A Highly-Available Simplified Distributed Banking System

Project #1: High Availability through Primary-Backup Replication (Passive Replication)

Hossein Kassaei, 9043160

Prosenjit Sinha, 9190740

Chung Mak, 2393336

COMP 6231, Distributed Systems Design

Fall 2007

Instructor: R. Jayakumar

Tutor: Liu, Yan

1 Requirements Specification and Analysis

1.1 Hight-Availabilty Requirements of the System

In this project high-availability is achieved through passive or Primary-backup replication. The Backup or "warm standby" is a Branch Server that is running in the background (normally on a different machine), receiving requests from the primary server to update its state and hence ready to jump in if the primary server fails. Thus, when the primary server receives a request from a client which will change its state, it performs the request, sends the state update request to the backup server and sends the reply back to the client (while the state update may not yet have been done at the backup server). Sending update only for requests that change state help reduce the number of unnecessary messages in the system and thus enhance performance. Since the primary and backup servers are usually on a local area network, they communicate using the unreliable UDP protocol. However, the communication between them should be reliable and FIFO.

Such a technique requires four main modules to meet the requirements:

1.1.1 Primary Server

Primary server receives requests from clients as CORBA invocations, performs the request, sends the state update request to the backup server using UDP data grams if necessary, and sends the response back to the client (while the state update may not yet have been done at the backup server). When the primary notices that the backup does not respond within a reasonable time, it assumes that the backup has failed and informs the Branch Monitor so that a new backup server can be created and initialized.

1.1.2 Backup Server

Backup server receives state update requests from the primary as UDP data grams and performs the state update. If the backup server does not receive any request from the primary for a reasonable time, it sends a request to the primary to check if the latter is working. If the primary server does not reply in a reasonable time, the backup server assumes that the primary has failed and takes over by configuring itself as the primary so that it can receive and handle all client requests from that point onwards; and also informs the Branch Monitor of the switchover so that the latter can create and initialize another backup server.

1.1.3 FIFO Communication Subsystem

FIFO subsystem provides reliable, ordered delivery of messages between primary and backup over the unreliable UDP layer.

1.1.4 Branch Monitor

Branch Monitor maintains the application fault tolerant. It Monitor initializes the primary and backup servers at the beginning, creates and initializes a backup server when the primary fails (and the original backup server takes over as the primary), and creates and initializes a backup server when the original backup server fails.

1.2 Functional Requirements of the Banking System

This simplified Distributed Banking System comprises a set of branches. Associated with each account, there is a unique seven digit number. The first three digits indicate the branch and the next four digits indicate the account within a specified branch.
Associated with each account, there is an ***account number*** and a ***balance***. There might be other information like the account holder's profile (name, address, phone number(s), etc.) that have been deemed unnecessary for the purposes of this assignment. But, the application is designed in such a way that they can be included in the records without affecting the fundamental components of the application.

1.2.1 Use case model

Customers (from anywhere), may invoke **four** operations on the server.

- *Deposit*(*acnt*, *amt*): Cause the balance of account number *acnt* to be increased by the specified amount *amt*. Returns the new account balance.

- *Withdraw*(*acnt*, *amt*): Cause the balance of account number *acnt* to be decreased by specified amount *amt*. Returns the new (possibly negative) account balance.

- *Balance*(*acnt*): Returns the balance of account number *acnt*.

- *Transfer*(*src_acnt*, *dest_acnt*, *amt*): Causes the balance of account number *src_acnt* to be decreased by *amt* and the balance of account number *dest_acnt* to be increased by *amt*. Returns the new account balance of account number *src_acnt*. Below is a use case diagram illustrating the interaction of a user with the system:

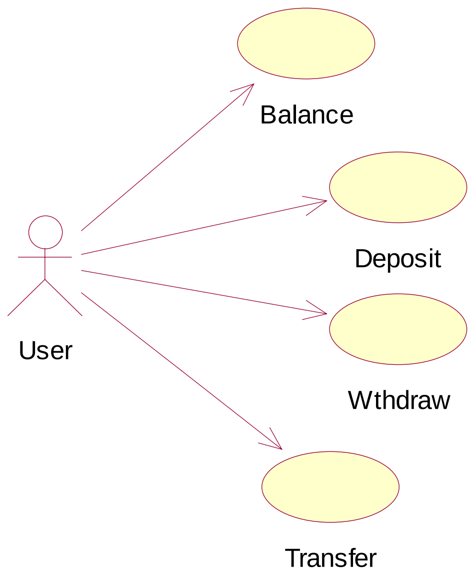


Figure 1 – Use Case Diagram

The four following scenarios describe the details of use-system interaction:

1.2.1.1 Scenario to get the balance

Precondition(s):

1. Viewing the application’s main menu

Trigger: Selecting “view balance” from the menu

Procedure:

1. System prompts user to enter the account number
2. User enters the account number
3. System retrieves the information
4. System displays the current balance

Postconiditon(s):

Viewing the balance

Exception(s):

2a. User enters incorrect account number
2a1. System displays an error message
2a2. User may repeat the procedure
3a. Database error
3a1. System displays an error message

1.2.1.2 Scenario to deposit

Precondition(s):

1. Viewing the application’s main menu

Trigger: Selecting “deposit” from the menu

Procedure:

1. System prompts user to enter the account number
2. User enters the account number
3. System prompts user to enter the amount to deposit
4. User enters the amount to deposit
5. System saves the new account balance
6. System displays the new balance

Postcondition(s):

1. Viewing the new balance of the account
2. Database updated

Exception(s):

2. User enters incorrect account number
2a1. System displays an error message
2a2. User may repeat the procedure
5a.Database error
5a1. System displays an error message

1.2.1.3 Scenario to withdraw

Precondition(s):

1. Viewing the application’s main menu

Trigger: Selecting “withdraw” from the menu

Procedure:

1. System prompts user to enter the account number
2. User enters the account number
3. System prompts user to enter the amount to deposit
4. User enters the amount to withdraw
5. System saves the new account balance
6. System displays the new balance

Postcondition(s):

1. Viewing the new balance of the account
2. Database updated

Exception(s):

2a. User enters incorrect account number
2a1. System displays an error message
2a2. User may repeat the procedure
5a.Database error
5a1. System displays an error message

1.2.1.4 Scenario to transfer

Precondition(s):

1. Viewing the application’s main menu

Trigger: Selecting “transfer” from the menu

Procedure:

1. System prompts user to enter the source account number
2. User enter the source account number
3. System prompts user to enter the destination account number
4. User enters the destination account number
5. System prompts user to enter the amount to transfer
6. User enters the amount to transfer
7. System saves the new accounts' balances
8. System displays the new balance of the source account

Postcondition(s):

1. Viewing the new balance of the source account
2. Database updated

Exception(s):

2a. User enters incorrect source account
2a1. System displays an error message
2a2. User may repeat the procedure
4a. User enters incorrect destination account
4a1. System displays an error message
4a2. User may repeat the procedure
6a. The amount specified exceeds the source account's balance
6a1. System displays an error message
6a2. User may repeat the procedure
7a. Database error
7a1. System displays an error message

1.3 Analysis of considerations and constraints

1.3.1 Concurrency and data integrity

Problem definition: As a distributed application, the available functions are naturally expected to be carried out concurrently by multiple users. For example, while a user is transferring money from a given account to another account, another user might deposit or withdraw an amount of money to/from the same account. This leads to data corruption. Therefore, taking the appropriate measures to provide conflict-free concurrent access to functions is crucial.

Solution: One simple way is to lock the individual functions, namely balance, deposit, withdraw and transfer. However, this approach severely affects the performance of the system making concurrent accesses very slow, particularly in the presence of a large number of clients interacting with the system at a given time.
A better solution is to synchronize the critical sections of the code that try to access and probably alter shared resources.

In order to achieve this goal, the two basic functions of deposit and withdraw have been synchronized. The following code snippets from the “AccountInfo” class illustrates the mechanism:

```
public class AccountInfo {
  String name;
  int accountNo;
  float balance;


....
  public float depositAmount(float damount) {
    synchronized(this){
      this.balance=this.balance+damount;
    }
    return this.balance;
  }
  public float withdrawAmount(float wamount){
    synchronized(this){
      this.balance=this.balance-wamount;
    }
    return this.balance;
  }
...
}
```

As illustrated, using different locks for different critical sections helps enhance the system’s performance considerably.

1.3.2 Deadlock prevention

Problem definition: This problem can be caused by using the “transfer function” in the system. Suppose user 1 intends to transfer some money from account X in the first branch server to account Y in the second branch server and simultaneously user 2 intends to transfer some money in the opposite direction between the same two accounts. As seen in the previous section, in order to provide conflict-free concurrent access, account objects are locked whenever a deposit or withdrawal is made. Locking the account objects may lead to a dead-lock situation in this scenario because each client is waiting for the other one to release the lock before being able to deposit the specified amount to the destination account and neither does so! As an extension, this scenario can be applied to three or more clients intending to transfer money from and to the same accounts on different branch servers. Therefore, resolving this problem is deemed crucial to the correct performance of the system.

Solution: The solution to this problem is using the “action, roll back” scheme.
In order to prevent the clients from waiting for each other indefinitely, each branch server first locks the account object and then withdraws the amount specified form the source account. Once withdrawal has been completed, the lock is released. The next step involves sending a message to the branch server holding the destination account and waiting for the reply. If the reply message indicates success, then the operation has

been completed successfully; otherwise, the server needs to roll back by locking the account object again and depositing back the money withdrawn in the first step. The following pseudo-code illustrates the solution programmatically:

```
1. Lock the account object
2. Withdraw the amount specified
3. Unlock the account object
4. Send a message to the branch server holding the destination account
5. Wait for the reply
6. If the reply indicates failure
        6a. Lock the account object again
        6b. Deposit the amount withdrawn in step 2
        6c. Unlock the account object
```

The above algorithm has been implemented in the “transfer method” of the server object and is as the following snippet:

```
synchronized (accounts[i])
      {
        accounts[i].decreaseAmount(amt);
      }
        //compose the message to send

      String message = dest_acnt + "$" + Float.toString(amt);

      RemoteBranch rb = new RemoteBranch();

      //send the UDP packet and get the reply

      result = rb.connect(branchServerName, 9999, message);

       //roll back if not successful, otherwise carry on
      if ( result.compareTo("Successful") != 0)
      {
         synchronized (accounts[i])
          {
            accounts[i].raiseAmount(amt);
          }
         ...
       }
```

1.3.3 Crucial Assumptions

1. The correct and reliable performance of this system is seriously degraded in the absence of the crucial assumption that a daemon should always run on any machine that hosts a server object or branch monitor and that this daemon does not crash. So, the current design choices have been made holding this critical assumption. It is also notable for people who are going to run and maintain the system.

2. Failures tolerated in this system are all “crash failures”, i.e. a server object or the Branch Monitor stop running altogether. Otherwise, they function accurately when up and running. In other words, “Byzantine failures”, whether malicious or non-malicious, are not tolerated in this system. To provide tolerance as such, more replicas and a “majority vote” subsystem should be integrated into the system.

3. As the system follows “passive replication standard”, the service is temporarily unavailable if the primary server crashes. In fact, during the “switch-over” phase (when Backup detects that Primary has crashed and reconfigures itself as the new Primary) the service is not available and clients receive an error message. However, it should be noted that, given the fact that the system normally runs on a LAN, this switch-over time can be set to around 10 seconds which is an acceptable performance constraint.

4. Since concurrency and performance is very important in distributed systems, all the operations are carried out in server memory and only occasionally serialized to a file. Initially, the server loads all the information form a file into the memory and proceeds with the memory content from that point onwards.

2 Design and Implementation

2.1 Domain Model

In order to bring about high availability, two systems are absolutely essential: a branch monitor and a daemon. These two systems cooperate with the built-in failure detection subsystem in the servers to provide high availability. The figure below shows the high level relation between these three modules:

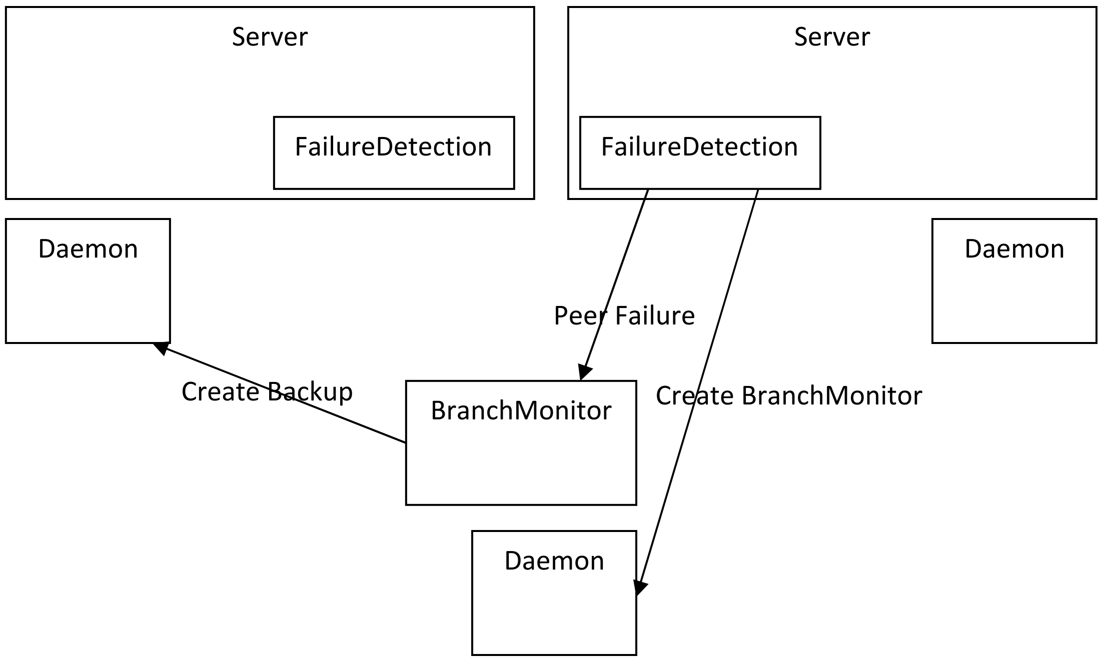


Figure 2 – Domain Model

So, the next two sections describe the design of the daemon and Branch Monitor. Note that daemon should run on every machine that hosts the server objects or Branch Monitor.

2.1.1 Daemon

Daemon is constantly listening on a specific port and handles incoming requests. The UDPReceiver class listens on a designated port and receives the incoming requests. Once a request is received, it is passed to the requesthandler class to take the appropriate action. The incoming requests are the following:

1. Build_Server_Rrequest: It's a request that the branchMonitor sends to the remote daemon. The action to take is to create a new server on the local machine.

2. Build_BranchMonitor_Request: It's a request that a server sends to the local daemon. The action to take is to create a new BranchMonitor on the local machine.

