CSC3002F–Networks Assignment 2015

Lecturer: Dr. Maria Keet
email: mkeet@cs.uct.ac.za
Department of Computer Science
University of Cape Town, South Africa

This assignment consists of two parts, being about TCP socket programming and network sniffing.

In the first part, you will explore sockets and implement a chat server and client in two ways and compare them. It won't look as fancy as, say, WhatsApp or the Facebook Chat, but you will get a taste of what's happening behind the scenes of such applications.

The second part makes you look into what is actually sent around on the transmission medium. You may have heard of IP packets, but what exactly is 'inside' it, or sending HTTP request, but how is that put across the transmission medium? The Wireshark tool will be used for that. There is a separate Wireshark intro pdf file you may want to read through, which give a little overview, and there are there's the manual online. As they keep on changing things and there are slight differences across OSs, it may be that the screenshots further below in this document aren't exactly the same as what you'll see, but worry not, the questions can be answered.

The last page of this document lists what you have to submit.

Contents

1 Multi-threaded Client/Server Applications—Sockets Programming in Java

1.1 What is a socket?

A socket is the one end-point of a two-way communication link between two programs running over the network. Running over the network means that the programs run on different computers, usually referred as the local and the remote computers. However one can run the two programs on the same computer. Such communicating programs constitutes a client/server application. The server implements a dedicated logic, called **service**. The clients connect to the server to get served, for example, to obtain some data or to ask for the computation of some data. Different client/server applications implement different kind of services.

To distinguish different services, a numbering convention was proposed. This convention uses integer numbers, called port numbers, to denote the services. A server implementing a service assigns a specific port number to the entry point of the service. There are no specific physical entry points for the services in a computer. The port numbers for services are stored in configuration files and are used by the computer software to create network connections.

A socked is a complex data structure that contains an internet address and a port number. A socket, however, is referenced by its descriptor, like a file which is referenced by a file descriptor. That is why, the sockets are accessed via an application programming interface (API) similar to the file input/output API. This makes the programming of network applications very simple. The two-way communication link between the two programs running on different computers is done by reading from and writing to the sockets created on these computers. The data read from a socked is the data wrote into the other socket of the link. And reciprocally, the the data wrote into a socket in the data read from the other socket of the link. These two sockets are created and linked during the connection creation phase. The link between two sockets is like a pipe that is implemented using a stack of protocols. This linking of the sockets involves that internally a socket has a much more complex data structure, or more precisely, a collaboration of data structures. Thus, a socket data structure is more than just an internet address and a port number. You have to imagine a socket as a data structure that contains at least the internet address and the port number on the local computer, and the internet address and the port number on the remote computer.

1.2 How is a network connection created?

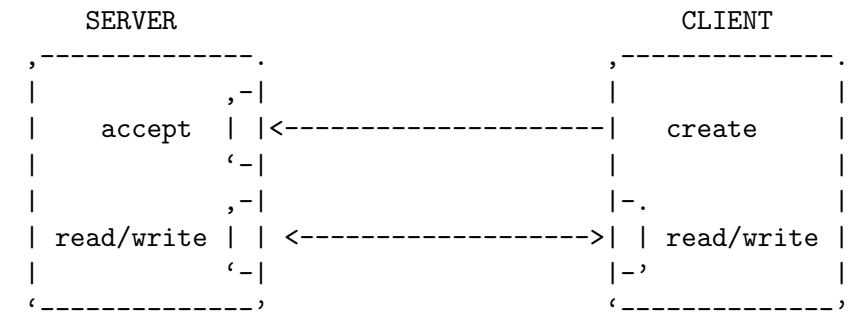
A network connection is initiated by a client program when it creates a socket for the communication with the server. To create the socket in Java, the client calls the **Socket** constructor and passes the server address and the the specific server port number to it. At this stage the server must be started on the machine having the specified address and listening for connections on its specific port number.

The server uses a specific port dedicated only to listening for connection requests from clients. It can not use this specific port for data communication with the clients because the server must be able to accept the client connection at any instant. So, its specific port is dedicated only to listening for new connection requests. The server side socket associated with specific port is called

server socket. When a connection request arrives on this socket from the client side, the client and the server establish a connection. This connection is established as follows:

1. When the server receives a connection request on its specific server port, it creates a new socket for it and binds a port number to it.
2. It sends the new port number to the client to inform it that the connection is established.
3. The server goes on now by listening on two ports:
   - it waits for new incoming connection requests on its specific port, and
   - it reads and writes messages on established connection (on new port) with the accepted client.

   The server communicates with the client by reading from and writing to the new port. If other connection requests arrive, the server accepts them in the similar way creating a new port for each new connection. Thus, at any instant, the server must be able to communicate simultaneously with many clients and to wait on the same time for incoming requests on its specific server port. The communication with each client is done via the sockets created for each communication.



The **java.net** package in the Java development environment provides the class **Socket** that implements the client side and the class **serverSocket** class that implements the server side sockets.

The client and the server must agree on a protocol. They must agree on the language of the information transferred back and forth through the socket. There are two communication protocols:

- stream communication protocol
- datagram communication protocol

The stream communication protocol is known as TCP (transfer control protocol). TCP is a connection-oriented protocol. It works as described in this document. In order to communicate over the TCP protocol, a connection must first be established between two sockets. While one of the sockets listens for a connection request (server), the other asks for a connection (client). Once the two sockets are connected, they can be used to transmit and/or to receive data. When we say "two sockets are connected" we mean the fact that the server accepted a connection. As it was explained above the server creates a new local socket for the new connection. The process of the new local socket creation, however, is transparent for the client.

The datagram communication protocol, known as UDP (user datagram protocol), is a connectionless protocol. No connection is established before sending the data. The data are sent in a packet called datagram. The datagram is sent like a request for establishing a connection. However, the datagram contains not only the addresses, it contains the user data also. Once it arrives to the destination the user data are read by the remote application and no connection is established. This protocol requires that each time a datagram is sent, the local socket and the remote socket addresses must also be sent in the datagram. These addresses are sent in each datagram.

The **java.net** package in the Java development environment provides the class **DatagramSocket** for programming datagram communications.

UDP is an unreliable protocol. There is no guarantee that the datagrams will be delivered in a good order to the destination socket. For, example, a long text, split in several pages and sent one

page per datagram, can be received in a different page order. On the other side, TCP is a reliable protocol. TCP guarantee that the pages will be received in the order in which they are sent.

When programming TCP and UDP based applications in Java, different types of sockets are used. These sockets are implemented in different classes. The classes **ServerSocket** and **Socket** implement TCP based sockets and the class DatagramSocket implements UDP based sockets as follows:

- Stream socket to listen for client requests (TCP): the class **ServerSocket**.
- Stream socket (TCP): the class **Socket**.
- Datagram socket (UDP): the class **DatagramSocket**.

This document shows how to program TCP based client/server applications. The UDP oriented programming is not covered in document.

1.2.1 Opening a socket

The client side When programming a client, a socket must be opened like below:

```
Socket MyClient;
MyClient = new Socket("MachineName", PortNumber);
```

This code, however, must be put in a **try/catch** block to catch the **IOException**:

```
Socket MyClient;
try {
    MyClient = new Socket("MachineName", PortNumber);
  }
  catch (IOException e) {
    System.out.println(e);
  }
```

where

- **MachineName** is the machine name to open a connection to and
- **PortNumber** is the port number on which the server to connect to is listening.

When selecting a port number, one has to keep in mind that the port numbers in the range from 0 to 1023 are reserved for standard services, such as email, FTP, HTTP, etc. For our service (the chat server) the port number should be chosen greater than 1023.

The server side When programming a server, a server socket must be created first, like below:

```
ServerSocket MyService;
try {
    MyServerice = new ServerSocket(PortNumber);
  }
  catch (IOException e) {
    System.out.println(e);
  }
```

The server socket is dedicated to listen to and accept connections from clients. After accepting a request from a client the server creates a client socket to communicate (to send/receive data) with the client, like below :

```
Socket clientSocket = null;
try {
    serviceSocket = MyService.accept();
  }
  catch (IOException e) {
    System.out.println(e);
  }
```

Now the server can send/receive data to/from the clients. Since the sockets are like the file descriptors the send/receive operations are implemented like read/write file operations on the input/output streams.

1.2.2 Creating an input stream

On the client side, you can use the **DataInputStream** class to create an input stream to receive responses from the server:

```
DataInputStream input;
try {
   input = new DataInputStream(MyClient.getInputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

The class **DataInputStream** allows you to read lines of text and Java primitive data types in a portable way. It has several read methods such as **read**, **readChar**, **readInt**, **readDouble**, and **readLine**. One has to use whichever function depending on the type of data to receive from the server.

On the server side, the **DataInputStream** is used to receive inputs from the client:

```
DataInputStream input;
try {
     input = new DataInputStream(serviceSocket.getInputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

1.2.3 Create an output stream

On the client side, an output stream must be created to send the data to the server socket using the class **PrintStream** or **DataOutputStream** of **java.io** package:

```
PrintStream output;
try {
   output = new PrintStream(MyClient.getOutputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

The class **PrintStream** implements the methods for displaying Java primitive data types values, like **write** and **println** methods. Also, one may want to use the **DataOutputStream**:

```
DataOutputStream output;
try {
   output = new DataOutputStream(MyClient.getOutputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

The class **DataOutputStream** allows you to write Java primitive data types; many of its methods write a single Java primitive type to the output stream.

On the server side, one can use the class **PrintStream** to send data to the client.

```
PrintStream output;
try {
   output = new PrintStream(serviceSocket.getOutputStream());
}
catch (IOException e) {
   System.out.println(e);
}
```

1.2.4 Closing sockets

Closing a socked is like closing a file. You have to close a socket when you do not need it any more. The output and the input streams must be closed as well but before closing the socket.

