Software Failure Tolerant CORBA Distributed Staff Management System

DISTRIBUTED SYSTEM DESIGN PROJECT

COMP 6231

SUMMER 2016

Team Members

Ananta Purohit 26594530

Leila Abdollahi Vayghan 26722792

Mandeep Kaur 40014707

The purpose of this document is to elicit the design, implementation and testing of the fault tolerant and highly available Distributed Staff Management System.

1. Introduction

The key features of a Distributed System is that it should be fault tolerant and most importantly, highly available. To assure that a software system has attained both of these features depends on the way the system is designed and on the underlying architecture. In order to achieve these qualities for a software system, special methods such as process replication can be used to improve both the fault tolerance and the availability of the system.

A Software Failure Tolerant CORBA Distributed Staff Management System is a Distributed System which allows a particular task performed by three different processes or replicas (on different hosts in a network) controlled by a front end process and a client, collectively makes a highly available system by performing active replication. If for some reason any of the replicas is failed/crashed, the Replica Manager is able to cope with this issue by replacing the failed with a good one. To induce the qualities of fault tolerance and high availability to the DSMS, we adopt an active replication process, wherein the DSMS server system is replicated into several copies each running different instance of the resources in different hosts within a network. One should note that the replication transparency is a requirement when the data are replicated. It says that the client should not be aware that multiple copies of data exist. One of the tasks of Front End is to make replication transparent.

1.1 Active Replication

In the distributed systems research area replication is mainly used to provide fault tolerance. The entity being replicated is a process. Two replication strategies have been used in distributed systems: **Active** and **Passive** replication. We are working with active replication. In active replication each client request is processed by all the servers. Active Replication was first introduced by Leslie Lampard under the name **state machine replication**. This requires that the process hosted by the servers is deterministic. **Deterministic** means that, given the same initial state and a request sequence, all processes will produce the same response sequence and end up in the same final state. In order to make all the servers receive the same sequence of operations, an atomic broadcast protocol must be used. An atomic broadcast protocol guarantees that either all the servers receive a message or none, plus that they all receive messages in the same order. The big disadvantage for active replication is that in practice most of the real world servers are non-deterministic. Still active replication is the preferable choice when dealing with real time systems that require quick response even under the presence of faults or with systems that must handle byzantine faults.

2. Problem Description

The DSMS system we designed will tolerate a single replica crash using active replication. By using active replication instead of a single server executing the client request, we have three replicas with different server implementations processing the client request so that a crashed server can be restored with a healthy replica. The front end will receive requests from clients through a CORBA communication. After receiving the request, the FE multicasts the request to all replicas through IP multicast using UDP protocol and uses total ordering so that all requests are executed in the same order. Moreover, total ordering will make the UDP reliable. We are using two buffers, one to send the requests and another one to receive the request. By implementing the buffers in total order we can ensure the order of the request sent by a replica will be in the same order as received by another replica. Once the FE receives the responses from replicas it will sends a single correct response back to the client (through CORBA). In case a replica does not reply for 10 seconds then the FE informs the replica manager (RM) and the replica manager will confirm the crash and if it concludes that the replica is indeed crashed, will replace the replica as well. After restoring the crashed replica, the corresponding RM will communicate with another RM and will update both the request queue and hashmap of the previously crashed replica.
Since this is a distributed application, the communication between the replicas, RM and front end take place using an unreliable UDP communication. The unreliability is due to absence of virtual connection which leads to out of order delivery of messages, and receiving duplicate messages. The communication is made reliable by implementing total order buffers to tackle the out of order delivery of messages and duplicate messages can be filtered out using the sequence numbering.

3. Technologies and Techniques

The following technologies, algorithms and Protocols are used in the design of the DSMS. This section also describes the need for using CORBA and UDP.

3.1. User Datagram Protocol

UDP is one of the transport layer protocols that sits above the Internet Protocol (IP) ; Thus, uses IP to get data unit (datagram) from one computer to another. Unlike TCP, UDP does not divide message into packets and does not guarantee delivery and the order at which messages arrive at its destination.

However, UDP does not establish end to end connections between communicating end systems. UDP communication consequently does not incur connection establishment and teardown overheads and there is minimal associated end system state, hence UDP can offer a very efficient communication to some applications. In our system, servers are communicating among themselves through UDP.
UDP adds the unreliability which leads to out of order delivery of messages, and receiving duplicate messages. The communication is made reliable by implementing total order buffers to tackle the out of order delivery of messages and duplicate messages can be filtered out using the sequence numbering.[1]

3.2. CORBA

CORBA is a standard architecture for a distributed object system. It enables communication between distributed objects implemented in different programming languages. One of the components of CORBA whose role is to carry out the communication between object is called Object Request Broker (ORB). ORB helps in locating an object, activating the object if necessary and then communicating the client's request to the object, which carries it out and replies [2].

