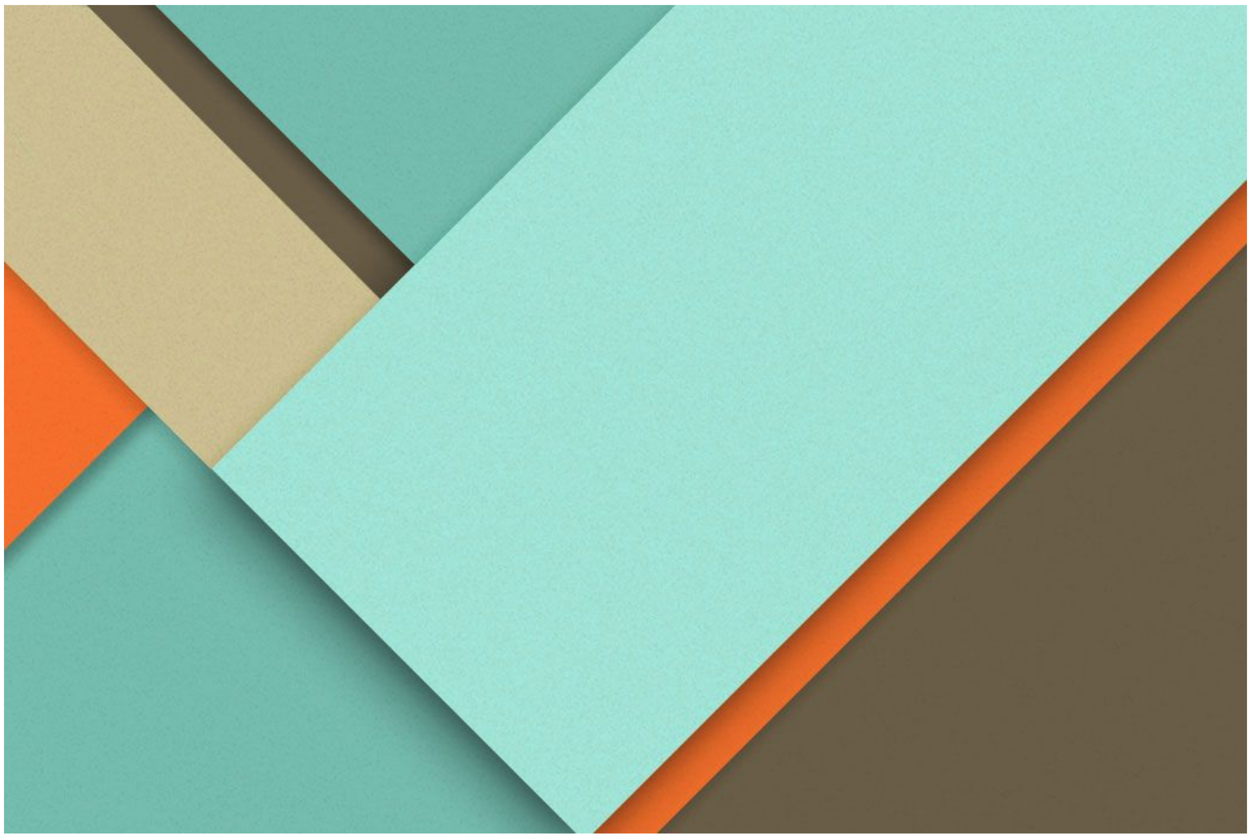


Computer Networks I

CNT 4004.001 - Dr. Miguel Labrador

Socket Programming Assignment

Due: 6 October 2016

Matthew Kramer
U20891900
Sterling Price
U49743231

Table of Contents

Introduction

In a nutshell, the focus of this project was socket programming. Making use of both the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), we used Java to provide a foundational understanding . Simple client and server programs were either provided (**`TCPEchoClient.java`**, **`TCPEchoServer.java`**, **`PingServer.java`**) or created (**`PingClient.java`**) in order to communicate using the respective protocols. This exercise was primarily a proof-of-concept, though it served to create a real-world construct of concepts discussed in class, helping not only to provide experience in programming using sockets, but also to further reinforce our existing knowledge of networks .