On the client side you have to close the input and the output streams and the socket like below:

```
try {
   output.close();
   input.close();
   MyClient.close();
}
catch (IOException e) {
   System.out.println(e);
}
```

On the server you have to close the input and output streams and the two sockets as follows:

```
try {
   output.close();
   input.close();
   serviceSocket.close();
   MyService.close();
}
catch (IOException e) {
   System.out.println(e);
}
```

Usually, on the server side you need to close only the client socket after the client gets served. The server socket is kept open as long as the server is running. A new client can connect to the server on the server socket to establish a new connection, that is, a new client socket.

2 A simple client/server application

We present a simple client/server application in this section that shows communication between the server and the client.

2.0.5 The client

This is a simple client which reads a line from the standard input and sends it to the echo server. The client keeps then reading from the socket till it receives the message "`Ok`" from the server. Once it receives the "`Ok`" message then it breaks.

```
//Example 23

import java.io.DataInputStream;
import java.io.PrintStream;
import java.io.BufferedInputStream;
import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;

public class Client {
  public static void main(String[] args) {

    Socket clientSocket = null;
    DataInputStream is = null;
    PrintStream os = null;
    DataInputStream inputLine = null;

    /*
     * Open a socket on port 2222. Open the input and the output streams.
     */
```

```
    try {
      clientSocket = new Socket("localhost", 2222);
      os = new PrintStream(clientSocket.getOutputStream());
      is = new DataInputStream(clientSocket.getInputStream());
      inputLine = new DataInputStream(new BufferedInputStream(System.in));
    } catch (UnknownHostException e) {
      System.err.println("Don't know about host");
    } catch (IOException e) {
      System.err.println("Couldn't get I/O for the connection to host");
    }

    /*
     * If everything has been initialized then we want to write some data to the
     * socket we have opened a connection to on port 2222.
     */
    if (clientSocket != null && os != null && is != null) {
      try {

        /*
         * Keep on reading from/to the socket till we receive the "Ok" from the
         * server, once we received that then we break.
         */
        System.out.println("The client started. Type any text. To quit it type 'Ok'.");
        String responseLine;
        os.println(inputLine.readLine());
        while ((responseLine = is.readLine()) != null) {
          System.out.println(responseLine);
          if (responseLine.indexOf("Ok") != -1) {
            break;
          }
          os.println(inputLine.readLine());
        }

        /*
         * Close the output stream, close the input stream, close the socket.
         */
        os.close();
        is.close();
        clientSocket.close();
      } catch (UnknownHostException e) {
        System.err.println("Trying to connect to unknown host: " + e);
      } catch (IOException e) {
        System.err.println("IOException:  " + e);
      }
    }
  }
}
```

2.0.6 The server

This is a simple echo server. The server is dedicated to echo messages received from clients. When it receives a message it sends the message back to the client. Also, it appends the string "`From server :`" in the echoed message to clarify who is saying what.

```
//Example 24

import java.io.DataInputStream;
import java.io.PrintStream;
import java.io.IOException;
import java.net.Socket;
import java.net.ServerSocket;

public class Server {
```

```
  public static void main(String args[]) {

    ServerSocket echoServer = null;
    String line;
    DataInputStream is;
    PrintStream os;
    Socket clientSocket = null;

    /*
     * Open a server socket on port 2222. Note that we can't choose a port less
     * than 1023 if we are not privileged users (root).
     */
    try {
      echoServer = new ServerSocket(2222);
    } catch (IOException e) {
      System.out.println(e);
    }

    /*
     * Create a socket object from the ServerSocket to listen to and accept
     * connections. Open input and output streams.
     */
    System.out.println("The server started. To stop it press <CTRL><C>.");
    try {
      clientSocket = echoServer.accept();
      is = new DataInputStream(clientSocket.getInputStream());
      os = new PrintStream(clientSocket.getOutputStream());

      /* As long as we receive data, echo that data back to the client. */
      while (true) {
        line = is.readLine();
        os.println("From server: " + line);
      }
    } catch (IOException e) {
      System.out.println(e);
    }
  }
}
```

2.0.7 Compiling and running the application

To try this application you have to compile the two programs: *Example 23* and *Example 24*.

Save these programs on your computer. Name the files **Client.java** and **Server.java**. Open a shell window on your computer and change the current directory to the directory where you saved these files. Type the following two commands in the shell window.

```
javac Server.java
javac Client.java
```

If java compiler is installed on your computer and the PATH variable is configured for the shell to find **javac** compiler, then these two command lines will create two new files in the current directory: the files **Server.class** and **Client.class**. Start the server in the shell window using the command:

```
java Server
```

You will see the following message in this window

```
The server started.  To stop it press <CTRL><C>.
```

telling you that the server is started.

Open a new shell window and change the current directory to the directory where you saved the application files. Start the client in the shell window using the command:

```
java Client
```

You will see the following message in this window

```
The client started.  Type any text.  To quit it type 'Ok'.
```

telling you that the client is started. Type, for example, the text `Hello` in this window. You will see the following output.

```
hello
From server:  hello
```

telling you that the message `Hello` was sent to the server and the echo was received by the client from the server.

2.1 A multi-threaded client/server application

The next example is a chat application. A chat application consists of a chat server and a chat client. The server accepts connections from the clients and delivers all messages from each client to other clients. This is a tool to communicate with other people over Internet in real time.

The client is implemented using two threads - one thread to interact with the server and the other with the standard input. Two threads are needed because a client must communicate with the server and, simultaneously, it must be ready to read messages from the standard input to be sent to the server.

The server is implemented using threads also. It uses a separate thread for each connection. It spawns a new client thread every time a new connection from a client is accepted. This simplifies a lot the design of the server. Multi-threading, however, creates synchronization issues. We will present two implementations of the chat server. An implementation that focus on multi-threading without considering the synchronization issues will be presented first. Then we will focus on the synchronization issues that a multi-threaded implementation creates. Finally, an updated version of the multi-threaded chat server that fixes the synchronization issues is presented.

2.1.1 The chat client

The code below is the multi-threaded chat client. It uses two threads: one to read the data from the standard input and to sent it to the server, the other to read the data from the server and to print it on the standard output.

```
//Example 25

import java.io.DataInputStream;
import java.io.PrintStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
```

```
import java.net.Socket;
import java.net.UnknownHostException;

public class MultiThreadChatClient implements Runnable {

  // The client socket
  private static Socket clientSocket = null;
  // The output stream
  private static PrintStream os = null;
  // The input stream
  private static DataInputStream is = null;

  private static BufferedReader inputLine = null;
  private static boolean closed = false;

  public static void main(String[] args) {

    // The default port.
    int portNumber = 2222;
    // The default host.
    String host = "localhost";

    if (args.length < 2) {
      System.out
          .println("Usage: java MultiThreadChatClient <host> <portNumber>\n"
              + "Now using host=" + host + ", portNumber=" + portNumber);
    } else {
      host = args[0];
      portNumber = Integer.valueOf(args[1]).intValue();
    }

    /*
     * Open a socket on a given host and port. Open input and output streams.
     */
    try {
      clientSocket = new Socket(host, portNumber);
      inputLine = new BufferedReader(new InputStreamReader(System.in));
      os = new PrintStream(clientSocket.getOutputStream());
      is = new DataInputStream(clientSocket.getInputStream());
    } catch (UnknownHostException e) {
      System.err.println("Don't know about host " + host);
    } catch (IOException e) {
      System.err.println("Couldn't get I/O for the connection to the host "
          + host);
    }

    /*
     * If everything has been initialized then we want to write some data to the
     * socket we have opened a connection to on the port portNumber.
     */
    if (clientSocket != null && os != null && is != null) {
      try {

        /* Create a thread to read from the server. */
        new Thread(new MultiThreadChatClient()).start();
        while (!closed) {
          os.println(inputLine.readLine().trim());
        }
        /*
         * Close the output stream, close the input stream, close the socket.
         */
        os.close();
        is.close();
        clientSocket.close();
```

```
      } catch (IOException e) {
        System.err.println("IOException:  " + e);
      }
    }
  }

  /*
   * Create a thread to read from the server. (non-Javadoc)
   *
   * @see java.lang.Runnable#run()
   */
  public void run() {
    /*
     * Keep on reading from the socket till we receive "Bye" from the
     * server. Once we received that then we want to break.
     */
    String responseLine;
    try {
      while ((responseLine = is.readLine()) != null) {
        System.out.println(responseLine);
        if (responseLine.indexOf("*** Bye") != -1)
          break;
      }
      closed = true;
    } catch (IOException e) {
      System.err.println("IOException:  " + e);
    }
  }
}
```

