Object1
Thread 1
Object1
Object2

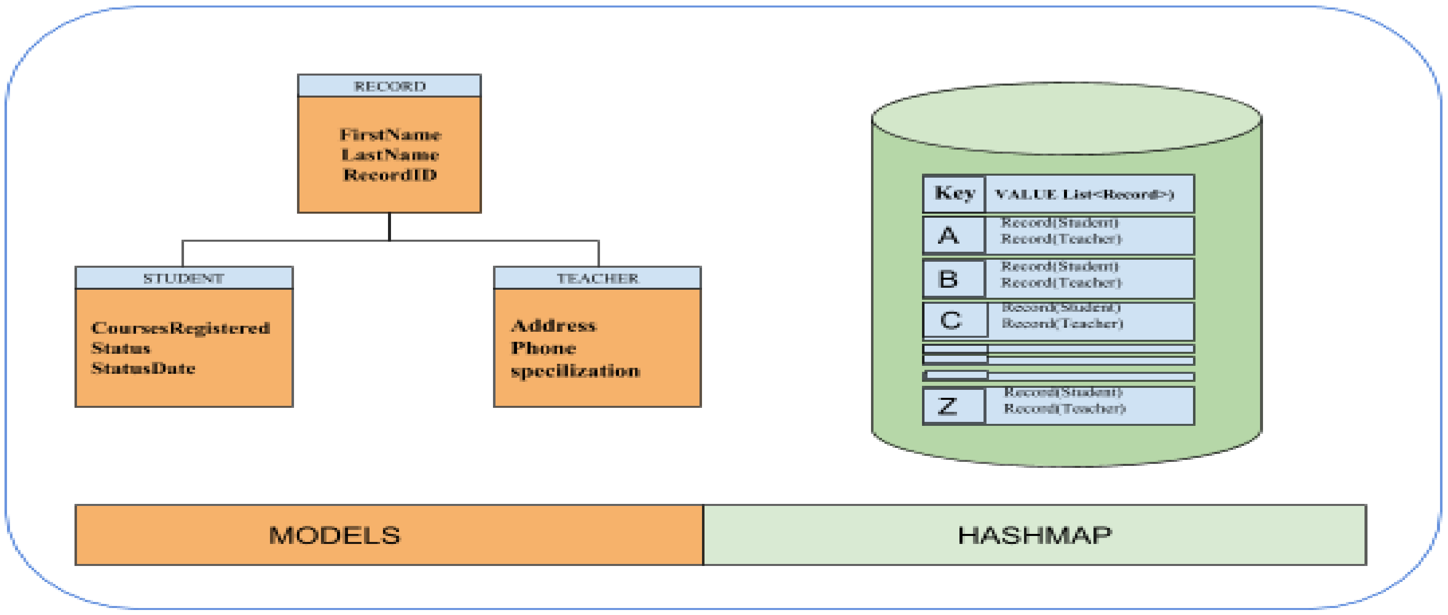
Data Structures

Req Buffer & FIFO Broadcast





HashMap & Class Models



Reliability Measures

Consistent DCMS backup
Reason: Incase of sudden power failure of any server, there’s always a risk of losing precious data.
Working: For every data that is being added into the server database(HashMap), a copy of it is maintained in a separate file. If at all a power failure occurs, the data can be retrieved from the file.

Acknowledgements
Reason: With many incoming UDP messages, there should be a way to confirm the reception of a message for security and reliability purposes.
Working: ‘ACK’ messages are sent as a response for all the incoming messages to provide a more reliable form of communication.

Temporary Buffer
Reason: When the primary fails, all the incoming requests may get lost if no dealt with properly.
Working: A temporary buffer is maintained at the ‘Front End’ of the DCMS to store the incoming requests temporarily until the next backup server is up.

Advantages:

Fault Tolerant: With the usage of proper algorithms and strong support from the replicas, we were able to achieve a highly available, fault tolerant system.

Concurrency: As we have implemented synchronization from the block level with the use of locks, we were able to handle the behaviour of multiple threads simultaneously.

