Lab #6: Concurrent TCP Servers in Java

Lab #6 aims for familiarize you with the programming of concurrent TCP servers sockets. To do this we will briefly introduce the Thread class, or the threads of execution, as a basic tool to achieve concurrency in Java. Later we will focus on different types of concurrent servers.

At the end of Lab #6 you should be able to:

1) Write a basic program in Java that uses threads in order to perform several tasks concurrently.
2) Explain the basic structure of a concurrent TCP server, including the distribution of the server code between the different methods of the Thread class.
3) Make a simple iterative TCP server into a concurrent TCP server that offers the same service, but to several clients at the same time.

1. Concurrent Programming in Java

In https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html you can find a tutorial about concurrent programming in Java. Below, we can see a summary of it.

In concurrent programming, there are two basic units of execution: processes and threads. In the Java programming language, concurrent programming is mostly concerned with threads.

Threads are sometimes called lightweight processes. Threads exist within a process — every process has at least one. They share the process's resources, including memory and open files. Each thread is associated with an instance of the class Thread.

There are two basic strategies for using Thread objects to create a concurrent application.

- To directly control thread creation and management, simply instantiate Thread each time the application needs to initiate an asynchronous task.
- To abstract thread management from the rest of your application, pass the application's tasks to an executor.

Lab #6 documents the use of Thread objects.

The simplest way to work with threads is to declare a subclass of the **`Thread`** class. Each object of this subclass allows to launch a new thread that will run in parallel with the existing ones. The syntax to create a new subclass from the **`Thread`** class is:

```
class Hilo extends Thread
```

The **Thread** class (and by extension the subclass created from it) always includes a **run** () method that contains the code that must be executed concurrently with the rest of the program. It also includes a constructor in which you can include the necessary code to initialize the thread after creating it.

On the other hand, it is possible to particularise the behaviour of each instantiated object of the **Hilo** class by the attributes of this specific class. These attributes are declared before the constructor of the class, they can be initialized with the parameters of its constructor and used in **run()**method.

Putting all these details together, our new class will look something like:

```
class Hilo extends Thread {
  Socket client; // attribute of the class
  public Hilo(Socket s) { // constructor of the class
    client = s; // Code to execute during initialization
 }
 public void run() {
     // Code of thread
     // Here we put the dialogue with the client
     Scanner input = new Scanner(client.getInputStream());
     ...
 }
}
```

Regarding the **run()** method, it is important to note that it is not possible to modify the prototype of the function, and therefore it is not possible to modify its header to add the throws of exceptions. This implies the need to treat the exceptions within the **run()** itself using the **try/catch** clauses. It is also not possible to pass parameters in the method call.

After having seen how to declare a subclass of the **Thread** class, let's see how to use it. The first step is to create an object of the new subclass, in our example, the **Thread** class. For example:

```
...
Hilo h=new Hilo ();
```

The previous instruction creates the thread, but does not start it. It is simply an empty **Thread** that has not yet allocated system resources. Therefore, the next step is to "start" the thread, that is, start the concurrent execution of the corresponding code. This is done by invoking the **start()** method, which will create a context for the thread,

assigning the thread the necessary resources and starting its execution. Also, the **`start()`** method will execute the code included in the **`run()`** method concurrently with the other threads in the system. Therefore, to create a thread and implement it, we need to write a code similar to the following:

```
...
Hilo h=new Hilo ();
h.start();
...
```

Also, as we have seen before, **`Hilo`** must be declared as a subclass of the **`Thread`** class and providing its own implementation of **`run()`** method.

A useful method of **`Thread`** class is the **`sleep()`** method. It causes the current thread to suspend execution for a specified period. Two overloaded versions of sleep are provided: one that specifies the sleep time to the millisecond and one that specifies the sleep time to the nanosecond. (i.e. **`sleep(long milliseconds)`** and **`sleep(long milliseconds, int nanos)`**

To get used to programming of the threads, you will do an exercise that shows you how the threads work. **Attention: the exercise DOES NOT implement a client or a server. Therefore, it will NOT use `Socket` objects.**

Exercise 1:

Write a Java program that, being a subclass of **`Thread`** class, throw three threads.

Each thread will write to the standard output (stdout) ten times the text received as parameter in its constructor.

The thread will suspend execution for 100 milliseconds between each text printed. Use the **`sleep()`** method to do it (i.e. **`sleep(100)`**).

