Week 7 Lab - Introduction to TCP

COSC244

1 Introduction

A computer network is an interconnected collection of computers and other devices which are able to exchange information. Machines which make up a network are often called *nodes*. If a node is an autonomous computer, it is often called a *host*.

During the next few labs, we will be writing programs which can send information between multiple hosts on a network. You can send information between two terminal windows on the same machine. You can also communicate with a friend in the lab (once both of your programs are functioning).

2 Assessment

This lab is worth 1%. The marks are awarded for the written answers to the preparation questions (worth 0.5%) and completing two programming exercises (worth 0.5%).

The preparation questions should be answered **before** coming to the lab. They should be stored electronically in a plain text file with a .txt suffix in your ∼/244/07 directory. They will be checked by the demonstrators at the start of the lab.

3 Introduction to Networks

In lectures, you were introduced to the OSI layered protocol model which consists of seven layers. The Internet (which is the network we are using) has four layers. They are shown in Figure **??**. The Internet model combines the OSI Application, Presentation and Session layers into the Application layer. Usually we consider the OSI Physical and Datalink layer to be the Host-to-Network layer.

The reason we separate networks into layers is to make changes and modifications easier. Each layer performs a separate function and can have a design independent of other layers. As long as a layer meets the interface specifications of the layers immediately above and below it, its internal details can change without other layers being affected. This is similar to the concept of modular program design.

3.1 Protocols

The most widely used Network layer protocol is the Internet Protocol (IP). You will learn more details about it in lectures. For now it is sufficient to say that IP specifies a datagram service between nodes. Delivery is not guaranteed, corruption may occur in transit, packets may arrive in a different order, and individual packets may travel different routes to their destination.

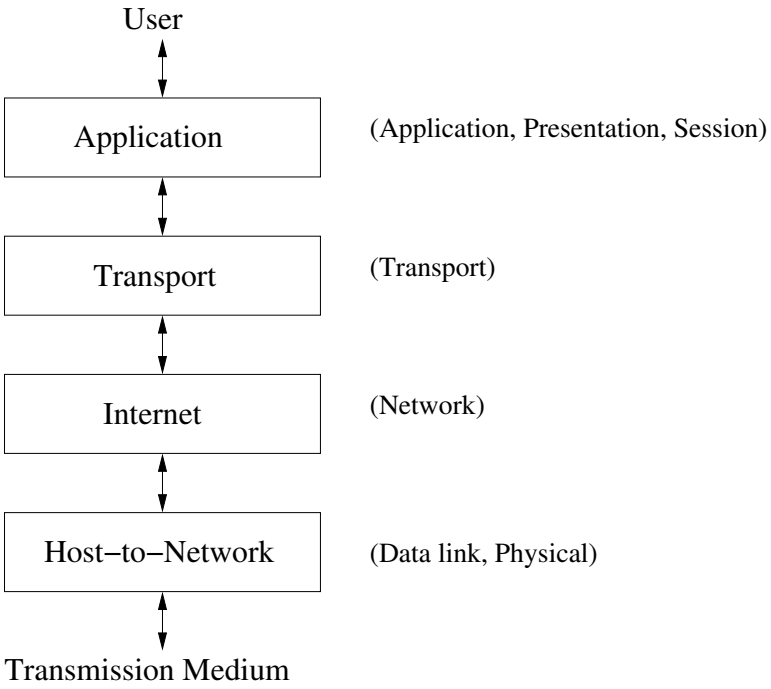

Figure 1: TCP/IP Network Layers

The Transport layer protocol we use in the labs is called the Transmission Control Protocol (TCP). It is known as a *reliable* protocol because it guarantees the order and integrity of packets delivered. In lectures you will learn about another Transport layer protocol called User Datagram Protocol (UDP) which is known as an *unreliable* protocol. It provides no guarantees about the delivery of packets or their arrival order.

3.2 Addresses

Every node, connected to an IPv4 (Internet Protocol version 4) network has a unique four byte address. This address is represented in *dotted-quad* format. For example, the email server (chasm.otago.ac.nz) has the IP address 139.80.32.68. Nodes on the Internet also have an IP name. The email server's IP name is *chasm.otago.ac.nz*. The form of this name is called the fully-qualified name. You can use this name to get to the machine from anywhere. If you are within the university network, you can refer to it simply as *chasm*.