Reliable UDP: It is a known fact that UDP is unreliable. Hence, we have implemented a version of UDP which provides an acknowledgement and response for every datagram packet received. In this way, we have achieved reliable UDP.

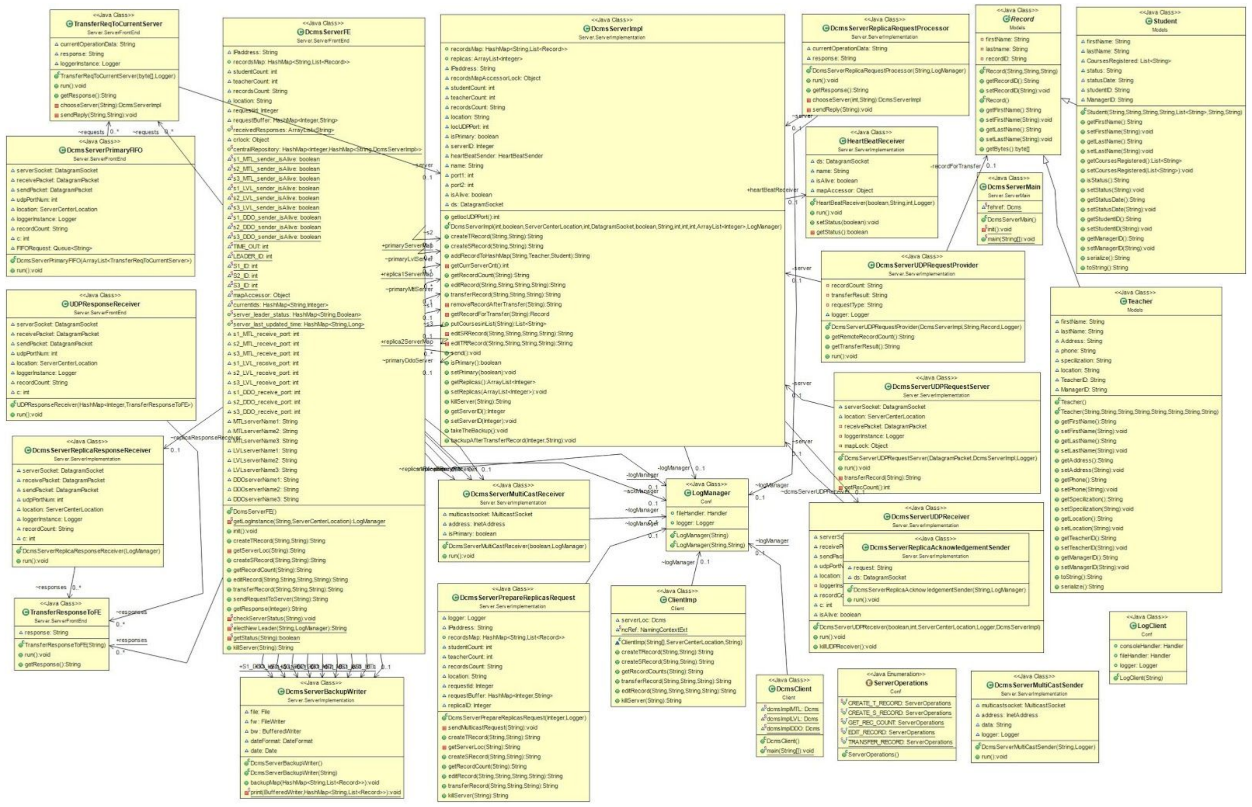
Data Backup: Since we are dealing with an application that is susceptible to hardware and software failures, precautionary steps have been taken to safeguard the repository(HashMap). This has been achieved by writing the contents of the HashMap to a file constantly for every operation performed.

