FALL 2013

COMP 6231 – DISTRIBUTED SYSTEMS DESIGN

HIGHLY AVAILABLE DISTRIBUTED POLICE INFORMATION SYSTEM

Submitted by:

Jaiminkumar Patel	6775330
Pratik Bidkar	6746349
Sohan Argulwar	6757871
Hiren Tarsadiya	6742904

Contents

Design and Architecture:

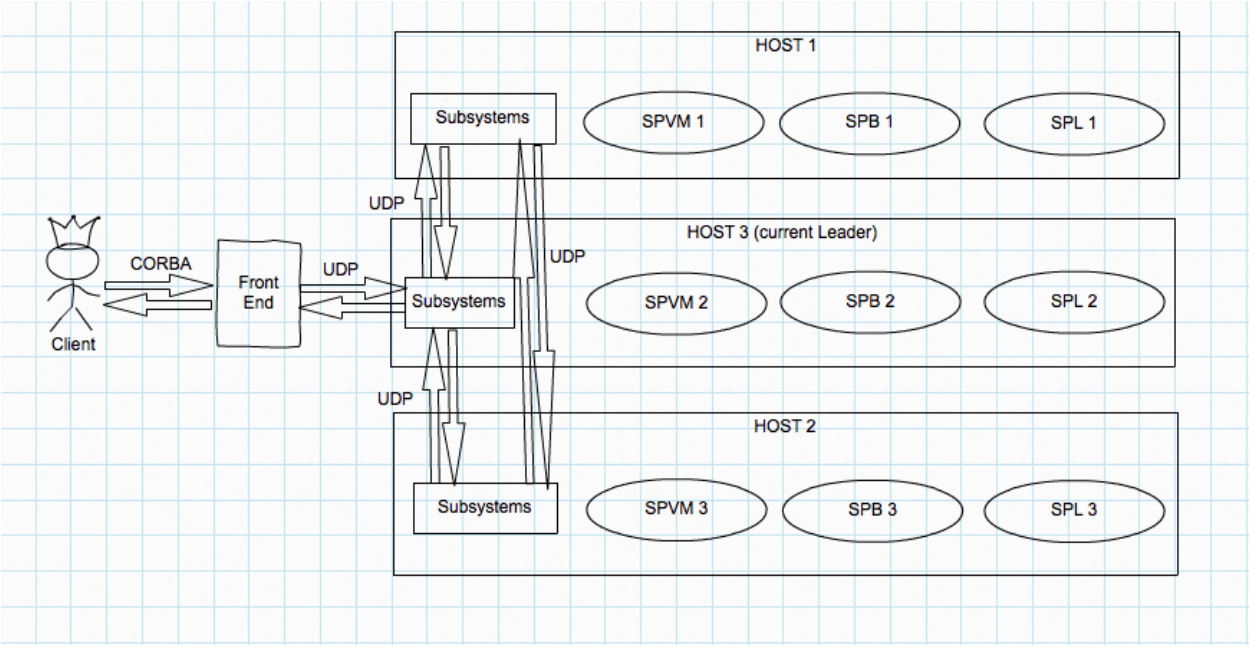
The primary goal of the system is to tolerate a single benign process failure. The following techniques are used to achieve this:

- Replicated Servers
- A failure detection subsystem
- Subsystems (Leader Election) to restore system integrity after a server failure.

However the system performance could be affected during critical periods related to process failure and initialization.

Overview of the Distributed System:

The client and the front end communicate using CORBA. The front end and the leader communicate using UDP. The client will send request to front end and front end will send the request to leader process. The leader process will execute the operation in its own host, also will broadcast the client request atomically to all the servers replicas, receive the responses back from the servers, compares the results and sends a single response back to the client. The leader process and the front end communicate using the reliable UDP protocol. The unreliable UDP protocol has to be made reliable using some approaches. We would be using the Acknowledgement technique to achieve this.



