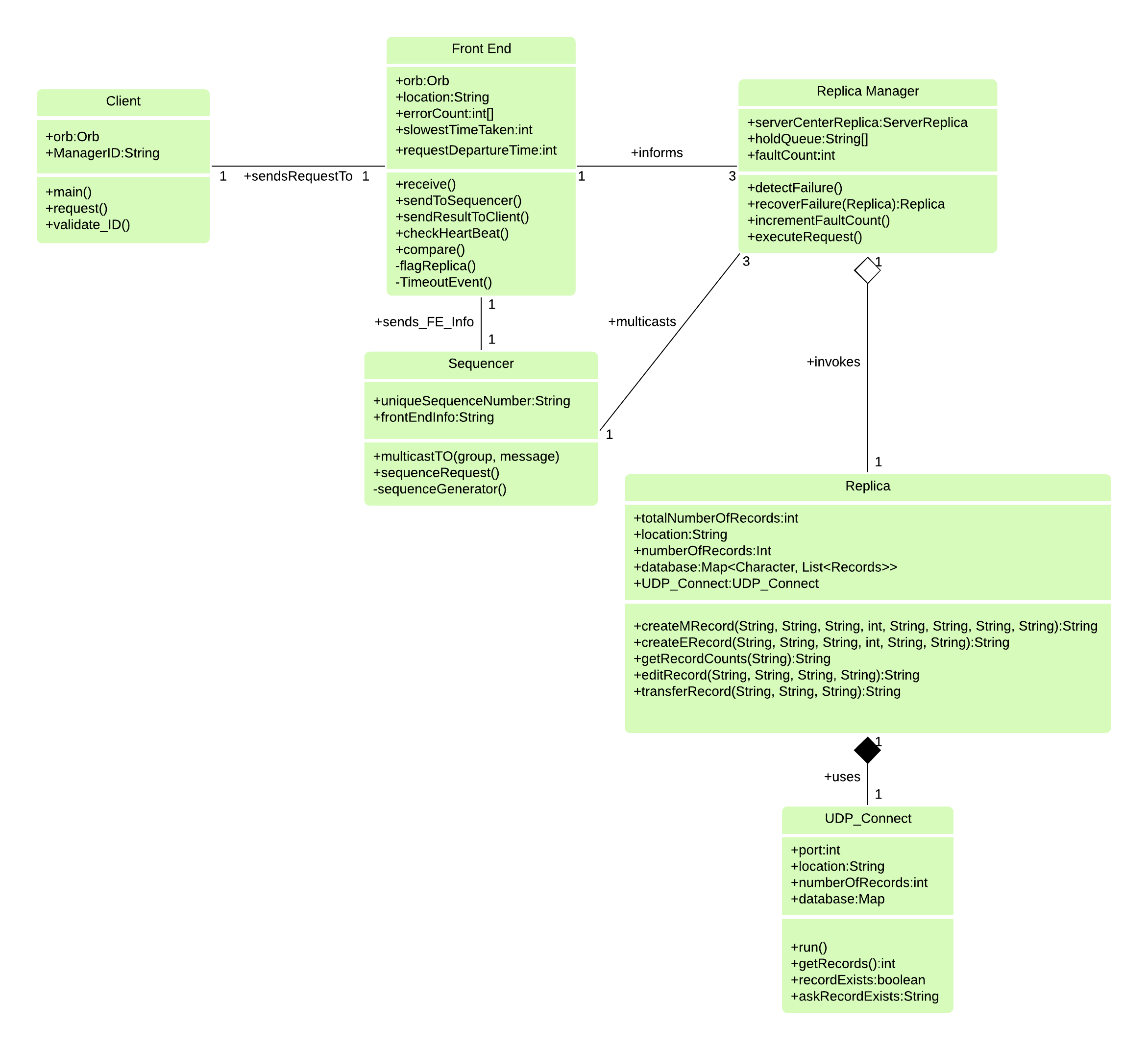
SOEN 423: Architecture and Design Documentation

Jordan Hubscher: 27019696

Dave Tetreault: 27762801

Olivier Mercier-Peetz: 27181000

**Introduction**

**Figure 1: Class Diagram for DEMS**

Our project will be incorporating six modules for each corresponding level of functionality and communication within the system. Each module will be represented in our class diagram (i.e. denoted in Figure 1) depicting the relationships amongst each other and their expected functions. This project will be composed of a client interface application, a server front end, a request sequencer, followed by a replica manager for each corresponding replica instance, as well as a UDP Connection class to handle UDP communications. The client will have very simple command for which they can call and they will strictly communicate with the front end.

