



Bulletin Board System

LAB② - DISTRIBUTED SYSTEMS

Ahmed Reda (8)

Ahmed Hamdy (13)

Mohammed Deifallah (59)

Phase 1 (TCP/IP)

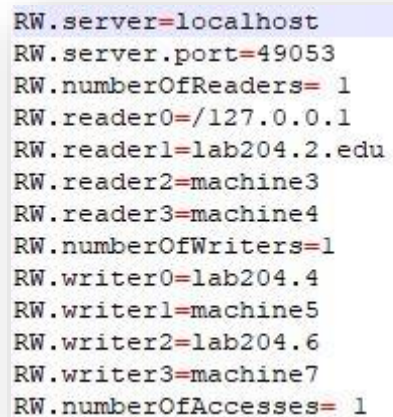
OVERVIEW

In this assignment we will be simulating a news bulletin board system. In this system there are several processes accessing the system, either getting the news from the system or updating the news.

The system will consist of a server that implements the actual news bulletin board and several remote clients that communicate with the server using the TCP/IP protocol suite. The remote clients will have to communicate with the server to write their news to the bulletin board and also to read the news from the bulletin board.

START.JAVA

It's the main class that's supposed to initiate the server and all the clients according to the **system.properties** file. The following figure shows a screenshot of this configuration file:

A screenshot of a text file named system.properties. The file contains configuration parameters for a news bulletin board system, including server address, port, number of readers and writers, and their respective hostnames or IP addresses. The text is as follows:

```
RW.server=localhost
RW.server.port=49053
RW.numberOfReaders= 1
RW.reader0=/127.0.0.1
RW.reader1=lab204.2.edu
RW.reader2=machine3
RW.reader3=machine4
RW.numberOfWriters=1
RW.writer0=lab204.4
RW.writer1=machine5
RW.writer2=lab204.6
RW.writer3=machine7
RW.numberOfAccesses= 1
```

This main file uses a parser as a **Decorator** to parse the properties file as follows:

```
serverAddress = props.getProperty("RW.server");
serverPort = props.getProperty("RW.server.port");
numReaders = Integer.valueOf(props.getProperty("RW.numberOfReaders"));

numWriters = Integer.valueOf(props.getProperty("RW.numberOfWriters"));

requests = Integer.valueOf(props.getProperty("RW.numberOfAccesses"));
for (int i = 0; i < numReaders; i++) {
    readers.add(props.getProperty("RW.reader" + i));
}
System.out.println(readers.size());
for (int i = 0; i < numWriters; i++) {
    writers.add(props.getProperty("RW.writer" + i));
}
```

After that, a **Builder** component is used to compose a particular server with respect to the specifications passed to it as follows:

```
public NewsServer getServer() throws RemoteException {
    return new NewsServer(portNumber, numReaders, numWriters, maxReqs);
}

public NewsServer getServer(int init) throws RemoteException {
    return new NewsServer(portNumber, numReaders, numWriters, maxReqs, init);
}
```

SERVER SIDE

NewsServer.java

This is the main class of server side which has the server itself. It implements **Runnable** Interface, so that it can serve all the requests it receives. The figure below shows the attributes it has:

```
private SharedObject<Integer> object;
private SharedInt curReaders;
private String port_number;
private int num_readers, num_writers;
private SharedLog readLog, writeLog;
private int sseq = 0;
private int maxReqs;
private ClientBuilder builder ;
```

Note that the most important part of that class is the loop inside the overridden method `run()`:

```
ServerSocket server_socket ;
sseq = 0;
server_socket = new ServerSocket(Integer.parseInt(port_number));
List<Thread> handlers = new ArrayList<>();
while (sseq < maxReqs) {
    Socket client = server_socket.accept();
    builder.sseq = ++sseq;
    builder.connection = client;
    ClientHandler handler = builder.get();
    handlers.add(handler);
    handler.start();
}
for(Thread handler:handlers) {
    handler.join();
}
exportReport();
```

This figure could be a good starting point to demonstrate the responsibility of each component inside this class. **ClientBuilder** is a generator for a thread to serve its correspondent client.

```
ClientHandler get() throws IOException {
    return new ClientHandler(sseq,connection,o,readers,readLog,writeLog);
}
```

This **ClientHandler** extends **Thread** so it can be spawned easily and be a lightweight individual.

The last point is the **SharedObject** class that is a generic class, that has two subclasses. The first one is **SharedInt**, *Integer*, which represents the number of readers so that it can be incremented and decremented flexibly by the client itself. And the second one is **SharedLog**, *String*, which represents the line of log inside the log file.

CLIENT SIDE

Client.java

```
public abstract class Client {
    protected int id;
    protected String ServerIP;
    protected Integer ServerPort;
    protected String file;
    protected FileWriter fstream;
    public Client(Integer id) {
        this.id = id;
        file = "log"+id+".log";
    }
    public void setServer(String serverIP,Integer serverPort) {
        this.ServerIP = serverIP;
        this.ServerPort = serverPort;
    }
    public abstract boolean request();

    protected abstract boolean log();

    public void close() throws IOException {
        fstream.close();
    }
}
```

As shown above, it's an abstract class, that is supposed to be extended by subclasses, each for its functionality.