A node's IP name and IP address are tied together. In lectures, you will learn how this is accomplished. The main reason for IP names is for us humans. It is easier to remember chasm.otago.ac.nz than to remember 139.80.32.68.

3.3 The Lab Machines

Each of the machines in the Linux lab has an IP name and an IP address since they are connected to the network. The names of the machines are of the form, *oucsNNNN*, where *NNNN* is a 4 digit number. You can determine the IP name of the machine you are using in one of two ways:

- On the top right of the screen, there is a sticker with the name of the computer printed on it.

- At the command line, type `hostname`. Linux will respond with the fully-qualified name of your machine.

Remember if you are on the university network, you only need the first part of the name. All computers have a special name called *localhost* which refers to itself. Your computer is *localhost* to you; your friend's computer is *localhost* to her or him. If you want to communicate with your friend's computer, you must refer to it by its *oucsNNNN* name.

3.4 Ports

While it is necessary to know the IP name to communicate with another computer, that is not sufficient to complete the plot. At any time, you might have one or more web browser windows open, maybe some chat sessions, maybe *ssh* or *sftp* sessions, or other programs communicating with remote computers. Just delivering a packet to your computer does not tell the operating system what to do with it. We need a way to specify what application to give the packet to.

The way to specify the application is by means of a *port* number. Each instance of an application which communicates via the network will have a unique port number. These numbers range from 1 to 65535. They are logical constructs only. They are not physical entities like serial or parallel ports. The operating system associates a network application to a given port number.

More details about port numbers will be covered in lectures. What you need to know for the labs is that you can use ports 7770 - 7779. Other port numbers are reserved for *well-known* applications or are blocked for security reasons.

3.5 End-to-End Communications

We will concentrate on communications between applications you write, so we will work at the application layer. The application for this lab allows one side to type messages to the other side. Modifications will be made to the application in subsequent labs to enhance its capabilities. Ultimately you will end up with a basic chat program.

The combination of a port number and an IP address makes up what is called a *socket*. A socket is an abstraction of an endpoint for sending and receiving data. See Figure **??**. The network and operating systems transfer packets between the socket for the application program running on *Host1* and the socket for the application program running on *Host2* and vice versa.

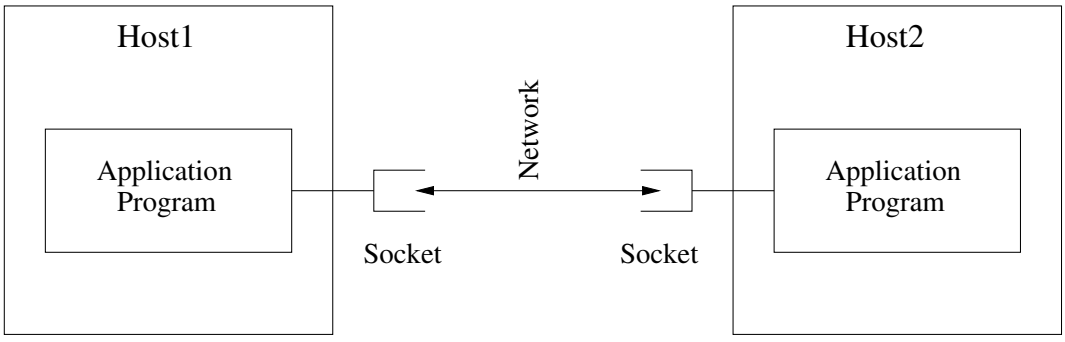

Figure 2: Applications Communicating via Sockets

There is a certain startup sequence that must be followed for an application on two different hosts to communicate. One host's application must start first, and it must use a port known

to the other host's application. Then the second host's application starts and specifies the port number chosen by the first host.

Referring to Figure **??**, let's assume the application on *Host1* starts first and specifies port 7772 on startup. At this point in time, the application will be *listening* on port 7772 for an application to connect to it. It just sits there patiently listening (and computers can be very patient).

Assume a user on *Host2* wants to communicate with *Host1*. The user on *Host2* starts the application and specifies the IP name of *Host1* and the port number of the application (in this case 7772). 7772 is the number of the port on *Host1*. The port number that *Host2*'s application actually uses is chosen by the operating system. There are ways you can find out the port number, but it is not important for the two applications to communicate.

There can be a variation of the above scenario. Instead of communicating between two different hosts, it is possible for the applications to communicate when running in two terminal windows on the same machine. That is what we will do in this lab. It is also what you will do as you create and debug your programs. To run both sides on the same machine, simply specify *localhost* as the IP name when you start the second application.