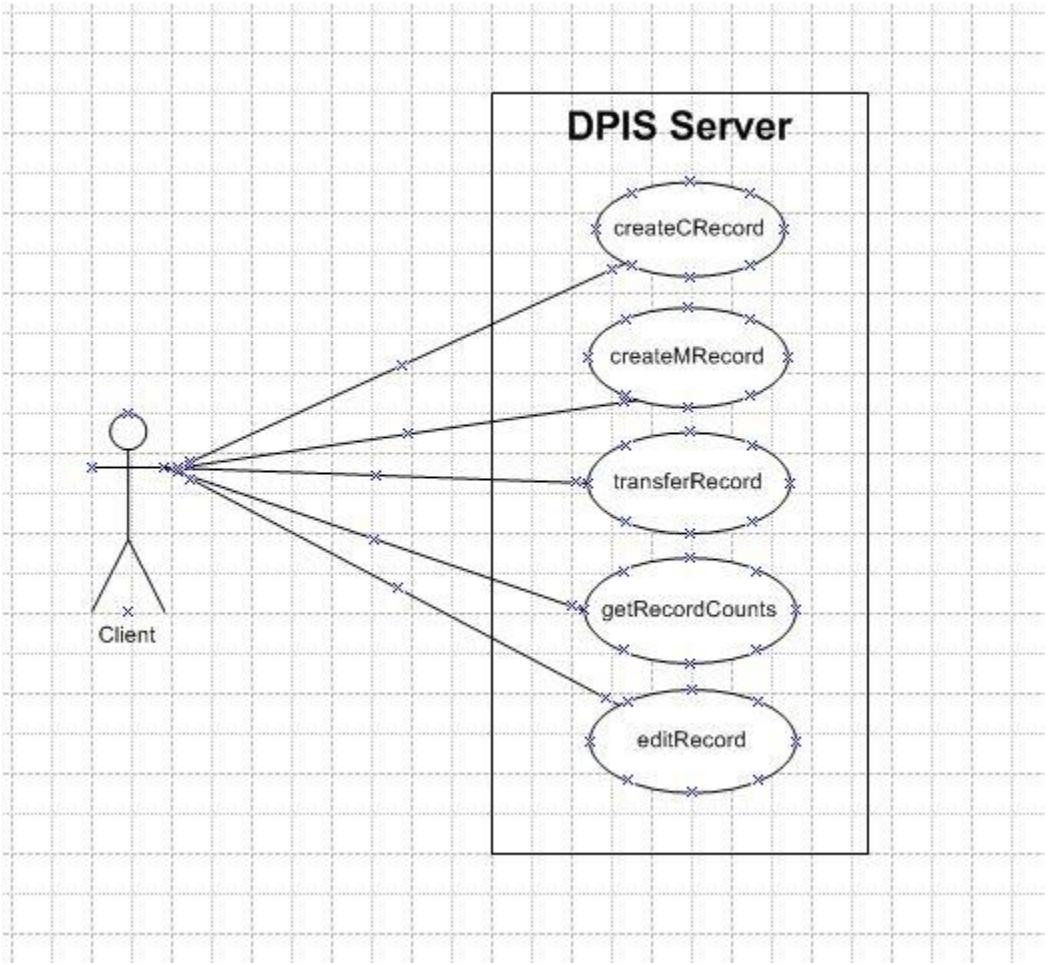
Subsystems explanation: This comprises of the following mechanisms:

- Reliable FIFO broadcast subsystem.
- Leader election subsystem.
- Failure detection subsystem.

These subsystems will be a part of each replica server process. The failure detection subsystem residing in each host will periodically check with each other for the host availability and remove the failed server process. It will trigger the leader election subsystem in case of group leader process failure.

The FIFO broadcast mechanism (IP multicast) residing in the leader process will forward the request to other server processes in the group whenever it will receive the request from the Front End.

Use Case Diagram:



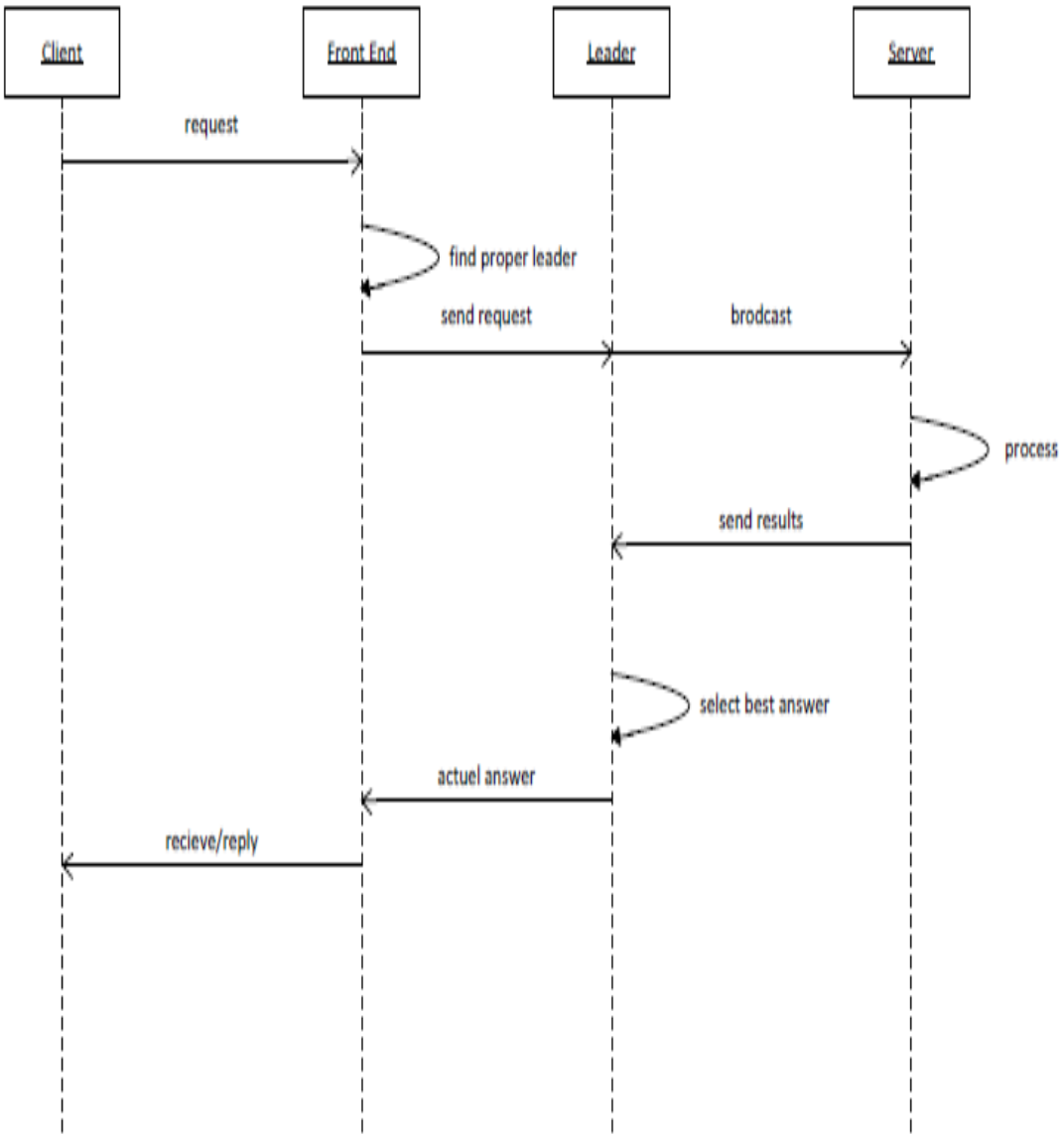
Each client can send his/her request to server and receives the result without knowing in which server or host the processes have been performed. So the system will be transparent from the client view.

The operations that a client can ask for are:

- Create a criminal/missing record for any station for a badgeID.
- Edit an existing record and update its status for a badgeID.
- Get the total record counts for a badgeID.

- Transfer an existing record from one station to other station for a badgeID.

System Sequence:

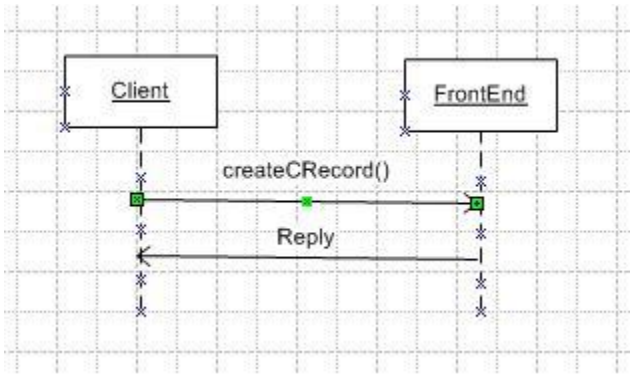


Client invokes a remote object through the front end; Client and front end communicate with each other through CORBA. Actually, Front end is the entry point of the server.

Detailed Functional Implementation:

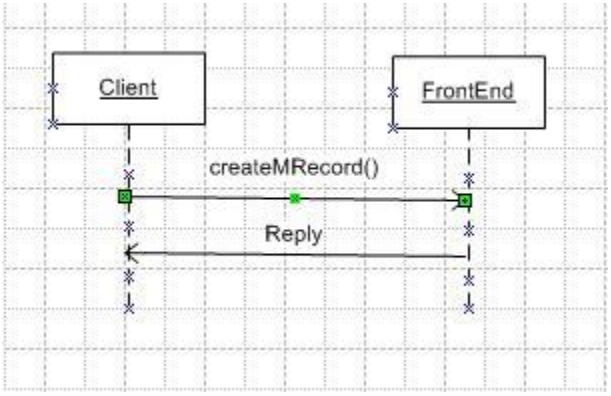
All operations will send by client to server through front end are:

1. Creating a Criminal Record:



When a client sends a request to create a criminal record for one of the stations through the front end. Front end sends the request to right leader and waits for the reply. After receiving the request response, the front end sends result to the Client.