2.1.2 The chat server

We continue with the multi-threaded chat server. It uses a separate thread for each client. It spawns a new client thread every time a new connection from a client is accepted. This thread opens the input and the output streams for a particular client, it ask the client's name, it informs all clients about the fact that a new client has joined the chat room and, as long as it receive data, echos that data back to all other clients. When the client leaves the chat room, this thread informs also the clients about that and terminates.

```
//Example 26

import java.io.DataInputStream;
import java.io.PrintStream;
import java.io.IOException;
import java.net.Socket;
import java.net.ServerSocket;

/*
 * A chat server that delivers public and private messages.
 */
public class MultiThreadChatServer {

  // The server socket.
  private static ServerSocket serverSocket = null;
  // The client socket.
  private static Socket clientSocket = null;

  // This chat server can accept up to maxClientsCount clients' connections.
  private static final int maxClientsCount = 10;
  private static final clientThread[] threads = new clientThread[maxClientsCount];

  public static void main(String args[]) {
```

```
    // The default port number.
    int portNumber = 2222;
    if (args.length < 1) {
      System.out
          .println("Usage: java MultiThreadChatServer <portNumber>\n"
              + "Now using port number=" + portNumber);
    } else {
      portNumber = Integer.valueOf(args[0]).intValue();
    }

    /*
     * Open a server socket on the portNumber (default 2222). Note that we can
     * not choose a port less than 1023 if we are not privileged users (root).
     */
    try {
      serverSocket = new ServerSocket(portNumber);
    } catch (IOException e) {
      System.out.println(e);
    }

    /*
     * Create a client socket for each connection and pass it to a new client
     * thread.
     */
    while (true) {
      try {
        clientSocket = serverSocket.accept();
        int i = 0;
        for (i = 0; i < maxClientsCount; i++) {
          if (threads[i] == null) {
            (threads[i] = new clientThread(clientSocket, threads)).start();
            break;
          }
        }
        if (i == maxClientsCount) {
          PrintStream os = new PrintStream(clientSocket.getOutputStream());
          os.println("Server too busy. Try later.");
          os.close();
          clientSocket.close();
        }
      } catch (IOException e) {
        System.out.println(e);
      }
    }
  }
}

/*
 * The chat client thread. This client thread opens the input and the output
 * streams for a particular client, ask the client's name, informs all the
 * clients connected to the server about the fact that a new client has joined
 * the chat room, and as long as it receive data, echos that data back to all
 * other clients. When a client leaves the chat room this thread informs also
 * all the clients about that and terminates.
 */
class clientThread extends Thread {

  private DataInputStream is = null;
  private PrintStream os = null;
  private Socket clientSocket = null;
  private final clientThread[] threads;
  private int maxClientsCount;
```

```
  public clientThread(Socket clientSocket, clientThread[] threads) {
    this.clientSocket = clientSocket;
    this.threads = threads;
    maxClientsCount = threads.length;
  }

  public void run() {
    int maxClientsCount = this.maxClientsCount;
    clientThread[] threads = this.threads;

    try {
      /*
       * Create input and output streams for this client.
       */
      is = new DataInputStream(clientSocket.getInputStream());
      os = new PrintStream(clientSocket.getOutputStream());
      os.println("Enter your name.");
      String name = is.readLine().trim();
      os.println("Hello " + name
          + " to our chat room.\nTo leave enter /quit in a new line");
      for (int i = 0; i < maxClientsCount; i++) {
        if (threads[i] != null && threads[i] != this) {
          threads[i].os.println("*** A new user " + name
              + " entered the chat room !!! ***");
        }
      }
      while (true) {
        String line = is.readLine();
        if (line.startsWith("/quit")) {
          break;
        }
        for (int i = 0; i < maxClientsCount; i++) {
          if (threads[i] != null) {
            threads[i].os.println("<" + name + "&gr; " + line);
          }
        }
      }
      for (int i = 0; i < maxClientsCount; i++) {
        if (threads[i] != null && threads[i] != this) {
          threads[i].os.println("*** The user " + name
              + " is leaving the chat room !!! ***");
        }
      }
      os.println("*** Bye " + name + " ***");

      /*
       * Clean up. Set the current thread variable to null so that a new client
       * could be accepted by the server.
       */
      for (int i = 0; i < maxClientsCount; i++) {
        if (threads[i] == this) {
          threads[i] = null;
        }
      }

      /*
       * Close the output stream, close the input stream, close the socket.
       */
      is.close();
      os.close();
      clientSocket.close();
    } catch (IOException e) {
    }
  }
```

```
}
```

2.1.3 Synchronization issues of the multi-threaded chat server implementation

Consider now the synchronization issues such an implementation creates. To simplify our task let us split the chat server code as follows, see the partitioned code below.

```
//section 1 begin

//Example 26

import java.io.DataInputStream;
import java.io.PrintStream;
import java.io.IOException;
import java.net.Socket;
import java.net.ServerSocket;

/*
 * A chat server that delivers public and private messages.
 */
public class MultiThreadChatServer {

  // The server socket.
  private static ServerSocket serverSocket = null;
  // The client socket.
  private static Socket clientSocket = null;

  // This chat server can accept up to maxClientsCount clients' connections.
  private static final int maxClientsCount = 10;
  private static final clientThread[] threads = new clientThread[maxClientsCount];

  public static void main(String args[]) {

    // The default port number.
    int portNumber = 2222;
    if (args.length < 1) {
      System.out
          .println("Usage: java MultiThreadChatServer <portNumber>\n"
              + "Now using port number=" + portNumber);
    } else {
      portNumber = Integer.valueOf(args[0]).intValue();
    }

    /*
     * Open a server socket on the portNumber (default 2222). Note that we can
     * not choose a port less than 1023 if we are not privileged users (root).
     */
    try {
      serverSocket = new ServerSocket(portNumber);
    } catch (IOException e) {
      System.out.println(e);
    }

    /*
     * Create a client socket for each connection and pass it to a new client
     * thread.
     */
    while (true) {
      try {
        clientSocket = serverSocket.accept();
        int i = 0;
```

```
//section 1 end
//section 2 begin

        for (i = 0; i < maxClientsCount; i++) {
          if (threads[i] == null) {
            (threads[i] = new clientThread(clientSocket, threads)).start();
            break;
          }
        }

//section 2 end
//section 3 begin

        if (i == maxClientsCount) {
          PrintStream os = new PrintStream(clientSocket.getOutputStream());
          os.println("Server too busy. Try later.");
          os.close();
          clientSocket.close();
        }
      } catch (IOException e) {
        System.out.println(e);
      }
    }
  }
}

/*
 * The chat client thread. This client thread opens the input and the output
 * streams for a particular client, ask the client's name, informs all the
 * clients connected to the server about the fact that a new client has joined
 * the chat room, and as long as it receive data, echos that data back to all
 * other clients. When a client leaves the chat room this thread informs also
 * all the clients about that and terminates.
 */
class clientThread extends Thread {

  private DataInputStream is = null;
  private PrintStream os = null;
  private Socket clientSocket = null;
  private final clientThread[] threads;
  private int maxClientsCount;

  public clientThread(Socket clientSocket, clientThread[] threads) {
    this.clientSocket = clientSocket;
    this.threads = threads;
    maxClientsCount = threads.length;
  }

  public void run() {
    int maxClientsCount = this.maxClientsCount;
    clientThread[] threads = this.threads;

    try {
      /*
       * Create input and output streams for this client.
       */
      is = new DataInputStream(clientSocket.getInputStream());
      os = new PrintStream(clientSocket.getOutputStream());
      os.println("Enter your name.");
      String name = is.readLine().trim();
      os.println("Hello " + name
          + " to our chat room.\nTo leave enter /quit in a new line");

//section 3 end
```

```
//section 4 begin

      for (int i = 0; i < maxClientsCount; i++) {
        if (threads[i] != null && threads[i] != this) {
          threads[i].os.println("*** A new user " + name
              + " entered the chat room !!! ***");
        }
      }

//section 4 end
//section 5 begin

      while (true) {
        String line = is.readLine();
        if (line.startsWith("/quit")) {
          break;
        }

//section 5 end
//section 6 begin

        for (int i = 0; i < maxClientsCount; i++) {
          if (threads[i] != null) {
            threads[i].os.println("<" + name + "> " + line);
          }
        }

//section 6 end
//section 7 begin

      }

//section 7 end
//section 8 begin

      for (int i = 0; i < maxClientsCount; i++) {
        if (threads[i] != null && threads[i] != this) {
          threads[i].os.println("*** The user " + name
              + " is leaving the chat room !!! ***");
        }
      }

//section 8 end
//section 9 begin

      os.println("*** Bye " + name + " ***");
      /*
       * Clean up. Set the current thread variable to null so that a new client
       * could be accepted by the server.
       */

//section 9 end
//section 10 begin

      for (int i = 0; i < maxClientsCount; i++) {
        if (threads[i] == this) {
          threads[i] = null;
        }
      }

//section 10 end
//section 11 begin

      /*
```

```
        * Close the output stream, close the input stream, close the socket.
        */
       is.close();
       os.close();
       clientSocket.close();
     } catch (IOException e) {
     }
   }
}

//section 11 end
```

Consider the section 2, 4, 6, 8, and 10 of the code above. All these sections use the array **threads**[]. This array, however, is shared by all threads of the server. The array is passed by reference to the constructor of the thread every time a new thread is created. The modification of this array by a thread is visible by all other threads. These portions of code are called *critical sections* because, if used uncontrolled, they can cause unexpected behaviour and even exceptions (explained below).

Since all threads run concurrently, the access to this array is also concurrent. Suppose now that a thread (Thread 1) enters section 4 while another thread (Thread 2) enters section 10 of the code. Section 4 uses the array **threads**[] to inform the clients about a new client, whereas section 10 removes from this array the thread references of the client that leaves the chat room. It can happen that a **threads[i]** reference, while being used in section 4, is set to null in section 10 by another thread—by the thread of the client leaving the chat room. The sequence below shows this scenario (explained afterward).

```
        Thread 1                                                Thread 2

4        for (int i = 0; i < maxClientsCount; i++) {
         if (threads[i] != null && threads[i] != this) {

10                                                      for (int i = 0; i < maxClientsCount; i++) {
                                                                if (threads[i] == this) {
                                                                  threads[i] = null;
                                                                        }
                                                                    }

4        threads[i].os.println("*** A new user " + name
          + " entered the chat room !!! ***");
          }
         }
```

In this scenario, Thread 1 executes the **if** statement in section 4. Suppose **threads[i]** is not null at this instant. Suppose, also, Thread 1 is interrupted by the operating system immediately after evaluating the **if** statement condition, meaning that Thread 1 is put in a waiting queue, while Thread 2 starts executing section 10. Such kind of execution is called interleaving. Suppose, Thread 2 sets **threads[i]** to null when executing portion 10. Finally, Thread 1 is resumed and executes **threads[i].os.println()** statement. But **threads[i]** is null at this instant. This will cause a null pointer exception, and it will close abnormally the connection with a client. And all that because of another client that decided to leave the chat room. The same situation can arise if we consider the concurrency of any of the sections 2, 6, 8 with section 10. Such situations are unacceptable and must be resolved correctly in a concurrent multi-threaded application.

