School of Computer Science: 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 Monday 24th 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 multi-threaded server applications for a simple voting system in which the server is initialised with the options to vote for, and clients can view the current number of votes for each option. Clients can also vote for one of the options, which is then updated on the server.

   This coursework specification is for school Unix machines only, including the remote access `feng-linux.leeds.ac.uk`. We cannot guarantee it will work on any other environment.

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
     |
     -- server --- Server.java
```

   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.

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

   - Accept at least two options that can be voted for, each consisting of a single word, as command line arguments when launched, *e.g.*
     `java Server rabbit squirrel duck`
   - The server should immediately quit with an error message for less than two options.
   - Otherwise, it should run continuously.
   - Use an `Executor` to manage a fixed thread-pool with 30 connections.
   - If the client makes a vote for `<option>`, the vote count for `<option>` should be increased by 1 (note all vote counts should initially be zero). **For the purposes of this assessment, you must include explicit thread synchronisation to ensure this update is thread-safe**[1] – see the first part of Lecture 10 for details.
   - However, if `<option>` does not exist, an error message should be returned instead.
   - Following a `list` request by a client, return the current state of the poll with one line per option, where each line contains at least the option and the current count. See below for an example of valid output.
   - 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`

[1] Note that simply adding 1 to a variable is *not* thread-safe as it requires the old value to be loaded into a register, incremented, and then stored back to global memory; if two threads do this, there is a risk of interleaving these operations, *i.e.*, a data race. Also note that although some Java containers are already thread-safe, for this assessment, you must include your own synchronisation mechanism, as we need to see that you know how to use synchronisation to avoid data races.

where `request` is one of `list` or `vote`, *i.e.* you do not need to log the option for `vote` operations. Do not add other rows (*e.g.* headers, blank lines) 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.

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

- Accept one of the following commands as command line arguments, and perform the stated task:
  - `list`, which displays the current state of the poll from the server and displays it to the user. Each option should be output on the same line as their current vote count, with a different line for each option. See below for an example of valid output.
  - `vote <option>`, which requests that the server increases the vote count for `<option>` by 1, and display the message returned by the server.
- Exits after completing each command.

Your server application should listen to the port number `7777`. Both the client and the server should run on the same host, *i.e.* with hostname `localhost`.

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 with two options to vote for:

```
> java Server rabbit squirrel
```

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
'rabbit' has 0 vote(s).
'squirrel' has 0 vote(s).

> java Client vote rabbit
Incremented the number of votes for 'rabbit'.

>java Client list
'rabbit' has 1 vote(s).
'squirrel' has 0 vote(s).

> java Client vote duck
Cannot find option 'duck'.
```

Note your application does not need to follow *exactly* the same output as in this example, as long as the requirements above are followed.

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.

   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 21 installed as follows,

   ```
   apt-get -y install openjdk-21-jdk
   ```

   This version of Java most closely matches the RedHat machines in the Bragg teaching cluster and on `feng-linux.leeds.ac.uk`.

   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 *e.g.* `java Server mouse rabbit`

(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.

There is a three-tier traffic light categorisation for using Gen AI in assessments. This assessment is **amber** category: AI tools can be used in an assistive role. Use comments in your code to declare any use of generative AI, making clear what tool was used and to what extent.

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.

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