Approach

The goals of this assignment were accomplished using a remote desktop connection to the USF Central Instructional and Research Computing Environment (CIRCE). Client-facing machines run Scientific Linux v6.3 which was sufficient for our purposes. While we could have used any number of methods to develop, compile, and execute these programs, we found this to be the best option as it allowed us to access a Linux-based system on-the-fly and, moreover, provided a secure and accessible space in which to store files. In the end, we did resort to GitHub for better version control and more fluid collaborative working.

Part 1: TCP Echo

Execution

The TCP echo client and server programs were borrowed from *TCP/IP Sockets in Java* (2nd Edition) by Kenneth L. Calvert and Michael J. Donahoo. The source code for these programs can be found through the following links: TCPEchoClient.java and TCPEchoServer.java. If you download the code from the provided links, you may compile it by running the following commands in a terminal of your choice. If you have downloaded our source code, you may also compile it by using the **`make`** command from within the directory.

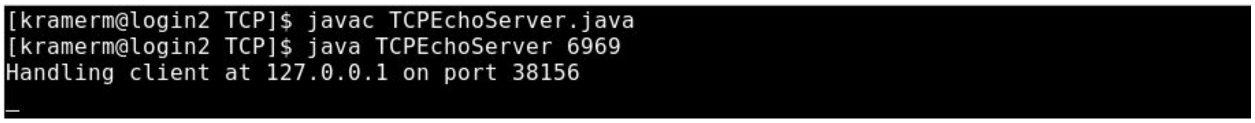
```
javac TCPEchoServer.java

javac TCPEchoClient.java
```

To execute the program properly, the **`TCPEchoServer`** program must be running before the **`TCPEchoClient`**. This is accomplished by running the following commands in separate terminal windows.

```
java TCPEchoServer [PORT]
java TCPEchoClient [HOST] [MESSAGE] [PORT]
```

Output

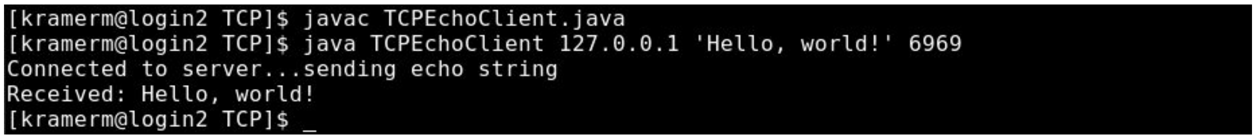
When the TCP echo server and client are run successfully, you can expect output from both of the running programs. On the server side (Fig. 0), nothing will appear in the terminal until a packet has been sent from the client to the server, after which it will notify the user that it is handling a client.



```
[kramerm@login2 TCP]$ javac TCPEchoServer.java
[kramerm@login2 TCP]$ java TCPEchoServer 6969
Handling client at 127.0.0.1 on port 38156
_
```

***[Figure 0]:** Sample output from the TCPEchoServer.java program.*

When running the client side of the program (Fig. 1), however, you will immediately see the attempt to connect to the server in addition to what the client received as a response from the server.



```
[kramerm@login2 TCP]$ javac TCPEchoClient.java
[kramerm@login2 TCP]$ java TCPEchoClient 127.0.0.1 'Hello, world!' 6969
Connected to server...sending echo string
Received: Hello, world!
[kramerm@login2 TCP]$ _
```

***[Figure 1]:** Sample output from the TCPEchoClient.java program.*

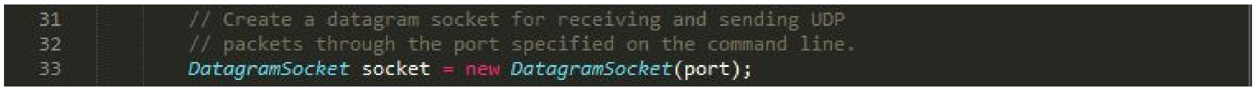
Analysis

As we know, TCP is a connection-oriented protocol, meaning that while it is extremely reliable, it is consequently slower than, for example, UDP. Using the **`TCPEchoServer`** and **`TCPEchoClient`**, the message that you send will never be lost and will always be sent back once it is received by the server.