Solution to the issue To avoid such kind of exception, the threads must be synchronized so that they execute the critical section of code sequentially (as if it were an atomic action), and thus, without interleaving. For example, in the sequence below, the execution of the two sections of code is sequential—the critical sections execute without interruption. We call such execution *synchronized*. To archive this synchronization we have to use the **synchronized(this){}** statement, like below.

Thread 1

```
4    synchronized (this){
     for (int i = 0; i < maxClientsCount; i++) {
       if (threads[i] != null && threads[i] != this) {
         threads[i].os.println("*** A new user " + name
             + " entered the chat room !!! ***");
       }
     }
   }
```

Thread 2

```
10                                          synchronized(this) {
                                             for (int i = 0; i < maxClientsCount; i++) {
                                              if (threads[i] == this) {
                                               threads[i] = null;
                                                  }
                                                }
                                              }
```

All **synchronized(this){}** statements mutually exclude each other. This means that when a thread enters the **synchronized(this){}** statement, it verifies first that any other **synchronized(this){}** statement is not being executed by another thread. If a such a statement is being executed by a thread, then this thread, as well as all other threads trying to execute a **synchronized(this){}** statement, are forced to wait until the thread executing the **synchronized(this){}** terminates this statement. When the thread executing a synchronized(this) statement leaves the critical section, that is, when it terminates the **synchronized(this){}** statement, a thread waiting for critical section enters its **synchronized(this){}**. When a thread enters **synchronized(this){}** statement, it blocks all other threads from entering their **synchronized(this){}** statements. Thus, by putting all critical sections in **synchronized(this){}** statements we are guaranteed that the chat server will execute without raising null pointer exceptions caused by concurrent execution of other critical sections.

The **synchronized(this){}** statement is a powerful tool. However, using it requires a good understanding of the synchronization issue, as incorrect use of **synchronized(this){}** statement can cause deadlocks of the program. A *deadlock* is a scenario when one thread waits for another thread to leave its critical section forever. To quickly explain this scenario, suppose we extended the critical section 6 like below, where now the **synchronized(this){}** statement includes a loop that potentially can execute forever.

```
6      synchronized(this) {
        while (true) {
        String line = is.readLine();
            if (line.startsWith("/quit")) {
              break;
            }
            for (int i = 0; i < maxClientsCount; i++) {
              if (threads[i] != null) {
                threads[i].os.println("<" + name + "> " + line);
              }
            }
          }
        }
```

The **while (true)** loop will execute until it receives a "`/quit`" command from the input stream. Now suppose the `/quit` command never arrives or it arrives after a very long time. The thread executing this loop inside the **synchronized(this){}** statement will block all other threads from executing their synchronized code because they will wait at their **synchronized(this){}** statements. For example, the section of code labeled 10, below, will never be executed by Thread 2 if Thread 1 entered the **while (true)** loop and stays in there forever.

```
        Thread 1                                                    Thread 2

6       synchronized(this) {
         while (true) {
           String line = is.readLine();
           if (line.startsWith("/quit")) {
             break;
           }
           for (int i = 0; i < maxClientsCount; i++) {
             if (threads[i] != null) {
               threads[i].os.println("<" + name + "> " + line);
             }
           }
         }
       }

10                                                          synchronized(this) {
                                                             for (int i = 0; i < maxClientsCount; i++) {
                                                              if (threads[i] == this) {
                                                              threads[i] = null;
                                                              }
                                                             }
                                                            }
```

Thus, when synchronizing programs, an appropriate solution must be implemented to solve such issues, otherwise the **synchronized(this){}** statement can cause very long delays and even deadlocks. And certainly, you have to avoid putting unnecessary **synchronized(this){}** statements in the program. For example, it is not necessary to synchronize section 2 of the code (see the partitioned code earlier). Even if this code modifies the **threads**[] array, a better inspection of the code discovers that there is no risk that this modification will create null pointer exceptions or other problems to the program.

2.1.4 The synchronized version of the chat server

Now we are ready for the updated version of the chat server that fixes the synchronization issues described in the previous section, where the **synchronized(this){}** statement is used to solve the synchronization issues. Also, this version of chat server is improved so as to deliver private messages to clients.

```
//Example 26 (updated)

import java.io.DataInputStream;
import java.io.PrintStream;
import java.io.IOException;
import java.net.Socket;
import java.net.ServerSocket;

/*
 * A chat server that delivers public and private messages.
 */
public class MultiThreadChatServerSync {

  // The server socket.
  private static ServerSocket serverSocket = null;
  // The client socket.
  private static Socket clientSocket = null;

  // This chat server can accept up to maxClientsCount clients' connections.
  private static final int maxClientsCount = 10;
  private static final clientThread[] threads = new clientThread[maxClientsCount];

  public static void main(String args[]) {
```

```
    // The default port number.
    int portNumber = 2222;
    if (args.length < 1) {
      System.out.println("Usage: java MultiThreadChatServerSync <portNumber>\n"
          + "Now using port number=" + portNumber);
    } else {
      portNumber = Integer.valueOf(args[0]).intValue();
    }

    /*
     * Open a server socket on the portNumber (default 2222). Note that we can
     * not choose a port less than 1023 if we are not privileged users (root).
     */
    try {
      serverSocket = new ServerSocket(portNumber);
    } catch (IOException e) {
      System.out.println(e);
    }

    /*
     * Create a client socket for each connection and pass it to a new client
     * thread.
     */
    while (true) {
      try {
        clientSocket = serverSocket.accept();
        int i = 0;
        for (i = 0; i < maxClientsCount; i++) {
          if (threads[i] == null) {
            (threads[i] = new clientThread(clientSocket, threads)).start();
            break;
          }
        }
        if (i == maxClientsCount) {
          PrintStream os = new PrintStream(clientSocket.getOutputStream());
          os.println("Server too busy. Try later.");
          os.close();
          clientSocket.close();
        }
      } catch (IOException e) {
        System.out.println(e);
      }
    }
  }
}

/*
 * The chat client thread. This client thread opens the input and the output
 * streams for a particular client, ask the client's name, informs all the
 * clients connected to the server about the fact that a new client has joined
 * the chat room, and as long as it receive data, echos that data back to all
 * other clients. The thread broadcast the incoming messages to all clients and
 * routes the private message to the particular client. When a client leaves the
 * chat room this thread informs also all the clients about that and terminates.
 */
class clientThread extends Thread {

  private String clientName = null;
  private DataInputStream is = null;
  private PrintStream os = null;
  private Socket clientSocket = null;
  private final clientThread[] threads;
  private int maxClientsCount;
```

```
public clientThread(Socket clientSocket, clientThread[] threads) {
  this.clientSocket = clientSocket;
  this.threads = threads;
  maxClientsCount = threads.length;
}

public void run() {
  int maxClientsCount = this.maxClientsCount;
  clientThread[] threads = this.threads;

  try {
    /*
     * Create input and output streams for this client.
     */
    is = new DataInputStream(clientSocket.getInputStream());
    os = new PrintStream(clientSocket.getOutputStream());
    String name;
    while (true) {
      os.println("Enter your name.");
      name = is.readLine().trim();
      if (name.indexOf('@') == -1) {
        break;
      } else {
        os.println("The name should not contain '@' character.");
      }
    }

    /* Welcome the new the client. */
    os.println("Welcome " + name
        + " to our chat room.\nTo leave enter /quit in a new line.");
    synchronized (this) {
      for (int i = 0; i < maxClientsCount; i++) {
        if (threads[i] != null && threads[i] == this) {
          clientName = "@" + name;
          break;
        }
      }
      for (int i = 0; i < maxClientsCount; i++) {
        if (threads[i] != null && threads[i] != this) {
          threads[i].os.println("*** A new user " + name
              + " entered the chat room !!! ***");
        }
      }
    }
    /* Start the conversation. */
    while (true) {
      String line = is.readLine();
      if (line.startsWith("/quit")) {
        break;
      }
      /* If the message is private sent it to the given client. */
      if (line.startsWith("@")) {
        String[] words = line.split("\\s", 2);
        if (words.length > 1 && words[1] != null) {
          words[1] = words[1].trim();
          if (!words[1].isEmpty()) {
            synchronized (this) {
              for (int i = 0; i < maxClientsCount; i++) {
                if (threads[i] != null && threads[i] != this
                    && threads[i].clientName != null
                    && threads[i].clientName.equals(words[0])) {
                  threads[i].os.println("<" + name + "> " + words[1]);
                  /*
```

```
                    * Echo this message to let the client know the private
                    * message was sent.
                    */
                   this.os.println(">" + name + "> " + words[1]);
                   break;
                 }
               }
             }
           }
         } else {
           /* The message is public, broadcast it to all other clients. */
           synchronized (this) {
             for (int i = 0; i < maxClientsCount; i++) {
               if (threads[i] != null && threads[i].clientName != null) {
                 threads[i].os.println("<" + name + "> " + line);
               }
             }
           }
         }
       }
       synchronized (this) {
         for (int i = 0; i < maxClientsCount; i++) {
           if (threads[i] != null && threads[i] != this
               && threads[i].clientName != null) {
             threads[i].os.println("*** The user " + name
                 + " is leaving the chat room !!! ***");
           }
         }
       }
       os.println("*** Bye " + name + " ***");

       /*
        * Clean up. Set the current thread variable to null so that a new client
        * could be accepted by the server.
        */
       synchronized (this) {
         for (int i = 0; i < maxClientsCount; i++) {
           if (threads[i] == this) {
             threads[i] = null;
           }
         }
       }
       /*
        * Close the output stream, close the input stream, close the socket.
        */
       is.close();
       os.close();
       clientSocket.close();
     } catch (IOException e) {
     }
   }
}
```

2.1.5 Compiling and running the application

To try this application you have to compile the two programs: *Example 25* and *Example 26 (updated)*.

Save these programs on your computer. Name the files **MultiThreadChatClient.java** and **MultiThreadChatServerSync.java**. Open a shell window on your computer and change the current directory to the directory where you saved these files, and type the following two commands in the shell window.

```
javac MultiThreadChatServerSync.java
javac MultiThreadChatClient.java
```

If java compiler is installed on your computer and the PATH variable is configured for the shell to find **javac** compiler, then these two command lines will create two new files in the current directory, being the files **MultiThreadChatServerSync.class** and **MultiThreadChatClient.class**.