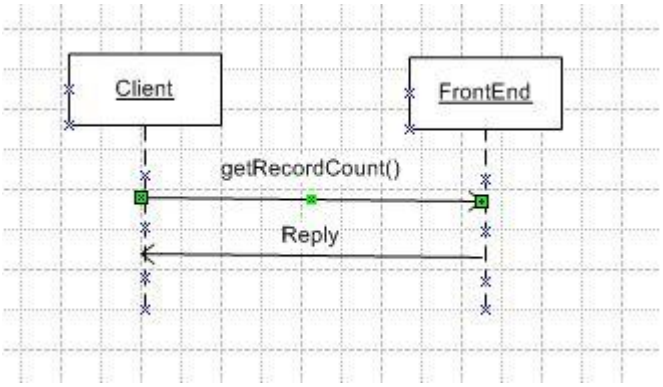
2. Create a Missing Record



When a client sends a request to create a criminal record for one of the stations through the front end, front end sends the request to right leader and waits for the reply. After receiving the request response, the front end sends result to the Client.

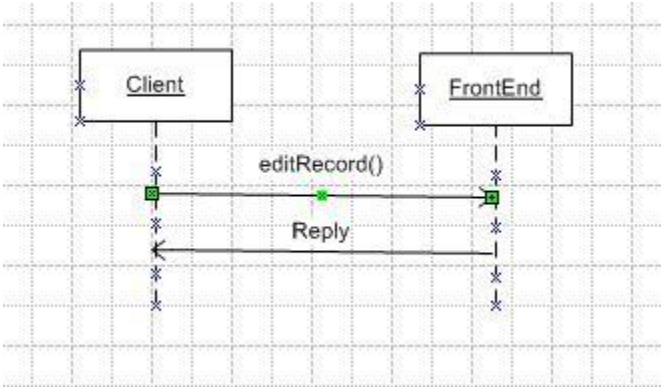
3. Get the record counts.

   When a client sends a request to retrieve the count of records from one of the stations through the front end, front end sends the request to right leader and waits for the reply. After receiving the request response, the front end sends result to the Client.



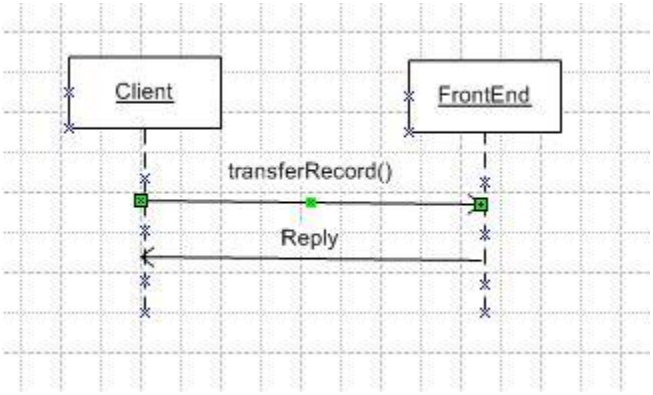
.

4. Edit an existing record.



When a client sends a request to edit a record for one of the stations through the front end, front end sends the request to right leader and waits for the reply. After receiving the request response, the front end sends result to the Client.

5. Transfer a record from one station to another station.



When a client sends a request to transfer a record for one of the stations through the front end, front end sends the request to right leader and waits for the reply. After receiving the request response, the front end sends result to the Client.

Implementation:

Limitations and Assumptions:

Assumptions:

Before designing the distributed systems we have made certain assumptions that are out of the scope of this project.

1. The CORBA architecture will not fail.
   Explanation:
   The whole system works on the CORBA architecture and if it fails the application will not work correctly.
2. Processor failures are benign.
   Explanation:
   We need to design a system which handles process crashes only.
3. No partition network failure.
   Explanation:
   We cannot distinguish between the broken network and server crash and this is an assumption in the project description.
   Explanation:
   There will be no UDP message loss during the subsystem communication is going on. Thus the test case for starting the election subsystem due to a UDP packet loss will not be handled.
4. The client request that comes in when a current leader crashes is not buffered.
   Explanation:
   The client request that appears during the time interval between a current leader crash and the election of a new leader will not appear again in the system, i.e. it will be lost.

Limitations:

When the leader crashes the processes start electing to choose an alternative leader. During this period every request, which is sent to group, will be failed, because the FE sends the request only to leader and during this time there is no leader in the group.

Future enhancements to the project limitations:

1. To solve this problem we can have a mechanism in the FE to send the request again at a later time when it receives notification that a new leader has been elected. So the system remains highly available.