Disadvantages:

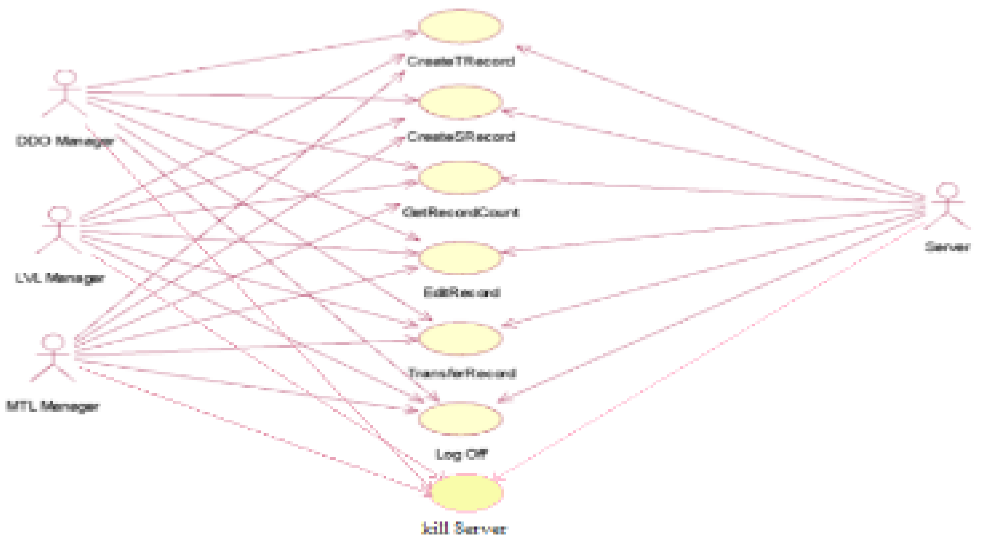
CPU Usage: Due to the use of multiple threads in our current DCMS project, many client-server interactions need to handled and processed. This might leads to the slow down CPU working due to higher CPU usage.
Restarting Server: After crashing any of the server (Primary or Replica), the server does not come back online automatically.

UML Diagrams

Class Diagram


TransferReqToCurrentServer
DcmsServerFE
DcmsServerImpl
DcmsServerReplicaRequestProcessor
Record
Student
DcmsServerPrimaryFIFO
HeartBeatReceiver
DcmsServerMain
UDPResponseReceiver
DcmsServerUDPRequestProvider
Teacher
DcmsServerUDPRequestServer
DcmsServerReplicaResponseReceiver
DcmsServerMultiCastReceiver
LogManager
DcmsServerUDPReceiver
DcmsServerReplicaAcknowledgementSender
TransferResponseToFE
DcmsServerPrepareReplicasRequest
ClientImp
LogClient
DcmsServerBackupWriter
DcmsClient
ServerOperations
DcmsServerMultiCastSender

Use case diagram



Challenges Faced

Synchronization:

With many threads running concurrently and continuous UDP messages, several functions needed to be synchronized in order to achieve reliability and stability. Therefore, several synchronization measures were taken.

Multicast UDP Communication between servers:

Primary along with the replicas have a lot of communication involved between them. This in includes status messages, getRecCount() functionality, transferRecord() functionality etc. Especially with multiple threads involved, communication with the right socket/port was quite the challenge.

Election Algorithm Implementation:

Implementing election algorithm and communicating among multiple servers after the new leader was elected was challenging.

Transfer Record and Get record count functionalities with multiple replicas:

When multiple replicas communicate with primary server, transfer record and get record count becomes complex and to return the correct result and maintain correct result in all the maps was challenging.

Heartbeat communication among the multiple servers:

Continuous and consistent communication among the servers and sending periodic messages to other servers in the group using UDP communication was challenging.

Front End and Primary Server Communication Implementation:

Communication between FE and Primary Server using UDP communication and implementing request buffer for serving requests was tedious.

Improvements:

Modified Election Algorithm: With various forms of Bully algorithm available, we can implement a type of bully that asks each process to have queue and flag to maintain a record history of all the leader and failed servers. This way, the redundancy involved with the messages can be reduced.

Increased Security: Due to the implementation of CORBA architecture, the DCMS project is prone to several network-based based. This includes, encryption attacks and man-in-the-middle attacks. Hence, using a more modified and recent architecture such as openDCE would help resolve such issues.