3. GetBMHostName_Request: It's a request that a server sends to the local daemon thereby asking the BranchMonitor hostname. The action to take is to send the BranchMonitor's hostname that has been saved from the UDP datagrams received from the BranchMonitor to the server.

Below is the class diagram of the daemon.

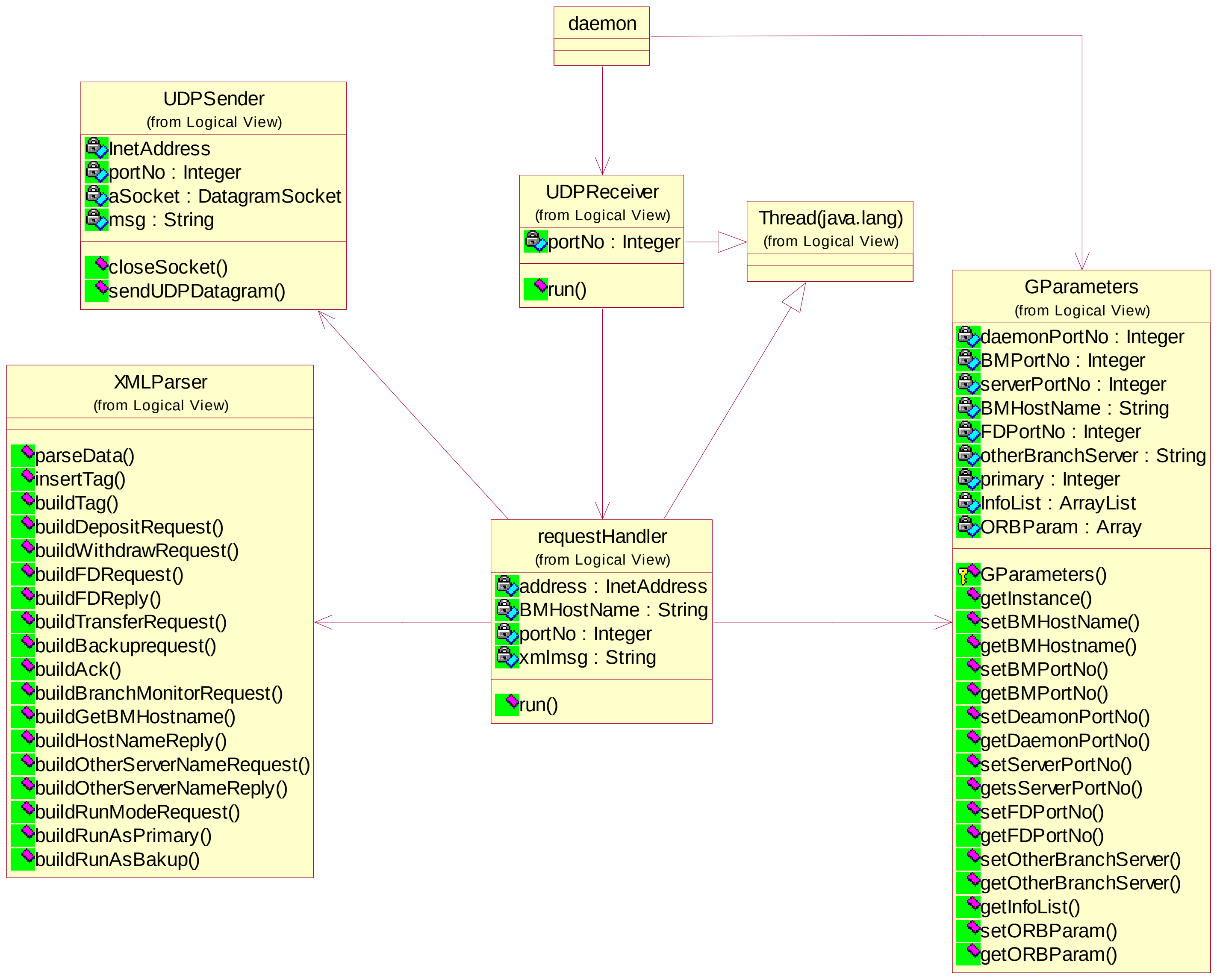


Figure 3 – Daemon Class Diagram

2.1.2 Branch Monitor

Branch Monitor is the component that plays an important role in providing high availability in this replicated system. Upon start-up, it creates and initializes two server objects on two hosts. It receives the hostname of the two machines from the administrator of the system. It then continues to monitor the server objects and handles the incoming requests. In doing so, it listens on a specific port and handles the messages that are received at that port. When it receives a message from a server object indicating that its peer had crashed, it recreates a backup to replace the failed server. It also helps each server object discover its peer when the object is created. Finally, it dictates the object its run mode, i.e. as primary or backup. Basically, it tells one of the servers to configure itself as primary at the very beginning and from that point onwards, dictates all other server objects to run a backup. To recap, below is the list of the requests that a Branch Monitor handles:

1. Build_ Backup_ Request: This is a request that a server object sends to the Branch Monitor when it detects a failure has occurred. Using the IP address of the UDP datagram received, Branch Monitor decides which server has sent the message and which one has crashed. It then sends a message to the remote daemon which lies at the failed server host thereby asking it to recreate another server object. When the server object is recreated, it asks the Branch Monitor about its peer and Branch Monitor gives it back the information necessary to communicate with its peer. It also tells the newly created object to run as backup.

2. OtherServerName_Request: This is a request that a server object sends to the Branch Monitor when it is going through the "discovery" phase. At this phase, the server gathers the information necessary to operate as expected. Through this message it asks the Branch Monitor the host name of the machine on which the other server object is running. The Branch Monitor responds to this request by sending back the host name of the other server object.

3. RunMode_Request: This is another request that a server object sends to the Branch Monitor in the discovery phase. Once a server object finds out where Branch Monitor is (by asking the local daemon), and also discovers the other server object, it asks the Branch Monitor for the Run Mode. Branch Monitor responds to this request by sending back the run mode (0 for primary, and 1 for backup).

2.1.2.1 Handling the Branch Monitor's failure

In a robust and highly available distributed system, no single point of failure is tolerated. Therefore, Branch Monitor's failure should be taken care of in an appropriate way. In order to handle Branch Monitor's failure, there are two possible solutions. The first one is to design and implement the Branch Monitor as a replicated system, i.e. provide a backup for the Branch Monitor such that it can jump in should the primary Branch Monitor fails. The other solution is to have the server objects detect such a failure and take the appropriate measure. In this project we chose the second approach. The main reason why the second approach was adopted is the presence of an already existing "failure detection module" in the system. As stated earlier, the failure detection subsystem which is built into the server object can be delegated to take care of Branch

Monitor as well. So there is no need to develop a new replicated system from scratch. The built-in failure detection system works as follows:
When it detects a failure in the other server object, it sends a request to the Branch Monitor asking it to create a new backup to replace the failed server object. When Branch Monitor creates a backup, it sends an acknowledgement to the server object that detected the failure. If the server that detects the failure does not receive and acknowledgement from the Branch Monitor in a timely manner, it concludes that the Branch Monitor has also failed. At this point, it sends a request to the remote daemon on the Branch Monitor's host ordering it to create a new Branch Monitor. When a new Branch Monitor is created, it resends the build_backup_request to the newly created Branch Monitor.
Below is the class diagram for the Branch Monitor:

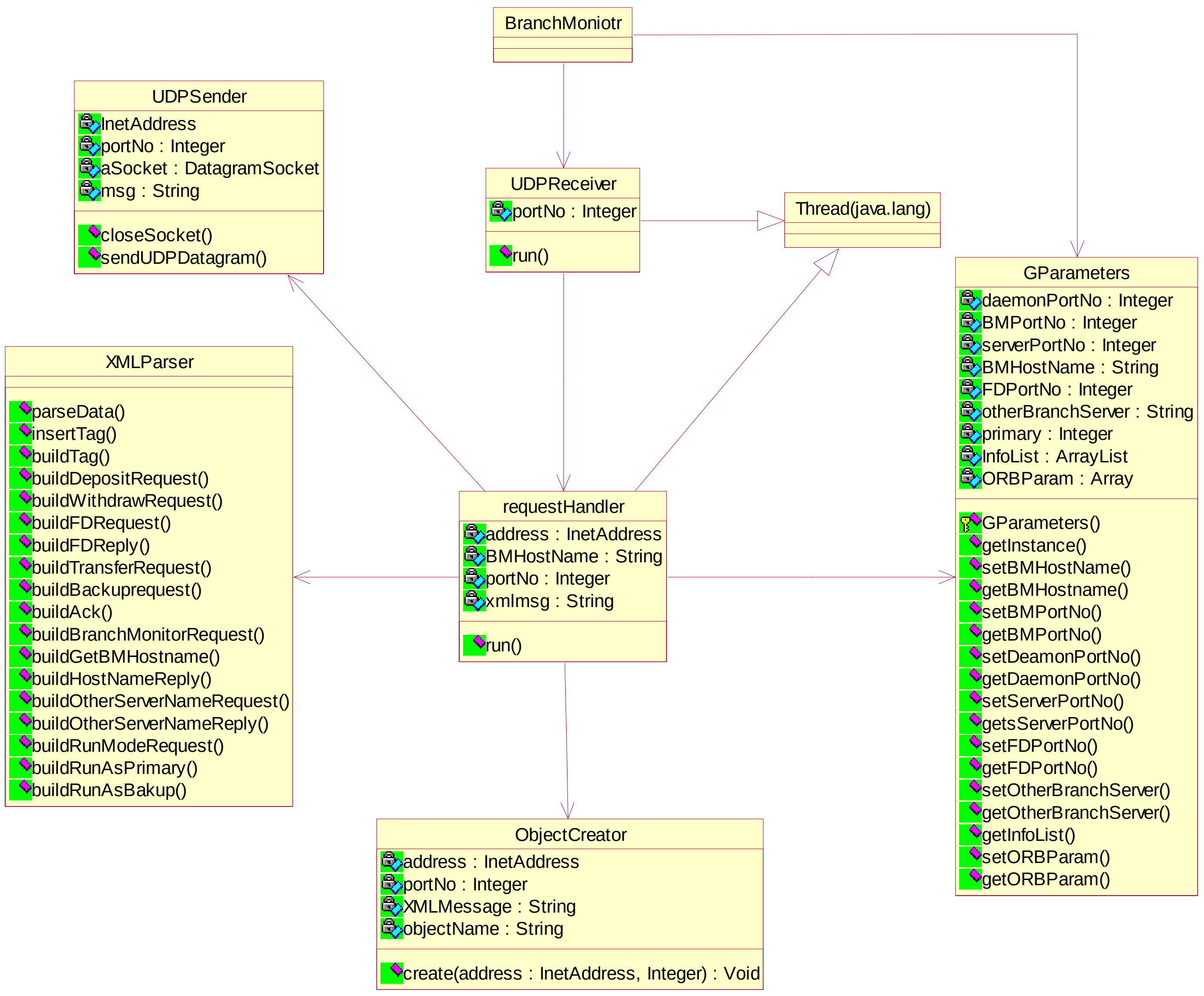


Figure 4 – BranchMonitor Class Diagram

Note that this class diagram mainly illustrates the participating classes. The detailed and complete relationships between classes have not been shown as they unnecessarily complicate the diagram and might make it difficult to comprehend.

2.1.3 Server

Obviously, the server is the core in this system. Not only should the server handle clients' requests, it must also handle additional tasks such as sending and receiving update requests for those client requests that "change server state", as well as detecting failures and reporting them to the Branch Monitor or the daemon. As stated earlier, for performance reasons, all server-server communications are carried out on top of UDP layer, so an additional mechanism should provide the correctness and reliability required for update requests. Such requirements imply that a server object needs to be much more complicated than a usual server that assumes no failures and crashes and totally relies on the underlying network. We subdivided this server into four main subsystems:

2.1.3.1 Discovery subsystem

Discovery subsystem is the module that starts running immediately after the UDP receiver starts its job. Once it is started, it carries out crucial tasks that resolve parameters absolutely essential for the server to communicate with the other components in the system. The first parameter to resolve is the Branch Monitor's hostname. In order to discover the Branch Monitor, a message is sent to the local daemon asking it to send back the host name of the machine on which Branch Monitor is running. The local daemon has already saved this information by extracting the IP address of the Branch Monitor from the UDP datagram request it initially sent to daemon asking it to create the server object. Once the Branch Monitor hostname is resolved, the Discovery subsystem sends a message to the Branch Monitor asking it to send the host name of the other server. Once the host name of the other server is resolved, Discovery subsystem sends a message to the Branch Monitor asking it to determine the Run Mode. It should be noted that these three steps should be taken sequentially because they are inherently inter-related. Below is the snippet from the Discovery class that resolves these three parameters:

```
//1. send a message to the local daemon to ask the hostname of BranchMonitor
      BranchMonitorName = parser.BuildGetBMHostname();
      InetAddress localhostAddress = InetAddress.getLocalHost();
      sender = new UDPSender(localhostAddress, gp.getDaemonPortNo(), BranchMonitorName);
      sender.SendUDPDatagram();
      //wait until BMHostname is resolved and set in Global Parameters
      while (gp.getBMHostName().isEmpty()){
         System.out.println("Resolving BranchMonitor's hostname...");
         Thread.sleep(1000);
      }
      System.out.println("BranchMonitor's hostname is:"+ gp.getBMHostName());
```

```
//2. Now contact the BranchMonitor to ask the hostname of the other server Object
OtherServerName = parser.BuildOtherServerNameRequest();
InetAddress BMaddress = InetAddress.getByName(gp.getBMHostName());
sender = new UDPSender(BMaddress, gp.getBranchMonitorPortNo(), OtherServerName);
sender.SendUDPDatagram();
//wait until OtherServerName is resolved and set in global Parameters
while (gp.getOtherHostName().isEmpty()){
  System.out.println("Resolving other server's hostname...");
  Thread.sleep(1000);
}
System.out.println("The other server's hostname is: "+gp.getOtherHostName());
//3. send a message to the BranchMonitor to ask the RunMode
RunMode = parser.BuildRunModeRequest();
sender = new UDPSender(BMaddress, 8888, RunMode);
sender.SendUDPDatagram();
//wait until BranchMonitor send a reply telling the runmode
while (gp.getMode() == -1){
  System.out.println("Resolving Function Mode...");
  Thread.sleep(2000);
}
```

Once these three parameters are resolved, Discovery takes the last step. At this point, based on the "RunMode" dictated by the Branch Monitor, it either starts the server object as a Primary or Backup server. If it is to set up the server as primary, it registers the "DBSServant " with the "CORBA naming service" using the "EnablePrimary" class. Otherwise, it simply instantiates and run the DBSServant without registering it with the CORBA naming service. The snippet below demonstrates how the logic is implemented:

```
if (gp.getMode()==0){
        //if primary, then run EP
        EnablePrimary ep = new EnablePrimary();
        ep.start();
      }
      else //Run as Backup
      {
        DBSServant dbs = DBSServant.instance();
      }
```

