Lab 10: Sockets
12:00 PM, Apr 4, 2018

Contents

1 The Client-Server Model 1
  1.1 Constructing Java Sockets 2
  1.2 Communicating via Java Sockets 2

2 Client 3

3 Server 5

Objectives

By the end of this lab, you will know:

- How the client-server model interacts
- All about Java’s built-in sockets

By the end of this lab, you will be able to:

- Write a client that connects to the TAs’ server
- Write your own server, and connect your client to it

Run this lab from the command line!

In this lab, you’ll be creating and running both a client and a server. These have to respond to \texttt{CTRL-D}, but Eclipse can be a little funky with \texttt{CTRL-D}, so we encourage you to run this from the command line instead, should you encounter an issue.

To compile your code cd into the directory containing your bin, sol, and src directories. Run the following:

```
scalac -d bin */lab10/*/*.scala
```

This will create binaries in your bin file. To run this code, cd into the bin directory and run the following:

```
scala lab10.sol.[NAME OF OBJECT/CLASS HERE] [ARGUMENTS HERE]
```

When you get to testing your client and server, you should be running these from separate terminals, and you should run your server first so that it can accept your clients connection!
1 The Client-Server Model

So far in CS 18, you’ve explored with how different parts of one single program can communicate with one another. In this lab, you’ll learn how different programs, possibly running on different machines, can communicate with one another, through the **client-server model**. Most modern programming languages support this model, or some similar abstraction.

To understand the client-server model, imagine placing a telephone call to a customer service center. The client is the person placing the call. The server is the customer service center. The client asks questions or requests information, and is the active entity. The server responds with information, thus serving the client. The server is a passive entity– the customer service agent doesn’t offer to help with something the client didn’t ask for.

Computer programs can also communicate in this fashion, with some more active entities (clients), and other more passive entities (servers). In this lab, you will learn about Java’s support for the client-server model, namely *sockets*. Scala does not have its own implementation of sockets, so we’ll be relying on Java sockets for both this lab and your two remaining projects.

1.1 Constructing Java Sockets

**Sockets** are endpoints in a two-way communication channel. The idea is this: you connect two sockets together, and each end of the connection can both send and receive data.

In this lab, you’ll be using the Java classes `Socket` and `ServerSocket`. There are many `Socket` constructors. The one you will be using today is `Socket(host: String, port: Int)`. This will:

- Construct a `Socket` and
- Connect that `Socket` to a `ServerSocket` running on the specified host and port.

The word **host** in this context refers to the server. So, if you’re running a server on the machine `cslab1a`, then you would specify `cslab1a` as the host in the `Socket` constructor. Or, if you are running a server on the *same machine* as your client, you would use either your machine’s name, or `localhost`.

A host provides **ports**, each of which is a communication endpoint corresponding to a *unique* client. No two clients can have the same port number on the same server. A server listens on a port to wait for an incoming connection on that port number. To service multiple clients, a server listens on multiple port numbers. On the other end, client Sockets can connect to the same server on different ports. Together, this allows multiple clients to interact with a single server without interfering with each other.

Following the earlier call center analogy, a port could be represented by a single customer service agent: each agent can only listen to one Client at a time.

To construct a `ServerSocket`, you will use the `ServerSocket(port: Int)` constructor. For a `ServerSocket` to connect with a client `Socket`, both must specify the same port number.

Today in lab, you will write two programs: a client and a server.
1.2 Communicating via Java Sockets

As previously mentioned, one can both send and receive data from both a Socket and a ServerSocket. To service this, every Socket has an InputStream and an OutputStream. You have already worked with these before! System.in, which reads from the keyboard, is an InputStream, and System.out, which prints to the console, is an OutputStream.

Imagine two programs, A and B, that are communicating via a socket. Program A can write to its OutputStream anything it wants to send to program B. Furthermore, program A can read on its InputStream anything that program B writes to program B’s own OutputStream.

![Diagram](A output/B input

A input/B output)

To obtain these streams from the Socket, use the methods `getInputStream` and `getOutputStream`. For example, if `socket` is a Socket, then:

```java
val iStream = socket.getInputStream
val oStream = socket.getOutputStream
```

When you’re done sending data, you must call the `shutdownOutput` method on `oStream` to close the OutputStream and send an EOF (end of file). Similarly, when you’re done listening for data, you must call `shutdownInput` on `iStream`.

Now, you might be saying, how exactly can one send and read this data?

To send data through an OutputStream, you should create a BufferedWriter, then use the `write` method, then flush the writer. Suppose `oStream` is an OutputStream we want to write to:

```java
BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(oStream));
writer.write(<something>);
writer.flush();
```

To receive data from InputStream, you can equivalently use a BufferedReader. Suppose `iStream` is an InputStream we are reading from:

```java
BufferedReader reader = new BufferedReader(new InputStreamReader(iStream));
reader.read();
```

Tip: Think of reading and writing to a stream exactly like reading and writing to a File, as seen in your previous labs and assignments!
2 Client

Now that you know all about the client-server model, how to connect two programs through Sockets, and how to communicate through these Sockets, it’s time to write a client!