Part 2: UDP Pinger

Development

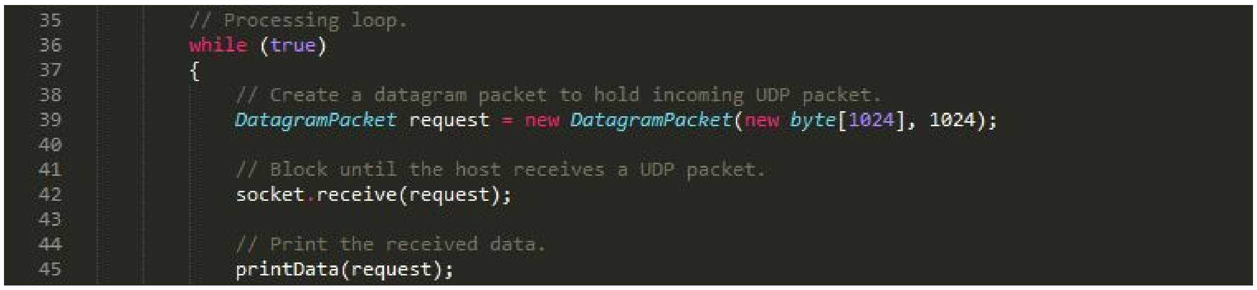
As the project guidelines suggest, the UDP ping server code may be used as an aid to develop the UDP ping client program. The TCP echo client code, in addition to the program's interaction with the TCP echo server, proved to be a useful reference for this part of the project. The UDP ping server's true functionality begins with the creation of a new datagram socket (Fig. 2). Since this is the server side of the program, it simply needs to know through which port it should open the socket, as it will use the local host machine address: **`127.0.0.1`**.



```
        // Create a datagram socket for receiving and sending UDP
        // packets through the port specified on the command line.
        DatagramSocket socket = new DatagramSocket(port);
```

***[Figure 2]:** Datagram socket creation using user-specified port.*

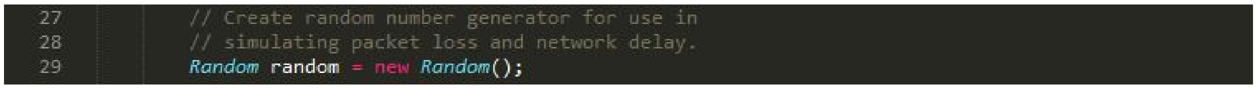
With the socket now established, the server can begin waiting to receive incoming packets from clients. To accomplish this, an infinite **`while`** loop (Fig. 3) is started that continually receives incoming packets. For each iteration, a new datagram packet is created that will hold the client's request. Once complete, the server waits until a packet is received, placing it in the newly created request packet. Finally, the server outputs the data from the packet using the included **`printData`** method.



```
        // Processing loop.
        while (true)
        {
            // Create a datagram packet to hold incoming UDP packet.
            DatagramPacket request = new DatagramPacket(new byte[1024], 1024);

            // Block until the host receives a UDP packet.
            socket.receive(request);

            // Print the received data.
            printData(request);
```

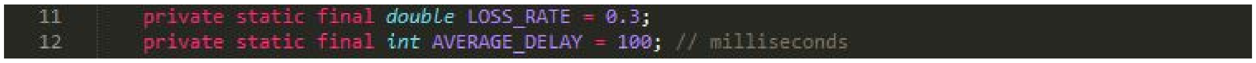
***[Figure 3]:** Infinite while loop to process incoming requests to the server.*

For each iteration of this processing loop, the **`PingServer`** program additionally simulates packet loss and network delay by using a random number class (Fig. 4) in conjunction with global variables that can be adjusted by the user (Fig. 5).



```
        // Create random number generator for use in
        // simulating packet loss and network delay.
        Random random = new Random();
```

