2020

SOFTENG 364: Lab 7
Java Sockets

Craig Sutherland

Contents

Overview 2
Preparation 2
A Simple Daytime Client 2
A Simple Echo Server 3
A Multi-threaded Server 5

Overview

By the end of this lab you should be able to:

1. Implement a basic client-server echo application in Java,
2. Use threads to handle multiple client.

To receive credit for completing the worksheet, please complete the Lab 7 Quiz on Canvas. You'll receive feedback immediately afterwards, and may choose to redo the quiz if you wish. This worksheet and the associated quiz comprise Lab 6 and contribute 1% to the final mark. The due date (for the quiz) is 11:59 pm, Friday, 22nd May.

Several activities on the lab worksheet are framed as "questions", but responses needn't be submitted (i.e. the on-line quiz is the only submission required). Nonetheless, please don't hesitate to speak to a member of the 364 team during the lab if you are unsure of what a suitable response might be.

Preparation

The software we need this week is installed in the Engineering computer labs. If you're working on your own PC, please install the Java SE 8u251 (https://www.oracle.com/java/technologies/javase-downloads.html) and Eclipse[1] (https://www.eclipse.org/downloads/).

There is also a resource file on Canvas that contains template code for you to use in this lab. Download the code and extract it into a folder on your machine.

You may like to make a new git repository in which to track your SOFTENG 364 lab and assignment work.

A Simple Daytime Client

For the first part of this lab, you will write a simple Java Socket client that will connect to NIST[2] time server and download the current date and time.

Next start Eclipse and create a new project:

1. Start Eclipse
2. Click on "Create a project…"
3. Select Java Project from the list and click on "Next >"
4. Type in SOFTENG364-Lab-7 as the project name and click on "Finish"
5. Click on "Don't Create"
6. Your project is now ready
7. Add the following files to the project and compile it:
   - DaytimeClient.java*: main entry point for your project*
8. Execute your program

This project should produce some output like the following:

```
Requesting time
Received message
58990 20-05-21 21:13:38 50 0 0 504.9 UTC(NIST) *
Done
```

The following are the important lines in this class that you need to understand:

[1] You can use an alternate IDE if desired (e.g. IntelliJ IDEA or Apache NetBeans). However this manual assumes you are using Eclipse, as this is installed on the lab computers.
[2] NIST is the National Institute of Standards and Technology.

```
import java.io.*;
import java.net.*;
```

These lines are importing the classes from the `java.io` and `java.net` packages. The `java.io` package contains classes for working with data streams, while the `java.net` package contains the classes for working with sockets.

```
Socket socket = new Socket(hostname, port);
```

This line initialises the new `Socket` instance and attempts to connect to a host and port. Using this construction will automatically resolve the hostname as part of the connection. The port number is required as the connection does not know what higher level application the socket is for.

```
DataInputStream dataIn=new DataInputStream(socket.getInputStream());
```

Retrieves the input stream from the socket and wraps it in a `DataInputStream` instance. While it is possible to directly use the input stream from the socket, using a `DataInputStream` simplifies working with the stream. For this example, it does not add any value but in later examples we will use other methods of this class.

```
StringBuilder msg = new StringBuilder();
while ((character = dataIn.read()) != -1) {
    msg.append((char)character);
}
```

Reads the input stream one character at a time and appends each character to a `StringBuilder` instance. The `StringBuilder` instance is a helper class that allows us to easily build a `String` instance in memory (in Java `String`s are immutable, so if you modify a `String` you end up with a copy of the original `String` with the modifications.)

After these lines, you now have a `String` containing the output from the time server that you can use. In our example, we merely write the output to a console, but other applications would parse the data and do something meaningful with it.

The other line of importance is:

```
catch (IOException e)
```

This line catches any IO errors (e.g. due to the socket failing). For example, if the socket could not connect this line would catch the error.

Questions

What is the port this example is using? Can you explain why this port is used? *Hint: look up the Daytime protocol.*

Why does the read loop check until -1 is received?

A Simple Echo Server

The Daytime client you tested shows you how to read a simple message format from a server. But the client is only half of a network application: there is also the server half. So, let's write a simple server that will read a message from an input socket and repeat the message back on an output socket.

1. Add the following files to the project and compile it:
   - EchoServer.java*: main entry point for your project*
2. Execute your program

When you read your program, you will get the following output:

```
Starting echo server
Listing in port 3333
```

And your program will remain running.