2. Concurrent Servers

Alternatively to iterative servers, that attends only one client at time, a server can handle multiple clients at the same time in parallel. This type of a server is called a concurrent server. There are several ways we can implement concurrent servers, the simplest technique is to use **`Thread`** class. The idea is to create a service thread for each client receiving the service.

The basic design of the concurrent server, therefore, may use several threads. In the main thread, the server will remain waiting for the client connection requests. When a TCP connection request is received, the server will accept the connection and create a new thread to attend it, while the main thread of the server is still waiting to receive requests from new clients.

Assuming that we have a class descendant of the **Thread** class called **Service (Socket s)**, which serves the client through the socket **s** that is passed as a parameter. The main thread will look similar to the following code:

```
public class ConcurrentServer {
   public static void main(String argv[]) throws
                          UnknownHostException, IOException {
       int port=8000; //wellknown port of the server
       ServerSocket server=new ServerSocket(port);
       while (true) {
         Socket client=server.accept(); // waiting for a client
          // To serve a client request, it creates a Service(Socket s)object
          // passing the socket "client" as a parameter in the constructor
         Service Cl=new Service(client);
          // and start the thread that give service to the client in parallel
         Cl.start();
         } // While End
     } // Main End
} // ConcurrentServer End
```

The **Service** class would be defined as we have seen in the previous section, extending the **Thread** class. In the **run()** method of the **Service** class, we would write the Java code to carry out the dialogue with the client and provide the desired service.

Exercise 2:

Write a TCP Concurrent Echo Server that listens on port 7777.

The server must return to the client each line of text that the client sends, until string "FIN" is received. After sending back the string "FIN", the server will close the connection with the client.

Implementation:

To be able to serve several clients at the same time, the server must use several threads. The main thread, which executes the main procedure, will wait for clients on port 7777. Each time a connection is established with a TCP client, a new service thread will be launched. The connected socket with the client will be passed as a parameter to the new thread. Using this socket, the new thread will perform the dialogue with the client, until the string "FIN" is received. At that moment, the server will close the connection with the client and end the execution of that thread.

You can check the right behaviour of your server with **nc** or **telnet.**

3. Other uses of concurrency

The possibility of launching threads within the same code makes easier the implementation of other types of servers, such as multiservice servers and multiprotocol servers.

A multiservice server is capable of accepting clients through different ports, offering different services for each of them. For example, a multiservice server can provide both daytime services, which returns the system time as a String; as time, which returns the system time in seconds from January 1, 1900. These services would be offered through two different ports (the standard ports are 13 and 37, respectively; you can try them at time.nist.gov). Another use of concurrency is a multiprotocol server, which offers the same service over two transport protocols, usually TCP and UDP. A typical multiprotocol example is an echo server offering the service over TCP and over UDP.

Some clients also require concurrency for its implementation. For example, suppose a chat client. After connecting the user to a chat server, the client program must be aware of two possibilities at the same time. On the one hand, the client must collect the text typed by the user by its keyboard and send it to the server. On the other hand, the client must also wait for messages from other users sent to it by the chat server and display them on the screen. The simplest solution is to implement both functionalities in concurrent threads, which allow both situations to be managed.

Also, keep in mind that read operations are blocking. Therefore, when you execute a line code that reads from the System.in or reads from a socket, the execution of the program stops on that line until there is data to read.

Exercise 3:

Write a **Java Chat Client** program that connects to the server at TCP port indicated by your teacher. The program will require at least one additional thread to the main procedure.

You will need two threads one of them will be used to:

read from the keyboard the data that the user types and

send the data to the server through the socket,

and the other thread will be used to:

read the data that arrives from the server through the socket and

display the data on the screen.

The client will be running until the user types the "quit" command that will be transmitted to the server before terminating the client program.

Note: Remember that the Scanner class has the hasNext() method available, which will return false when it detects that the Socket is closed. This method can help you to terminate the loop that reads the lines sent by the server.

Implementation Help:

It has been assumed that there is already a socket connected to the client. One of the 2 threads may be the **`main`** procedure, although it is not essential. The other thread will have to be created.

Thread A

…

Create **`Scanner`** bound to the input stream of the **`Socket`**.

While something to read{

Read **`Socket`**.

Print to stdout.

}

Thread B

…

Create **`PrintWriter`** bound to the output stream of the **`Socket`**.

Create **`Scanner`** bound to the keyboard.

While ¡quit {

Read from keyboard.

Send read line out to the server. (Remember to do **`flush`**).

}

Close **`Socket`**.