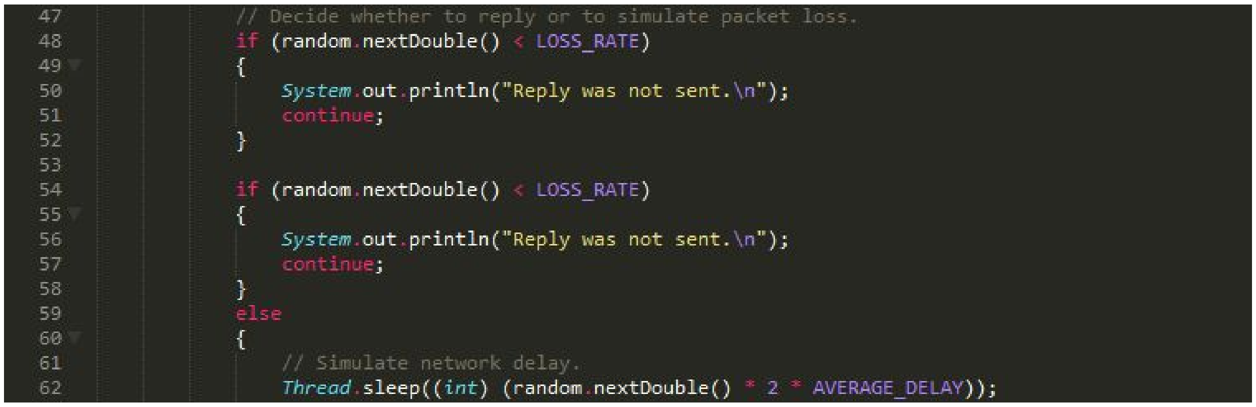
***[Figure 4]:** Random number generator used in simulating packet loss and network delay.*



```
    private static final double LOSS_RATE = 0.3;
    private static final int AVERAGE_DELAY = 100; // milliseconds
```

***[Figure 5]:** Global variables used in simulating packet loss and network delay.*

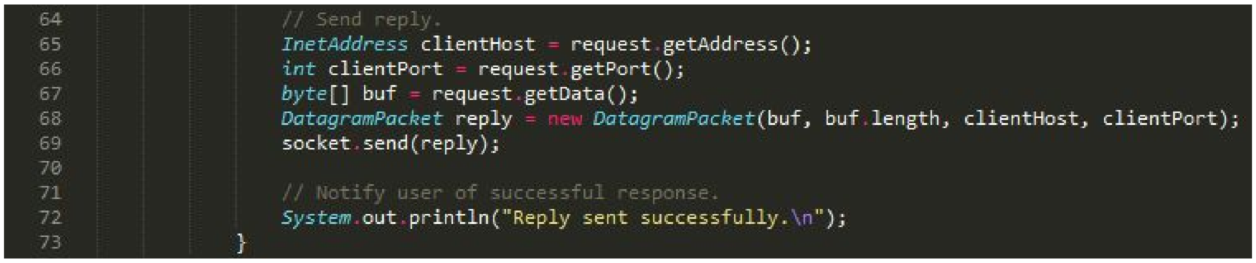
Using this random number, the program determines whether or not to send a reply by testing against the global variable **`LOSS_RATE`** twice (Fig. 6). To simulate network delay, **`PingServer`** sleeps for a random time before sending the reply. This delay is the product of the random number and a preset average in **`AVERAGE_DELAY`** (Fig. 6, Line 62).



```
// Decide whether to reply or to simulate packet loss.
if (random.nextDouble() < LOSS_RATE)
{
    System.out.println("Reply was not sent.\n");
    continue;
}

if (random.nextDouble() < LOSS_RATE)
{
    System.out.println("Reply was not sent.\n");
    continue;
}
else
{
    // Simulate network delay.
    Thread.sleep((int) (random.nextDouble() * 2 * AVERAGE_DELAY));
```

***[Figure 6]:** Simulating packet loss and network delay.*

Finally, a reply can be sent back to the client (Fig. 7). The address and port of the client as well as the data in the packet is found in the datagram packet that was received with the **`getAddress`**, **`getPort`**, and **`getData`** methods of datagram packet classes, respectively. Using these parameters, a new datagram packet is created and addressed back to that client. This packet contains the same data that was received. Once sent, the user is notified of a successful response.