The sequence diagram below illustrates how a server resolves the parameters it needs to communicate with the other components and the order in which it runs the different modules:

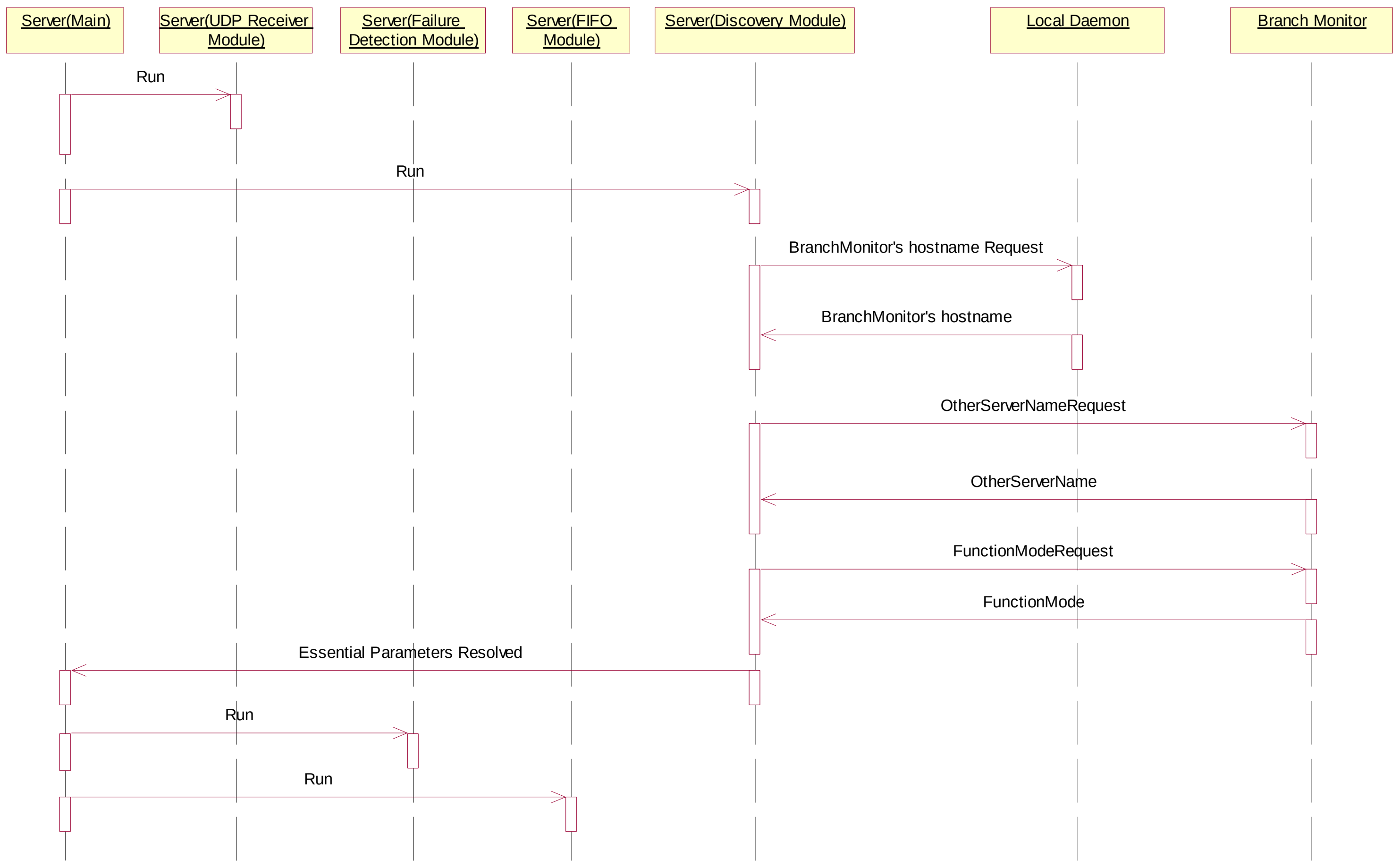


Figure 5 –Sequence Diagram (Illustrates how a server is initially configured)

2.1.3.2 Client Requests Handling Subsystem

This system is pretty much the same as the DBSServant from the second assignment. When the interface written in CORBA IDL is compiled using "idlj", six classes are automatically produced that help link the server and client and provide the services to the client. What DBSServant basically does is that it implements the interface compiled and kept in "DBSInterfaceOperations" class. Using the DBSInterfaceHelper class, the server object is registered with the CORBA naming service and its remote methods are advertised to clients. The client in turn uses the DBSInterfaceHelper and DBSInterfaceStub, finds the server object that is registered with the ORB and obtains a handle to it. From this point onwards, all remote invocations are treated as local calls. The diagram below shows the details of client-server interaction through the interface.

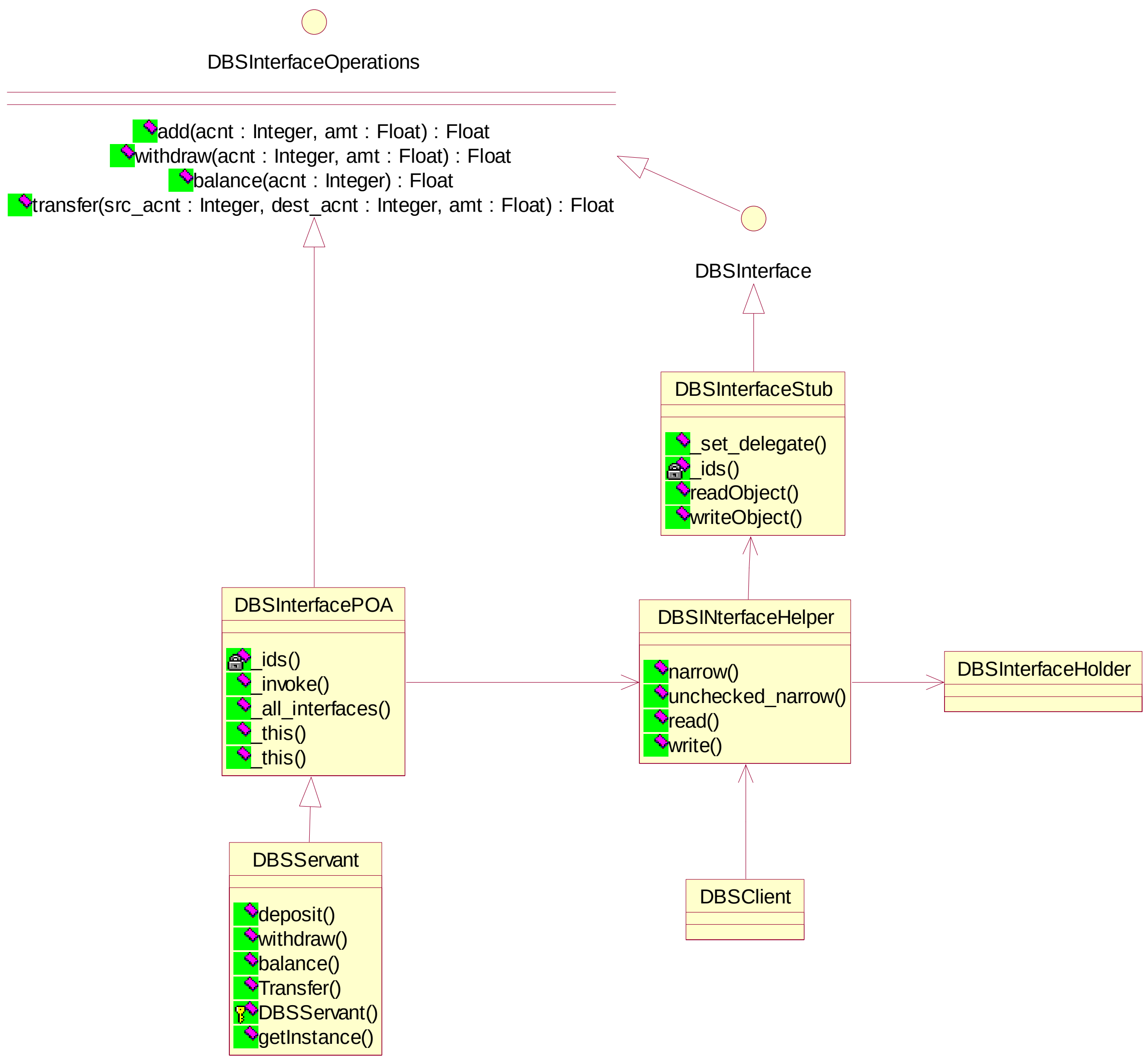


Figure 6 – DBSServant Class Diagram

The snippet below shows how the "EnablePrimary" class registers the server object with the CORBA naming service thereby providing the clients with the information necessary to invoke the remote methods.

```
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(gp.getORBParam(), null);

        //get reference to rootpoa & activate the POAManager
```

```
POA rootpa = (POA) orb.resolve_initial_references("RootPOA");
rootpa.the_POAManager().activate();

//create servant and register it with the ORB
//Also gets the database path from the admin to initialize the servant

 DBSServant dbs = DBSServant.instance();
dbs.setORB(orb);
//get object reference from the servant

org.omg.CORBA.Object ref = rootpa.servant_to_reference(dbs);

//and cast the reference to a CORBA reference

DBSInterface dref = DBSInterfaceHelper.narrow(ref);

//get the root naming context

org.omg.CORBA.Object objref = orb.resolve_initial_references("NameService");

//Use NamingContextExt, which is part of the interoperable Naming
//service (INS) specification.

NamingContextExt ncRef = NamingContextExtHelper.narrow(objref);

//bind the object reference in naming

String name = "DBSInterface";
NameComponent path[] = ncRef.to_name(name);

ncRef.rebind(path,dref);

System.out.println("DBS Server ready and waiting...");

//wait for invocations
orb.run();
```

2.1.3.3 FIFO Subsystem

FIFO subsystem is responsible for maintaining reliable delivery with FIFO ordering in a synchronous environment over the unreliable UDP protocol. Whenever Primary server does any operation which results a state change, it use the FIFO to communicate with the Backup server in order to update its state to remain in a identical consistent state. Central idea is to use sending and receiving sequence number to maintain integrity and ordering of the message. In one sense FIFO use TCPs sliding window protocol with stop and wait method. Whole FIFO subsystem is implemented in one class. It has dual functionality aggregated in one subsystem. It can work either at primary mode or at backup mode, depending on the role of the server. Also it can automatically switch from primary to backup and vice versa. As per the statement: when it works as primary its functionality is to receive command from the associated servant, send them to the backup server and receive acknowledgement maintaining FIFO ordering. On the other hand when it works as backup it pivotal job is to receive the state change request from the primary server and send him acknowledgement maintain reliability and ordering.

Working mechanism is:

FIFO maintains two global LinkedBlockingQueue. Whenever Servant receives any operation from the client that results state change, it creates corresponding message and put it into the rQueue (LinkedBlockingQueue). Also all network message received by the UDPReceiver class of type FIFO are put on the queue. ProcessMessage class check all the message and if the type is FIFO it checks the purpose of the message. Depending of the purpose of the message it takes the decision to which queue to put the message. If it is an Ack message than it put it into the aQueue and if it is a request then it put it into the rQueue queue. Now getting and setting of messages are specified; here I want to describe a scenario which will simplify to understand the functionality of the FIFO subsystem.

Lets server get a deposit request from the client: as it is a state change operation after serving the client servant made a BuildFIFODepositRequest() message and put it into the rQueue. FIFO, which is running as a thread is polling the queue using its built in BlockingQueue.Take() method. FIFO gets that message and check the content of the <from > tag. If it is servant then it knows that it comes from the own server (this checking is crucial, will describe later). Then it changes the value of <from> tag from “servant” to “network”, insert the corresponding sequence number of the message into the message, and sends it to the other server. UDPReceiver of the other server receive this message and as its purpose is “deposit “, it put that message into the rQueue. FIFO receives that message and checks its sequence number. From now on two case may happen: in one case sequence number is the number that it expected, that means a correct message or a wrong sequence number, indication wrong, misplaced message. If the message is correct one then it should do all the necessary operation. If the message is wrong one it will send the Ack of previous correct message that it received. Sender, on the other side receives that Ack, analyze the received sequence number and work accordingly. FIFO always keeps the record of the last sent message for future use.

2.1.3.4 Failure Detection Subsystem

The concept of failure detection in this project is to detect if a crash failure occurs in the primary server, backup server or branch monitor. We accomplish this by having the primary server ping Backup synchronously at time intervals to detect if it is alive. By sending UDP datagrams and waiting for acknowledgements, they can detect each other's failure by waiting for timeouts. The timeouts of course have to be large enough so that network latency is taken into account and network congestion is avoided. Upon failure detection, the branch monitor is notified of this event and proper action can be taken like restarting the server. If the branch monitor does not respond in a timely manner, then who ever detected the failure will restart the branch monitor. This implementation can be illustrated from the following piece of code. This code is used for both primary and backup servers.

```
String msg=parser.BuildFDRequest();              //the message sent to Back at regular intervals
String replyFD=parser.BuildFDRereply();          //the acknowledgment sent back by the backup
while(true){
        //if not primary break
        if(gp.getMode()==1)
          break;
       //send failure request
        sender.SendUDPDatagram(failureRequest, OtherServer, gp.getFDPortNo());

        try{
        packet = new DatagramPacket(receiveBuf, receiveBuf.length);
        socket.setSoTimeout(10000);
        socket.receive(packet);
        Thread.sleep(4000);
        }catch(SocketTimeoutException e){
          //call the branch monitor
          //to do
          sender.SendUDPDatagram(backupFailure, BMaddress, gp.getBranchMonitorPortNo());
          try {//receive the ack from BranchMonitor
            socket.setSoTimeout(4000);
            socket.receive(packet);
            //wait for the other server to come up
            Thread.sleep(10000);
          }catch(SocketTimeoutException ex) {
            //call the BMHostname daemon to create a BranchMonitor
            sender.SendUDPDatagram(branchMonitorFailure, BMaddress, gp.getDaemonPortNo());
            try {
              Thread.sleep(4000);
              //resend the backup request to the new branchmonitor
              sender.SendUDPDatagram(backupFailure, BMaddress, gp.getBranchMonitorPortNo());
              try {//receive the ack from BranchMonitor
                socket.setSoTimeout(4000);
                socket.receive(packet);
                //wait for the other server to come up
```

```
                        Thread.sleep(5000);
                    }catch (Exception exp){}
//….other catches….
```

And the other server object (Backup) sends acknowledgements to these messages. If Primary does not receive an acknowledgement within a specific period of time, it notifies the Branch Monitor. Likewise, if Backup does not receive any "FailureDetection" messages from the Primary for a considerable amount of time, it concludes that the primary has failed and notifies the Branch Monitor. As illustrated in the code above and explained earlier in the Branch Monitor section, if the Branch Monitor also crashes, Failure Detection module notices the problem and requests the remote daemon to create another Branch Monitor. The figure below shows a general class diagram of the server package.