4 Preparation

Let's begin by looking at 5 major classes which you will be using for communication: *InputStreamReader, BufferedReader, PrintWriter, Socket, and ServerSocket.*

4.1 InputStreamReader

Open the course web page, click on *Resources*, click on *Java API Documentation*, click on *InputStreamReader* in the package *java.io*.

Q1 What is this class meant for?

Q2 What does the explanation say to do for best efficiency? Write the suggested statement.

4.2 BufferedReader

Click on *BufferedReader*. You know from reading about *InputStreamReader* that this class makes reading character strings more efficient.

Q3 What method do you use to read a line from an input stream? What does it return?

4.3 PrintWriter

Click on *PrintWriter*. This class allows one to print formatted representations of objects to a text-output stream. When we are sending data over the network, our output stream will be a socket.

Look at the constructor with the following form:

```
PrintWriter(OutputStream out, boolean autoFlush)
```

If autoFlush is set to *true*, output will be done when one of the *println*, *printf*, or *format* methods is invoked, rather than only when *flush()* or *close()* is called.

Q4 What does the method, *println(String x)* do? What does it return?

4.4 ServerSocket

The *Socket* and *ServerSocket* classes are in the package called *java.net*.

In the upper left of the web browser, click on *java.net*. Then click on *ServerSocket*. Read the first paragraph. More will be said about client-servers later. For now, you need to know that the server is the program that must start first. Once it starts, it waits for a 'client to start and connect to it.'

Q5 What does the constructor *ServerSocket(int port)* do?

Q6 What does the method *accept()* do? What does it return?

4.5 Socket

Now click on *Socket*. Notice this class is for client sockets. The client starts after the 'server' starts. Remember from section 3.5 the client program must specify the IP name and port number of the server.

Q7 Which constructor do you think is the most appropriate?

Q8 What does the method *getInputStream()* return?

Q9 What method would you use to have a *socket* use an *OutputStream*?

4.6 Date

The first program you write in the lab will be a modification of a program we provide. You need to be able to get the current date and time. To do so, you can use the *java.util.Date* class.

In the upper left of the web browser, click on *java.util*. Then click on *Date*.

Look at the default constructor *Date()*.

Q10 When you use the default constructor to create an instance of *Date*, what is it initialized with?

Q11 Write the code fragment required to print the current date and time to standard output.

4.7 TCPExample Program

We are providing a single program that performs both the client and server functions. In this case, the 'client' sends text to the 'server' which receives it and prints it on the screen. We may refer to the client as the sender and the server as the receiver. Which function the program performs is determined by the command line parameters used to start the program. The program is called *TCPExample*. The server (receiver) is started by typing:

```
java TCPExample 7777
```

This command tells the server to start up and begin listening on port 7777 for a client to connect.

When running on the same machine as the server, the client (sender) is started by typing:

```
java TCPExample 7777 localhost
```

This command tells the client to start up and to connect to whatever application is listening on port 7777 of the local machine. In this case, that application is the expected server.

The port number can be any number 7770 - 7779. Both the client and server must use the same port number.

The source code listing for the program is attached to the end of this document. Let's examine the program.

Line 2 imports the *java.net* classes which includes *Socket* and *ServerSocket*. Line 7 declares *output* to be *PrintWriter*. It will be used to write data to a socket. Line 8 declares *input* to be *BufferedReader*. It will be used to read data from a socket.

Lines 10 - 14 are the constructor for the *TCPExample* class. It is passed a *Socket*. Line 11 sets *output* to be a socket output stream with a *PrintWriter* wrapped around it. Line 12 sets *input* to be a socket input stream with a *BufferedReader* wrapped around it.

Skip down to the *main()* method beginning on line 31. Since some network functions may fail, we need to use a try-catch block. This also takes care of the possibility the user did not

use the correct parameters on the command line. Line 32 declares a variable called *socket* to be an instance of *Socket*. Line 34 initialises *port* to the first command line parameter.

The *if-else* statement determines if the program is starting as a client (sender) or a server (receiver). First let's consider the server code first which is lines 41-46. Line 41 declares *serverSocket* to be an instance *ServerSocket* attached to the port number entered on the command line. Line 42 outputs a message that the receiver is waiting for someone to connect to it. Line 43 calls the *accept()* method. The program waits at this point for a client to connect to it.