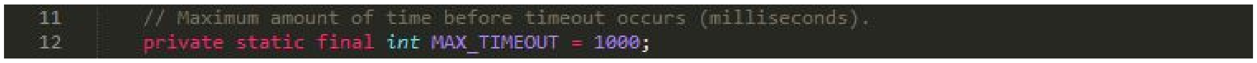


```
    // Send reply.
    InetAddress clientHost = request.getAddress();
    int clientPort = request.getPort();
    byte[] buf = request.getData();
    DatagramPacket reply = new DatagramPacket(buf, buf.length, clientHost, clientPort);
    socket.send(reply);

    // Notify user of successful response.
    System.out.println("Reply sent successfully.\n");
}
```

***[Figure 7]:** Retrieving address and port of client before sending a reply packet.*

Now that we understand how the **`PingServer`** program works, we can begin to see the process of developing its counterpart, the **`PingClient`**.

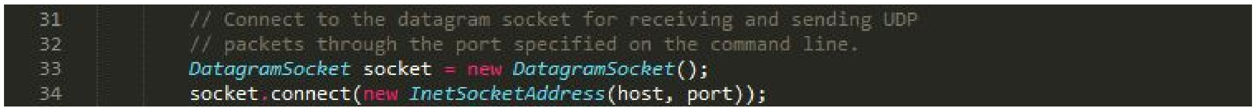
The 1 second delay between messages is defined by a global integer variable **`MAX_TIMEOUT`** (Fig. 8) set to 1000 (for milliseconds). This value is later used in the **`setSoTimeout`** function (Fig. 11, Line 61) of the datagram socket to establish how long the socket should wait for a reply from the server before timing out. This delay is necessary to ensure accuracy with the 'hit' or 'miss' of each message to the server.



```
// Maximum amount of time before timeout occurs (milliseconds).
private static final int MAX_TIMEOUT = 1000;
```

***[Figure 8]:** Global variable for setting the maximum delay before timeout in milliseconds.*

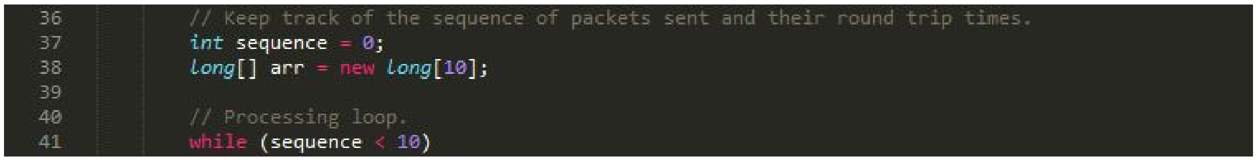
To communicate with the server, the client must connect to the same port that the server is looking for incoming packets on. To do this, the client connects to the user-specified port and address (Fig. 9). The client must connect to the existing socket as it would be unable to open it since the server has already done that.



```
31        // Connect to the datagram socket for receiving and sending UDP
32        // packets through the port specified on the command line.
33        DatagramSocket socket = new DatagramSocket();
34        socket.connect(new InetSocketAddress(host, port));
```

***[Figure 9]:** Connecting to the datagram socket.*

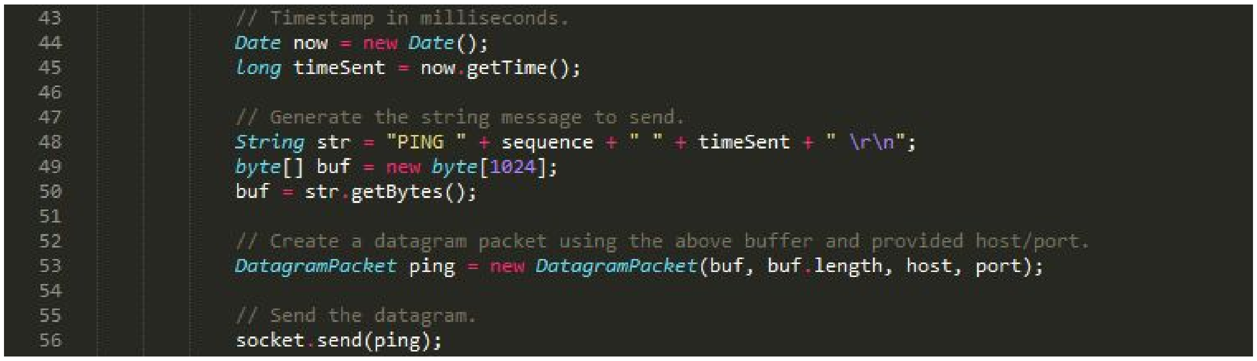
After connecting to the socket, the client must now begin sending packets. A **`while`** loop is used that iterates 10 times (for the 10 required packets). This same parameter **`sequence`** is also used when generating the message for the packet (Fig. 11, Line 48). Also, **`long`** array is declared (Fig. 10, Line 38) to store the round-trip time of each packet to be used later to calculate the minimum, maximum, and average round-trip time (Fig. 12).



```
36        // Keep track of the sequence of packets sent and their round trip times.
37        int sequence = 0;
38        long[] arr = new long[10];
39
40        // Processing loop.
41        while (sequence < 10)
```