Now start the server in the shell window using the command:

```
java MultiThreadChatServerSync
```

You will see the following message in this window

```
Usage:  java MultiThreadChatServerSync <portNumber>
Now using port number=2222
```

telling you that the chat server is started and that it is listening for connections on port number 2222. The phrase `Usage:  java MultiThreadChatServerSync <portNumber>` tells you that you can start the server specifying a parameter (the port number), though port 2222 is used by default. If you execute the command a gain, you will receive an error message, because the server is already running.

Open a new shell window and change the current directory to the directory where you saved the application files. Start the client in the shell window using the command:

```
java MultiThreadChatClient
```

You will see the following message in this window

```
Usage:  java MultiThreadChatClient <host> <portNumber>
Now using host=localhost, portNumber=2222
Enter your name.
```

telling you that the client is started. Type, for example, the name `Anonymous1` in this window. You will see the following output.

```
Hello Anonymous1 to our chat room.
To leave enter /quit in a new line
```

telling you that the client `Anonymous1` entered the chat room. It tells you also that to quit the chat room the client has to enter `/quit` command.

Open one more shell window and change the current directory to the directory where you saved the application files. Start a new client in the shell window using the command:

```
java MultiThreadChatClient
```

You will see the following message in this window

```
Usage:  java MultiThreadChatClient <host> <portNumber>
Now using host=localhost, portNumber=2222
Enter your name.
```

telling you that the client is started. Now you have two clients connected to the server. Type, for example, the text `Anonymous2` in this window. You will see the following output.

```
Hello Anonymous2 to our chat room.
To leave enter /quit in a new line
```

telling you that the client Anonymous2 entered the chat room. It tells you also that to quit the chat room the client has to enter `/quit` command. In the window of the client *Anonymous1* the following message will be printed.

```
*** A new user Anonymous2 entered the chat room !!!  ***
```

If we enter now a message in any of the client window the message will be printed also in the window of the other client. This kind of message exchange is a chat session.

2.2 Chatting over the network

The application you created runs on one machine in two different windows. This is due to the following two lines in the code *Example 25*:

```
//snip

String host = "localhost";

//snip

clientSocket = new Socket(host , portNumber);

//snip
```

One option is to modify the client code so that it will ask the client whom to contact (instead of defaulting to the localhost, i.e., your PC), so that when the client-side starts up, it will ask the user the address (name or IP address) of the server. If you like, you can modify the client code from *Example 25* so that it first asks the user for the address of the remote host.

Vula has a file for this tutorial called **GUIchat.java**, which has a basic GUI and it can be used across networked machines already. The class relevant for our client-server application is **ConnectionHandler**, which is responsible for opening the network connection and for reading incoming messages once the connection has been opened. By putting the connection-opening code in a separate thread, we make sure that the GUI is not blocked while the connection is being opened. Like reading incoming messages, opening a connection is a blocking operation that can take some time to complete. A **ConnectionHandler** is created when the user clicks the "Listen on port" or "Connect to" button in the GUI. The "Listen on port" button makes the thread act as a server, while "Connect to" makes it act as a client. So, when you run the application, make sure you have clicked on "Listen on port" in one of the running instances before writing either localhost or an IP address in the text field right from the "Connect to" button, and clicking that button.

The **ConnectionHandler** class has two constructors to handle each. The 'server-part' is:

```
ConnectionHandler(int port) {
   state = ConnectionState.LISTENING;
```

```
    this.port = port;
    postMessage("\nLISTENING ON PORT " + port + "\n");
    start();
}
```

and the 'client-part':

```
ConnectionHandler(String remoteHost, int port) {
    state = ConnectionState.CONNECTING;
    this.remoteHost = remoteHost;
    this.port = port;
    postMessage("\nCONNECTING TO " + remoteHost + " ON PORT " + port + "\n");
    start();
```

Once a thread has been started, it executes the **public void run()** method (scroll to the end of the **GUIchat.java** file to inspect it). After opening the connection as either a server or client, this **run()** method enters a **while** loop in which it receives and processes messages from the other side of the connection until the connection is closed. In order to close the connection, the GUIChat window has a "Disconnect" button that the user can click. The program responds to this event by closing the socket that represents the connection. It is likely that when this happens, the connection-handling thread is blocked in the **in.readLine()** method, waiting for an incoming message. When the socket is closed by another thread, this method will fail and will throw an exception; this exception causes the thread to terminate. (If the connection-handling thread happens to be between calls to **in.readLine()** when the socket is closed, the while loop will terminate because the connection state changes from CONNECTED to CLOSED.) Note that closing the window will also close the connection in the same way. It is also possible for the user on the other side of the connection to close the connection. When that happens, the stream of incoming messages ends, and the **in.readLine()** on this side of the connection returns the value null, which indicates end-of-stream and acts as a signal that the connection has been closed by the remote user. The code is richly annotated with other information. Inspect the code. The main aspect we are interested in here is the code that deals with `remoteHost`.

2.3 Final remarks and TCP Socket Exercises

Java sockets API (Socket and ServerSocket classes) is a powerful and flexible interface for network programming of client/server applications.

Java threads is another powerful programming framework for client/server applications. Multi-threading simplifies the implementation of complex client/server applications. However, it introduces synchronization issues. These issues are caused by the concurrent execution of critical sections of the program by different threads. The **synchronized(this){}** statement allows us to synchronize the execution of the critical sections. Using this statement, however, requires a good understanding of the synchronization issues. The incorrect use of **synchronized(this){}** statement can cause other problems, such as deadlocks and/or performance degradation of the program. In addition, there are several options as to what is part of the thread.

Finally, for networked applications—not just client-server on one machine—specification of both host and port are crucial. One can set it in the software code, or cater for it using input from the client or server.

Exercises

1. Try the examples presented in this document.
2. Can you get the chat client from Section 2.1 to work across machines (within the lab with your lab mate, another laptop, mobile)? If so, demonstrate how; if not, explain why not and how that can be resolved.
3. How does **GUIchat.java** implement chatting between networked machines?

4. Compare and contrast the *differences in approach of programming* the chat client/server application between what you implemented in Section 2.1 and that of Section 2.2.

Credits: this tutorial is based on `http://www.ase.md/~aursu/ClientServerThreads.html` and chapter 12, section 4 of 'Introduction to Programming Using Java', 6th ed. (v6.0.3, January 2014), by David J. Eck.

3 Sniffing with Wireshark

3.1 TCP Wireshark lab

In this part of the practical assignment, we'll investigate the behaviour of the celebrated TCP protocol in detail. We'll do so by analysing a trace of the TCP segments sent and received in transferring a 150KB file from your computer to a remote server. That file is allocated to you and is old enough to have had their copyright expired, such as Lewis Carrol's Alice's Adventures in Wonderland (many of those books are digitised and available from Project Gutenberg[1]). We'll study TCP's use of sequence and acknowledgement numbers for providing reliable data transfer; we'll see TCP's congestion control algorithm—slow start and congestion avoidance—in action; and there dis an optional exercises on TCP's receiver-advertised flow control mechanism. We'll also briefly consider TCP connection setup and we'll investigate the performance (throughput and round-trip time) of the TCP connection between your computer and the server.

Before beginning this lab, you'll probably want to review sections 3.5 and 3.7 in the text.

3.1.1 Capturing a bulk TCP transfer from your computer to a remote server

Before beginning our exploration of TCP, we'll need to use Wireshark to obtain a packet trace of the TCP transfer of a file from your computer to a remote server. You'll do so by accessing a Web page that will allow you to enter the name of a file stored on your computer (which contains the ASCII text of your assigned Gutenberg ebook), and then transfer the file to a Web server using the HTTP POST method (see section 2.2.3 in the text). We're using the POST method rather than the GET method as we'd like to transfer a large amount of data from your computer to another computer. Of course, we'll be running Wireshark during this time to obtain the trace of the TCP segments sent and received from your computer.

Do the following:

- Start up your web browser. Go to `http://download.cs.uct.ac.za/2015/CSC3002F/Networks/` and retrieve an ASCII copy of the Gutenberg ebook assigned to you, `yourgutenbergfile.txt`. Store this file somewhere on your computer.
- Next go to `http://gaia.cs.umass.edu/wireshark-labs/TCP-wireshark-file1.html`.
- You should see a screen that looks like the screenshot in Fig. 1.
- Use the Browse button in this form to enter the name of the file (full path name) on your computer containing yourgutenbergfile (or do so manually). Don't yet press the "Upload alice.txt file" button.
- Now start up Wireshark and begin packet capture (Capture −> Start) and then press OK on the Wireshark Packet Capture Options screen. Note: if it doesn't work, go to Capture −> Interfaces and select the NIC and then click Start
- Return to your browser and press the "Upload alice.txt file" button to upload the file to the gaia.cs.umass.edu server. Once the file has been uploaded, a short congratulations message will be displayed in your browser window.
- Stop Wireshark packet capture. Your Wireshark window should look similar to the screenshot in Fig. 2. Note: it may look slightly different, which is due to either difference in operating system or yet another version of the software.

NOTE: you certainly will be able to run Wireshark in the labs. It may well be the case that when you (want to) start doing this exercise only some 5 hours before the assignment deadline, some server is down or the network happens to be not working. *that's your problem* and not an excuse or a valid reason for extension. There's plenty of time to do these exercises, and most of the time, the network and servers are working.