3.3. Why do we need CORBA and UDP?

This distributed application aggregates the use of both CORBA as well as UDP implementation. CORBA enables software written in different languages and running on different computers to work with each other seamlessly. Implementation details from specific operating systems, programming languages, and hardware platforms are all removed from the responsibility of developers who use CORBA. With the help of CORBA the managers are able to invoke methods in the server transparently in order to perform various functions.

Apart from achieving the language independence as well as platform independence, by using CORBA we also attain various transparencies like access transparency, location transparency, replication transparency and concurrent transparency.

- *Access transparency* allows us to use the remote object in the similar way as the local object
- With *Location transparency* the details of the topology of the system will be of no concern to the client. The location of an object in the system may not be visible to the user or programmer.
- By using *Replication transparency* the player is not aware of how many replicas are actually executing the request. It will resemble to client as if only one server is executing the request.
- In *Concurrent transparency* the users and Applications will be able to access shared data or objects without interference between each other.

On the other hand, User Datagram Protocol (UDP) is a transport layer protocol which is used to send messages, in this case referred to as *datagram*, to other hosts on an Internet Protocol (IP) network without prior communications to set up special transmission channels or data paths. Here User Datagram Protocol (UDP) is employed to enable communication between the servers in each replica, the front end and the replicas and between the replicas.

3.5. Fault Tolerance Protocol

The replica managers are in charge of maintaining the replicas in an active and valid state of operation. It does so through pinging the replica, if the replica does not respond after 10 seconds, the replica manager will assume the replica has crashed and will replace the crashed replica with a good one.

3.5 Total Ordering

Total ordering takes cares about the processing of messages. There is one sequence class for all the replicas. This class increments the sequencer at the time of multicast. Then every replica calls the sequence class to get the sequencer and then compare it with the sequence number send by the front end in the messages. If order is correct then replica processes the message otherwise replica put the message in hold back queue and wait for the message with correct sequence. After receiving next messages, replica gets the message from queue and again check the sequence, if the sequence is correct, then delete that message from the hold- back queue and process it.

4. Architecture Design

Below is a high level look into the design of the system.

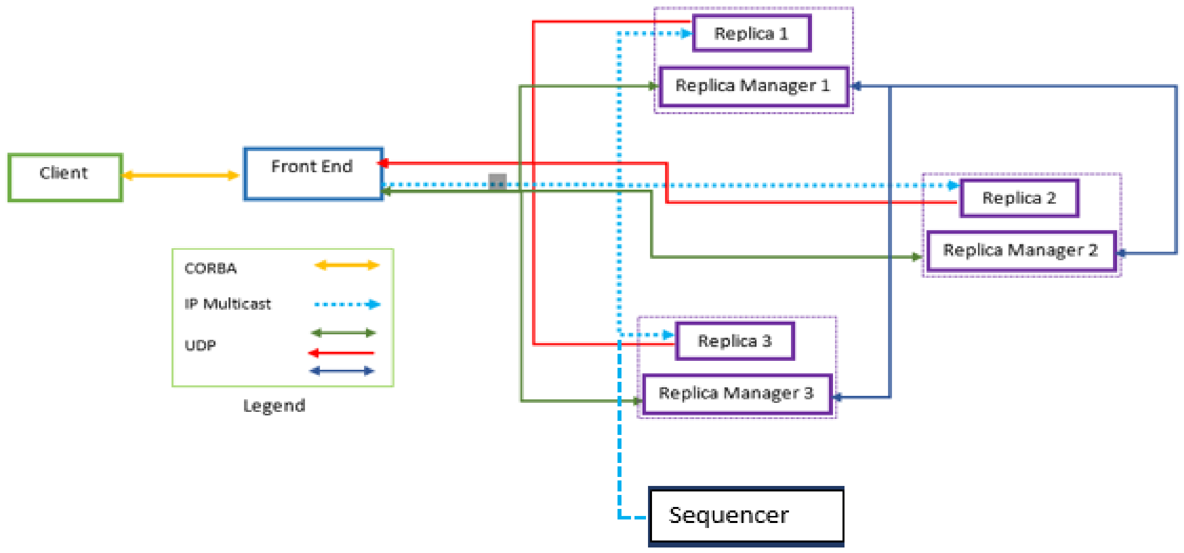


Figure 1High level look of the system [2]