Your program is still running because it is waiting to receive some input from a client (which you haven't written yet!) When the program starts, it initialises a `ServerSocket` instance and sets the port that the socket will listen on (3333 in our example.) Unlike the client instance, this hasn't open a TCP/IP connection yet. The connection is only opened when the following line finishes execution:

```
Socket socket = serverSocket.accept();
```

This line accepts a connection from a client and returns a `Socket` instance – an instance of the same `Socket` class that the client application uses. The server now how a connection to the client and can operate using the same commands as the client can.

After connecting to the client, the echo server enters a loop where it waits for input from the client, outputs the input to the console and then returns a modified version of the input to the client. This loop in implemented in the following lines:

```
String msg = "";
while (!msg.equals("stop")) {
    msg = dataIn.readUTF();
    System.out.format("Received message '%s' from client%n", msg);
    dataOut.writeUTF("You said " + msg);
    dataOut.flush();
}
```

In the first program you executed, we had to directly read each character from the input stream. In this program, you can instead use a convenience method exposed by `DataInputStream`: `readUTF()`. Unfortunately, we can't use `readUTF()` with Daytime servers as the Daytime protocol predates UTF by a decade[3]. Instead Daytime servers use the ASCII encoding for their messages.

But since we are writing our own echo server (and a client too), we can use whatever text encodings we want – and since UTF is included on `DataInputStream` we might as well use it.

Figure 1 shows the overall process flow for an application that reads from a network socket. The flow can be broken into two distinct phases:

1. Initialising the socket and preparing to accept incoming requests
2. Handling a client connection (incoming client request)

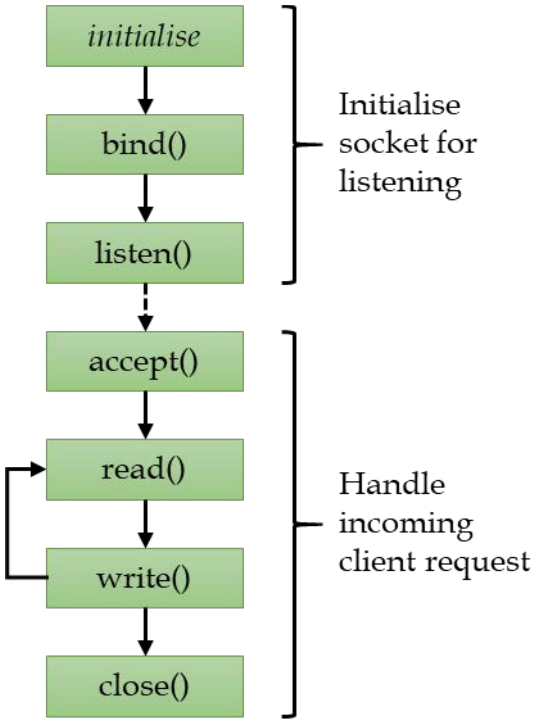

Figure 1: Server process flow.

In Java, the `ServerSocket` constructor handles both binding to a socket port and starting to listen. So we don't have to

[3] The Daytime protocol was defined in 1983 as part of the Internet Protocol suite – hence its low port number. In contrast. UTF was only proposed as a standard in 1993.

explicitly call these methods (although you can if you want to, there are other overloaded constructors for this class.) However we do have to explicitly handle accepting a connection, as well as reading and writing data.

The other important point to know with the `accept()` method is it is blocking: your application will stop and wait for a client to connect before it does anything. This is why the program has not finished execution: there is no client (yet) to connect to it and tell it to stop.

Task 1: Write an echo client

You now have all the pieces you need to write a simple echo client. Add a new class to your project called EchoClient and implement it.

The client should write some text to the server (you can choose what it says), then write the word "stop". To send data, use a `DataOutputStream` in the same way that the server does.

To initialise the connection, you will need to know the host name and port. You should know the port already (look at the server code if you don't.) For the hostname, you can use the loopback hostname: localhost.

Hint: you can use the basic code from DaylightClient.java but modify it to use `readUTF()` instead of reading each character individually.

Questions

For the Daytime client, you did not need to send any data to the server. How would you modify the echo server to behave in a similar manner?

If you change the order of write and read statements in the client program, does it change the output? For example, try doing two or more writes before your reads or interleaving writes and reads.

How many clients can the server handle at one time? How could you modify your program to handle more clients? Try to think of multiple ways you could extend your program.

What port is the server using to accept the client connection? How do you think this port is allocated? How is this port different from the server?