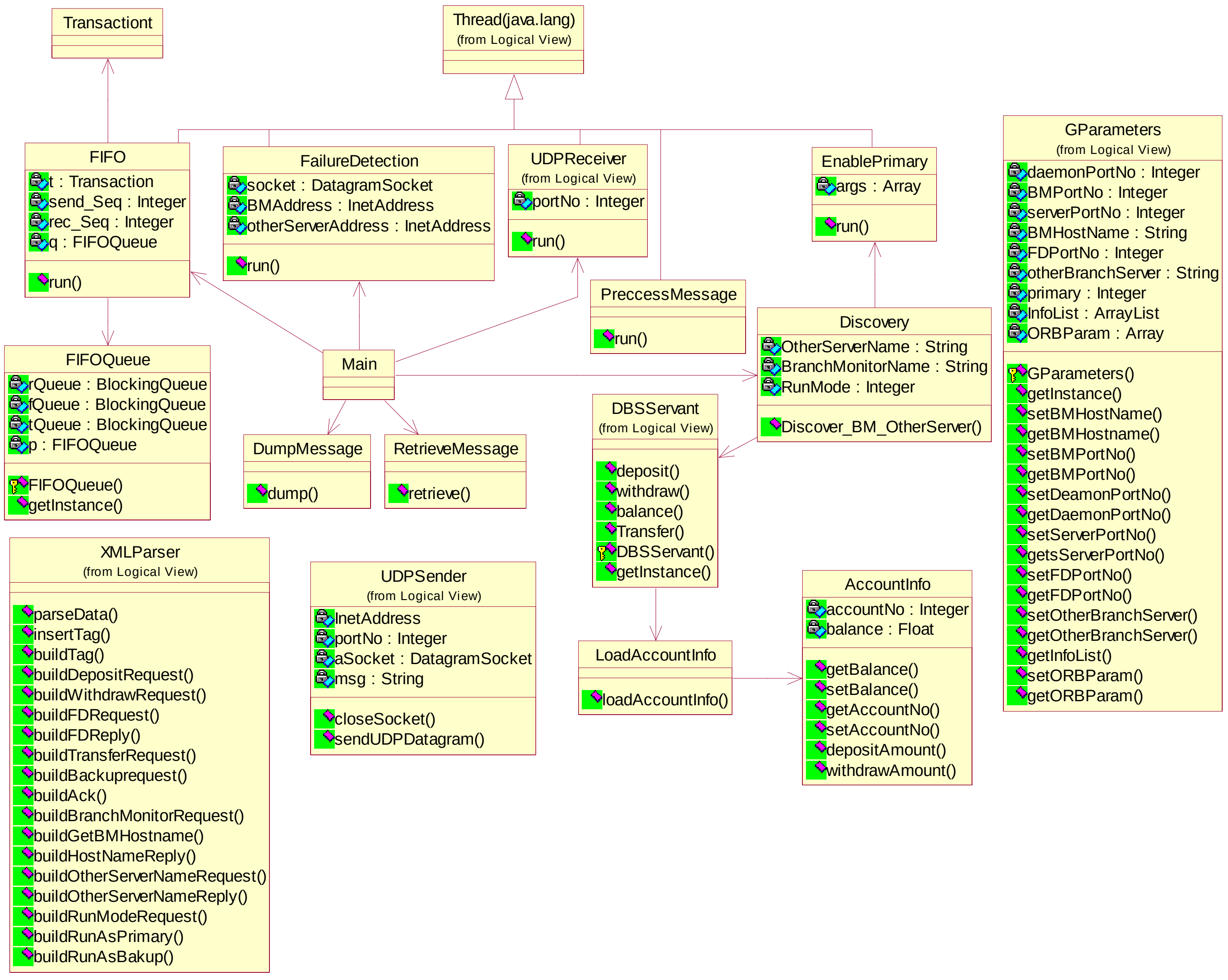


Figure 7 – Server Class Diagram

3 Fault Tolerance and Data Integrity

In this section, it is congruent to have an analysis of how fault-tolerance and data integrity are maintained in the system. As mentioned earlier the current triangular design of the Server, Branch Monitor, and Daemon renders the system very robust and resilient. At any given time, the system can survive if just one of the server objects (either primary or backup) is alive. This object can recreate a new Branch Monitor and the Branch Monitor in turn creates a new Backup. However, it should be noted again that all this requires the assumption that Daemon never fails and is up and running on all machines. In this section, we analyse the behaviour of the system in more details in terms of fault-tolerance and recovery and also maintaining data integrity.

3.1 Fault Tolerance

Let's consider the different crash failure scenarios and see how the system recovers from those crash failures.

3.1.1 Backup Failure

Let see how the system recovers from this crash. As mentioned earlier, Primary and Backup are continually contacting each other to see if there is a crash failure in the other one. If the Primary crashes, Backup detects the crash failure after around 10 seconds. Timing parameters can be set to meet the desirable performance. For example, if the likelihood of crash failures is considerable, then timing parameters can be shrunk so as to detect failures more quickly (within 4-5 seconds). However, this comes at the expense of extra message overhead in the system. On the other hand, if crash failures are not very often and happen only rarely, the timing parameters can be stretched (for example up to 30 seconds). This results in less communication overhead and higher performance in the network.

Once the Primary detects the crash Failure, it requests the Branch Monitor to recreate a new Backup to replace the failed one. Branch Monitor sends a message to the primary as an acknowledgement when it recreates the Backup successfully and Primary proceeds. The sequence diagram below illustrates this scenario:

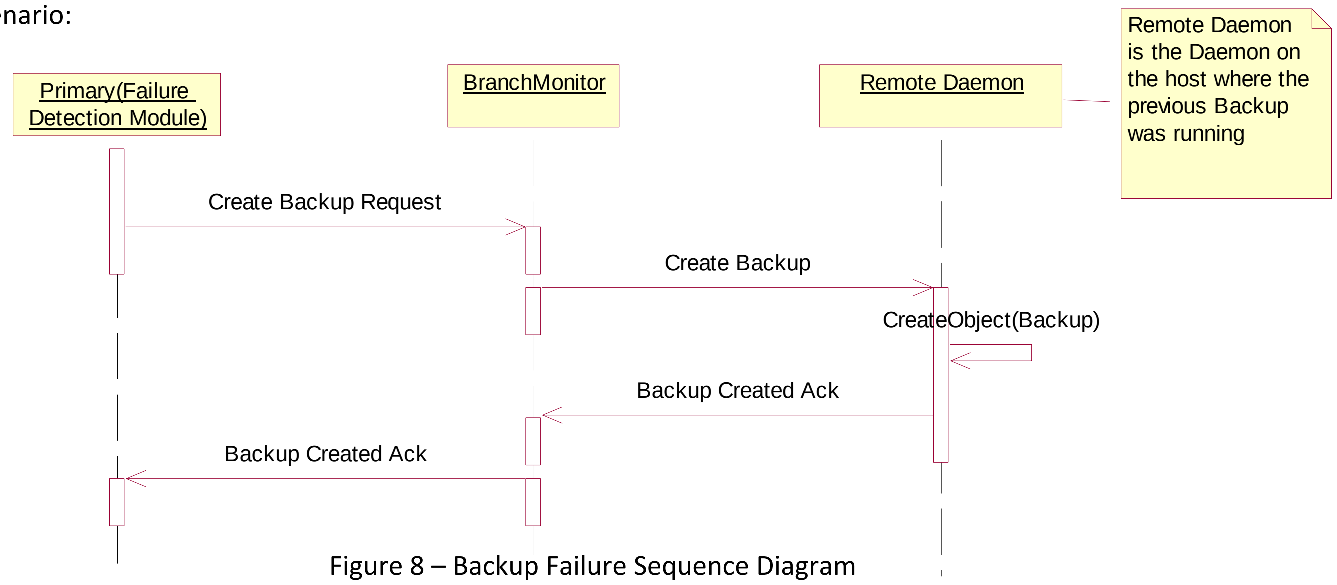


Figure 8 – Backup Failure Sequence Diagram

3.1.2 Primary Failure

In case of Primary failure, Backup pretty much goes through the same procedure except that it first reconfigures itself as primary to minimize the system down-time. Once it takes over as the new primary, it sends a request to the Branch Monitor to build a new backup to replace the failed server. The sequence diagram below illustrates the procedure:

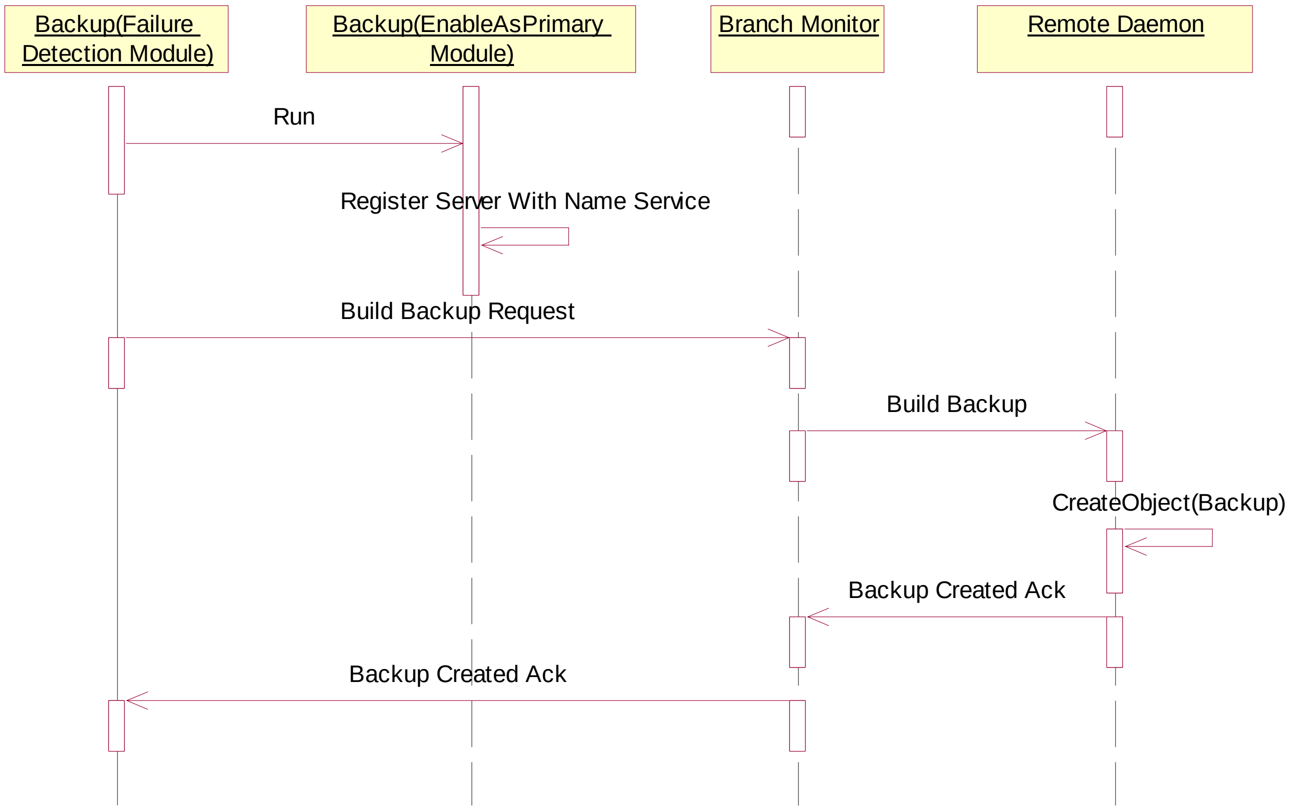


Figure 9 – Primary Failure Sequence Diagram

3.1.3 Primary/Backup and Branch Monitor Failure

What if Branch Monitor also fails? In this case, we can make use of the acknowledgement that Branch Monitor sends to the (new) primary server. As the system is running in a synchronous environment with bounded transmission times, the server that requests the Branch Monitor, can time out if it does not receive the acknowledgement form the Branch Monitor after a certain amount of time has elapsed. At this point, it concludes that the Branch Monitor has also failed. As a consequence, it sends the remote daemon a request to create a new Branch Monitor. When the new Branch Monitor is successfully created, it resends the request to the new Branch Monitor asking it to create a backup. The sequence diagram below illustrates the procedure:

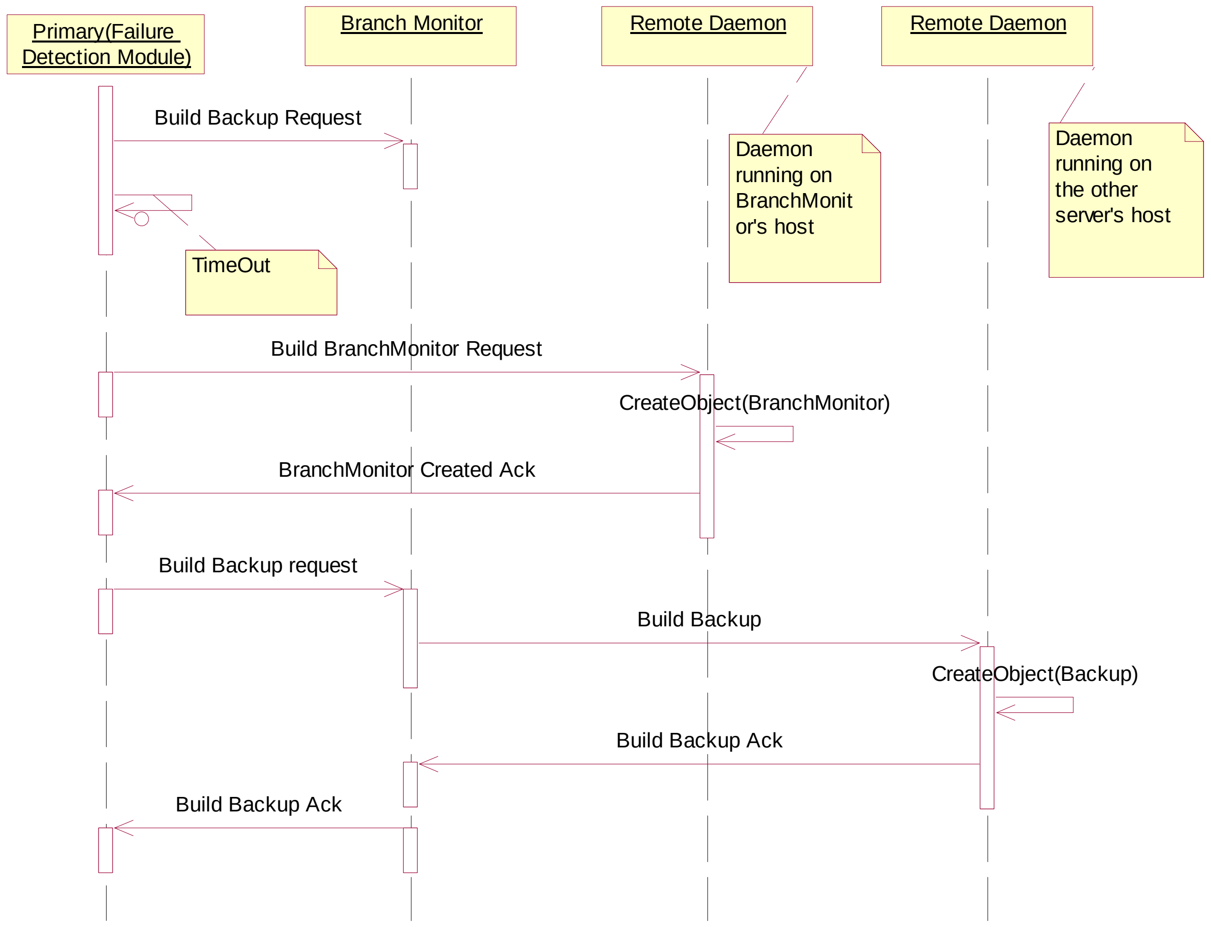

Primary(Failure Detection Module)
Branch Monitor
Remote Daemon
Remote Daemon
Daemon running on BranchMonitor's host
Daemon running on the other server's host
Build Backup Request
TimeOut
Build BranchMonitor Request
CreateObject(BranchMonitor)
BranchMonitor Created Ack
Build Backup request
Build Backup
CreateObject(Backup)
Build Backup Ack
Build Backup Ack