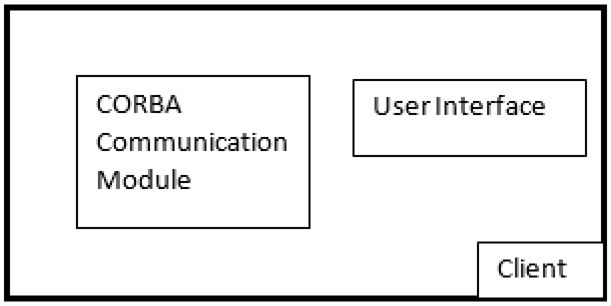
4.1. Design Description

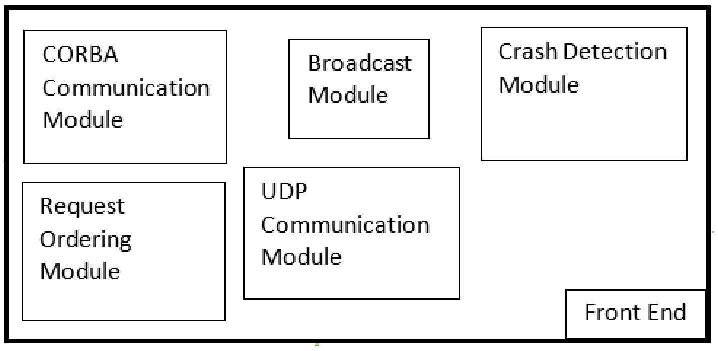
Under active replication, the sequence of events when a client requests an operation to be performed is as follows:

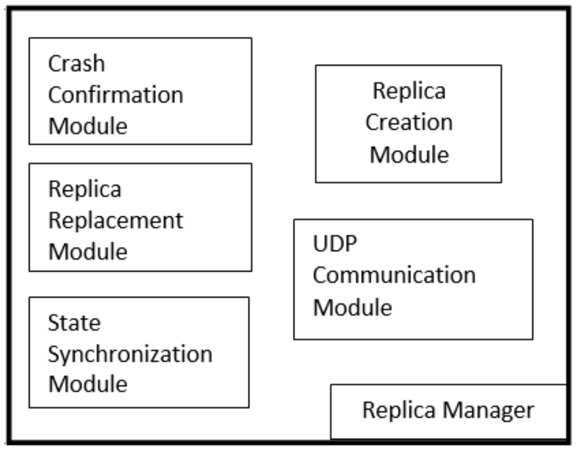
1. Request: The front end attaches a unique identifier to the request and multicasts it to the group of replica managers, using a totally ordered, reliable multicast primitive. The front end is assumed to fail by crashing at worst. It does not issue the next request until it has received a response.
2. Coordination: The group communication system delivers the request to every correct replica manager in the same (total) order.
3. Execution: Every replica manager executes the request. Since they are state machines and since requests are delivered in the same total order, correct replica managers all process the request identically. The response contains the client's unique request identifier.
4. Agreement: No agreement phase is needed, because of the multicast delivery semantics.
5. Response: Each replica manager sends its response to the front end. The number of replies that the front end collects depends upon the failure assumptions and the multicast algorithm. If, for example, the

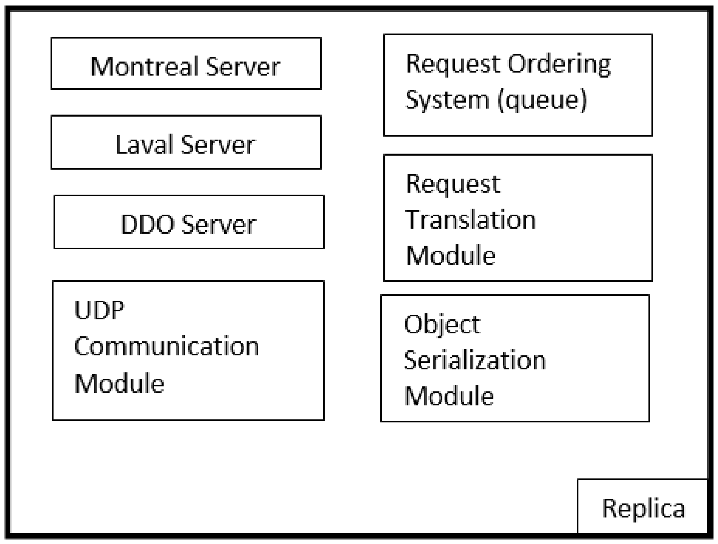
goal is to tolerate only crash failures and the multicast satisfies uniform agreement and ordering properties, then the front end passes the first response to arrive back to the client and discards the rest (it can distinguish these from responses to other requests by examining the identifier in the response).[2]

A detailed diagram of the modules seen in Figure 1 is shown here.