Ash NEEDS YOUR HELP! He needs to call customer support in order to reload his bus pass. He needs you to help him setup a communication channel with RIPTA. Start by making a client for Ash.

Task: Create a file named Client.scala, containing a class called `Client(host: String)`, where host is the hostname. Your file should contain an object `Client` with a main method that runs your client. Your class should also contain a run method which contains the full functionality, specified below. Its signature should be:

```scala
@throws(classOf[IOException])
def run() {
  // your code here
}
```

The `@throws` annotation is the equivalent of putting throws into a Java method signature. Because this method throws an `IOException`, you will need to catch it in your main method.

Task: Write an interactive client that performs the following functionality:

1. Connects to our provided server, then does the following until the user types a null:

2. Reads user input on `System.in`

3. Sends this input to the server

Once the user sends a null the client should:

4. Read in the server’s response until it reaches EOF

5. Print the server’s response to `System.out` (the console)

Note: Unlike in other assignments, you should not use `readLine`. Instead, use `BufferedReader`'s `read` method. This reads one character at a time.

Tips/Roadmap:

- Create a Socket using the `Socket(host: String, port: Int)` constructor to connect your client to the server. You should specify the port as a constant:

  ```scala
  private val Port = 1818
  ```

- Port numbers should be 4 digits long and greater than 1024. If you experience an error that indicates the port number is already in use, try a different port number.

- Create all your necessary readers and writers:

  1You can find the Javadocs for `readhere`
- A reader for `System.in`
- A writer for your client to write to the `Socket`
- A reader for your client to read from the `Socket`

- Remember to use try/finally to handle resources like BufferedReader, BufferedWriter, and Sockets.
- Your main method should take the hostname as input, to pass into your instance of the client!
- You should use `System.out.write` to print to the console! Because you are using `read`, you will be reading everything as an int, so using `println` will read the characters as integers, not as characters. Remember to flush the output using `System.out.flush` when you are done writing to it.
- In your first loop:
  - Read data input from the user on `System.in`
  - Send this input to the server through the socket’s `OutputStream` (using `write` and `flush`)
- Once you read a `null` from the user, the client should disconnect by shutting down output.
- After the client has disconnected, your second loop should:
  - Read the server’s response from the socket’s `InputStream`
  - Print this response to the console
- Once the server’s response has been fully printed back to the user, the program should terminate!
  - Don’t forget to close all of your various readers and writers.

**Task:** During this lab, the TAs will run an `echo` server that reads whatever input is sent, then sends the same input back. At the beginning of the lab, the TAs will announce their server’s host and port number. Test your client by connecting to this server!

**Note:** The server won’t echo the input until you send an `EOF` (ctrl + d).

**Task:** The TAs will also be running a mystery server! Try it out by sending it some text and seeing how it responds.

You’ve reached a checkpoint! Please call over a lab TA to review your work.

### 3 Server

We need some way for the RIPTA customer support agent to hear Ash. Create a server so that the customer support agent can hear Ash’s requests.

\(^2\text{You can find the Javadocs for `write` here}\)
Now that you’ve written your first client, it’s time to write your own server! As mentioned before, a server can service multiple clients. This means that the server shouldn’t terminate when a client terminates or disconnects—it should keep going. This is done through a deliberate infinite loop. During each iteration of this loop, the following should happen:

Task: Create a file named Server.scala, containing a class called Server(). Your file should contain an object Server with a main method that runs your client. Remember you can do extends App to avoid writing a main method! Your class should also contain a run method which contains the full functionality, specified below. Its signature should be the same as Client.scala’s run method.

1. Wait for a Socket to connect, using the ServerSocket’s accept method. Once a Socket connects, the accept method will return that Socket. Save it. The server will use this Socket to send and receive data to (and from) the client.

2. Once you’ve established a connection, your server will need to receive all the data sent to it by the client, and send a response back. So, you should read data on the Socket’s InputStream, and write a response to the Socket’s OutputStream. As when creating your client, we suggest you use a BufferedReader and BufferedWriter to accomplish this.

3. Flush the output, then call the shutdownInput and shutdownOutput methods on the Socket. Note that these are not called on the ServerSocket, but rather, the Socket representing the connection between the server and a particular client.

In summary, this loop will accept a client’s request, respond to it, then close the associated Socket and wait for another client.

Task: Write a server that does the following:

1. Receives text from a client.
2. Prints the number of characters it received to System.out.
3. Sends the same text it received back to the client.

Tips/Roadmap:

- Create a ServerSocket using the ServerSocket(port: Int) constructor with the same port number that you used for your client
- Be sure to write the infinite loop described above, but with the additional step that the received input is also printed to System.out
- As always, don’t forget to close any readers or writers you open!

Task: Use your client to test your server. To do this, start running your server. Once that is running, run your client (being sure to connect it to the right host and port) and start typing into the client’s console.

Just for Fun: Have your friends and neighbors run their servers and try to connect to them as well. (But don’t worry: your server does not need to handle multiple clients simultaneously. Multithreading is required for that functionality—stay tuned for CS 33!)

Note: This “Just for Fun” task is not required to exit the lab early.
Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: https://cs.brown.edu/courses/cs018/feedback