School of Computing, University of Leeds

COMP2221 Networks

Worksheet 2: Servers and clients

This worksheet covers material from the second half of Lecture 5 up until Lecture 11, including the `InetAddress` class, client applications, and single and multi-threaded server applications.

Questions

1. *[Lectures 5 and 6]* Download and inspect the `Lookup.java` code from Lecture 6, and check the documentation for the `InetAddress` class from Lecture 5 and online[1]. Test the code on the following hostnames:

   - `www.leeds.ac.uk`
   - `www.comp.leeds.ac.uk`
   - `localhost`
   - `dns6.leeds.ac.uk`

   Also try 'reverse-looking up', *i.e.* using the IP address as input, and see how this relates to the hostnames above. Then try making the following modifications.

   (i) Modify the code to return multiple IP addresses for a single host name. There is an existing method for this purpose that was covered in Lecture 5 and is in the online documentation (see footnote). Try your modified code on `www.google.com`. Note that what you see will depend on your local configuration.

   (ii) Further modify the code to also `ping` each of the IP addresses returned by part (i). Again, there is an appropriate method that performs this function that is available in the online documentation.

2. *[Lecture 7]* For parts (ii) and (iii) of this question you will need to launch a server before running the client. This can be done by compiling and launching the client and server applications in different shells/tabs. The server code corresponding to each client be found on Minerva alongside Lecture 8.

   (i) Download and inspect the `LowPortScanner.java` from Lecture 7, compile and execute it. Without command line arguments, the program will output all port numbers on `localhost` in the range 1 to 1023 inclusive for which a socket can be opened. You may find you get no ports when running on a mobile device, but you should get some when running the program on a School machine. Modify the code to scan all ports in the range 1024 to 65535 inclusive, *i.e.* make it a *high* port scanner. If you are using a

[1] `https://docs.oracle.com/javase/8/docs/api/java/net/InetAddress.html`

Unix machine, then for both the low and high port scanners, check the port numbers in `/etc/services` by using `grep` as in Worksheet 1, to see what service they correspond to.

(ii) Download `DailyAdviceClient.java` from Lecture 7 and the `DailyAdviceServer.java` from Lecture 8. Compile and launch the server in the background, then switch to a new shell/tab, and compile and launch the client. Check the output from both the client and server. If there is an error, check that both applications are using the same port number and host `localhost`. Now kill the server using Ctrl-C and try to launch the client again – notice how this time you get an error message because the server could not be found.

(iii) Download `KnockKnockClient.java` from Lecture 7, and the two associated server files `KnockKnockServer.java` and `KnockKnockProtocol.java`, from Lecture 8. As before, compile and launch the server on one shell/tab, then switch to another shell/tab and launch the client. Try to deliberately get the protocol wrong, *i.e.* do not follow the standard lines for the joke[2]. Also try running two clients simultaneously – launch the first, and then while it is still running, try launching a second. What happens?

3. *[Lecture 8]* Download either of the servers from parts (ii) or (iii) of the previous question. Launch the server, and then in a separate shell/tab, launch the same server again. What happens?

   If you can, try to coordinate with a friend to deploy the paired client-server applications over an actual network. Attempting this on the internet may be tricky because of firewalls *etc.*, but should be possible for a local-area network (LAN), *e.g.* two hosts in the undergraduate teaching lab. Log into a machine and type `hostname` in a shell, then launch the server. Now log into a second machine, and launch the client after first modifying the code to use the first machine's hostname as the server IP address (replacing `localhost`).

4. *[Lectures 7 and 8]* Write client and server classes called something like `PortReporterClient` and `PortReporterServer`, respectively, where the server is listening on port 5555, although you can use another port if you prefer. The client should connect to the server as per the examples in Lecture 7, print the source and destination ports of the socket to screen, then close the socket and quit. Similarly, the server should accept a connection from the client as per the examples in Lecture 8, output the source and destination ports for the corresponding socket, close the connection, and and then wait for another client connection in an infinite loop. Once you get your code working, what can you say about the port numbers that the client and server output for each connection?

5. *[Lecture 10]* Download the files for the multi-threaded 'knock knock' server using the thread-per-client architecture from Lecture 10. You will also need to download the corresponding client application from Lecture 7. Note that the client and protocol are unchanged, but the `KKMultiServer.java` has been altered, and the client handler `KKClientHandler.java` is new. Launch a single `KKMultiServer`, and then run two clients simultaneously. You should now find each can communicate with the server concurrently.

[2] For the uninitiated, 'knock knock' jokes run like this: (A) Knock! Knock! (B) Who's there? (A) [something] (B) [something] who? (A) [punchline based on B's previous answer, usually not very funny].

Try to understand how the multi-thread ‘knock knock’ server works, then apply similar design principles to write a multi-threaded version of the `DailyAdviceServer.java` from Lecture 8. You will need a separate class for the client handler, but do not need to write a separate protocol class unless you would like to.

Once you have checked your code works for a single client, confirm that it can communicate with multiple clients simultaneously. This is difficult with the daily advice example as the communication is so brief – it is difficult to start the second client before the first one has already finished. Therefore, add a `Thread.sleep(5000)` command in your client handler class, after the message has been sent to the client, but before the connection is closed. Also add a print message when the client socket is being closed. This should give you enough time (5 seconds) to launch two clients near-simultaneously, and confirm that they are being dealt with concurrently by the server.