ECS518U Operating Systems

Lab 10: Multi-Threaded Server

This Lab is Assessed

1 Aim

Use classes from java.util.concurrent to create a multi-threaded server.

2 Background

You can find on QM+ java code for a simple client and server. The server listens on a specified port for requests, which are the name of a text file. The file is then written, a line at a time, to the socket to be received by the client. The client prompts the user to enter a filename, requests the file from the server and prints it.

Note:

- Run the server and client in separate terminal windows:

  `java –jar Server.jar` *service_port_number*

  `java –jar Client.jar` *server_ip_address service_port_number client_name*

  (client_name can be any name you choose to give the client, e.g. client1, client2, etc.).

- You can run the client and server on the same machine. To do this use the ip address *localhost* and any ephemeral port number (e.g. 5000; must be greater than 1024).
- To stop the server type Ctrl-C. The client exits if you enter a blank filename.
- The text file must be in the same directory as the server jar file.
- If the file does not exist, the read in the client returns a null string and nothing is printed.
- You can run multiple clients but the requests are handled sequentially.

3 Requirements

You are to enhance the server code to allow for multiple concurrent requests.

4 Design Guidelines

The existing server code has two classes:

1. Main: this contains a main program which parses the command line arguments and creates an instance of Server.
2. Server: this class contains two methods: ‘startServer’ and ‘serveFile’. The method ‘startServer’ creates a server socket on the given port and calls ‘accept’. The ‘serveFile’ method takes the socket returned by accept and processes the request for the file.

You need to modify this design so that the ‘serveFile’ code is executed in a separate thread. You can follow these steps:

1. Create a new class (called e.g. RequestHandler or FileHandler or …):
   a. Move the ‘serveFile’ method into this class.

   b. The class must implement the Runnable interface, so add the necessary declaration and a 'run' method.
   c. The serveFile method should be called from the 'run' method. Since the 'run' method has no parameters, the socket object (needed by serveFile) needs to become a parameter of the constructor of this class.

2. In the Server class
   a. Create a thread pool using the static method `Executors.newFixedThreadPool` (with at least 2 threads); this is found in the package 'java.util.concurrent'. A good place to do this is in the Server class constructor.
   b. Use each socket returned by the call to 'accept' to create a new instance of the new 'RequestHandler' class.
   c. Pass the 'RequestHandler' object to the executor using the 'execute' method (from the Executor interface).

In order to show concurrent execution you may need to put a sleep between the reads of separate lines in the 'serveFile' method.

5 Bonus Problem

Give the server a command line interface with commands to start and stop the server. What happens if a request is made when the server is stopped?

5.1 Design Guidelines for the Challenge Problem

- The command line interface must run in a separate thread from the server. This means that you must have two threads (in addition to those controlled by the thread pool). One thread (e.g. the main thread) is for listening (the 'accept' call); a second thread is for the command line interface.
- A request to stop the server can be made using an 'atomic boolean' (see java.util.concurrent.atomic.AtomicBoolean) which can be set in one thread and read in another thread.
- The server thread that listens for requests must periodically check the request to stop. To do this, a time out is needed on the 'accept' method otherwise the server cannot be stopped when it is idle.
- The thread pool can be stopped by using the methods of the ExecutorService interface: 'shutdown' and 'awaitTermination' or ('shutdownNow').

6 Demonstration

There is no answer sheet. You should be ready to:

1. Briefly describe the functionality you have implemented.
2. Demo your code and prove that your server can handle requests concurrently.
3. Explain the changes you have made to the code. The code should be clearly commented.