Data Flow Description and Module Interaction Details:

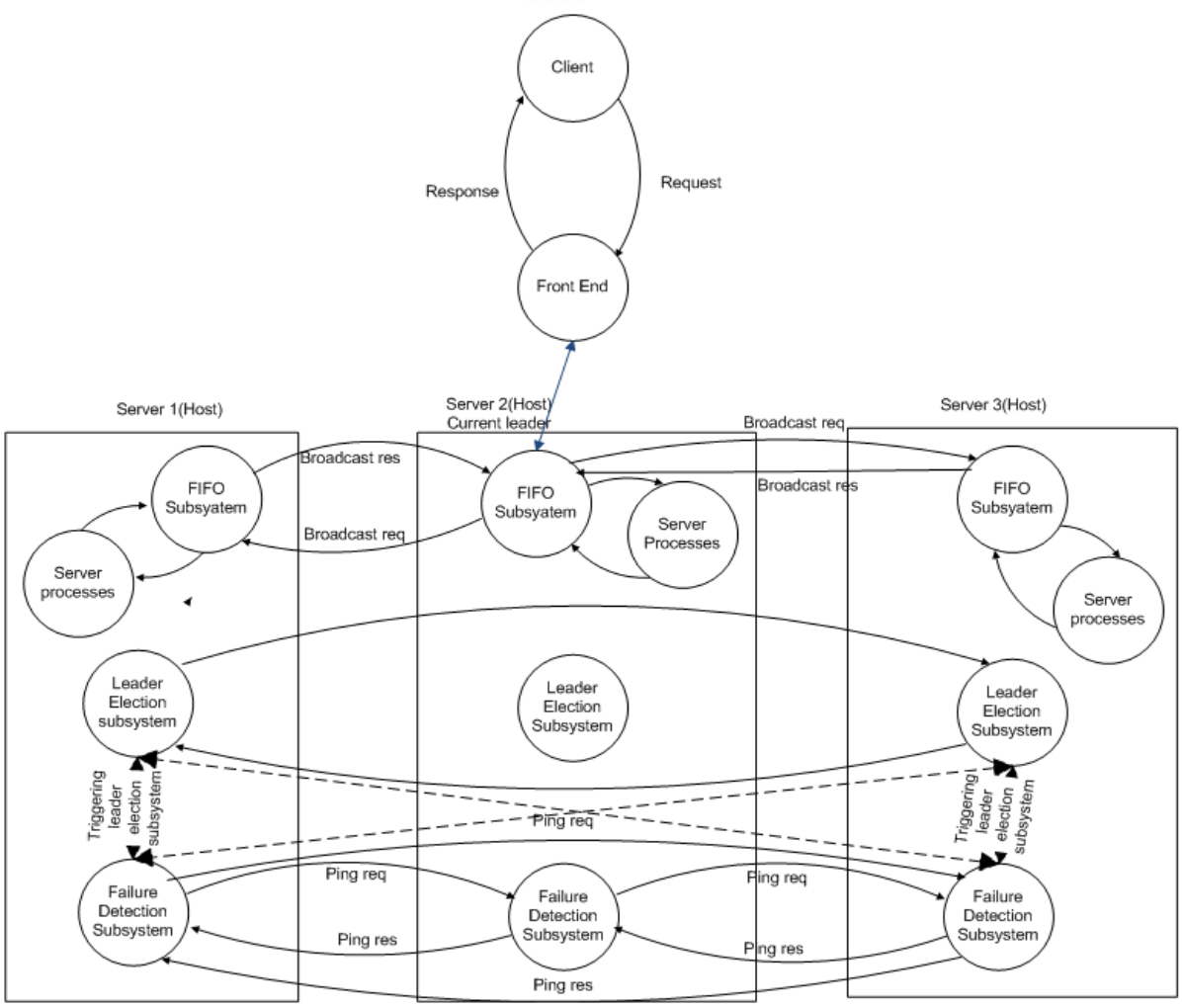


Figure: Data flow diagram

Client:

A client in this system represents an officer client who sends a request to the Front End and receives reply from the Front-End for the same request.

Front End:

A client sends a request to the server through the front end. Front end has its own repository where station server names, contact information are stored. In addition to this the front end will also be storing the information about who is the current leader by maintaining the processID in the repository. After receiving the response, the front end sends result to the Client.

Actually front end communicates with client directly and it is a bridge between client and group leader. It receives the requests from clients by CORBA invocation and sends the request to the appropriate group leader through UDP.

Front end functionalities:

1. FE receives concurrent requests from the client including the badgeID and other parameters required to fulfill the operations like createCRecord, createMRecord etc.
2. The front end will generate a sequenceID for each new request and add this sequenceID as a key and parameters as values in the request buffer. Front end will forward the client request to the group leader process (FIFO broadcast subsystem who will order the concurrent requests by prioritizing the request operation) in the group and wait for the response.
3. The front end will be having the knowledge of the current leader from its repository and in case of group leader process failure the front end will be notified with new leader information by the leader election subsystem.
4. The front end will be communicating over the reliable UDP communication with the leader process with the help of ACK or NAK piggybacked in the response message. With this ACK the front end will match the request to the exact response thus using the asynchronous Request-Reply protocol.