[1] `https://www.gutenberg.org/`

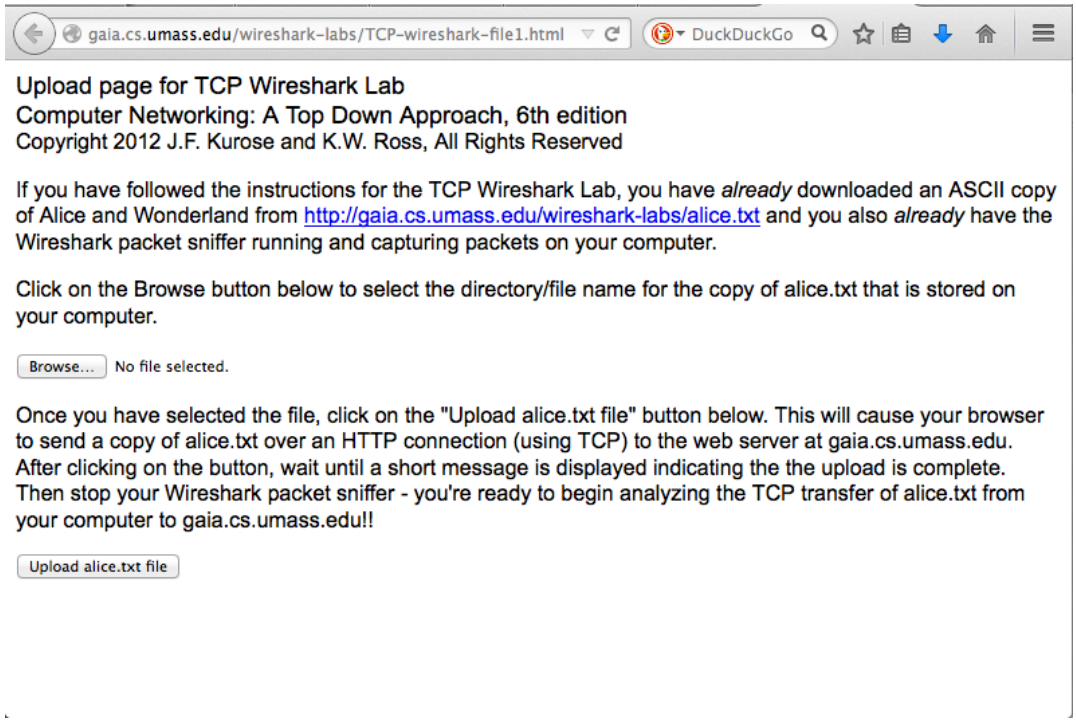


Figure 1: Upload page for yourgutenbergfile.txt

3.1.2 A first look at the captured trace

Before analysing the behaviour of the TCP connection in detail, let's take a high level view of the trace. First, filter the packets displayed in the Wireshark window by entering "tcp" (lowercase, no quotes, and don't forget to press return after entering!) into the display filter specification window towards the top of the Wireshark window. What you should see is series of TCP and HTTP messages between your computer and gaia.cs.umass.edu. You should see the initial three-way handshake containing a SYN message. You should see an HTTP POST message. Depending on the version of Wireshark you are using, you might see a series of "HTTP Continuation" messages being sent from your computer to gaia.cs.umass.edu. Recall from our discussion in the earlier HTTP Wireshark lab, that is no such thing as an HTTP Continuation message; this is Wireshark's way of indicating that there are multiple TCP segments being used to carry a single HTTP message. In more recent versions of Wireshark, you'll see "[TCP segment of a reassembled PDU]" in the Info column of the Wireshark display to indicate that this TCP segment contained data that belonged to an upper layer protocol message (in our case here, HTTP). You should also see TCP ACK segments being returned from gaia.cs.umass.edu to your computer.

Answer the following questions, by using your own trace. When answering a question, you should hand in a printout (screenshot) of the packet(s) within the trace that you used to answer the question asked. Annotate the printout to explain your answers. (To print a packet, use File −> Print, choose Selected packet only, choose Packet summary line, and select the minimum amount of packet detail that you need to answer the question.)

Exercises

5. What is the IP address and TCP port number used by the client computer (source) that is transferring the file to gaia.cs.umass.edu? To answer this question, it's probably easiest to select an HTTP message and explore the details of the TCP packet used to carry this HTTP message, using the "details of the selected packet header window" (refer to Figure 2 in the 'Getting Started with Wireshark' Lab if you're uncertain about the Wireshark windows.

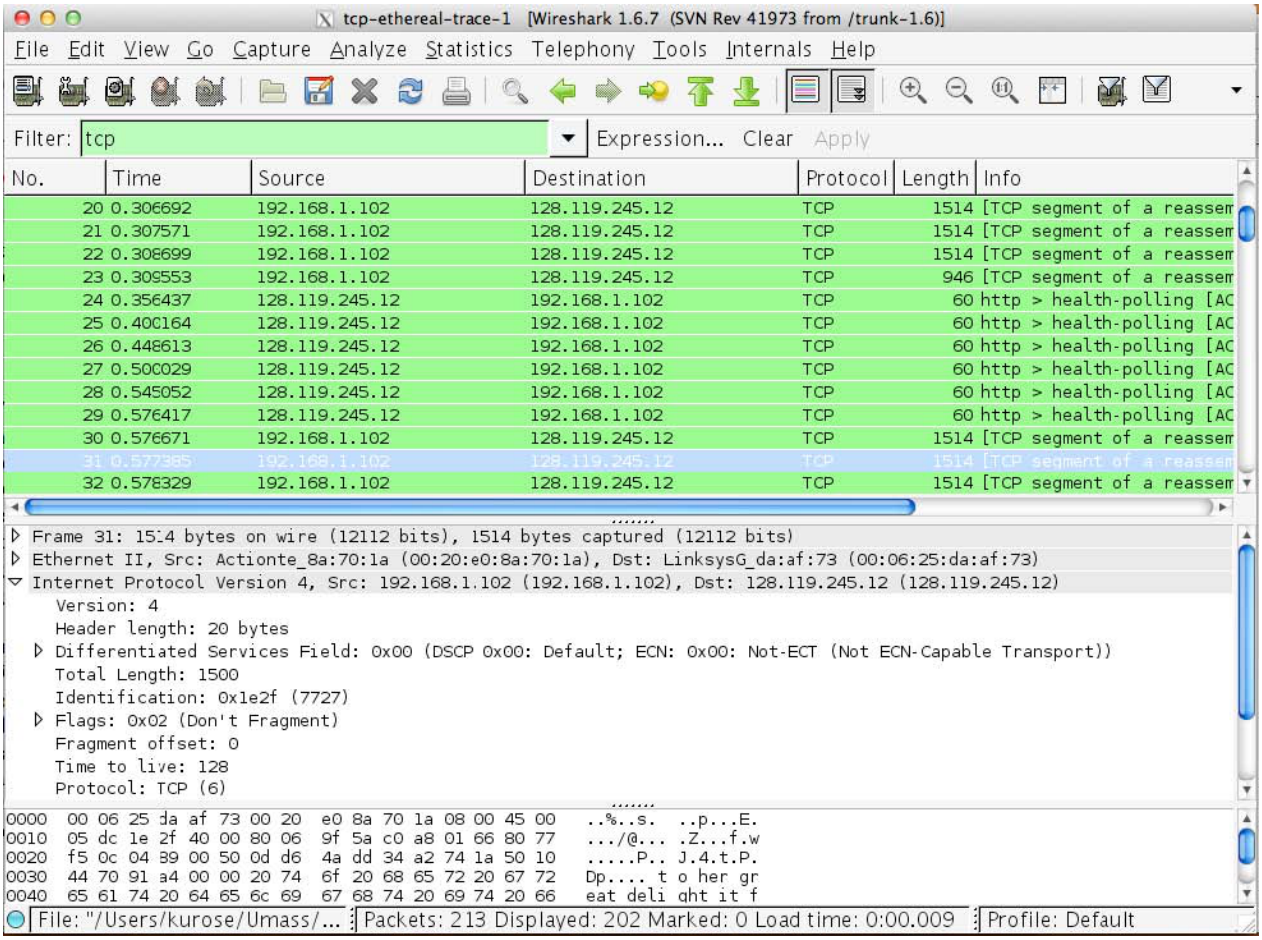


Figure 2: A screenshot of Wireshark after packet capture.

6. What is the IP address of gaia.cs.umass.edu? On what port number is it sending and receiving TCP segments for this connection?

3.1.3 TCP Basics

Since this lab is about TCP rather than HTTP, let's change Wireshark's "listing of captured packets" window so that it shows information about the TCP segments containing the HTTP messages, rather than about the HTTP messages. To have Wireshark do this, select Analyze −> Enabled Protocols. Then uncheck the HTTP box and select OK. You should now see a Wireshark window that looks like the one in Fig. 3.

This is what we're looking for—a series of TCP segments sent between your computer and gaia.cs.umass.edu. We will use the packet trace that you have captured to study TCP behaviour in the rest of this lab.

Exercises

7. What is the sequence number of the TCP SYN segment that is used to initiate the TCP connection between the client computer and gaia.cs.umass.edu? What is it in the segment that identifies the segment as a SYN segment?

8. What is the sequence number of the SYNACK segment sent by gaia.cs.umass.edu to the client computer in reply to the SYN? What is the value of the Acknowledgement field in the SYNACK segment? How did gaia.cs.umass.edu determine that value? What is it in the segment that identifies the segment as a SYNACK segment?