Crash Confirmation Module
Replica Creation Module
Replica Replacement Module
UDP Communication Module
State Synchronization Module
Replica Manager


Montreal Server
Request Ordering System (queue)
Laval Server
DDO Server
Request Translation Module
UDP Communication Module
Object Serialization Module
Replica

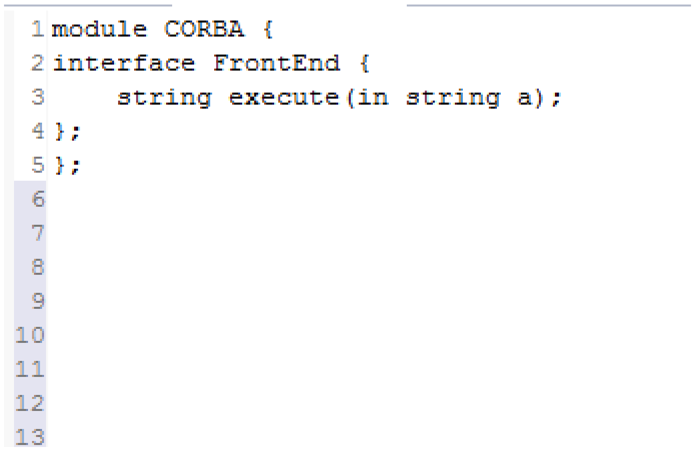
5. Explanation of tasks (based on the order mentioned in project description)

The following Section describes the individual implementations of the three defined parts of the Project by the team members-

5.1 Ananta (26594530):

This part includes the implementation of the Front End. The Front End acts as a forwarder of the request received from the client, to the three replicas and do not perform any computation. It also does the fault detection in case of replica crash, and informs the concerned Replica Manager about the crash.

It starts with defining the Interface Definition Language, the FronEnd.idl:-



```
module CORBA {
interface FrontEnd {
    string execute(in string a);
};
};
```

The Interface includes one method –

string execute – which takes the message of an operation to execute.

The following three classes do the defined task of Front End:-

My FrondEnd.java- It is the main class and extends the FrontEndPOA.java.It performs the initialization of the ORB. It uses the method MulticastMessageSender To send the message over UDP to the replicas and receives the response from it. It detects the crash in any of the replica if any and provides the fault tolerant feature by sending the failure message of the crashed replica to the concerned Replica Manager. Additionally, it also sends the message of request to the sequencer along with all the three replicas just to ensure the sequencer that the communication/request's sent receive has been started and the sequencer has to be started for providing the total ordering to the system.

MulticastMessageSender.java- It builds the structures of the message to be multicast to all the replicas. It uses the three predefined ports of the replicas defined in Config.java to send the message of requests.

To provide a guaranteed delivery of the request to the replicas, it counts the responses received before the final response is received from the replicas.

Login.java- It sends the operation request to the Front End through CORBA, to ensure more reliability, less latency and allow access transparency to the client. It contains the user interface to perform test scenarios. It uses the Front End port to send/receive the message / request, processed by the replicas.

5.2 Leila (26722792):

This part includes three replicas (the individual assignment2 of the three team members) and the Replica Managers which control the replicas. The Replica Manager communicates through UDP with the Front End to get the crash failure information of any of the replicas (if any). And, the replicas get the request from the Client through the Front End, using the reliable UDP implemented by Mandeep and Ananta.

The following classes which define the task of RM are as below:-

1.**RM.java-** It initializes the three replicas and the sequencer and provides four instances to each server (i.e. first to define the replica which sits between Front End and the server, the other three instances are for the three servers LVL,DDO,MTL).

The **“crash”** method is to execute a test scenario, used to kill the replica to simulate the crash of replica.

The **“confirm”** method confirms the crash information received by the Front End. It returns true if the replica is cashed and false if not. This is done by setting a time out on the socket for 10 seconds.

The method **“restore”** performs the restoring of the crashed replica by restarting the thread. After restoring, the method **‘stateUpdate’** sends request to another replica manager and asks for information to update the state of the crashed replica by using its own port. Note that the data hashmaps, the message hashmap and also message queue is also updated in this part.