Figure 10 – Backup and Branch Monitor Failure Sequence Diagram

3.2 Data Integrity

Another crucial aspect of the system is data integrity. In order to maintain data integrity, the backup server should always maintain the most update state so that in case Primary fails, it jumps in with the latest state, hence play the role of a "warm standby". In this system, data integrity is guaranteed through two subsystems: FIFO subsystem and "state_transfer" subsystem. In this section we look at each in more detail.

3.2.1 FIFO Update Requests

Whenever Primary handles a client request, it sends an update message through FIFO to the Backup asking it to update its state. FIFO subsystem provides reliable, ordered delivery of update messages over the unreliable UDP. So, the updates are executed in the Backup in the exact same order as Primary. It should be noted that update messages are sent only for those client requests that change the state (deposit, withdraw). Transfer is in fact a combination of these two operations.

3.2.1.1 Atomicity of Transfer Operation

In the case of "transfer", update request is sent to the Backup ***only if*** a message indicating success is received from the other Branch Server. Otherwise, no update request is sent to Backup and Primary rolls back. This helps maintain the "atomicity" of transfer operations. Using this mechanism, there is no need for the Backup to roll back in case Transfer fails.

3.2.2 State Transfer

When a new Backup server is created by the Branch Monitor, it should acquire the latest state of the Primary. Therefore, when the new Backup is created by the Branch Monitor and configures itself, it first requests the Primary to send the latest state. Upon receipt of this request, Primary dumps its memory state to an XML message and sends it to Backup. At the other side, Backup parses this message and loads it to its memory. Once this latest state is loaded, Backup proceeds with update requests and constantly updates its state to reflect the current state of the Primary at any given time. The sequence diagram below illustrates the procedure:

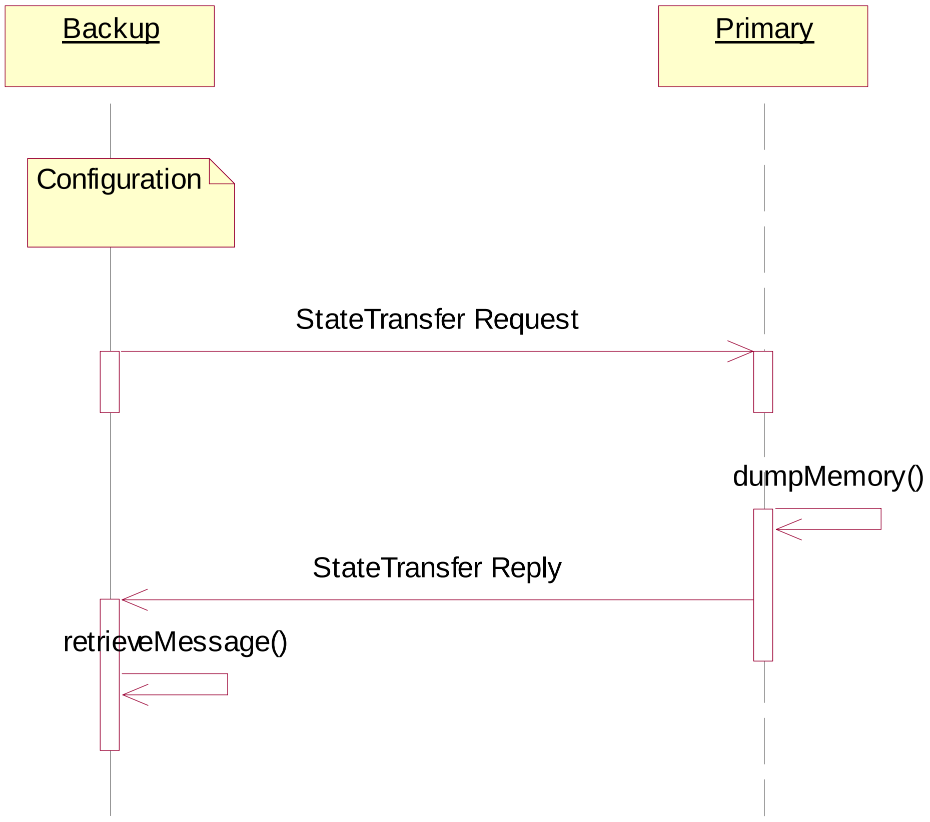
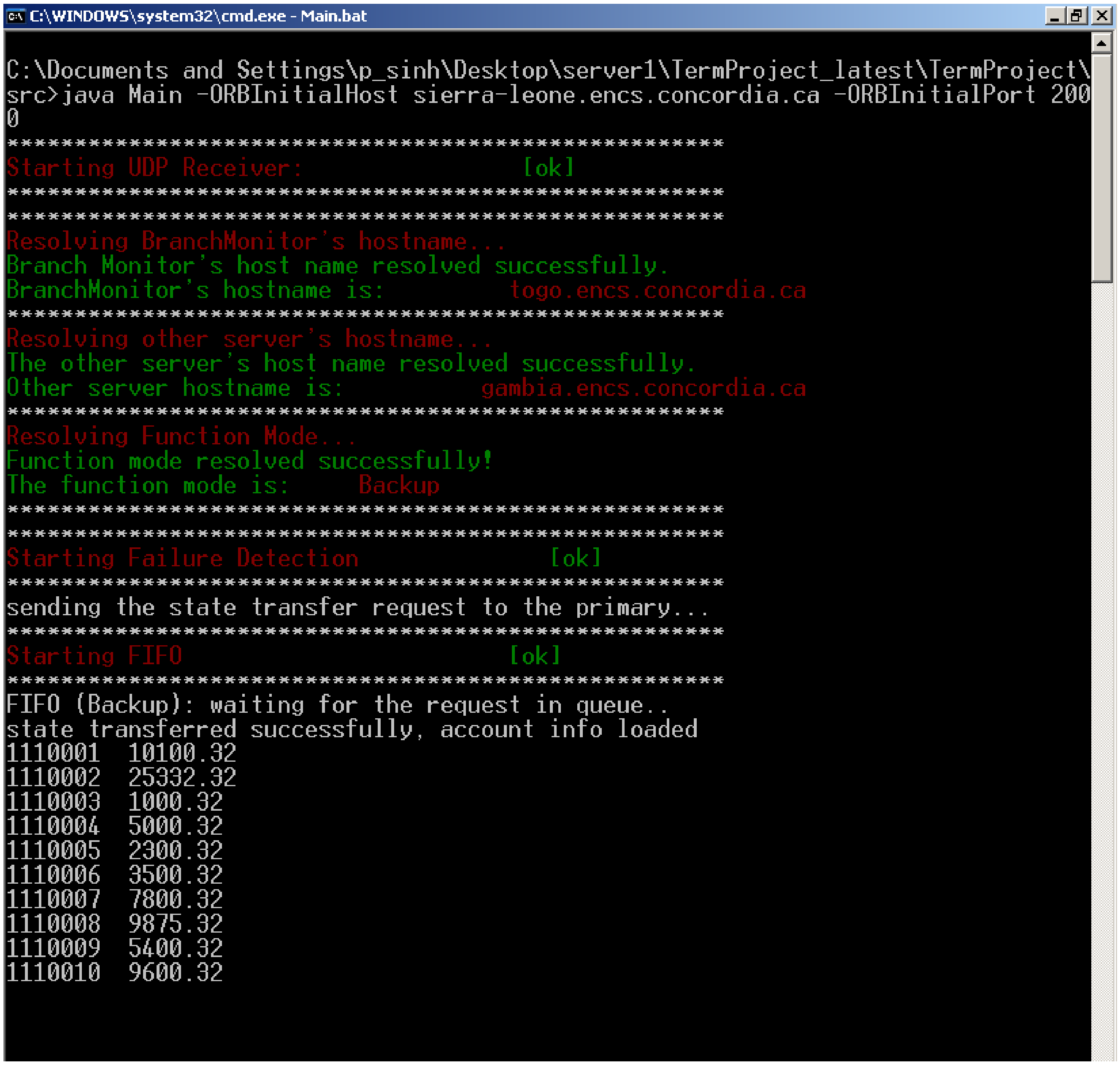


Figure 11 – State Transfer Sequence Diagram

The screenshot below shows how all this is implemented along with the latest update sent to the Backup from Primary:



C:\WINDOWS\system32\cmd.exe - Main.bat

```
C:\Documents and Settings\p_sinh\Desktop\server1\TermProject_latest\TermProject\
src>java Main -ORBInitialHost sierra-leone.encs.concordia.ca -ORBInitialPort 200
0
*****************************************************
Starting UDP Receiver:                      [ok]
*****************************************************
*****************************************************
Resolving BranchMonitor's hostname...
Branch Monitor's host name resolved successfully.
BranchMonitor's hostname is:          togo.encs.concordia.ca
*****************************************************
Resolving other server's hostname...
The other server's host name resolved successfully.
Other server hostname is:          gambia.encs.concordia.ca
*****************************************************
Resolving Function Mode...
Function mode resolved successfully!
The function mode is:     Backup
*****************************************************
*****************************************************
Starting Failure Detection                    [ok]
*****************************************************
sending the state transfer request to the primary...
*****************************************************
Starting FIFO                                [ok]
*****************************************************
FIFO (Backup): waiting for the request in queue..
state transferred successfully, account info loaded
1110001  10100.32
1110002  25332.32
1110003  1000.32
1110004  5000.32
1110005  2300.32
1110006  3500.32
1110007  7800.32
1110008  9875.32
1110009  5400.32
1110010  9600.32
```

Figure 12 – State Transfer Screenshot

4 Test Cases and Screenshots

4.1 Test Case 1

Name: **Deadlock** and data **consistency** check:

Goal:

The goal of the test is to verify that when to account transfer their balance mutually there should not be any deadlock to access the shared data. At the same time to check that data integrity and consistency is preserved, that is when we transfer balance mutually, the accounts are updated and in parallel warm backups are also updated.

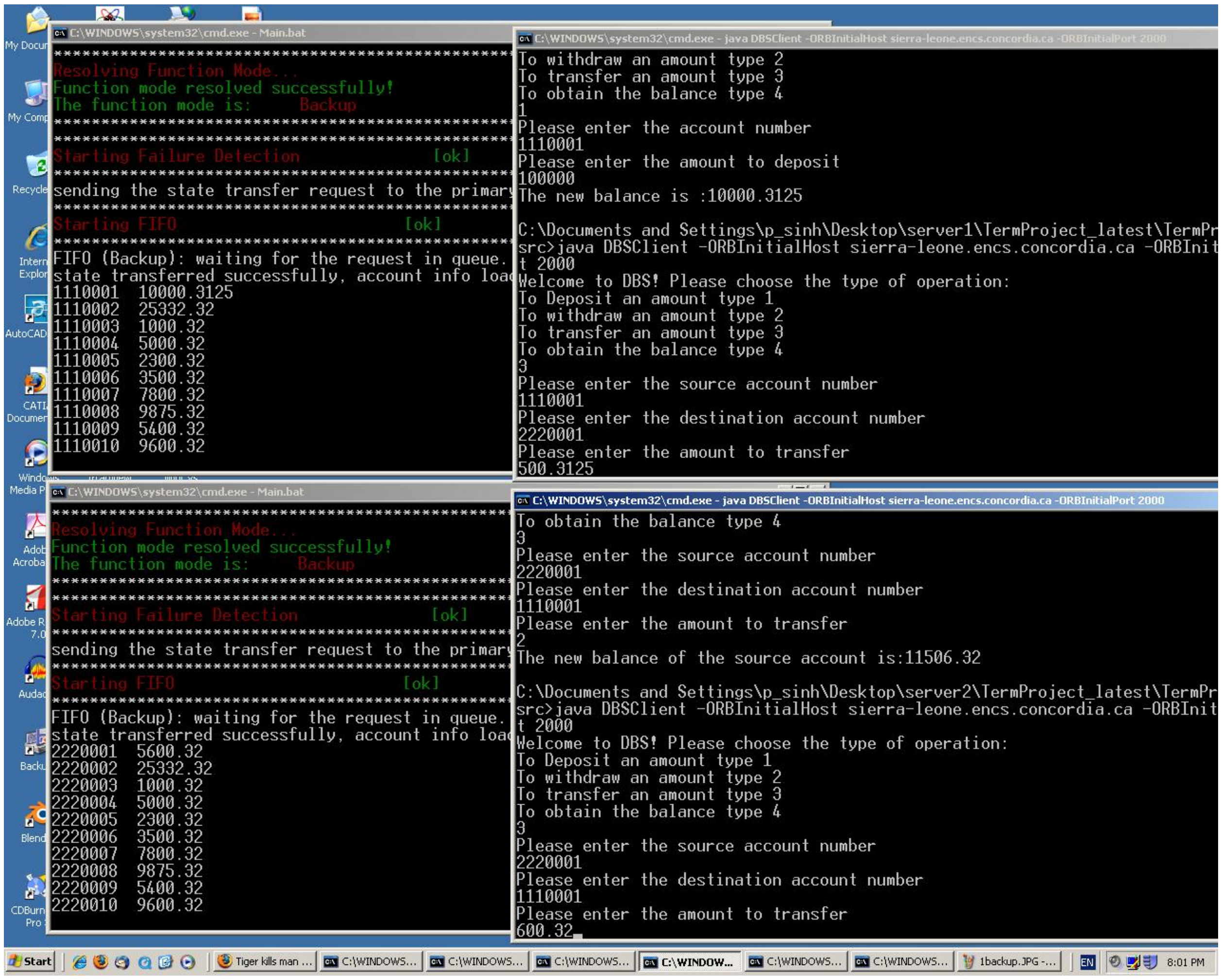
Analysis:

To perform this test case we maintain some specific condition. Here to simulate the scenario we extend our transfer operation using Thread.sleep() method. Every transfer is taking at least 10 seconds amount of time.

