

# ASSIGNMENT 2: CLIENT-SERVER NETWORKING

## Table of Contents:

<b>I.</b>	<b>Link of the Google Code SVN .....</b>	<b>2</b>
<b>II.</b>	<b>Coding exercise.....</b>	<b>2</b>
1.	Handin 1: Multithread .....	2
2.	Handin 2: Serialization .....	3
3.	Handin 3: Client Server comm .....	5
4.	Handin 4: File download .....	8
5.	Bonus: .....	10

## I. Link of the Google Code SVN

<http://simpleftp-chhim-mouchel.googlecode.com/svn/trunk>

## II. Coding exercise

### 1. Handin 1: Multithread

*Make the necessary changes to the server code so that it is multithread, Ie handle multiple incoming client connections. It should now be capable of handling multiple clients and file uploads.*

```
while (true) {  
    FTPServer server = new FTPServer(socketaccueil.accept());  
    server.call();  
    // TODO Modify for multithreading  
}
```

Figure 1: Uni-thread code sample

In this code<sup>1</sup>, we note a creation of a new FTPServer, which is a thread, followed by his start. Since there is only one FTPServer running, it is a uni-threading code.

---

<sup>1</sup> See figure 1

```

final int numberOfCores = Runtime.getRuntime().availableProcessors();
final double blockingCoefficient = 0.9;
final int poolSize = (int) (numberOfCores / (1 - blockingCoefficient));
ExecutorService pool = Executors.newFixedThreadPool(poolSize);

try {
    socketaccueil = new ServerSocket(PORT); // Creation of a server socket.
    System.out.println("Server up and running.");
    FTPServer.setDebug(true); // Set the debug mode to true.

    while(true)
    {
        Callable<Void> server = new FTPServer(socketaccueil.accept());
        pool.submit(server);
    }
}

```

Figure 2: Multi-thread code sample

To make a multi-thread code<sup>2</sup>, we can use an `ExecutorService` which can handle multi thread by submitting tasks for execution. Like the Yahoo project, the size of the pool is computed thanks the the variables “`numberOfCores`” and “`blockingCoefficient`”;

## 2. Handin 2: Serialization

*Make the necessary changes to the server and client so that when the client sends a `LIST` command to the server, the server returns a complete directory tree structure, and the client displays this structure in a `JList`. This will require you to recursively explore the working directory on the server.*

- Client side :

```
objectInput = new ObjectInputStream(socket.getInputStream());
```

Figure 3: connect method sample (client side)

We can receive the complete tree structure through an `ObjectInputStream`, which will be linked<sup>3</sup> to the input ends of the socket at the connection (connect method) and unserialize the incoming stream. We must at the begin declare it as this: `private ObjectInputStream objectInput = null`

---

<sup>2</sup> See figure 2.

<sup>3</sup> See figure 3.

Now it is possible to get the tree in the ls method, as the following code:

```
try {  
    fileTree = (File) objectInput.readObject();  
} catch (ClassNotFoundException e) {  
    // TODO Auto-generated catch block  
    e.printStackTrace();  
}
```

Figure 4 : ls method (client side)

- Server side :

We have to proceed in the same way, but instead it must be an `ObjectOutputStream` (contains the serialized object), which will be linked to the out ends of the socket.

Declaration : `private ObjectOutputStream objectOutput = null;`

Redirection of the out end of the socket to the stream:

```
objectOutput = new ObjectOutputStream(socket.getOutputStream());
```

Figure 5: connect method sample (server side)

Sending of the file when the signal *LIST* is received:

```
case LIST:  
    sb = new StringBuilder("200 ");  
    String[] fileNames = f.list();  
    for (String fileName : fileNames) {  
        sb = sb.append(fileName + "|");  
    }  
    sendLine(sb.toString());  
    // TO BE COMPLETED - 1 LINE :  
    writeObjectOutput(f);
```

Figure 6: case LIST (server side)

- In the CommandDispatcher :

```
File f = client.getFileTree();  
window.root.removeAllChildren();  
DefaultMutableTreeNode userRoot = new DefaultMutableTreeNode(f.getName());  
populateTree(userRoot, f);  
window.root.add(userRoot);
```

Figure 7: listDir method sample (CommandDispatcher class)

When the listDir method is called in the CommandDispatcher, the getFileTree method from the client class is called and a file is returned. We “re-initialize” the tree in the GUI, then we populate the tree thanks to the created method populateTree() (figure 8):

```
private void populateTree(DefaultMutableTreeNode userRoot, File f) {  
    for (File file: f.listFiles()) {  
        DefaultMutableTreeNode buffer = new DefaultMutableTreeNode(file);  
        buffer.setUserObject(file.getName());  
        userRoot.add(buffer);  
        if (file.isDirectory()) populateTree(buffer, file);  
    }  
}
```

Figure 8: populateTree (command Dispatcher class)

In this code, we create, for each file, a node, that is added to the root.