Better Scaling Efficiency: CPU usage can be optimized in a better manner to avoid slow down or crash of server and modules.

Test Cases

Test cases' inputs and outputs are stored in the **Test cases folder** in the project directory also with the Logs of the test cases.

Test ID	**Test Description**
T1	Client requests to create a record and Insert teacher record to server's hashmap
T2	Client requests to create a record and Insert student record to server's hashmap
T3	Login using valid credentials. Test Data: (MTLXXXX / LVLXXXX / DDOXXX)
T4	Login using invalid credentials.
T5	Get record count from all servers.
T6	Edit SR/TR record, with valid student ID.
T7	Validating input for Location field while editing record.
T8	Transfer record to itself
T9	Transfer record from one server to another server.
T11	● Add the teacher record in the server ● Get the record count ● kill the server, ● get the record count
T12	● add the student record. in the server. ● get the record count. ● kill the server. ● get the record count.

T13	● add the teacher record in the server. ● get the record count. ● kill the server. ● edit the record. ● get the record count.
T14	● add the student record in the server. ● get the record count ● kill the server ● edit the record ● get the record count.
T15	● add the teacher record in the server. ● get the record count ● kill the server in the LOC1 ● transfer the record from LOC1 to LOC2 ● edit the record in LOC2 ● get the record count
T16	● add the student record in the server. ● get the record count ● kill the server in the LOC1 ● transfer the record from LOC1 to LOC2 ● edit the record in LOC2 ● get the record count
T17	● add the teacher record ● get the record count ● kill the server ● transfer the teacher record ● get the record count
T18	● add the student record ● get the record count ● kill the server ● transfer the student record to LOC ● Kill LOC server ● get the record count
T19	● add the teacher record ● get the record count ● kill the server ● transfer the teacher record to LOC ● Kill LOC server ● get the record count
T20	● kill the server ● add the student record ● get the record count

T21	• kill the server • add the teacher record • get the record count
T22	• kill the server • add the student record • get the record count • edit the record • get the record count
T23	• kill the server • add the teacher record • get the record count • edit the record • get the record count
T24	• add the teacher record • transfer the record from LOC1 to LOC2 • kill the server LOC1 • kill the server LOC2 • get the record count
T25	• add the student record • transfer the record from LOC1 to LOC2 • kill the server LOC1 • kill the server LOC2 • get the record count
T26	• add the student record • get the record count • kill the server in LOC1 • transfer the student record from LOC1 to LOC2 • Kill LOC2 server • get the record count • transfer the student record from LOC2 to LOC3 • Kill LOC3 server • get the record count
T27	• add the teacher record • get the record count • kill the server in LOC1 • transfer the teacher record from LOC1 to LOC2 • Kill LOC2 server • get the record count • transfer the teacher record from LOC2 to LOC3 • Kill LOC3 server • get the record count

T28	● add the student record ● get the record count ● kill the server in LOC2 ● transfer the student record from LOC1 to LOC2 ● get the record count ● Kill LOC3 server ● get the record count ● transfer the student record from LOC2 to LOC3 ● get the record count
T29	● add the teacher record ● get the record count ● kill the server in LOC2 ● transfer the teacher record from LOC1 to LOC2 ● get the record count ● Kill LOC3 server ● get the record count ● transfer the teacher record from LOC2 to LOC3 ● get the record count
T30	● Kill all the Three servers ● Add a Teacher record ● Edit the Teacher record ● Get the record counts ● Transfer the records to another locations ● Get the record counts

References

1. http://blog.jbaysolutions.com/2011/10/06/java-threading-and-concurrency-introduction/
2. https://www.codeday.top/2017/08/18/33943.html
3. https://www.e-reading.club/chapter.php/143358/98/Tanenbaum_-_Distributed_operating_systems.html
4. https://www.techopedia.com/definition/3362/fault-tolerance
5. https://www.slideshare.net/sumitjain2013/fault-tolerance-in-distributed-systems
6. https://www.uml-diagrams.org/class-reference.html
7. https://docs.oracle.com/javase/7/docs/technotes/guides/idl/jidlExample.html