Expected results:

There is no deadlock. They can transfer their amount properly. If we see in the figure 1 below, we can see that the balance of account 1110001 is 10000.3125 and the balance of account 2220001 is 5600.32. then we initiate the transfer (showed in the right side of the figure 1). We are transferring 5000.3125 $from 1110001 to 2220001 and at the same time we are transferring 600.32$ from 2220001 to 1110001. Then we kill both of these primary servers. As a result two new backup servers wake up with the correct state transferred by the primary servers, which is shown in figure 2. Balances of account no 1110001 and 2220001 are consistent after that transfer and crash and wake up.

SCREEN SHOTS:



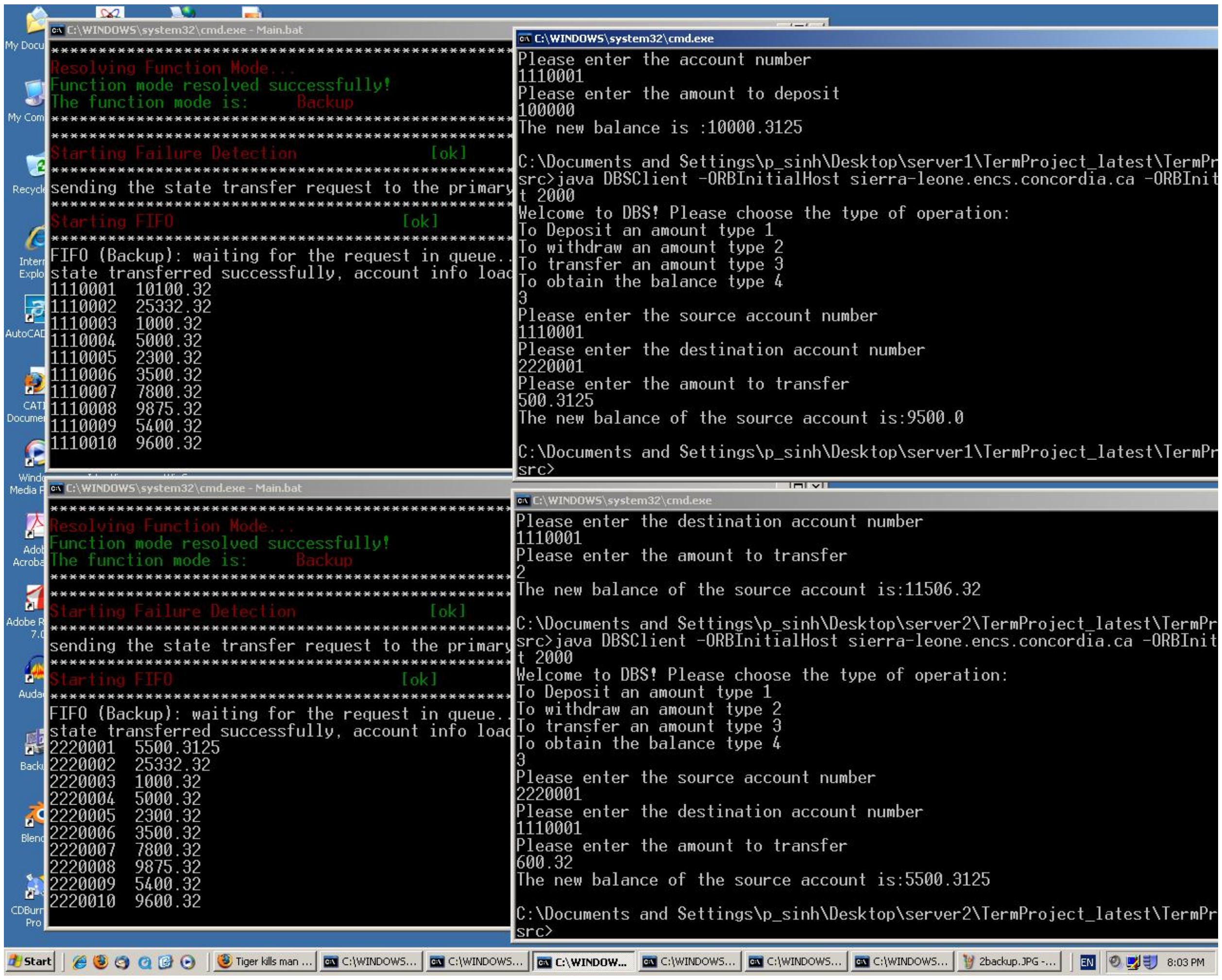
```
C:\WINDOWS\system32\cmd.exe - Main.bat
*************************************************
Resolving Function Mode...
Function mode resolved successfully!
The function mode is:     Backup
*************************************************
*************************************************
Starting Failure Detection                 [ok]
*************************************************
sending the state transfer request to the primary
*************************************************
Starting FIFO                           [ok]
*************************************************
FIFO (Backup): waiting for the request in queue.
state transferred successfully, account info loa
1110001  10000.3125
1110002  25332.32
1110003  1000.32
1110004  5000.32
1110005  2300.32
1110006  3500.32
1110007  7800.32
1110008  9875.32
1110009  5400.32
1110010  9600.32
```

```
C:\WINDOWS\system32\cmd.exe - java DBSClient -ORBInitialHost sierra-leone.encs.concordia.ca -ORBInitialPort 2000
To withdraw an amount type 2
To transfer an amount type 3
To obtain the balance type 4
1
Please enter the account number
1110001
Please enter the amount to deposit
100000
The new balance is :10000.3125

C:\Documents and Settings\p_sinh\Desktop\server1\TermProject_latest\TermPr
src>java DBSClient -ORBInitialHost sierra-leone.encs.concordia.ca -ORBInit
t 2000
Welcome to DBS! Please choose the type of operation:
To Deposit an amount type 1
To withdraw an amount type 2
To transfer an amount type 3
To obtain the balance type 4
3
Please enter the source account number
1110001
Please enter the destination account number
2220001
Please enter the amount to transfer
500.3125
```

```
C:\WINDOWS\system32\cmd.exe - Main.bat
*************************************************
Resolving Function Mode...
Function mode resolved successfully!
The function mode is:     Backup
*************************************************
*************************************************
Starting Failure Detection                 [ok]
*************************************************
sending the state transfer request to the primary
*************************************************
Starting FIFO                           [ok]
*************************************************
FIFO (Backup): waiting for the request in queue.
state transferred successfully, account info loa
2220001  5600.32
2220002  25332.32
2220003  1000.32
2220004  5000.32
2220005  2300.32
2220006  3500.32
2220007  7800.32
2220008  9875.32
2220009  5400.32
2220010  9600.32
```

```
C:\WINDOWS\system32\cmd.exe - java DBSClient -ORBInitialHost sierra-leone.encs.concordia.ca -ORBInitialPort 2000
To obtain the balance type 4
3
Please enter the source account number
2220001
Please enter the destination account number
1110001
Please enter the amount to transfer
2
The new balance of the source account is:11506.32

C:\Documents and Settings\p_sinh\Desktop\server2\TermProject_latest\TermPr
src>java DBSClient -ORBInitialHost sierra-leone.encs.concordia.ca -ORBInit
t 2000
Welcome to DBS! Please choose the type of operation:
To Deposit an amount type 1
To withdraw an amount type 2
To transfer an amount type 3
To obtain the balance type 4
3
Please enter the source account number
2220001
Please enter the destination account number
1110001
Please enter the amount to transfer
600.32
```

Figure13: balances before simultaneous transfer operation using Thread.sleep().


C:\WINDOWS\system32\cmd.exe - Main.bat
C:\WINDOWS\system32\cmd.exe
Start
8:03 PM

```
**************************************************
Resolving Function Mode...
Function mode resolved successfully!
The function mode is:     Backup
**************************************************
**************************************************
Starting Failure Detection                 [ok]
**************************************************
sending the state transfer request to the primary
**************************************************
Starting FIFO                          [ok]
**************************************************
FIFO (Backup): waiting for the request in queue..
state transferred successfully, account info load
1110001  10100.32
1110002  25332.32
1110003  1000.32
1110004  5000.32
1110005  2300.32
1110006  3500.32
1110007  7800.32
1110008  9875.32
1110009  5400.32
1110010  9600.32
```

```
Please enter the account number
1110001
Please enter the amount to deposit
100000
The new balance is :10000.3125

C:\Documents and Settings\p_sinh\Desktop\server1\TermProject_latest\TermPr
src>java DBSClient -ORBInitialHost sierra-leone.encs.concordia.ca -ORBInit
t 2000
Welcome to DBS! Please choose the type of operation:
To Deposit an amount type 1
To withdraw an amount type 2
To transfer an amount type 3
To obtain the balance type 4
3
Please enter the source account number
1110001
Please enter the destination account number
2220001
Please enter the amount to transfer
500.3125
The new balance of the source account is:9500.0

C:\Documents and Settings\p_sinh\Desktop\server1\TermProject_latest\TermPr
src>
```

```
**************************************************
Resolving Function Mode...
Function mode resolved successfully!
The function mode is:     Backup
**************************************************
**************************************************
Starting Failure Detection                 [ok]
**************************************************
sending the state transfer request to the primary
**************************************************
Starting FIFO                          [ok]
**************************************************
FIFO (Backup): waiting for the request in queue..
state transferred successfully, account info load
2220001  5500.3125
2220002  25332.32
2220003  1000.32
2220004  5000.32
2220005  2300.32
2220006  3500.32
2220007  7800.32
2220008  9875.32
2220009  5400.32
2220010  9600.32
```

```
Please enter the destination account number
1110001
Please enter the amount to transfer
2
The new balance of the source account is:11506.32

C:\Documents and Settings\p_sinh\Desktop\server2\TermProject_latest\TermPr
src>java DBSClient -ORBInitialHost sierra-leone.encs.concordia.ca -ORBInit
t 2000
Welcome to DBS! Please choose the type of operation:
To Deposit an amount type 1
To withdraw an amount type 2
To transfer an amount type 3
To obtain the balance type 4
3
Please enter the source account number
2220001
Please enter the destination account number
1110001
Please enter the amount to transfer
600.32
The new balance of the source account is:5500.3125

C:\Documents and Settings\p_sinh\Desktop\server2\TermProject_latest\TermPr
src>
```

Figure14: balances after simultaneous transfer operation and crash

4.2 Test Case 2

Name: Concurrency and Synchronization.

Goal:

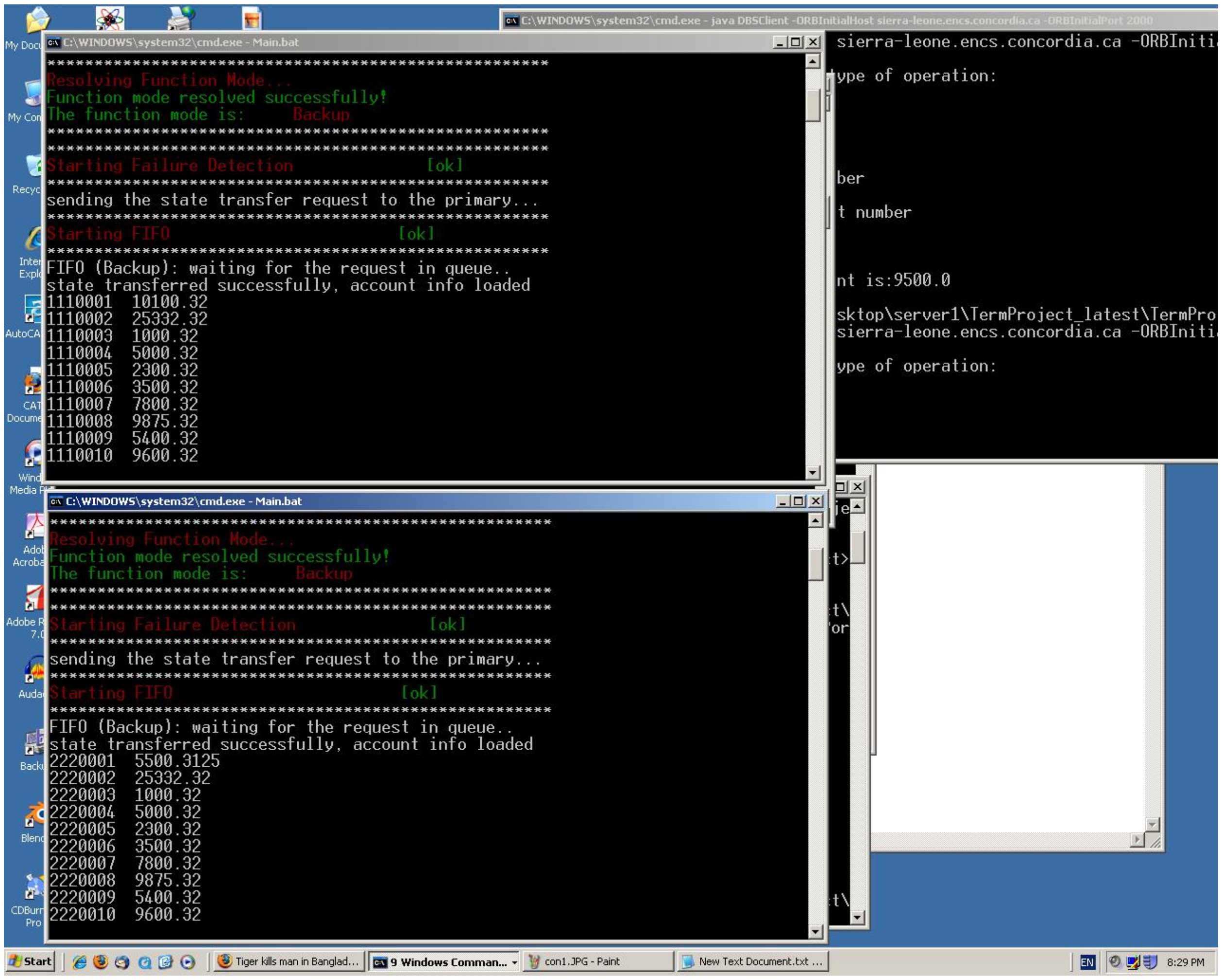
The goal of this test case is to check that server can handle multiple clients at a time (multithread environment), and also to ensure that it can do it by maintaining data consistency.