**Client**

The client application will be communicating directly with a respective FrontEnd via CORBA. This will involve both modules sharing the same IDL descriptions.

**Front End**

The FrontEnd will be in charge of receiving client requests and passing them along with their method name and arguments to the Sequencer. The FrontEnd and the Sequencer communicate via UDP. The FrontEnd will pass the method invocation and argument details to the Sequencer via it’s sendToSequencer() method. sendResultToClient(), checkHeartBeat(), compare() and increment\_fault\_count(). This will allow us to detect directly if a replica has had a system failure through the checkHeartBeat() method as well if it is responsive enough to continue using incrementFaultCount() which will accept a certain number of faults before taking action towards that replica.

**Sequencer**

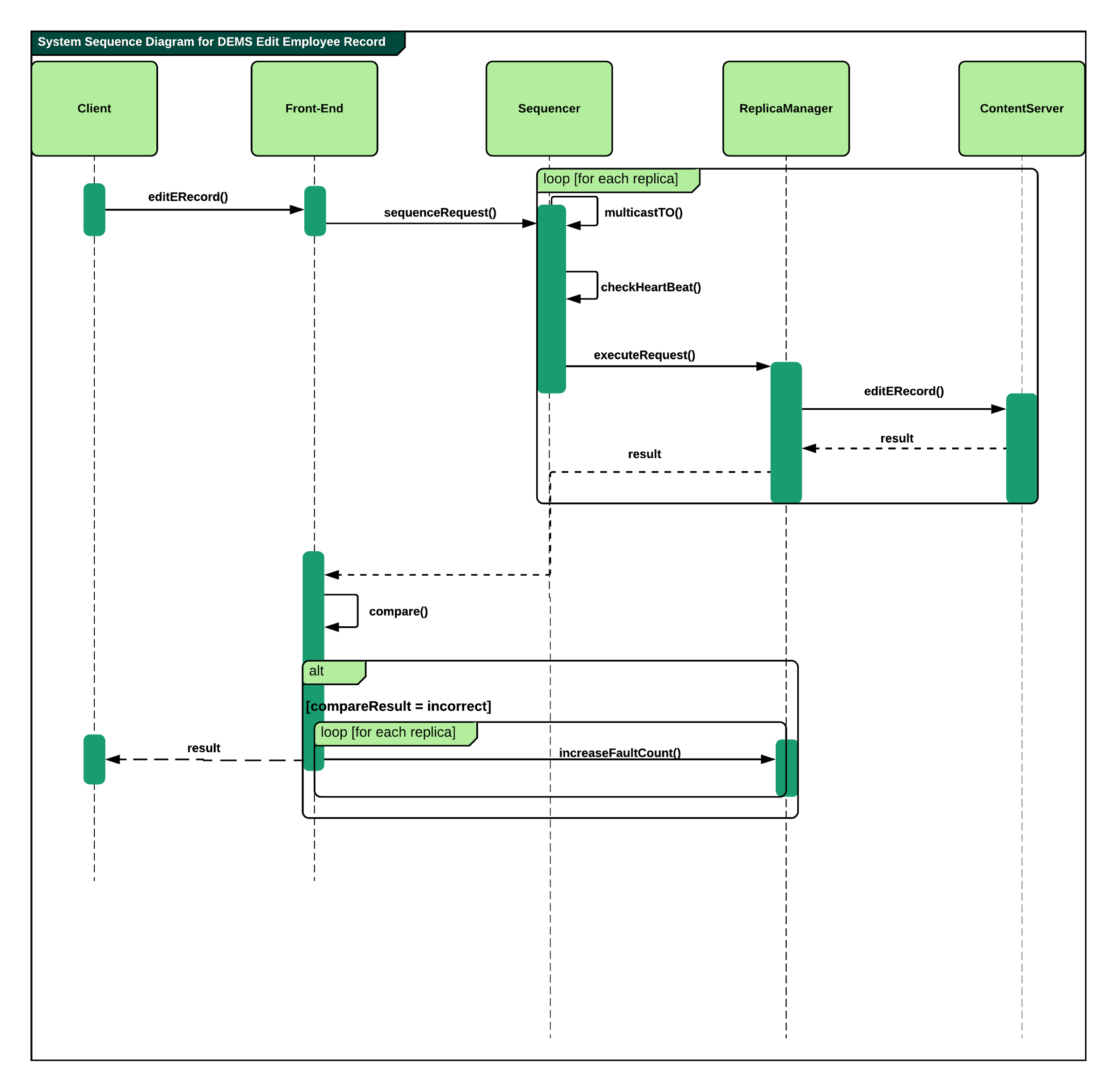
The Sequencer will have the ability to communicate with the FrontEnd as well as the ReplicManagers in order to allow proper total order multicast. To have reliable multicast we will aim to use total ordering as well as Kaashoeks protocol to allow the sequencer to store its history in case there is a need for re transmission. The sequencer will have the crucial method of multicastTO() which will allow it to send a message to multiple Replica Managers through UDP all at once.