Reader.java

The most important highlight of this subclass is to show the overriding of the prementioned **request()** method:

```
@Override
public boolean request() {
    try {
        Socket connection = new Socket(ServerIP, ServerPort);
        OutputStream outToServer = connection.getOutputStream();
        DataOutputStream out = new DataOutputStream(outToServer);

        out.writeInt(-1);
        out.writeInt(0);
        out.writeInt(id);
        InputStream inFromServer = connection.getInputStream();
        DataInputStream in = new DataInputStream(inFromServer);
        rseq = in.readInt();
        sseq = in.readInt();
        obval = in.readInt();
        log();
        System.out.println("Server response : " + in.readUTF());
        connection.close();
    } catch (IOException e) {
        e.printStackTrace();
        return false;
    }
    return true;
}
```


Writer.java

The same as the previous class. This class extends the abstract **Client** as a writer client. The following figure shows the concrete implementation of the abstract method `request()`:

```
public boolean request() {
    try {
        Socket connection = new Socket(ServerIP, ServerPort);
        OutputStream outToServer = connection.getOutputStream();
        DataOutputStream out = new DataOutputStream(outToServer);
        Random rand = new Random();
        newsID = rand.nextInt();
        out.writeInt(1);
        out.writeInt(newsID);
        out.writeInt(id);
        InputStream inFromServer = connection.getInputStream();
        DataInputStream in = new DataInputStream(inFromServer);
        rseq = in.readInt();
        sseq = in.readInt();
        log();
        System.out.println("Server response : " + in.readUTF());
        connection.close();
    } catch (IOException e) {
        e.printStackTrace();
        return false;
    }
    return true;
}
```

Phase 2 (RMI)

OVERVIEW

In this phase you will implement news bulletin board system using RPC/RMI instead of sockets. So, we will be developing another distributed client/server version of the solution using a service-oriented request/reply scheme.

Typically, we will use the same code we have written for Phase 1 but using RMI for all necessary communication instead of the Sockets. Also, while RMI will take care of the client threads (we don't need to handle it like sockets), we still need to provide the necessary synchronization.

SERVER SIDE

RemoteServer.java

This interface extends java.rmi.Remote as follows:

```
public interface RemoteServer extends Remote {  
  
    public Integer[] readData(Integer id) throws RemoteException;  
    public Integer[] writeData(Integer id ,Integer data) throws RemoteException;  
  
}
```

Server.java

This main class parses the arguments passed to the application through the command line, so it can handle both the cases of socket programming or RMI. For RMI, it initiates the server as follows:

```
public static void startRMI(Integer port,Integer readers,Integer writers, Integer reqs,  
                            String Ip,Integer RMIPort) throws RemoteException, AlreadyBoundException {  
    ServerBuilder builder = new ServerBuilder();  
    builder.setCurReaders(new SharedInt(0));  
    builder.setPortNumber(port+"");  
    builder.setNumReaders(readers);  
    builder.setNumWriters(writers);  
    builder.setMaxReqs(reqs);  
    Integer registry = (RMIPort);  
    NewsServer server = builder.getServer();  
    System.setProperty("java.rmi.server.hostname",Ip);  
    System.out.println("Starting RMI Server");  
    Registry registr = LocateRegistry.createRegistry(registry);  
    registr.bind("MyServer",server);  
  
}
```

CLIENT SIDE

AbstractClient.java

```
public abstract class AbstractClient {
    protected int id;
    protected String ServerIP;
    protected Integer ServerPort;
    protected String file;
    protected FileWriter fstream;
    public AbstractClient(Integer id) {
        this.id = id;
        file = "log"+id+".log";
    }
    public void setServer(String serverIP,Integer serverPort) {
        this.ServerIP = serverIP;
        this.ServerPort = serverPort;
    }
    public abstract boolean request();

    protected abstract boolean log();

    public void close() throws IOException {
        fstream.close();
    }
}
```

As shown above, it's an abstract class, so it can define the general behavior of the client.

RemoteClient.java

```
public abstract class RemoteClient extends AbstractClient {

    public RemoteClient(Integer id) {
        super(id);
    }
}
```

It's just an empty subclass of the previous one to preserve the ability of the hierarchy to be more extensible.

RemoteReader.java

It's a subclass of previous one to implement the reader functionality. The following functionality gives a brief about the **request()** overridden method, and the usage of the **Registry**.

```
@Override
public boolean request() {
    try {
        System.out.println("RMI Reading");
        System.setProperty("java.rmi.server.hostname",this.ServerIP);
        Registry registry = LocateRegistry.getRegistry(this.ServerIP, this.registry);
        look_up = (RemoteServer) registry.lookup("MyServer");
        Integer []res = look_up.readData(id);
        rseq = res[0];
        sseq = res[1];
        obval = res[2];
        log();

    }catch (RemoteException | NotBoundException e) {
        e.printStackTrace();
        return false;
    }
    return true;
}
```

RemoteWriter.java

```
@Override
public boolean request() {
    try {
        System.setProperty("java.rmi.server.hostname",this.ServerIP);
        Registry registry = LocateRegistry.getRegistry(this.ServerIP, this.registry);
        look_up = (RemoteServer) registry.lookup("MyServer");
        Random rand = new Random();
        int newsID = rand.nextInt();
        Integer []res = look_up.writeData(id,newsID);
        rseq = res[0];
        sseq = res[1];
        log();

    }catch (RemoteException | NotBoundException e) {
        return false;
    }
    return true;
}
```

This class is a subclass of **RemoteClient**, that implements the writer client functionality. The attached figure shows the implementation of **request()** method.