Analysis:

Multiple clients should access to the same server at same time, and server should serve them concurrently maintaining proper synchronization. The pivotal of this is to do all the transaction in thread, so that server can do all the operation concurrently. Deposit, withdraw are transfer function should be designed properly, so that there should not be any shared violation (using synchronization method of java).

Expected results:

In the figure below four clients are transferring their balance to account number 2220001 and it was successful. Screenshots below verify that data consistency is maintained.

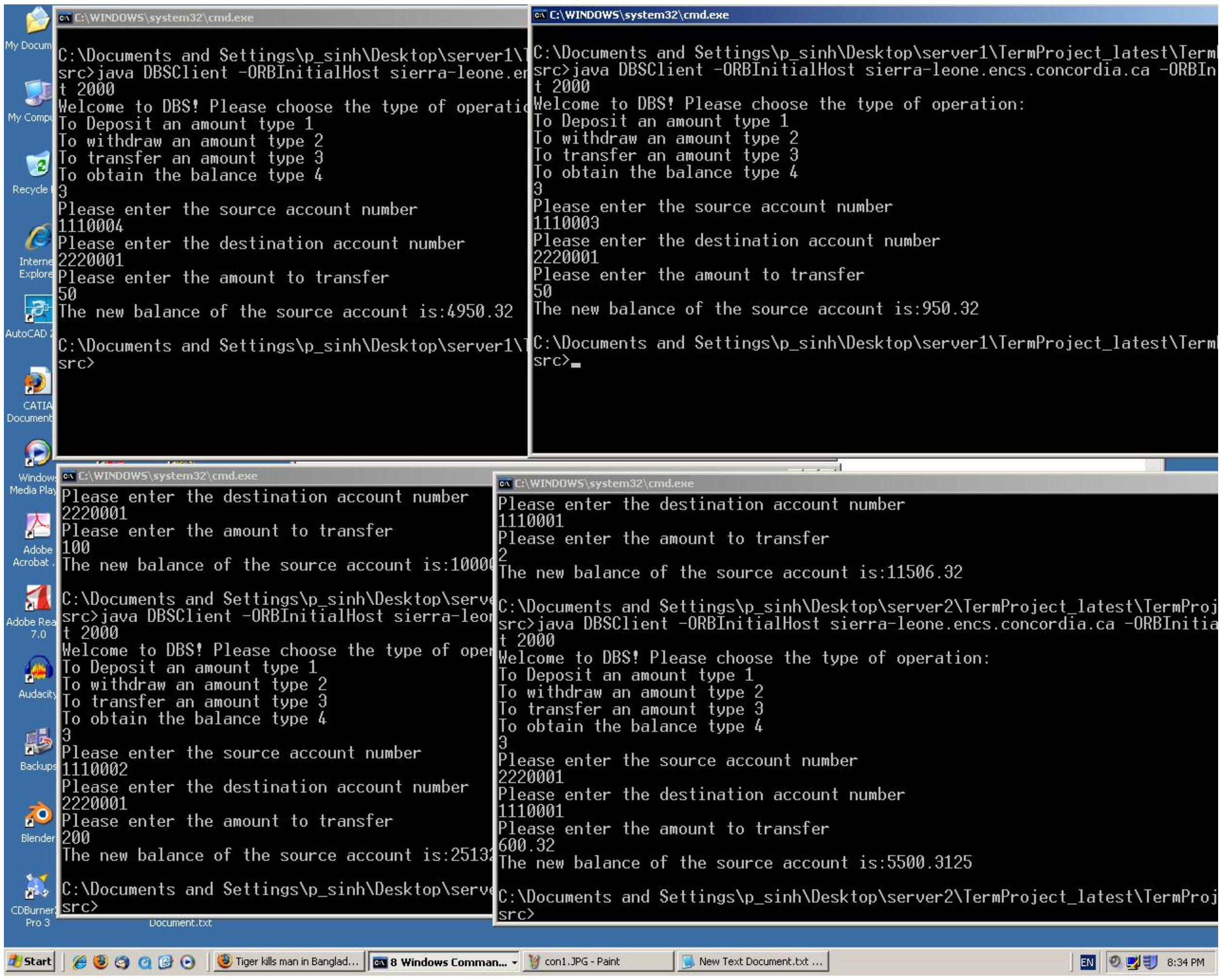


```
C:\WINDOWS\system32\cmd.exe - java DBSClient -ORBInitialHost sierra-leone.encs.concordia.ca -ORBInitialPort 2000
sierra-leone.encs.concordia.ca -ORBIniti
ype of operation:
ber
t number
nt is:9500.0
sktop\server1\TermProject_latest\TermPro
sierra-leone.encs.concordia.ca -ORBIniti
ype of operation:
```

```
C:\WINDOWS\system32\cmd.exe - Main.bat
*****************************************************
Resolving Function Mode...
Function mode resolved successfully!
The function mode is:     Backup
*****************************************************
*****************************************************
Starting Failure Detection               [ok]
*****************************************************
sending the state transfer request to the primary...
*****************************************************
Starting FIFO                        [ok]
*****************************************************
FIFO (Backup): waiting for the request in queue..
state transferred successfully, account info loaded
1110001  10100.32
1110002  25332.32
1110003  1000.32
1110004  5000.32
1110005  2300.32
1110006  3500.32
1110007  7800.32
1110008  9875.32
1110009  5400.32
1110010  9600.32
```

```
C:\WINDOWS\system32\cmd.exe - Main.bat
*****************************************************
Resolving Function Mode...
Function mode resolved successfully!
The function mode is:     Backup
*****************************************************
*****************************************************
Starting Failure Detection               [ok]
*****************************************************
sending the state transfer request to the primary...
*****************************************************
Starting FIFO                        [ok]
*****************************************************
FIFO (Backup): waiting for the request in queue..
state transferred successfully, account info loaded
2220001  5500.3125
2220002  25332.32
2220003  1000.32
2220004  5000.32
2220005  2300.32
2220006  3500.32
2220007  7800.32
2220008  9875.32
2220009  5400.32
2220010  9600.32
```

Figure 15


C:\WINDOWS\system32\cmd.exe
Start
Tiger kills man in Banglad...
8 Windows Comman...
con1.JPG - Paint
New Text Document.txt ...
8:34 PM

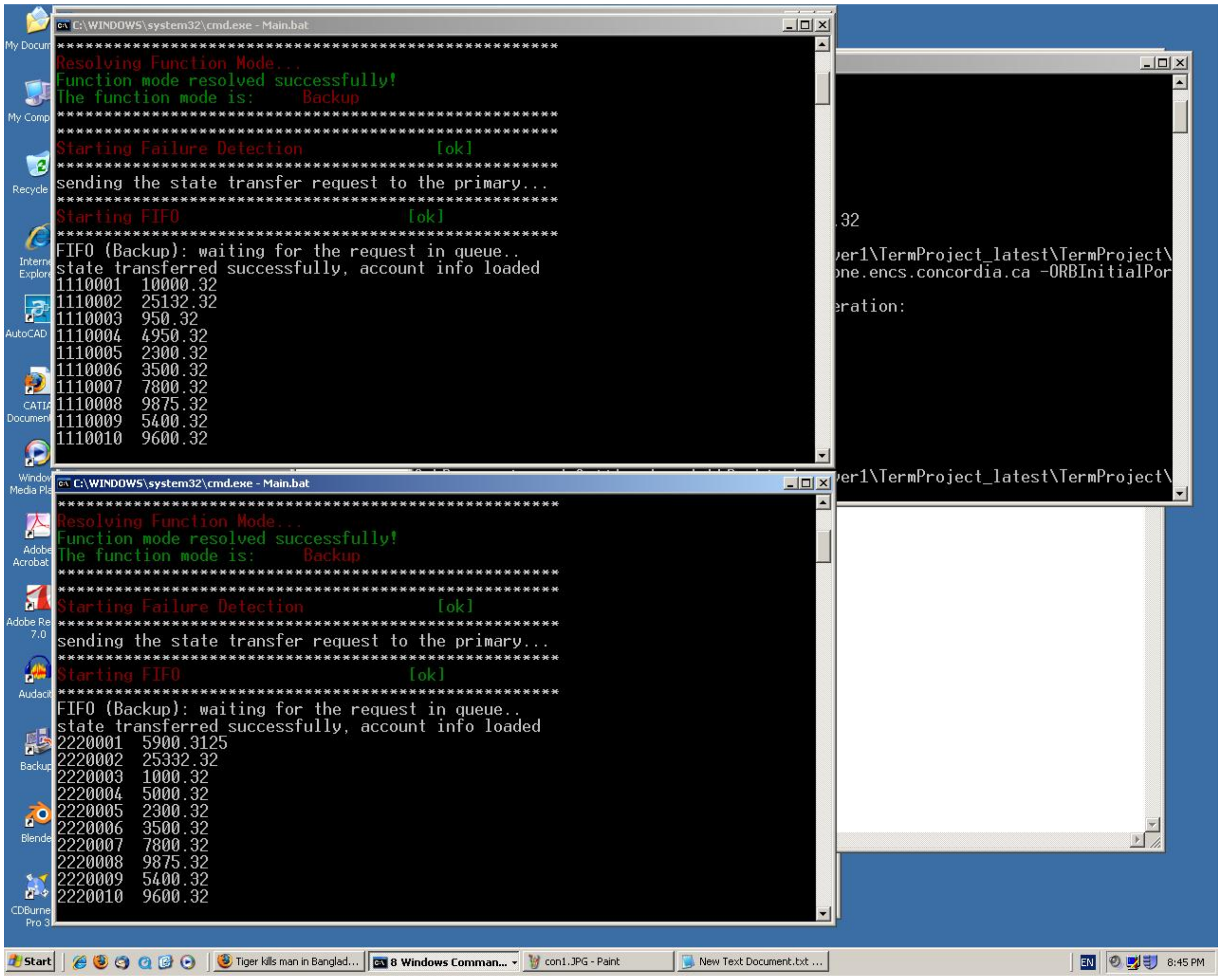
```
C:\Documents and Settings\p_sinh\Desktop\server1\T
src>java DBSClient -ORBInitialHost sierra-leone.en
t 2000
Welcome to DBS! Please choose the type of operatio
To Deposit an amount type 1
To withdraw an amount type 2
To transfer an amount type 3
To obtain the balance type 4
3
Please enter the source account number
1110004
Please enter the destination account number
2220001
Please enter the amount to transfer
50
The new balance of the source account is:4950.32

C:\Documents and Settings\p_sinh\Desktop\server1\T
src>
```

```
C:\Documents and Settings\p_sinh\Desktop\server1\TermProject_latest\Term
src>java DBSClient -ORBInitialHost sierra-leone.encs.concordia.ca -ORBIn
t 2000
Welcome to DBS! Please choose the type of operation:
To Deposit an amount type 1
To withdraw an amount type 2
To transfer an amount type 3
To obtain the balance type 4
3
Please enter the source account number
1110003
Please enter the destination account number
2220001
Please enter the amount to transfer
50
The new balance of the source account is:950.32

C:\Documents and Settings\p_sinh\Desktop\server1\TermProject_latest\Term
src>_
```

```
Please enter the destination account number
2220001
Please enter the amount to transfer
100
The new balance of the source account is:10000

C:\Documents and Settings\p_sinh\Desktop\serve
src>java DBSClient -ORBInitialHost sierra-leor
t 2000
Welcome to DBS! Please choose the type of oper
To Deposit an amount type 1
To withdraw an amount type 2
To transfer an amount type 3
To obtain the balance type 4
3
Please enter the source account number
1110002
Please enter the destination account number
2220001
Please enter the amount to transfer
200
The new balance of the source account is:2513

C:\Documents and Settings\p_sinh\Desktop\serve
src>
```

```
Please enter the destination account number
1110001
Please enter the amount to transfer
2
The new balance of the source account is:11506.32

C:\Documents and Settings\p_sinh\Desktop\server2\TermProject_latest\TermProj
src>java DBSClient -ORBInitialHost sierra-leone.encs.concordia.ca -ORBInitia
t 2000
Welcome to DBS! Please choose the type of operation:
To Deposit an amount type 1
To withdraw an amount type 2
To transfer an amount type 3
To obtain the balance type 4
3
Please enter the source account number
2220001
Please enter the destination account number
1110001
Please enter the amount to transfer
600.32
The new balance of the source account is:5500.3125

C:\Documents and Settings\p_sinh\Desktop\server2\TermProject_latest\TermProj
src>
```

Figure 16


```
C:\WINDOWS\system32\cmd.exe - Main.bat
*****************************************************
Resolving Function Mode...
Function mode resolved successfully!
The function mode is:     Backup
*****************************************************
*****************************************************
Starting Failure Detection               [ok]
*****************************************************
sending the state transfer request to the primary...
*****************************************************
Starting FIFO                         [ok]
*****************************************************
FIFO (Backup): waiting for the request in queue..
state transferred successfully, account info loaded
1110001  10000.32
1110002  25132.32
1110003  950.32
1110004  4950.32
1110005  2300.32
1110006  3500.32
1110007  7800.32
1110008  9875.32
1110009  5400.32
1110010  9600.32
```

```
.32
ver1\TermProject_latest\TermProject\
one.encs.concordia.ca -ORBInitialPor
eration:
ver1\TermProject_latest\TermProject\
```

```
C:\WINDOWS\system32\cmd.exe - Main.bat
*****************************************************
Resolving Function Mode...
Function mode resolved successfully!
The function mode is:     Backup
*****************************************************
*****************************************************
Starting Failure Detection               [ok]
*****************************************************
sending the state transfer request to the primary...
*****************************************************
Starting FIFO                         [ok]
*****************************************************
FIFO (Backup): waiting for the request in queue..
state transferred successfully, account info loaded
2220001  5900.3125
2220002  25332.32
2220003  1000.32
2220004  5000.32
2220005  2300.32
2220006  3500.32
2220007  7800.32
2220008  9875.32
2220009  5400.32
2220010  9600.32
```

Figure 17

5 Appendix: Message Library

The following XML messages are used to convey information between modules. Specifically it is the message format used for server to server communication. An XMLParser class is implemented to build these messages which essentially build the string representation of the requests as in the following example to inform the server to perform a deposit request.

```
public String BuildDepositRequest(String accNo, String amount,String type)
{
   return BuildTag("xml",(BuildTag("type",type)+BuildTag("purpose","deposit")
         +BuildTag("accNo",accNo)+ BuildTag("amount",amount)));
}
```

The details of these messages along with their purpose is illustrated in the following few pages along with some quick reference block diagrams.

