Networking

We are all now connected by the Internet, like neurons in a giant brain. - Stephen Hawking

Introduction

In this lab, you will add a FactoryWarehouse on the server that will deliver resources via a server connection to the Clients, which will be stored in the mailbox. This way, once the FactoryStockPersons go to the mailbox, they may have to wait for a new Resource to be delivered from the server. If you have done the Threading lab, this may look familiar.

Part 1 – Making a FactoryWarehouse

Create a new class called FactoryWarehouse in the Server package.

This class will constantly generate new Resources and ship them to the Factories, so let the class implement Runnable so it can operate by itself.

```java
public class FactoryWarehouse implements Runnable
```

The server already keeps track of all the clients to which it is connected, so pass that Vector of ServerClientCommunicators into the constructor.

```java
private Vector<ServerClientCommunicator> communicators;
```

```java
FactoryWarehouse(Vector<ServerClientCommunicator> sccVector) {
    communicators = sccVector;
}
```

The reference to the ServerClientCommunicator will never change, but the Factory and the respective resources can. If we switch the factory, we want the Warehouse to send the correct resources, create a method to set the current list of Resources, and store the list.

```java
private volatile Vector<Resource> resources;
```

```java
public void setFactory(Factory factory) {
    resources = factory.getResources();
}
```

*Note: We use the volatile keyword because we are going to be busy-waiting on the resources reference. This keyword guarantees visibility of changes to variables across threads. If the variable is still ‘null’ in one CPU’s cache, and another CPU changes the variable, the thread waiting for the variable to update may never see the change. Volatile gets rid of this problem by forcing the thread to always read from memory and never used a cached value of the variable.*

Lastly, implement the run() method. Generate a random resource every 7.5 seconds (or any value of your choice), and forward the resource to all the ServerClientCommunicators. If you have done the Threading lab's expansion, feel free to migrate the random resource generation code here.
Random rand = new Random();

@Override
public void run() {
    while(resources == null);
    while(true) {
        try {
            Thread.sleep(7500);
            int toStock = Math.abs(rand.nextInt() % resources.size());
            int number = Math.abs(rand.nextInt() % 25 + 1);
            for(ServerClientCommunicator communicator : communicators) {
                communicator.sendResource(new Resource(resources.elementAt(toStock).getName(), number));
            }
        } catch (InterruptedException e) {e.printStackTrace();}
    }
}

Note: Make sure to check if we have a factory loaded to send the resources.

Note: It is usually a good idea to not busy-wait but rather use conditions. Since this topic hasn’t been covered yet, busy-waiting is fine, but the program will run a bit slower.

To make busy-waiting not as much of a waste of CPU time, make the thread sleep or yield to let other threads run.

while(resources == null)
    try {
        Thread.sleep(1);
    } catch (InterruptedException e1) {e1.printStackTrace();}
Part 2 – Sending the resource

You will need to implement a new method in the ServerClientCommunicator in order to send the resource. Luckily this is very simple since this class can already handle sending entire factories.

```java
public void sendResource(Resource resource) {
    try {
        oos.writeObject(resource);
        oos.flush();
    } catch (IOException ioe) {
        Util.printExceptionToCommand(ioe);
    }
}
```

Note: The flush call is very important. You will want to call it every time you expect something to be sent across the network. However, if you were to send multiple things across a network at once, you only need to call it once everything has been written.

Part 3 – Setting up and running the warehouse

The warehouse is ready to start sending resources to the connected factories, but we need to make sure to create one in the ServerListener class, and make sure to update it when a new factory is selected.

```java
private FactoryWarehouse factoryWarehouse;
```

Modify the constructor to create and run the Warehouse as shown:

```java
public ServerListener(ServerSocket ss) {
    this.ss = ss;
    sccVector = new Vector<ServerClientCommunicator>();
    factoryWarehouse = new FactoryWarehouse(sccVector);
    new Thread(factoryWarehouse).start();
}
```

Right after we notify the clients about a new factory, update the FactoryWarehouse about the change as well.

```java
public void sendFactory(Factory factory) {
    this.factory = factory;
    for (ServerClientCommunicator scc : sccVector) {
        scc.sendFactory(factory);
    }
    factoryWarehouse.setFactory(factory);
}
Part 4 – Receiving the Resources

If you run the factory now, the server’s Warehouse will start sending off resources. You will also notice the client crashing!

Exception in thread "Thread-3" java.lang.ClassCastException: resource.Resource cannot be cast to resource.Factory
at client.FactoryClientListener.run(FactoryClientListener.java:52)

This is great – this means that the client is receiving the resource, but it just doesn’t know what to do with it.

We have assumed thus far that any object received from the server is a factory, and we try to cast this object to a Factory.

Change the run() method of the FactoryClientListener class in the Client package as follows:

```java
public void run() {
    try {
        mFClientGUI.addMessage(Constants.waitingForFactoryConfigMessage);
        Factory factory;
        while(true) {
            // in case the server sends another factory to us
            Object obj = cis.readObject();
            if (obj instanceof Factory) {
                factory = (Factory)obj;
                mFManager.loadFactory(factory, mFClientGUI.getTable());
                mFClientGUI.addMessage(Constants.factoryReceived);
                mFClientGUI.addMessage(factory.toString());
            } else if (obj instanceof Resource) {
                Resource toDeliver = (Resource)obj;
                mFManager.deliver(toDeliver);
                mFClientGUI.addMessage(Constants.resourceReceived);
                mFClientGUI.addMessage(toDeliver.toString());
            }
        }
    } catch (IOException ioe) {
        mFClientGUI.addMessage(Constants.serverCommunicationFailed);
    } catch (ClassNotFoundException cnfe) {
        Util.printExceptionToCommand(cnfe);
    }
}
```

This will read the object and check what type it is. Based on this, we can take different actions.

Print out a message to the GUI just as we did with the factory. You will need to add the constant String resourceReceived to the Constants class.
Part 5 – Adding the resource to the mailbox

Create a `deliver(Resource)` method in the `FactoryManager`.

We will need to implement some checks to make sure that we have a factory and a mailbox to put the resource into.

```java
public void deliver(Resource resource) {
    if (mFactorySimulation != null) {
        if (mFactorySimulation.getMailBox() != null) {
            mFactorySimulation.getMailBox().insert(resource);
        }
    }
}
```

Create an `insert(Resource)` method inside the `FactoryMailbox` class:

```java
private Queue<Resource> mail;

    mail = new LinkedList<Resource>();

    public void insert(Resource resource) {
        for (Resource r : available) {
            if (r.getName().equals(resource.getName())) {
                mail.add(resource);
                break;
            }
        }
    }
```

It is also a good idea to check if the resource type exists. **In case you did the previous lab's expansion code,** you will need to comment out the line of code that starts the thread. Otherwise we will still be generating Resources in the mailbox.

```java
//@Lab 7 expansion code
mLock = new ReentrantLock();
//@new Thread(this).start();
```

Part 6 – Retrieving an item from the mailbox

Now our mailbox has a queue of resources that the server’s warehouse has sent. Change the `getStock()` method to cause the worker to wait if there is no resource in the mailbox, but otherwise return the next resource. This may be similar to how you did the Threading lab’s expansion.

```java
public Resource getStock() throws InterruptedException {
    while (mail.isEmpty()) {
        Thread.sleep(200);
    }
    return mail.remove();
}
```
Expand on This

Instead of letting the Warehouse randomly generate resources on the server side, inform the server of what resource is needed most and let that determine what the Warehouse sends.

Right now, we just implemented a communication chain between the server and the client that works more or less like this:

**FactoryServer:** Hey, I got a Factory. ServerListener, can you send it to the Client?

**ServerListener:** Sure. Just let me open a connection (grabs ServerSocket from GUI and gives it to ServerClientCommunicator).

**ServerClientCommunicator:** Cool, the channel is open (new ObjectOutputStream (socket.getOutputStream())). Let's package the Factory in a nice envelope (oos.writeObject(factory);) and fire it off.

(After the buffer is flushed)

**FactoryClientListener:** Oh hey, I got a Factory from the server. I'm going to open up this envelope (factory = (Factory) obj;), and give it to the FactoryManager to get this Factory working.

Unfortunately, this dialogue does not work in the reverse direction. The FactoryClientListener, currently, cannot write objects in the reverse direction and have it read by the FactoryServer in the end. However, there is another way.

As you may have learned in class, networking can be done in two ways. You can send either an object via an ObjectOutputStream, or a String via a PrintWriter. You might not be aware, but there is a hidden, unused, client-to-server dialogue channel that uses the second method.

**FactoryClientListener:** Oh boy, I need to tell the server <some string>. Better send a message to the server using sendMessage(). I don't even know why I have this, but I'm going to use it.

(String is sent)

**ServerClientCommunicator:** I'm in my run() method, oh hey, a string just came in. It says <some string>. Okay, since I don't have anything else to do, I'll just post the string to the GUI.

You have two choices to do this expansion, either make your own Object writer that goes in the opposite direction, or utilize this unused (and very convenient, cough cough) client-to-server channel. Here are some hints to get you going.

1. Find a way to encode your Strings to be sent. You don't want to destroy the original purpose of the sendMessage() method.
2. When the ServerClientCommunicator receives the String, take a **decision** as to what to do with it. Is it a String that requests a resource? Is it something else that should just be posted in the GUI?

3. The FactoryClientCommunicator already contains a run() method to asynchronously read in an object. Maybe it can also be **rigged** to determine if a resource request should be sent to the server.

4. Because the Client is asynchronously requesting resources now, there is no immediate need for the Warehouse to generate **random** resources continuously. Maybe the ServerClientCommunicator should **tell or toggle** which resource is generated. You can still generate a random **amount** if you wish.

5. You might want to look at these three files: **FactoryClientListener**, **FactoryWarehouse**, and **ServerClientCommunicator**

6. You may notice that the StockPersons will refill the lowest resource several times in a row before going off to refill the next lowest resource. That is okay for this lab. As long as it is obvious that the StockPersons are not refilling arbitrary resources and that the StockPersons are attempting to fill the lowest resource, then it is ok.

Remember, there is no one right way of doing this. Be **creative** and ask your Lab CP) for help if needed! Good luck!

Did you read all the way to here before your lab section? Good. Start this lab early, it's a bit beefy.