Replica: (Individual Host)

It consists of server process for each station server and subsystems.

Server Processes:

We have three server processes in each host, which receives a request from the group leader host, execute the request and return the reply back to the group leader. The server processes do the same operation for the request and are dedicated to one host. But they are replicated over three different hosts.

Subsystems Explanation:

Reliable FIFO broadcast:

The FIFO broadcast subsystem in group leader process will receive concurrent requests from client. After receiving the request FIFO ordering subsystem will transform concurrent request into order based on priority of operation and each request has unique sequenceID which is used to make the UDP communication between subsystem and server processes reliable. FIFO subsystem also uses the acknowledgements (ACKs) to ensure the reliability of requests.

The idea is to sequence the messages and delay delivery until we see the “right” sequence number. The receiver does not deliver a message from a sender unless the sequence number is equal to the next message sequence number expected.

If ACK is lost then we are sending the request again to the server and are matching the sequence ID again at the server. If the sequenceID is matched then we are sending only ACK again without executing the request again to reduce the inconsistency.

If ACK is received then it will proceed with the next request in the buffer.We are also maintaining the timeout for ACK. If an ACK is not received in this timeout period it will resend the request and performs the action with sequenceID.

We consider the implementation of a single instance (a protocol for broadcasting multiple messages is obtained in a straightforward way by aggregating as many instances as there are messages). Assume that the sender of every broadcast instance is known implicitly; server Pi executes the following steps :

-broadcast(m): // only when Pi is the sender

send the message (send; m) to itself

upon receiving a message (send, m):

if message m has not been r-delivered yet **then**

send the message (send, m) to all

r-deliver(m)

FIFO Broadcast from Reliable Broadcast

When multiple messages are reliably broadcast concurrently, Reliable Broadcast does not guarantee anything about the order in which the messages are delivered. FIFO broadcast, which guarantees that messages from the same sender are delivered in the same sequence as the sender broadcast them provides one of the simplest orderings; this does not affect messages from different senders.

A protocol for FIFO broadcast is a protocol for reliable broadcast defined in terms of two events f-broadcast and f-deliver that also satisfies:

FIFO Order: If a server f-broadcasts a message m before it f-broadcasts a message m', then no server f-delivers m' unless it has previously f-delivered m.
Algorithm: Given an implementation of re-liable broadcast, server Pi executes the following steps:

Initialization:
M ← Φ // set of received but not f-delivered messages

s ← 0 // Pi's sequence number

sj ← 0 (for every j ε [1, n]) // next sequence number to be f-delivered from Pj

f-broadcast(m): // only when Pi's the sender

r-broadcast the message (s, m)

s ← s + 1

upon r-deliver (s', m') with sender Pj:

M ← M *U* {(j, s', m')}

while ∃(j, t, m) ε M such that t = sj **do** f-deliver(m)

FIFO ordered communication (between subsystems and server processes):

The FIFO mechanism will be using the requestID received in the request message as the unique sequence number.
The group leader (sender) will broadcast the message to FIFO subsystems in other replicas in the group.
The receiver (replicas) receives the messages, orders them in the FIFO queue if needed (decided by looking at the sequential number). Sequential messages too far in the past or too far in front of the most recent processed sequential number are rejected.
The Server Node processes the messages in FIFO order, produces an answer, and gives it back to the subsystem.
The receiver keeps track of the answer and unicasts it to the sender.
The sender echoes the answer to the leader.

Leader Election Subsystem:

The bully algorithm is a method in distributed computing for dynamically electing a coordinator by process ID number. The process with the highest process ID number is selected as the coordinator.

Assumptions for Bully Algorithm

- The system is synchronous and uses timeout for identifying process failure
- Allow processes to crash during execution of algorithm
- Message delivery between processes should be reliable
- Prior information about other process id's must be known

Steps for Bully Algorithm:

The Bully Algorithm

BEGIN
1. A process *Pi* does not receive a response within T1 from the coordinator.
   1.1 *Pi* sends an *election* message to every process with higher priority number.
   1.2 *Pi* waits for *answer* messages for the interval T2.
   1.3 If no answer within T2, /*Pi is the new coordinator*/
      1.3.1 *Pi* sends a *coordinator* message to other processes with lower priority number.
      1.3.2 *Pi* stops its election procedure.
   1.4 If the *answer* messages are received,
      1.4.1 *Pi* begins the interval T3 to wait for a *coordinator* message.
   1.5 If the *coordinator* message is received,
      1.5.1 Admit the sender as the new coordinator.
      1.5.2 *Pi* stops its *election* procedure.
   1.6 If no *coordinator* message within T2, *Pi* restarts the election procedure.
2. A process *Pj*(i<j) may receive an *election* message from *Pi*.
   2.1 *Pj* sends an *answer* message to *Pi*.
   2.2 *Pj* begins its *election* procedure (step 1).
3. A process *Pj*(i>j) may receive a *coordinator* message from *Pi*.
   3.1 Admit *Pi* as the new coordinator.
   3.2 Stops the election procedure.
END