//This message is used by the client to issue a deposit request from the server object.
<u>BuildDepositRequest</u>:

```
<xml>
        <type>type</type>
        <purpose>deposit</purpose>
        <accNo>1110001</accNo>
        <amount>900.32</amount>
</xml>
```

//This message is used by the client to issue a withdraw request from the server object.
<u>BuildWithdrawRequest</u>:

```
<xml>
        <type>type</type>
        <purpose>withdraw</purpose>
        <accNo>1110001</accNo>
        <amount>900.32</amount>
</xml>
```

//Comment
<u>BuildFDRequest</u>:

```
<xml>
        <type>internal</type>
        <purpose>fd</purpose>
        <req/rep>request</req/rep>
</xml>
```

//Comment
<u>BuildFDReply</u>:

```
<xml>
        <type>internal</type>
        <purpose>fd</purpose>
        <req/rep>reply</req/rep>
</xml>
```

//This message is used by the client to issue a transfer request from the server object.
<u>BuildTransferRequest</u>:

```
<xml>
        <type>external</type>
        <purpose>transfer</purpose>
        <req/rep>reply</req/rep>
        <accNo>1110001</accNo>
        <amount>900.32</amount>
</xml>
```

//This message is used by the BranchMonitor. BranchMonitor sends this request to
//daemon thereby asking it to run a Primary server
<u>BuildPrimaryRequest</u>:

```
<xml>
        <type>control</type>
        <value>Primary</value>
</xml>
```

//This message is used by the BranchMonitor. BranchMonitor sends this request to
//daemon thereby asking it to run a Backup server
<u>BuildBackupRequest</u>:

```
<xml>
        <type>control</type>
        <value>Backup</value>
</xml>
```

//This message is used by the BranchMonitor. BranchMonitor sends this message
//as an acknowledgement to the surviving object's request for creating a new
//backup.
<u>BuildAck</u>:

```
<xml>
        <type>control</type>
        <value>BMAck</value>
</xml>
```

//This message is used by the surviving server object (primary or backup). The server
//sends this request to daemon thereby asking it to run a BranchMonitor
<u>BuildBranchMonitorRequest</u>:

```
<xml>
        <type>control</type>
        <value>BranchMonitor</value>
</xml>
```

//This message is used by both server objects (primary and backup). The server
//sends this request to daemon thereby asking it to return the hostname of BranchMonitor
//for communication
<u>BuildGetBMHostname</u>:

```
<xml>
	<type> control </type>
	<value>GetBMHostName</value>
</xml>
```

//This message is sent by daemon in response to server's request of the BranchMonitor hostname
//upon startup
<u>BuildHostNameReply</u>:

```
<xml>
	<type>control</type>
	<value>BMHostName</value>
</xml>
```

//This message is used by both server objects. Each server object sends this request to
//BranchMonitor to ask the host name on which the other server object is running.
<u>BuildOtherServerNameRequest</u>:

```
<xml>
	<type>control</type>
	<value>OtherServerNameRequest</value>
</xml>
```

//This message is used by the BranchMonitor. BranchMonitor sends this message as a reply
//to each of the serever object's request for the name of the other object. It contains
//the hostname on which the requested server object resides.
<u>BuildOtherServerNameReply</u>:

```
<xml>
	<type>control</type>
	<value>OtherServerNameReply</value>
</xml>
```

//This message is used by server objects. Server objects send this message to the BranchMonitor thereby
//asking it the RunMode(whether to run as primary or backup
<u>BuildRunModeRequest</u>:

```
<xml>
	<type>control</type>
	<value>RunMode</value>
</xml>
```

//This message is used by the BranchMonitor. BranchMonitor sends this message to server objects as a reply
//to RunMode request. It tells the server object to configure itself as primary. (in the beginning)
<u>BuildRunAsPrimary</u>:

```
<xml>
	<type>control</type>
	<value>RunAsPrimary</value>
</xml>
```

//This message is used by the BranchMonitor. BranchMonitor sends this message to server objects as a reply
//to RunMode request. It tells the server object to configure itself as primary. (in the beginning)
<u>BuildRunAsBackup</u>:

```
<xml>
	<type>control</type>
	<value>RunAsBackup</value>
</xml>
```

//Comment
<u>BuildFIFOAck</u>:

```
<xml>
	<type>FIFO</type>
	<purpose>Ack</purpose>
	<seq_no>1</seq>
</xml>
```

//Comment
<u>BuildFIFODepositRequest</u>:

```
<xml>
	<type>FIFO</type>
	<from>servant</from>
	<purpose>deposit</purpose>
	<accNo>1110001</accNo>
	<amount>1000.32</amount>
</xml>
```

//Comment
<u>BuildFIFOWithdrawRequest</u>:

```
<xml>
	<type>FIFO</type>
	<from>servant</from>
	<purpose>deposit</purpose>
	<accNo>1110001</accNo>
	<amount>1000.32</amount>
</xml>
```

//Comment
<u>BuildTransferRequest</u>:

```
<xml>
        <type>transferl</type>
        <value>request</value>
        <sender>burkina-faso.encs.concordia.ca</sender>
        <accNo>1110001</accNo>
        <amount>1000</amount>
</xml>
```

//Comment
<u>BuildTransferReplyAck</u>:

```
<xml>
        <type>transfer</type>
        <value>reply</value>
        <content>Ack</content>
</xml>
```

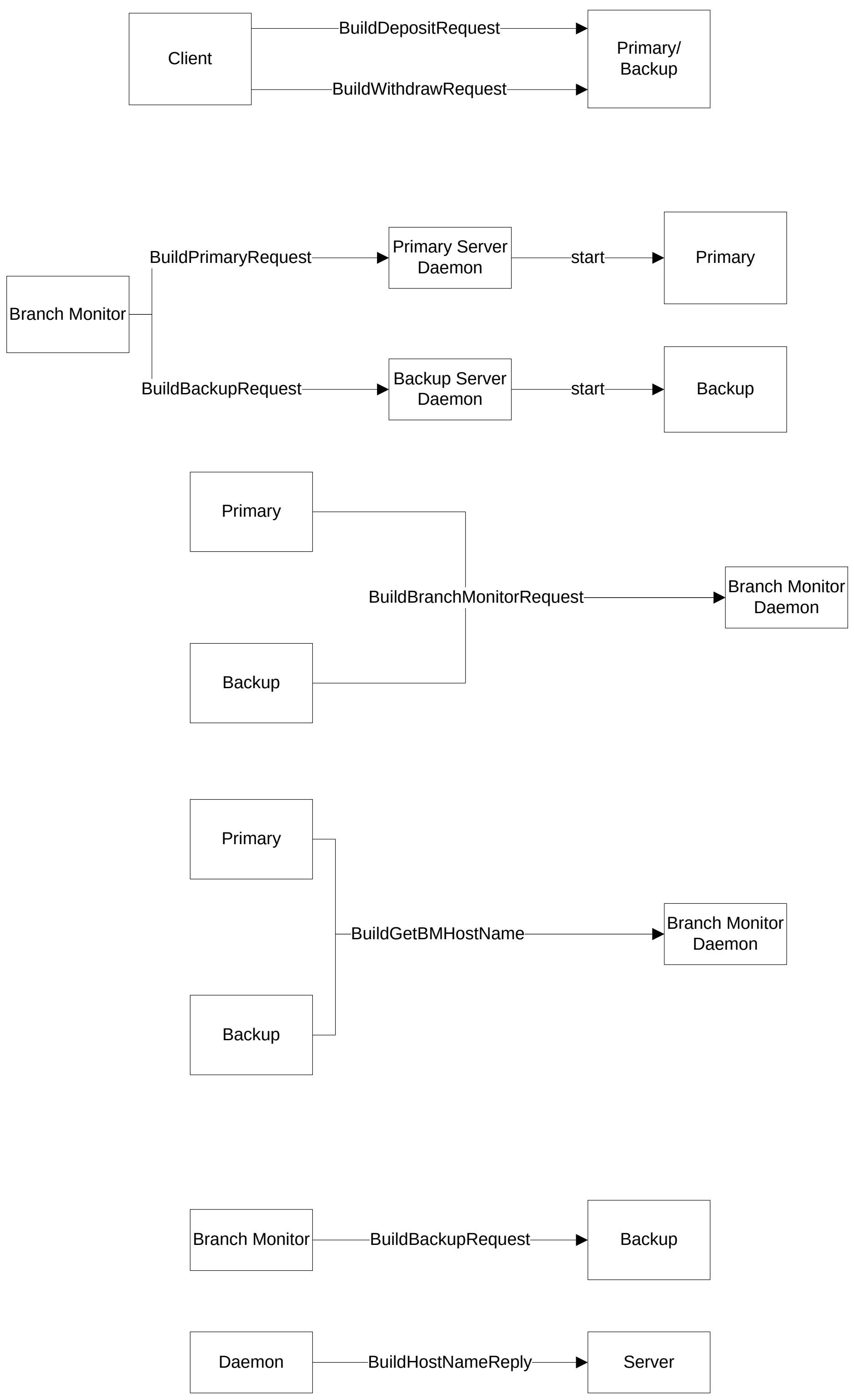
//Comment
<u>BuildTransferReplyNack</u>:

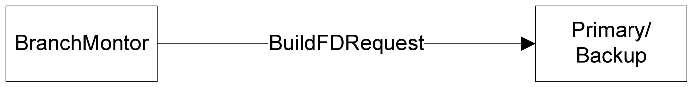
```
<xml>
        <type>transfer</type>
        <value>reply</value>
        <content>Nack</content>
        <altHost>tahiti.cs.concordia.ca</altHost>
</xml>
```

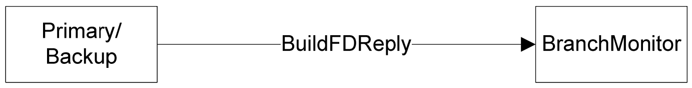
// This message contains the accounts information representing the state of the client accounts
// the time the memory was read and output to a string as an xml message.
<u>AccountInfoMessage</u>:

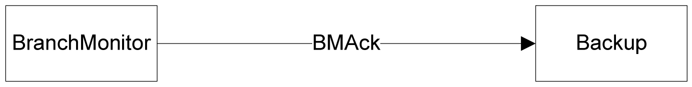
```
<xml>
        <type>ST</type>
        <purpose>reply</purpose>
        <no_of_clients> 50 </no_of_clients>
        <client 1>
                <acc_no> 1110001 </acc_no>
                <amount> 900.32 </amount>
        </client 1>
        <client 2>
                <acc_no> 1110002 </acc_no>
                <amount> 901.32 </amount>
        </client 2>
        <client 3>
                <acc_no> 1110003 </acc_no>
                <amount> 902.32 </amount>
        </client 3>
        <client 4>
                <acc_no> 1110004 </acc_no>
                <amount> 903.32 </amount>
        </client 4>
        <client 5>
                <acc_no> 1110005 </acc_no>
                <amount> 904.32 </amount>
        </client 5>

.
.
.
        <client 50>
                <acc_no> 1110050 </acc_no>
                <amount> 21009.89 </amount>
        </client 50>
</xml>
```

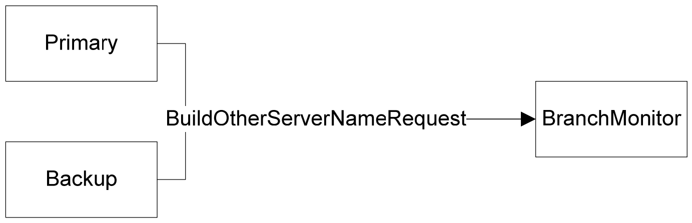
XML Message
Communication

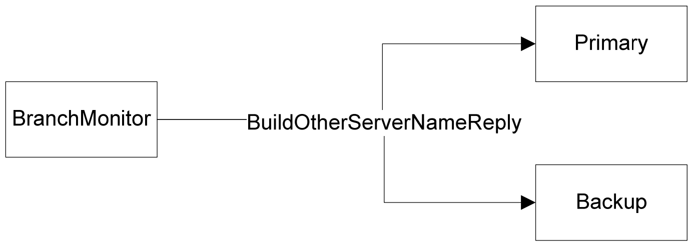


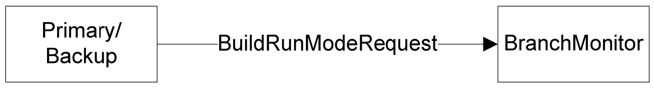

BranchMontor
BuildFDRequest
Primary/ Backup

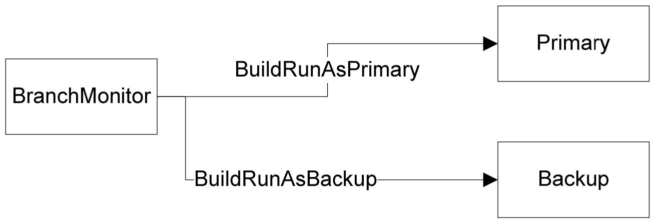

Primary/ Backup
BuildFDReply
BranchMonitor

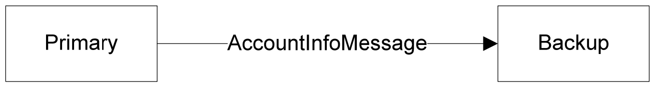

BranchMonitor
BMAck
Backup

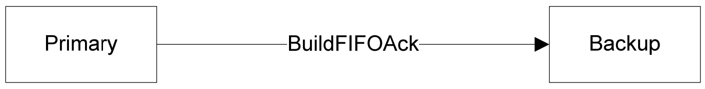

Primary
Backup
BuildOtherServerNameRequest
BranchMonitor

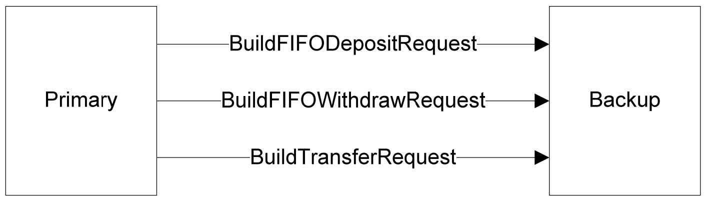

BranchMonitor
BuildOtherServerNameReply
Primary
Backup

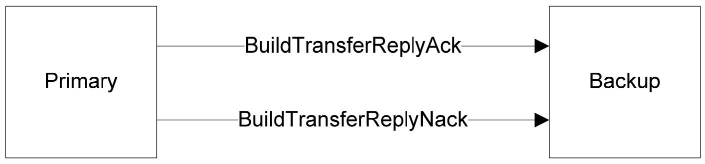

Primary/ Backup
BuildRunModeRequest
BranchMonitor


BranchMonitor
BuildRunAsPrimary
Primary
BuildRunAsBackup
Backup


Primary
AccountInfoMessage
Backup


Primary
BuildFIFOAck
Backup


Primary
BuildFIFODepositRequest
BuildFIFOWithdrawRequest
BuildTransferRequest
Backup


Primary
BuildTransferReplyAck
BuildTransferReplyNack
Backup