Once a client connects, the *accept()* method returns. Since *accept()* returns a *Socket* it is assigned to *socket*, and the program continues. Line 44 prints a message that it accepted a connection on the port. Line 45 creates an instance of the *TCPExample* class called *example*. Line 46 invokes the *startReceiving()* method.

Lines 16-21 are the *startReceiving()* method. It is simply a *while* loop that invokes the *readLine()* method on *input* and outputs what it read to *System.out*. Remember *input* is attached to our socket.

Let's return to *main()*. If the user enters a port and an IP name, the *if* part of the *if-else* statement is executed. This is lines 36-39. Line 36 declares *socket* to be a *Socket* connected to the host at the IP name and port number specified on the command line. Assuming the connection was made, line 37 outputs text stating the host name and port connected to.

Line 38 creates an instance of the *TCPExample* class called *example*. Line 39 invokes the *startSending()* method.

Lines 23-29 are the *startSending()* method. Line 24 creates a *Scanner* to read data from the keyboard. Line 25 outputs a prompt. Lines 26-27 are a *while* loop that reads a line from the keyboard and outputs it to the socket.

The program is terminated by typing ctrl-D at the client (sender) end.

If there is an exception when first running the program, it is most likely that the user did not run the program correctly from the command-line.

Q12 In the Usage error message, what is meant by `<port> [host]`?

5 Programming Exercises

5.1 Try It Out

Type the program into a file and then save it. Compile the program by typing:

```
javac TCPExample.java
```

Make sure you have two terminal windows open and `cd` both of them to the directory you are working in. Start the server by typing:

```
java TCPExample 7770
```

You can use any number 7770 - 7779. The message, 'Waiting for someone to connect' should appear. The program is waiting for someone to connect to it.

In the other terminal window, start the client by typing:

```
java TCPExample 7770 localhost
```

The client will respond that it connected and output a prompt. The server will respond that it accepted a connection.

Type some lines into the client and they will appear in the server's window. Terminate the programs by typing ctrl-D in the client window.

5.2 Program 1

Once you are comfortable running and using the program, copy *TCPExample.java* to another file. Make the following modifications.

- Modify the receiver part of the program to send the current date and time back to the sender whenever it receives a line of text.
- Modify the sender part of the program to read and print out the date and time it gets sent.

You cannot test the program until you have made both modifications.

HINT: Your program already contains code which is able to send data to and receive data from a socket.

5.3 Program 2

Copy *Program 1* to another file. Modify this program so the receiver can read a line from standard input and send it instead of the date and time.

At this point your program can perform primitive communications between two different computers. In order for this to work however, you must alternate the sending of messages.

Make sure you get your work signed off by a demonstrator before leaving the lab.

To be continued in the next lab...

Source code listing of TCPExample.java.

```
1  import java.io.*;
2  import java.net.*;
3  import java.util.*;
4
5  public class TCPExample {
6
7     private PrintWriter output;
8     private BufferedReader input;
9
10    public TCPExample(Socket socket) throws Exception {
11       output = new PrintWriter(socket.getOutputStream(), true);
12       input =
13          new BufferedReader(new InputStreamReader(socket.getInputStream()));
14    }
15
16    public void startReceiving() throws Exception {
17       String line;
18       while ((line = input.readLine()) != null) {
19          System.out.println(line);
20       }
21    }
22
23    public void startSending() {
24       Scanner stdin = new Scanner(System.in);
25       System.err.println("Please enter data here");
26       while (stdin.hasNextLine()) {
27          output.println(stdin.nextLine());
28       }
29    }
30
31    public static void main(String[] args) {
32       Socket socket = null;
33       try {
34          int port = Integer.parseInt(args[0]);
35          if (args.length > 1) {
36             socket = new Socket(args[1], port);
37             System.err.println("Connected to " + args[1] + " on port " + port);
38             TCPExample example = new TCPExample(socket);
39             example.startSending();
40          } else {
41             ServerSocket serverSocket = new ServerSocket(port);
42             System.err.println("Waiting for someone to connect");
43             socket = serverSocket.accept();
44             System.err.println("Accepted connection on port " + port);
45             TCPExample example = new TCPExample(socket);
46             example.startReceiving();
47          }
48       } catch (Exception e) {
49          e.printStackTrace();
50          System.err.println("\nUsage: java TCPExample <port> [host]");
51       }
52    }
53 }
```