Some descriptions of this algorithm also call for any process joining a process group to begin an election process as though the leader has failed. This ensures that the leader of the group is always that process with greatest process ID and provides permits the joining process to discover the group leader.
The election subsystem will update the repository in front end about the new leader whenever it is elected.
In our implementation, each process will be a registered CORBA Object, and the IOR is necessarily different for distinct CORBA objects. As such the IOR of the process is suitable to use as the process ID in the algorithm, provided that each is aware of its IOR.

Failure Detection Subsystem:

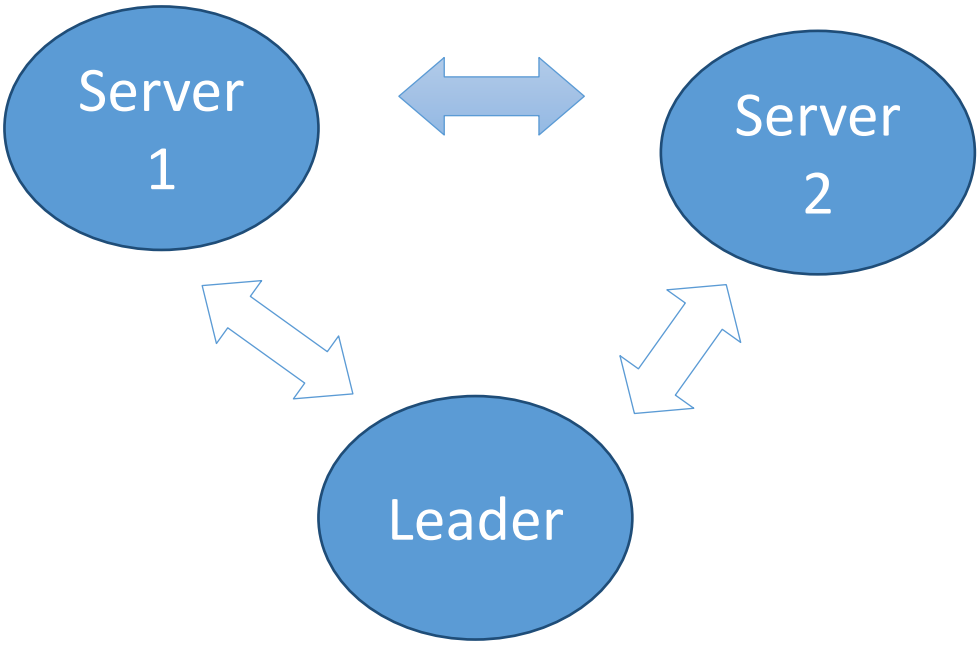
Goal: Find a failed server

Assumptions:

- There are three servers (Seevers that are responsible for a officer request)
- Only one failure happens in the system and it can not be front end and leader election subsystem.

Algorithm design:

Every server is responsible for testing one other server and if that server fails it should report the failure to other two servers.In every 300 miliseconds the node checks each other, this indicates the timeout for ping response. After timeout other servers will send ping request three times and if it fails to do so they declare about the failure of that node.



If the failed server is leader then any ther server node will start the leader election subsystem module.The checking process of each node is done through UDP connection with empty message and acknowledgement.

Test Cases:

Specific test cases:

To test this system when all the components are connected together we consider following scenarios:

Normal system behavior:

Multiple clients send request concurrently and no server stops working.
Example:

In this testing scenario we assume that three police officers are going to crate a criminal reocrd using his badgeID

Input: createCRecord(badgeID, firstName, lastName, description, status)

createCRecord("spvm1111", "Mary", "Kom", "Murderer", "Captured");
createCRecord("spvm2222", "steve", "Bond", "Theft", "OntheRun");
createCRecord("spvm3333", "Jason", "McDonald", "Pickpocketer", "Captured");
Expected output for above request

Criminal record created at SPVM station with recordID CR30000
Criminal record created at SPVM station with recordID CR30001
Criminal record created at SPVM station with recordID CR30002

One of the replicas fails:

1. Multiple clients send request concurrently.
2. One server which is not a leader stops working.
3. Leader will come to know that server is not responding.
4. This failed server is removed from the cluster and leader will not send any request to that node thereafter.

A group leader process fails:

1. Multiple clients send request concurrently.
2. Group leader stops working.
3. Any node will come to know that leader is not responding as they are communicating periodically.
4. Leader election subsystem selects a new leader using the Bully algorithm.
5. Leader election module will notify the front end about new leader.