2.**ServerAnanta.java-** It initializes the replica 1 and initiates all the variables necessary (for eg. Udpserver port, current location, all the three servers) and the repository /hashmap to store the data. It also initiates the queue to store the request from the Front End, it is also transferred along with the state information in case of replica crash. The Server gets initialized by passing the server port and the location where to send the operation. It also implements all the defined operations/methods required to perform on the replicas.

3.**ServerLeila.java-** It initializes the replica 2 and initiates all the variables necessary (for eg. Udpserver port, current location, all the three servers) and the repository /hashmap to store the data.It also initiates the queue to store the request from the Front End, it is also transferred along with the state information in case of replica crash. The Server gets initialized by passing the server port and the location where to send the operation. It also implements all the defined operations/methods required to perform on the replicas.

4.**ServerMandeep.java-** It initializes the replica 3 and initiates all the variables necessary (for eg. Udpserver port, current location, all the three servers) and the repository /hashmap to store the data.It also initiates the queue to store the request from the Front End, it is also transferred along with the state information in case of replica crash. The Server gets initialized by passing the server port and the location where to send the operation. It also implements all the defined operations/methods required to perform on the replicas.

4.State.java- This class is for running threads of outputstream in order to send a hashmap to another replica.

5. Config.java- All ports and addresses related to system members are accessed through this class.

6.Crasher.java- This class sends a crash request to a replica manager for testing purposes.

Extra: All system members save their activities in log files so it is easy to track the fellow of the procedures. Log4j is used for this part.

5.3 Mandeep (40014707):

tSequencer.java - This class contains two methods. Increase (), this method increments the sequencer when FrontEnd multicast the request. dSequence (), this method is called by all the replicas to get the updated sequence number.

ServerLeila.java, ServerAnante.java, ServerMandeep.java – checkSequence() method calls the dSequence() method of tSequencer.java class to check the sequence of messages. If message is out of order the put the sequence number in queue (message) and sequence + message in hashmap (hmessage). If message is in correct order. checkSequence () calls the processMsg() function for further processing.

6. Assumptions

1. The Front End will never crash in spite of being considered as a single point of failure.
2. The Sequencer will never crash in spite of being considered as a single point of failure.
3. The Replica Manager will never crash.
4. Assuming no network failure as we are using the system on local machine.
5. We cannot expect total efficiency in case of getting the response from the replica.

7. Challenges

1. It was difficult to work with the integration of the each participant's code.
2. Especially, we faced many issues regarding the message format sending and receiving from the Front End to the Replicas. We tried to solve the issue but at the end it resulted in replacing the message build functionality of the Front End part with Replica Manager .We also had to change the FrontEnd.idl.

The Message Format used to send the request from the Front End to replica was structured as shown below-

Message [messageID=1, message=<null>, method=DR, managerID=DDO1111, firstName=null, lastName=DILINGER, address=Sherbrook, phone=514324234, specialization=ORTHO, location=Halifax, designation=null, status=null, statusDate=null, recordType=null, fieldName=JOHN, newValue=null, recordID=null, remoteServerName=null, messageType=OPERATION, processID=<process id>]

The Message Format expected from the Replica to the Front End was structured as shown below-

Message [messageID=1, message="<Actual-response from method call>", method=DR, managerID=DDO1111, firstName=null, lastName=DILINGER, address=Sherbrook, phone=514324234, specialization=ORTHO, location=Halifax, designation=null, status=null, statusDate=null, recordType=null, fieldName=JOHN, newValue=null, recordID=null, remoteServerName=null, messageType=RESPONSE, processID=<replica-id>]

Now, the revised message format is-
"FE,,ReplicaLeila,,2,,added records are: " + server.getRecord()+" and the currentcount is: " + server.currentCount," ";

3. To understand the functionality of total ordering was time consuming and difficult to implement and integrate.
4. Sending a hashmap through sockets was very difficult to understand and implement.

8. References

[1] S. Deering, "Host Extensions for IP Multicasting," August 1989. [Online]. Available: http://tools.ietf.org/html/rfc1112. [Accessed 14 06 2014].

[2] C. George, D. Jean , K. Tim and B. Gordon, Distributed System: Concept and Design, Pearson, 2012.

[3] G. Fairhurst, "erg," 19 11 2008. [Online]. Available: http://www.erg.abdn.ac.uk/~gorry/eg3561/inet-pages/udp.html. [Accessed 13 6 2014].