***[Figure 10]:** Tracking the packet sequence and establishing an array for round-trip times.*

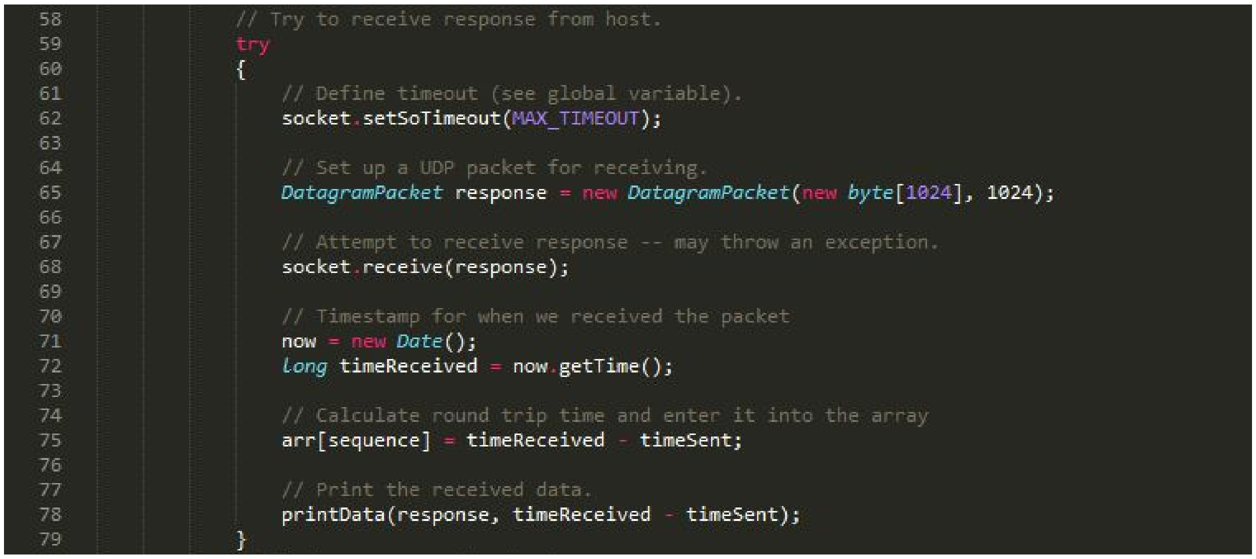
Within each iteration of this **`while`** loop, the client generates a packet that it will send to the server. The message sent is a string composed of four parts: a generic **`'PING'`** header, the packet's sequence number (0 to 9), the time that the packet was sent, and the carriage return ('**`\r`**') and line feed ('**`\n`**') characters. This string is entered into a byte buffer using the **`getBytes`** method before being encapsulated in the datagram packet **`ping`**.



```
43            // Timestamp in milliseconds.
44            Date now = new Date();
45            long timeSent = now.getTime();
46
47            // Generate the string message to send.
48            String str = "PING " + sequence + " " + timeSent + " \r\n";
49            byte[] buf = new byte[1024];
50            buf = str.getBytes();
51
52            // Create a datagram packet using the above buffer and provided host/port.
53            DatagramPacket ping = new DatagramPacket(buf, buf.length, host, port);
54
55            // Send the datagram.
56            socket.send(ping);
```