A Multi-threaded Server

One of the current limitations of the echo server is it can only handle a single client at a time. As soon as a client connects, it blocks the thread the server is processing on (see Figure 2). This behaviour occurs because your code can only do one thing at a time. As shown in Figure 2, as long as a client is connected to the server, it will block that server thread. While the sockets can potentially handle multiple connections at the same time, you need to add code to handle for multiple connections.

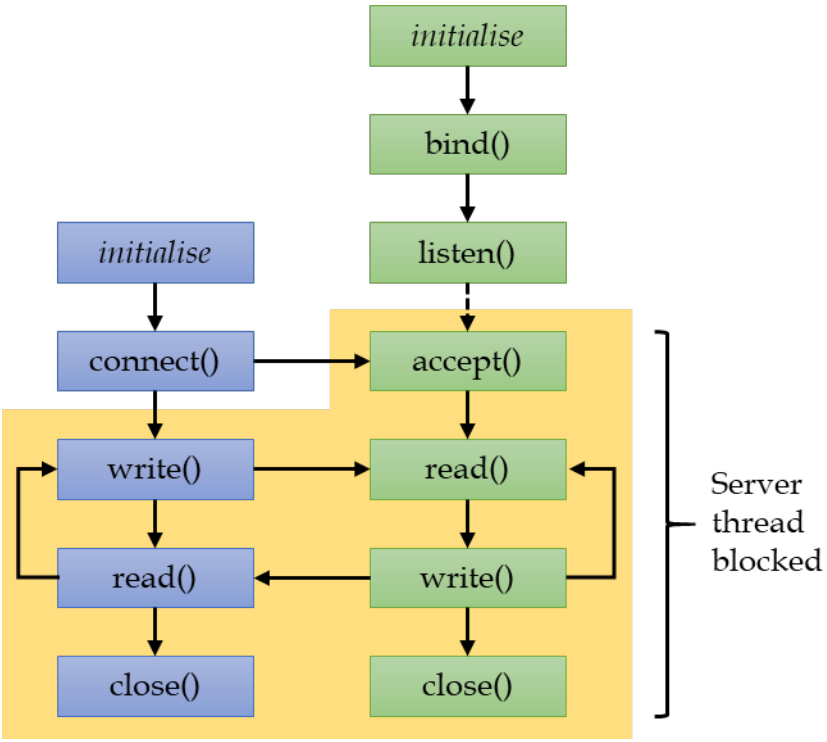

Figure 2: Relationship between the client and the server sockets.

The simple answer is to spin up a new thread every time a client connects. Open EchoServer.java and find the following two lines:

```
Socket socket = serverSocket.accept();
handleClientConnection(socket);
```

Replace them with the following code:

```
while (true) {
    Socket socket = serverSocket.accept();
    new Thread(new Runnable() {
        public void run() {
            try {
                handleClientConnection(socket);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }).start();
}
```

These lines add multithreading to your application. The code will enter the program into an endless loop. Inside the loop, the code will block and wait for a client to connect. When a client connects, it starts a new thread that executes the same processing code as the single threaded server. However, because the code is in a separate thread, it does not block the server from accepting additional clients.

Questions

What will happen when a client stops running without closing the connection first?

What are the port numbers being allocated to the clients for the multi-threaded server? How do they differ from the single thread server?

Task 2: Write a store and reply server

The current server is very limited. It only echos the messages received from a single client to the same client. Extend your application so it stores the last five messages received, from any client, and sends them to any clients that connects. You will also need to modify the client so it handles the new server.

You will need to implement a simple protocol for your store and reply server to help it work. The first message the client sends should not be stored, instead this message should be used as the name of the client. When the server is replaying messages to a client, the server should report who said the message. For example, if the original client was called Trudy, other clients would get the message "Trudy said Hello".

When a client sends a stop message, the server should respond with a done message so the client knows there are no more messages to receive.

You will also need to use locks in your program to ensure your data is not corrupted. While there are a variety of locking mechanisms in Java, one of the simplest is the `ReentrantLock` class. This class has `lock()` and `unlock()` methods that you can use to guarantee exclusivity to a resource.

For example, you could use the following code to lock and unlock a resource:

```
lock.lock();
try {
    messages[messageIndex++] = myName + " said " + msg;
}finally {
    lock.unlock();
}
```

The following is an example of potential output that your client might receive:

```
Received message 'Welcome Trudy'
Received message 'Bob said Hello'
Received message 'Jane said Hello'
Received message 'Bill said Hello'
Received message 'You said Hello'
```

Feel free to expand your server with any other functionality you want to try :-)

You have now finished lab 7. Remember to complete the Canvas quiz for this lab (https://canvas.auckland.ac.nz/courses/47894/quizzes/50657).