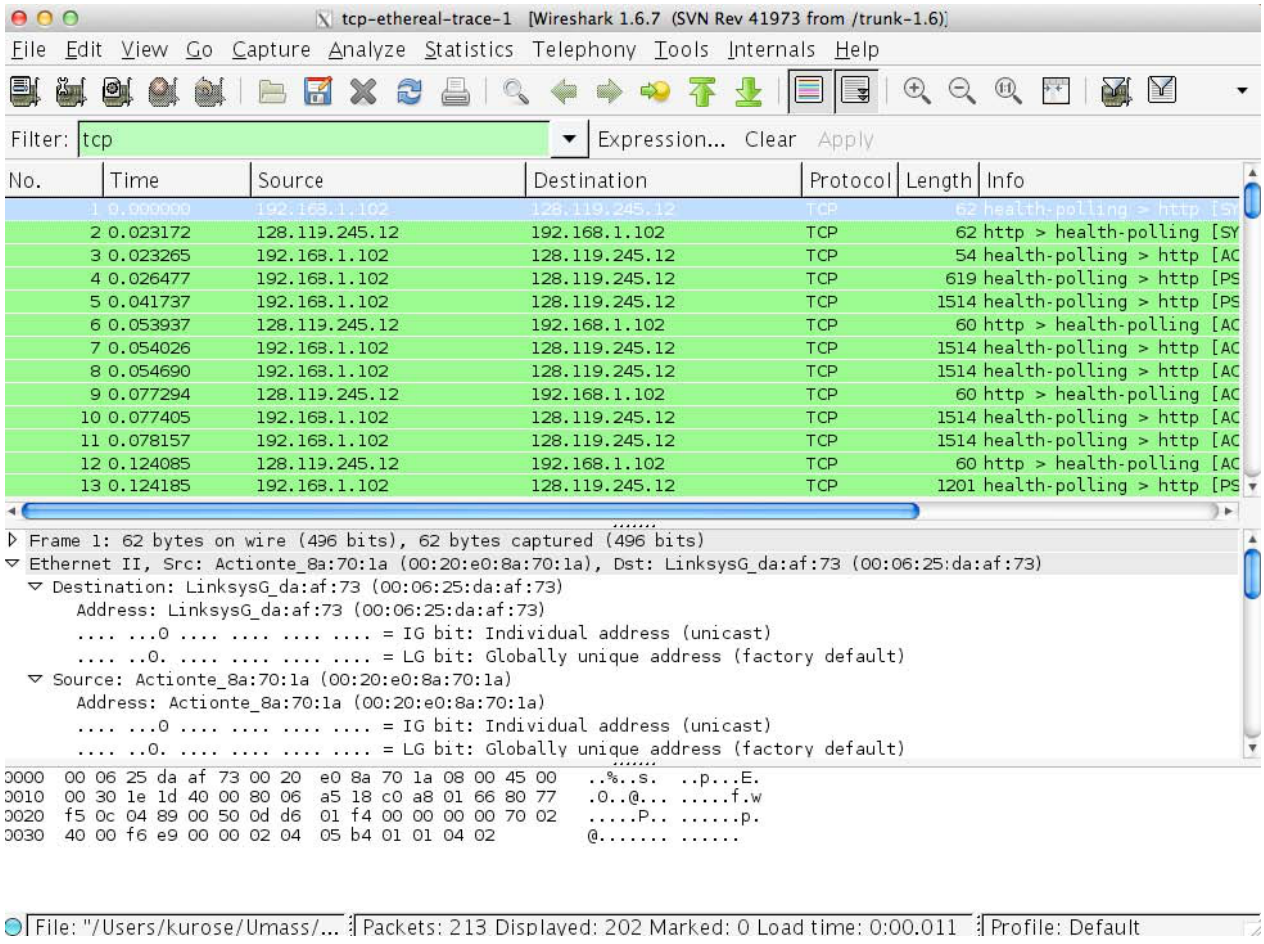


Figure 3: Another screenshot of Wireshark after packet capture.

9. What is the sequence number of the TCP segment containing the HTTP POST command? Note that in order to find the POST command, you'll need to dig into the packet content field at the bottom of the Wireshark window, looking for a segment with a "POST" within its DATA field.

10. Consider the TCP segment containing the HTTP POST as the first segment in the TCP connection. What are the sequence numbers of the first six segments in the TCP connection (including the segment containing the HTTP POST)? At what time was each segment sent? When was the ACK for each segment received? Given the difference between when each TCP segment was sent, and when its acknowledgement was received, what is the RTT value for each of the six segments? What is the `EstimatedRTT` value (see Section 3.5.3, page 239 in text) after the receipt of each ACK? Assume that the value of the `EstimatedRTT` is equal to the measured RTT for the first segment, and then is computed using the `EstimatedRTT` equation on page 239 for all subsequent segments. (Note: Wireshark has a nice feature that allows you to plot the RTT for each of the TCP segments sent. Select a TCP segment in the "listing of captured packets" window that is being sent from the client to the gaia.cs.umass.edu server. Then select: Statistics −> TCP Stream Graph −> Round Trip Time Graph.)

11. What is the length of each of the first six TCP segments?

12. What is the minimum amount of available buffer space advertised at the received for the entire trace? Does the lack of receiver buffer space ever throttle the sender?

13. Are there any retransmitted segments in the trace file? What did you check for (in the trace) in order to answer this question?

3.1.4 TCP congestion control in action (optional)

Let's now examine the amount of data sent per unit time from the client to the server. Rather than (tediously!) calculating this from the raw data in the Wireshark window, we'll use one of Wireshark's TCP graphing utilities—Time-Sequence-Graph (Stevens)—to plot out data.

Select a TCP segment in the Wireshark's "listing of captured-packets" window. Then select the menu : Statistics −> TCP Stream Graph −> Time-Sequence-Graph(Stevens). In theory, you should see a plot that looks similar to the plot as shown in Figure 4, which was created from the captured packets in the packet trace tcp-ethereal-trace-1 in http://gaia.cs.umass.edu/wireshark-labs/wireshark-traces.zip. In practice, you're unlikely to see such neat results with your own trace.

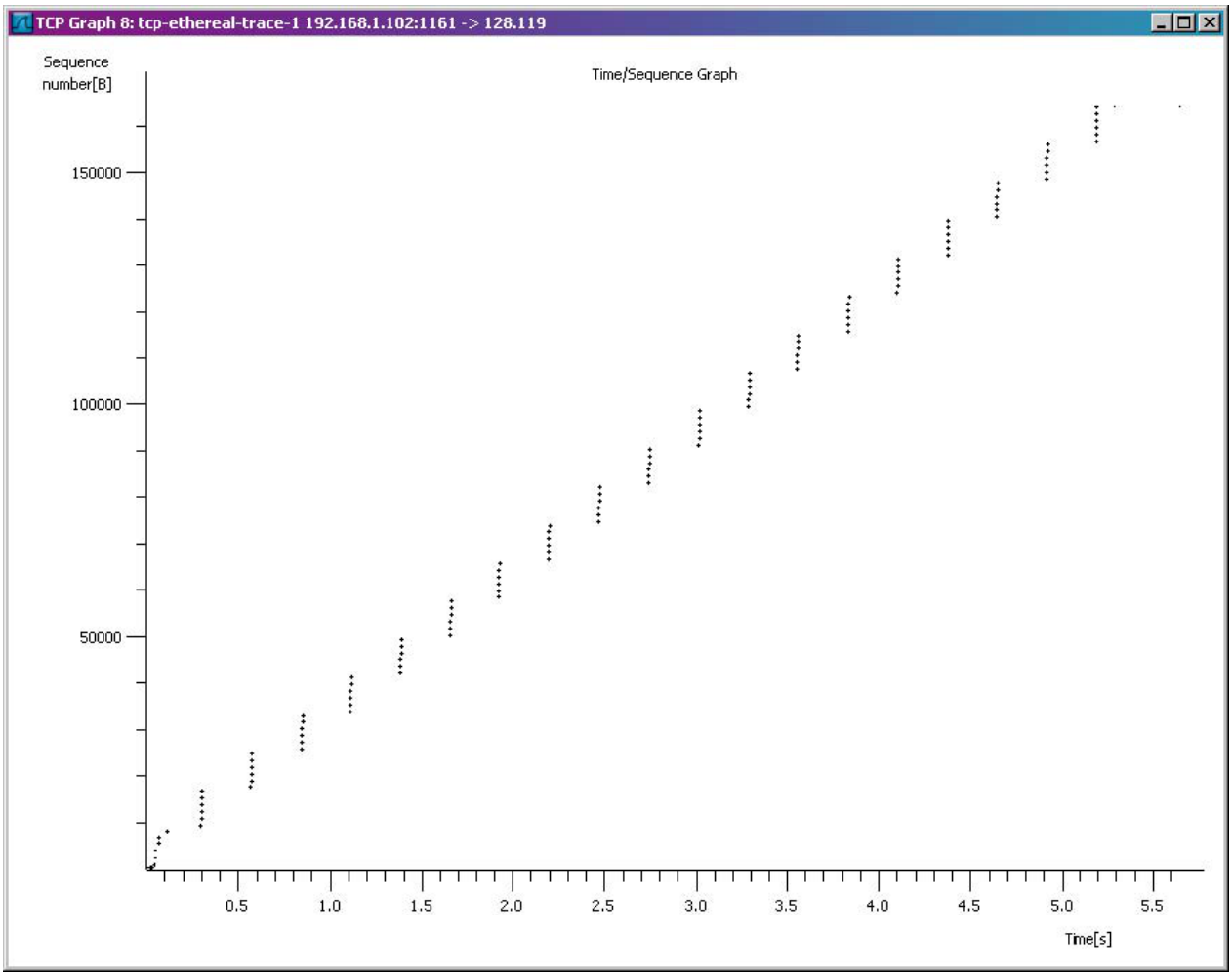


Figure 4: Time sequence graph with the book's clean data.

Here, each dot represents a TCP segment sent, plotting the sequence number of the segment versus the time at which it was sent. Note that a set of dots stacked above each other represents a series of packets that were sent back-to-back by the sender.

Use the Time-Sequence-Graph(Stevens) plotting tool to view the sequence number versus time plot of segments being sent from the client to the gaia.cs.umass.edu server. Can you identify where TCP's slowstart phase begins and ends, and where congestion avoidance takes over? If not (or extremely hard to find out form the graph), why not? Comment on ways in which the measured data differs from the idealised behaviour of TCP that we've studied in the text.

3.2 IP Wireshark lab

In this part of the practical assignment, you'll investigate the IP protocol, focusing on the IP datagram. We'll do so by analysing a trace of IP datagrams sent and received by an execution of the `traceroute` program[2]. We'll investigate the various fields in the IP datagram, and study IP fragmentation in detail.