**Replica Manager**

The ReplicaManager will be in charge of detecting system failure and one will be assigned to every replica. It will have the detectFailure() method which allows for it to detect failure as well as recoverFailure() which will allow for replica recovery. It will also be equipped with a holdQueue to make sure that the order of the message passing is respected. The Replica Manager will also be able to directly inform the FE of any system failures to make sure that it is aware and can properly communicate the correct information to the client.

**Replica (i.e. CenterServer instance)**

The Replica class will be in charge of the main actions taken by the system. These methods are createMRecord(), createERecord(), getRecordCounts(), editRecord() and transferRecord(). This class will interact directly with UDP\_Connect class in order to have access the local database stored in a hash map.

**Figure 2: System Sequence diagram for editERecord()**

**High Availability**

We will demonstrate that the distributed system is highly available by crashing a process and then recover by using active replication. Once the FE forwards a request to the sequencer, it stores the time in millisecond in the variable requestDepartureTime. When the requests makes its way to the replica, a value is returned to the Front End afterwards. The FE is expecting 3 returns, once the first is received the FE calls checkHeartBeat(). There are two possible scenarios to check for a crash.

The first is when the FE still hasn’t received any values in the past, this means that there are no values for slowestTimeTaken yet. The FE stores the current time in milliseconds of when the first value was received. It does the same for the second value. It then starts a timer with the difference of the time of the second time with the first time X 2. When the timer ends, it checks if it received a third value. If it did, it takes the difference between the requestDepartureTime and the time of reception of the third value as slowestTimeTaken. No faults were found. If it did not receive a third value after the timer has run out, it informs the RMs which in turn ping the replicas to see if one has crashed. If it is the case, the crashed replica is replaced by another one with recoverFailure(replica).

The second scenario is when slowestTimeTaken already has a value. A timer starts with twice the value of slowestTimeTaken. Once the timer runs out, checkHeartBeat() checks if the FE received 3 values. If it did, no faults were found. If not, it recovers by doing the same process as in the other scenario.

**Fault tolerant**

We will show that the distributed system is fault tolerant by forcing a single software failure ; A replica will return a random value. When the FE receives three values from the replicas, it uses compare() to see if the results are the same.If at least two of the three returned values are the same, it forwards that same value to the client. If the value received from one the replicas differs, it increments the element in errorCount[replica1, replica2, replica3] array that represents the error prone replica. The FE informs all the RM’s of the error. The RM in charge of that replica increments the fault count. The other RM’s set the fault count of their replicas to 0. (The fault count must be consecutive errors). If the fault count of an RM reaches 3, the replica assigned to that RM is replaced by another one with recoverFailure(replica).