***[Figure 11]:** Generating the packet to be sent to the server.*

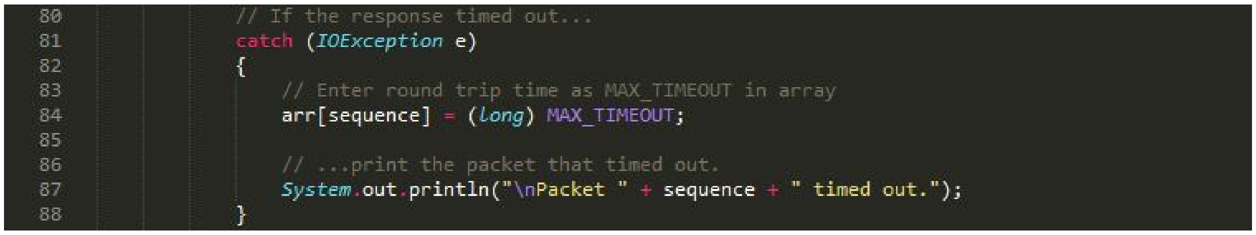
A datagram packet **`response`** is created and used to receive the response from the server confirming that the transmission was successful. If a response is received (Fig.12, Line 68), the round-trip time of the packet is calculated by retrieving the current time when the packet is sent (Fig. 11, Line 44) and when the response from the server is received (Fig. 12, Line 71) before printing this data.



```
// Try to receive response from host.
try
{
    // Define timeout (see global variable).
    socket.setSoTimeout(MAX_TIMEOUT);

    // Set up a UDP packet for receiving.
    DatagramPacket response = new DatagramPacket(new byte[1024], 1024);

    // Attempt to receive response -- may throw an exception.
    socket.receive(response);

    // Timestamp for when we received the packet
    now = new Date();
    Long timeReceived = now.getTime();

    // Calculate round trip time and enter it into the array
    arr[sequence] = timeReceived - timeSent;

    // Print the received data.
    printData(response, timeReceived - timeSent);
}
```