FIFO Broadcast Subsystem Test:

We can have a test case for this module. This module is located in each server node:
We have two classes for this subsystem:

1. FIFO Sender (constructor of the class that receives the address and port number of process) on the leader Broadcast function in the FIFO Sender class receives the request, and sends it to remaining two server nodes.
2. FIFO receiver class on the server node receives the message and checks for the FIFO queue order and hand it over to server node process to execute the operation.
   (For example createCRecord() from the leader(by broadcast function).

Example:
remotePort[0] = 6000; (self)
remotePort[1] = 6001; (port number of server2)
remotePort[2] = 6002; (port number of server3)
aHost[0] = "localhost" (self)
aHost[1] = "localhost"
aHost[2] = "localhost"

Then we start receiving these requests at the server nodes (by listening to UDP Socket) and print the suitable output results to be sure that this module works correctly.

Failure detection subsystem:

For testing the failure detection algorithm we have to verify two scenarios

- A server is down, if it is unable to ping other two servers.
- Other two servers will wait for ping reply till less than 300 milliseconds, if any server reports an unexpected failure or not.

  After this timeout it will receive three requests for the same and if it fails to reply then other server will conclude about the failure of that node.

General Test Cases:

ID #	Test Scenario	Expected Result	Procedure
1.	Police officers perform createCRecord() operation from respective police station	FE will receive request and generate unique sequence ID and forward the request to group leader and server responsible for respective officer client request gets connected and creates Criminal Record (CR) using details provided by client and also generates unique recordID in each replica in the group. Also the log file gets generated on server as well as on client side.	Client runs the test case 1 using OfficerClient program. The response will be cached in to log files for respective clients and servers. If the client or server file doesn't exist then it will be created.
2.	Police officer perform createMRecord() operation from respective police station	Server responsible for respective officer client request gets connected and creates Missing Record (MR) using details provided by client and also generates unique recordID.	Client runs the test case 2 using OfficerClient program. Response from the front end will be logged in respective log files.
3.	Police officer perform getRecordCount() operation to	Server responsible for respective client request will return the	Client runs the test case 3 using OfficerClient program.

	get the record counts of all three police stations	record count (both CR and MR) and also display the count with time from other two police stations. At the end log files will be updated.	Client first gets the count from its respective server and concurrently sends UDP request to other two police stations for record counts and return count to client. Response from the front end will be logged in respective log files.
4.	Police officer perform editRecod() operation to edit the record status	Server responsible for respective client request will update the status of requested record and respective log files will be updated.	Client runs the test case 4 using OfficerClient program. Based on recordID and lastName server checks whether record exists or not, if exists then it checks for the status is same or not. It returns either error or update the record and updates the log file.
5.	Police officer perform transferRecord() operation to transfer record to another station	Server responsible for respective client request will transfer the requested record, delete the record and update respective log files will be updated.	Client runs the test case 5 using OfficerClient program. Client provide the information like recordID and remote server name to transfer the record. Response from the front end will be logged in respective log files.
6.	Test Concurrency by running multiple clients	Server responsible for each client should perform the operation and update the respective log files.	Client runs the test case 6 using OfficerClient program. It will start multiple threads for each officer performing random operation simultaneously. Response from the front end will be logged in respective log files for each request.

Conclusion:

Will be added after implementation process is complete

References:

Available http://en.wikipedia.org/wiki/Bully_algorithm

Available http://link.springer.com/chapter/10.1007%2F3-540-45801-8_58#page-4

Team Tasks

Name	Task	Student ID
Team	• Assuming that processor failures are benign (i.e. crash failures) and not Byzantine, design your highly available active replication scheme using process group replication and reliable group communication. • Design and implement client based on test scenarios and front end (FE) • Modify the individual implementations of the server replica from Assignment 2, integrate all the modules properly, deploy your application on a local area network, and test the correct operation of your application using properly designed test runs. You may simulate a process crash by killing that process while the application is running.	
Pratik BIdkar	Design and implement the group leader process which receives a request from the front end, FIFO broadcasts the request to all the server replicas in the group using UDP datagrams, receives the responses from the server replicas and sends a single response back to the front end as soon as possible.	6746349
Hirenkumar Tarsadiya	Design and implement a reliable FIFO broadcast subsystem over the unreliable UDP layer.	6742904
Sohan Argulwar	Design and implement a failure detection subsystem in which the processes in the group periodically check each other and remove a failed process from the group. If the group leader has failed, a new leader is elected using a distributed election subsystem.	6757871
Jaiminkumar Patel	Design and implement a distributed leader election subsystem (based on the bully algorithm) which will be called when the current leader has crashed to elect a new leader for the process group.	6775330