### 3. Handin 3: Client Server comm

*Implement the FTP CWD command (and offer the corresponding feature on the client GUI via a button), taking as input the user selection in the tree view.*

- In the GUI:

First, we have to implement a button (like for pwd), with its handler (figure 9):

```
btnCwd = new JButton("CWD");  
panelCommands.add(btnCwd);  
btnCwd.addActionListener(new GUICommandListener<Command>(cmdQueue,  
    console, Command.CWD));
```

Figure 9: initialization of the button (GUI)

In addition, we have added an eventHandler for the tree, which fetch the current directory (stocked in the `DefaultMutableTreeNode curDir`) (figure 10)

```
tree.addTreeSelectionListener(new TreeSelectionListener() {  
    public void valueChanged(TreeSelectionEvent e) {  
  
        TreePath tp = e.getNewLeadSelectionPath();  
        if (tp != null) {  
            curDir = (DefaultMutableTreeNode) tp.getLastPathComponent();  
            System.out.println(curDir.toString());  
  
            StringBuilder remoteFileLocation = new StringBuilder();  
            for (int remoteDirectoriesIterator = 2; remoteDirectoriesIterator < tp.getPathCount();  
                Object tmp = tp.getPath()[remoteDirectoriesIterator];  
                if (tmp.equals(tp.getLastPathComponent()))  
                    remoteFileLocation.append(tmp.toString());  
                else  
                    remoteFileLocation.append(tmp.toString() + "/");  
            }  
  
            FTPClientWindow.this.treeSelectionPath = remoteFileLocation.toString();  
        }  
    }  
});
```

Figure 10: TreeSelectionListener

- In the CommandDispatcher:

We added the

```
case CWD:  
    cwd();  
    break;
```

Then we have coded the cwd function as following (figure 11):

```
private void cwd() {  
    try {  
        if (!alreadyConnected) {  
            window.console.append(NL + "You are not connected to any server.");  
            return;  
        }  
        File f = new File(window.curDir.toString());  
        if (f.isDirectory()) {  
            client.cwd(window.curDir.toString());  
            window.console.append(NL + "CWD on FTP server: " + window.curDir.toString());  
        }  
    } catch (IOException e) {  
        window.console.append(NL + e.getMessage());  
        e.printStackTrace();  
    }  
}
```

Figure 11: cwd method (Command Dispatcher class)

Here we created a new file with the current dir in the GUI. If it is a directory, we call the function `cwd()` in the client.

- Client side:

```
public boolean cwd (String dir) throws IOException  
{  
    sendLine("CWD " + dir);  
    String response = readLine();  
    return (response.startsWith("250 "));  
}
```

Figure 12: cwd (client side)

The command “CWD” with the path is sent to the server.

- Server side:

```
case CWD: // Current Working Directory  
    System.out.println("CWD");  
    System.setProperty("user.dir", response.substring(4));  
    System.out.println("CWD new current directory:"  
        + System.getProperty("user.dir").toString());  
    f = new File(System.getProperty("user.dir"));  
    sendLine("250 ");  
    break;
```

Figure 13: case CWD (server side)

The File `f`, representing the user current working directory, is changed.

#### 4. Handin 4: File download

*Implement the file (not directory) download feature offered by the client GUI taking as input the user selection in the tree view.*

- *The client should allow file download by clicking on the desired file in the tree representation.*
- *Study the working upload feature for inspiration.*
- *Implement multiple simultaneous downloads and/or directory download for extra credit (not required)*

Succinctly, it is the same functioning as for the upload, except that we don't need to code it multi-threaded. In addition, it is the opposite of the upload. Indeed:

- **Server side:**

The server receive the path of the remote file to sent in the InputStream and send it to the client via the OutputStream.

- **Client side:**

The client read the local file from the input of the dataSocket, and receives the requested remote file from the output of the socket (from the server).



- CommandDispatcher

```
private void download() throws Exception
{
    if (!alreadyConnected) {
        window.console.append(NL + "You are not connected to any server.");
        return;
    }
    File file = new File(window.getTreeSelectionPath().substring(window.getTreeSelectionPath().lastIndexOf('/')+1));
    System.out.println("here : " + window.getTreeSelectionPath().substring(window.getTreeSelectionPath().lastIndexOf('/')+1));
    System.out.println("DOWNLOAD starting for " + file.getName());
    Boolean result = false;
    result = client.retr(file, window.getTreeSelectionPath());
    System.out.println("here 2: " + window.getTreeSelectionPath());

    if (result) {
        window.console.append("Successfully transfered file "
            + file.getName());
        System.out.println("Successfully transfered file "
            + file.getName());
    } else {
        window.console.append("Failed transferring file " + file.getName());
        System.out.println("Failed transfered file "
            + file.getName());
    }
}
```

Figure 14: download method (Command Dispatcher class)

Here the path of the file to download is retrieved, and the function retr() of the client is called.

## 5. Bonus:

*Update the client GUI console display so that it always displays the last message (auto scroll) [easy]*

```
// Adding : The client GUI console always display the last message.  
consoleScroll.getVerticalScrollBar().addAdjustmentListener(new AdjustmentListener() {  
    public void adjustmentValueChanged(AdjustmentEvent e) {  
        console.selectAll();  
    }  
});
```

Figure 15: AdjustmentListener (in the GUI)

We added a new event handler to the vertical ScrollBar: If there is a modification to it, we select all the text in the console. The consequence of that is that the last message (logically at the bottom of the console) is always displayed.