School of Computing: assessment brief

Module title	Networks
Module code	COMP2221
Assignment title	Coursework
Assignment type and description	Programming assignment in Java
Rationale	Design and develop client and multi-threaded server applications in Java to solve a specified problem
Guidance	Detailed guidance provided later in this document
Weighting	30%
Submission deadline	2pm Thursday 21st March
Submission method	Gradescope
Feedback provision	Marks and comments for the submitted code returned *via* Gradescope
Learning outcomes assessed	Design, implement and test network protocols and applications.
Module lead	David Head

1. **Assignment guidance**

   For this coursework, you will implement client and server applications for a simple file transfer system that allows clients to request a list of text files contained on the server, and to upload new files to the server. In this way it is somewhat akin to a basic ftp application, but restricted to text files and file upload only.

2. **Assessment tasks**

   To get started, unarchive the file `cwk.zip` (you can do this from the command line by typing `unzip cwk.zip`). You should then have a directory `cwk` with the following structure:

```
cwk --- client --- Client.java
     |          |
     |          --lipsum2.txt
     |
     -- server --- Server.java
                 |
                 -- serverFiles --- lipsum1.txt
```

   Empty `.java` files for the client and server have been provided. **Do not change the names of these files**, as we will assume these file and class names when assessing. You are free to add additional `.java` files to the `client` and `server` directories. You are also provided with some example text files, one already on the server (`lipsum1.txt`), and one on the `client` directory ready to upload (`lipsum2.txt`).

   The requirements for the **server** application are as follows:

   - Run continuously.
   - Use an `Executor` to manage a fixed thread-pool with 20 connections.
   - Following a request by a client, query the local folder `serverFiles` and return a list of the files found there to the same client.
   - Receive a request from a client to upload a new file to `serverFiles`.
   - If a file with the same name already exists, return an error to the client; otherwise transfer the file from the client and save to `serverFiles`.
   - Create the file `log.txt` on the `server` directory and log every **valid** client request, with one line per request, in the following format:
     `date|time|client IP address|request`
     where `request` is one of `list` or `put`, *i.e.* you do not need to log the filename for `put` operations. Do not add other rows (*e.g.* headers) to the log file.

   Note that you must **create** the log file, not overwrite or append an existing file. Any `log.txt` file in your submission will be deleted at the start of the assessment (see below).

   The requirements for the **client** application are as follows:

   - Accepts one of the following commands as command line arguments, and performs the stated task:
     - `list`, which lists all of the files on the server's folder `serverFiles`.

- `put fname`, which uploads the file `fname` to the server to be added to `serverFiles` (see above), or returns an error message to say that this file already exists.
- Exits after completing each command.

Your server application should listen to a port number in the range `9100` to `9999`. Both the client and the server should run on the same host, *i.e.* with hostname `localhost`.

Your solution should work in principle for any text files, not just those provided. You can assume all text files will be less than 64Kb in size. Only text files need be transferred; there is no need to consider binary file transfer. When accessing local files, ensure UNIX filenames **only** are used (*e.g.* do not use Windows' backslashes), as your submission will be tested with a Linux image.

All communication between the client and server must use sockets – they cannot access each other's disk space directly. Your solution must use TCP, but otherwise you are free to devise any communication format you wish, provided the requirements above are met.

Neither the client nor the server should expect interaction from the user once they are executed. In particular, instructions to the `Client` application **must** be *via* command line arguments. In the case of an invalid input, your client application should quit with a meaningful error message.

3. **General guidance and study support**

If you have any queries about this coursework, visit the Teams page for this module. If your query is not resolved by previous answers, post a new message. Support will also be available during the timetabled lab sessions.

You will need the material up to and including Lecture 11 to complete this coursework.

You may like to first develop `Client.java` and `Server.java` to provide minimal functionality, following the examples covered in Lectures 7 and 8. You could then add another class that handles the communication with a single client. This will make it easier to implement the multi-threaded server using the `Executor`. Multi-threaded servers were covered in Lectures 10 and 11. You will need to use input and output streams; these were covered in Lecture 6.

Example session

First `cd` to `cwk/server`, compile, and launch the server:

```
> java Server
```

Now in another tab or shell, `cd` to `cwk/client` and compile. If you execute the following commands, the output should be something like that shown below.

```
>java Client list
Listing 1 file(s):
lipsum1.txt
```

```
> java Client put lipsum2.txt
Uploaded file lipsum2.txt
```

```
>java Client list
Listing 2 file(s):
lipsum1.txt
lipsum2.txt
```

```
> java Client put lipsum2.txt
Error:  Cannot upload file 'lipsum2.txt'; already exists on server.
```

```
> java Client put nonsuch
Error:  Cannot open local file 'nonsuch' for reading.
```

Note your application does not need to follow *exactly* the same output as in this example, as long as the requirements above are followed. This includes the exact filenames of all files stored on the server when performing the `list` operation.

4. **Assessment criteria and marking process**

   Your code will be checked using an autograder on Gradescope to test for functionality. Staff will then inspect your code and allocate the marks as per the provided mark scheme below. This includes the meaningful nature (or not) of error messages output by your submission.

5. **Submission requirements**

   Remove all extraneous files (*e.g.* `*.class`, any IDE-related files *etc.*). You should then archive your submission as follows:

   (a) `cd` to the `cwk` directory
   (b) Type `cd ..`
   (c) Type `zip -r cwk.zip cwk`

   This creates the file `cwk.zip` with all of your files. Make sure you included the `-r` option to zip, which archives all subdirectories recursively. It does not matter if you also include the test files `lipsum1.txt` and `lipsum2.txt`, as different text files will be copied over during testing.

   To check your submission follows the correct format, you should first submit using the link **Coursework: CHECK** on Gradescope. Only once it passes all of the tests should you then submit to the actual submission portal, **Coursework: FINAL**.

   The autograder is set up to use the standard Ubuntu image (base image version 22.04) with OpenJDK 8 installed as follows,

   ```
   apt-get -y install openjdk-8-jdk
   ```

   This version of Java most closely matches the RedHat machines in the Bragg teaching cluster, but may be older than the version on your own machine.

   The following sequence of steps will be performed when we assess your submission.

   (a) Unzip the `.zip` file.
   (b) `cd` to `cwk/client` directory and compile all Java files: `javac *.java`
   (c) `cd` to `cwk/server` directory and do the same.
   (d) If there is a `log.txt` file on the `server` directory, it will be deleted.

(e) To launch the server, `cd` to the `cwk/server` directory and type `java Server`
(f) To launch a client, `cd` to the `cwk/client` directory and type *e.g.* `java Client list`

If your submission does not work when this sequence is followed, you will lose marks.

6. **Academic misconduct and plagiarism**

Academic integrity means engaging in good academic practice. This involves essential academic skills, such as keeping track of where you find ideas and information and referencing these accurately in your work.

By submitting this assignment you are confirming that the work is a true expression of your own work and ideas and that you have given credit to others where their work has contributed to yours.

Code similarity tools will be used to check for collusion, and online source code sites will be checked.

7. **Assessment/marking criteria grid**

This coursework will be marked out of 30.

7 marks	:	Basic operation of the `Server` application, including use of thread pool and log file output.
13 marks	:	Implementation of the `list` and `put` commands.
4 marks	:	Meaningful error messages.
6 marks	:	Sensible code structure with good commenting.
Total: 30		