Before beginning this lab, you'll probably want to review sections 1.4.3 in the text and section 3.4 of RFC 2151[3] to update yourself on the operation of the `traceroute` program. You'll also want to read Section 4.4 in the text, and probably also have RFC 791[4] on hand as well, for a discussion of the IP protocol.

3.2.1 Capturing packets from an execution of `traceroute`

In order to generate a trace of IP datagrams for this lab, we'll use the `traceroute` program to send datagrams of different sizes towards some destination, X. Recall that `traceroute` operates by first sending one or more datagrams with the time-to-live (TTL) field in the IP header set to 1; it then sends a series of one or more datagrams towards the same destination with a TTL value of 2; it then sends a series of datagrams towards the same destination with a TTL value of 3; and so on. Recall that a router must decrement the TTL in each received datagram by 1 (actually, RFC 791 says that the router must decrement the TTL by at least one). If the TTL reaches 0, the router returns an ICMP message (type 11 – TTL-exceeded) to the sending host. As a result of this behaviour, a datagram with a TTL of 1 (sent by the host executing `traceroute`) will cause the router one hop away from the sender to send an ICMP TTL-exceeded message back to the sender; the datagram sent with a TTL of 2 will cause the router two hops away to send an ICMP message back to the sender; the datagram sent with a TTL of 3 will cause the router three hops away to send an ICMP message back to the sender; and so on. In this manner, the host executing `traceroute` can learn the identities of the routers between itself and destination X by looking at the source IP addresses in the datagrams containing the ICMP TTL-exceeded messages.

We'll want to run `traceroute` and have it send datagrams of various lengths.

- Windows. The `tracert` program (used for our ICMP Wireshark lab) provided with Windows does not allow one to change the size of the ICMP echo request (ping) message sent by the `tracert` program[5].
- Linux/Unix/MacOS. With the Unix/MacOS `traceroute` command, the size of the UDP datagram sent towards the destination can be explicitly set by indicating the number of bytes in the datagram; this value is entered in the `traceroute` command line immediately after the name or address of the destination. For example, to send `traceroute` datagrams of 2000 bytes towards gaia.cs.umass.edu, the command would be:
  `%{\tt traceroute} gaia.cs.umass.edu 2000`

Do the following:

– Start up Wireshark and begin packet capture (Capture −> Start) and then press OK on the Wireshark Packet Capture Options screen (we won't need to select any options here).

– If you are using a Windows platform, use the command prompt[6]. If you are using a Unix or Mac platform, enter three `traceroute` commands, one with a length of 56 bytes, one with

[2]The `traceroute` program itself is explored in more detail elsewhere in the Wireshark ICMP lab that we don't cover in the pracs, but is available on Vula in case you're interested

[3]`ftp://ftp.rfc-editor.org/in-notes/rfc2151.txt`

[4]`ftp://ftp.rfc-editor.org/in-notes/rfc791.txt`

[5]A nicer Windows `traceroute` program is pingplotter, available both in free version and shareware versions at http://www.pingplotter.com. Download and install `pingplotter`, and test it out by performing a few `traceroute`s to your favorite sites. The size of the ICMP echo request message can be explicitly set in `pingplotter` by selecting the menu item Edit − > Options − > Packet Options and then filling in the Packet Size field. The default packet size is 56 bytes. Once `pingplotter` has sent a series of packets with the increasing TTL values, it restarts the sending process again with a TTL of 1, after waiting Trace Interval amount of time. The value of Trace Interval and the number of intervals can be explicitly set in `pingplotter`.

[6]If you use pingplotter, then: start up pingplotter and enter the name of a target destination in the "Address to

a length of 2000 bytes, and one with a length of 3500 bytes. You can do the same with `pingplotter`, but not with `tracert`.
- Next, send a set of datagrams with a longer length, by selecting Edit −> Advanced Options −> Packet Options and enter a value of 2000 in the Packet Size field and then press OK. Then press the Resume button2.
- Stop Wireshark tracing when it has completed the trace.

3.2.2 A look at the captured trace

In your trace, you should be able to see the series of ICMP Echo Request (in the case of Windows machine) or the UDP segment (in the case of Unix) sent by your computer and the ICMP TTL-exceeded messages returned to your computer by the intermediate routers. In the questions below, we'll assume you are using a Windows machine; the corresponding questions for the case of a Unix machine should be clear. Whenever possible, when answering a question below you should hand in a softcopy of the packet(s) within the trace that you used to answer the question asked. When you hand in your assignment, annotate the output so that it's clear where in the output you're getting the information for your answer. (To print a packet, use File −> Print, choose Selected packet only, choose Packet summary line, and select the minimum amount of packet detail that you need to answer the question.)

Exercises

14. Select the first ICMP Echo Request message sent by your computer, and expand the Internet Protocol part of the packet in the packet details window (see also Fig. 5). What is the IP address of your computer?

15. Within the IP packet header, what is the value in the upper layer protocol field?

16. How many bytes are in the IP header? How many bytes are in the payload of the IP datagram?

17. Has this IP datagram been fragmented? Explain how you determined whether or not the datagram has been fragmented.

Next, sort the traced packets according to IP source address by clicking on the Source column header; a small downward pointing arrow should appear next to the word Source. If the arrow points up, click on the Source column header again. Select the first ICMP Echo Request message sent by your computer, and expand the Internet Protocol portion in the "details of selected packet header" window. In the "listing of captured packets" window, you should see all of the subsequent ICMP messages (perhaps with additional interspersed packets sent by other protocols running on your computer) below this first ICMP. Use the down arrow to move through the ICMP messages sent by your computer.

18. Which fields in the IP datagram always change from one datagram to the next within this series of ICMP messages sent by your computer?

19. Which fields stay constant? Which of the fields *must* stay constant? Which fields must change? Why?

20. Describe the pattern you see in the values in the Identification field of the IP datagram

Next (with the packets still sorted by source address) find the series of ICMP TTL-exceeded replies sent to your computer by the nearest router.

Trace Window." Enter 3 in the "# of times to Trace" field, so you don't gather too much data. Select the menu item Edit −> Advanced Options −> Packet Options and enter a value of 56 in the Packet Size field and then press OK. Then press the Trace button.

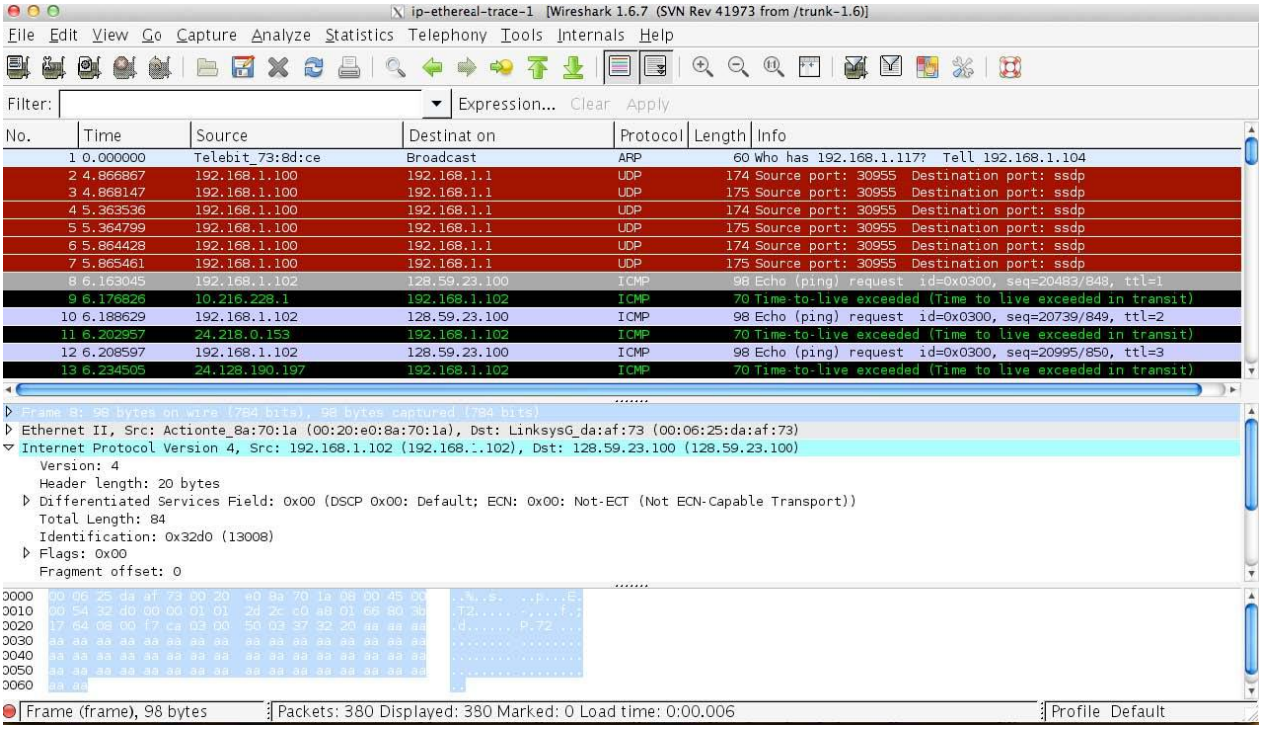


Figure 5: Screenshot related to question 14.

21. Describe how you found it. What is the value in the Identification field and the TTL field?

22. Do these values remain unchanged for all of the ICMP TTL-exceeded replies sent to your computer by the nearest (first hop) router? Why?

Recall: include the following in the compressed file for submission:

1. *The chat client class and the server class.*
2. *A screenshot with a chat session where at least two clients participate (using the one from Section 2.1), one having your student number, the other your last name.*
3. *Your Wireshark traces*
4. *The answers to the Questions*

Credits: these Wireshark exercises are heavily based on Kurose and Ross' material for the Wireshark labs accompanying the book, but note that *some strategic modifications have been made* to prevent you from copying the answers from those online available.