[Figure 12]: *Try block for receiving a reply from the server.*

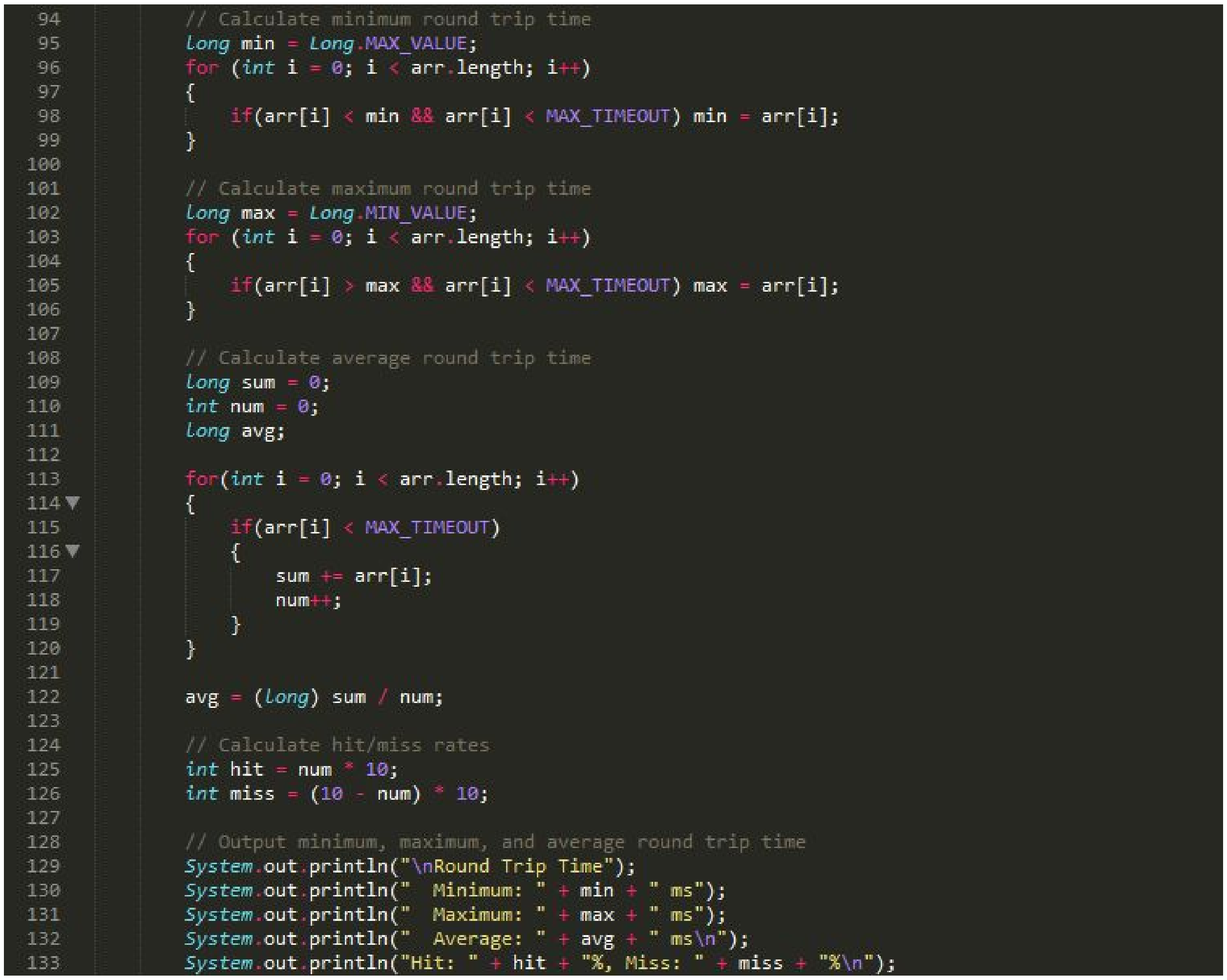
If a response is not received within the **`try`** block, an exception is thrown by the **`receive`** method and is caught in the corresponding **`catch`** block (Fig. 13). Here, the round-trip time for packet is entered into the array as the preset **`MAX_TIMEOUT`** value before notifying the user of what happened.



```
// If the response timed out...
catch (IOException e)
{
    // Enter round trip time as MAX_TIMEOUT in array
    arr[sequence] = (Long) MAX_TIMEOUT;

    // ...print the packet that timed out.
    System.out.println("\nPacket " + sequence + " timed out.");
}
```

[Figure 13]: *Catch block for receiving a reply from the server.*

Finally, the program performs some calculations (Fig. 14) to assess performance. Note that in each case, the program does not include the timed-out packets in its calculations. The minimum round-trip time, maximum round-trip time, average round-trip time, hit rate, and miss rate is all calculated before that data is output for the user.



```java
        // Calculate minimum round trip time
        Long min = Long.MAX_VALUE;
        for (int i = 0; i < arr.length; i++)
        {
            if(arr[i] < min && arr[i] < MAX_TIMEOUT) min = arr[i];
        }

        // Calculate maximum round trip time
        Long max = Long.MIN_VALUE;
        for (int i = 0; i < arr.length; i++)
        {
            if(arr[i] > max && arr[i] < MAX_TIMEOUT) max = arr[i];
        }

        // Calculate average round trip time
        Long sum = 0;
        int num = 0;
        Long avg;

        for(int i = 0; i < arr.length; i++)
        {
            if(arr[i] < MAX_TIMEOUT)
            {
                sum += arr[i];
                num++;
            }
        }

        avg = (Long) sum / num;

        // Calculate hit/miss rates
        int hit = num * 10;
        int miss = (10 - num) * 10;

        // Output minimum, maximum, and average round trip time
        System.out.println("\nRound Trip Time");
        System.out.println("  Minimum: " + min + " ms");
        System.out.println("  Maximum: " + max + " ms");
        System.out.println("  Average: " + avg + " ms\n");
        System.out.println("Hit: " + hit + "%, Miss: " + miss + "%\n");
```

[Figure 14]: *Post-communication data processing and output.*

Execution

This code may be compiled by entering the following commands into a terminal of your choice. Alternatively, you may simply run the **`make`** command to have this done for you.

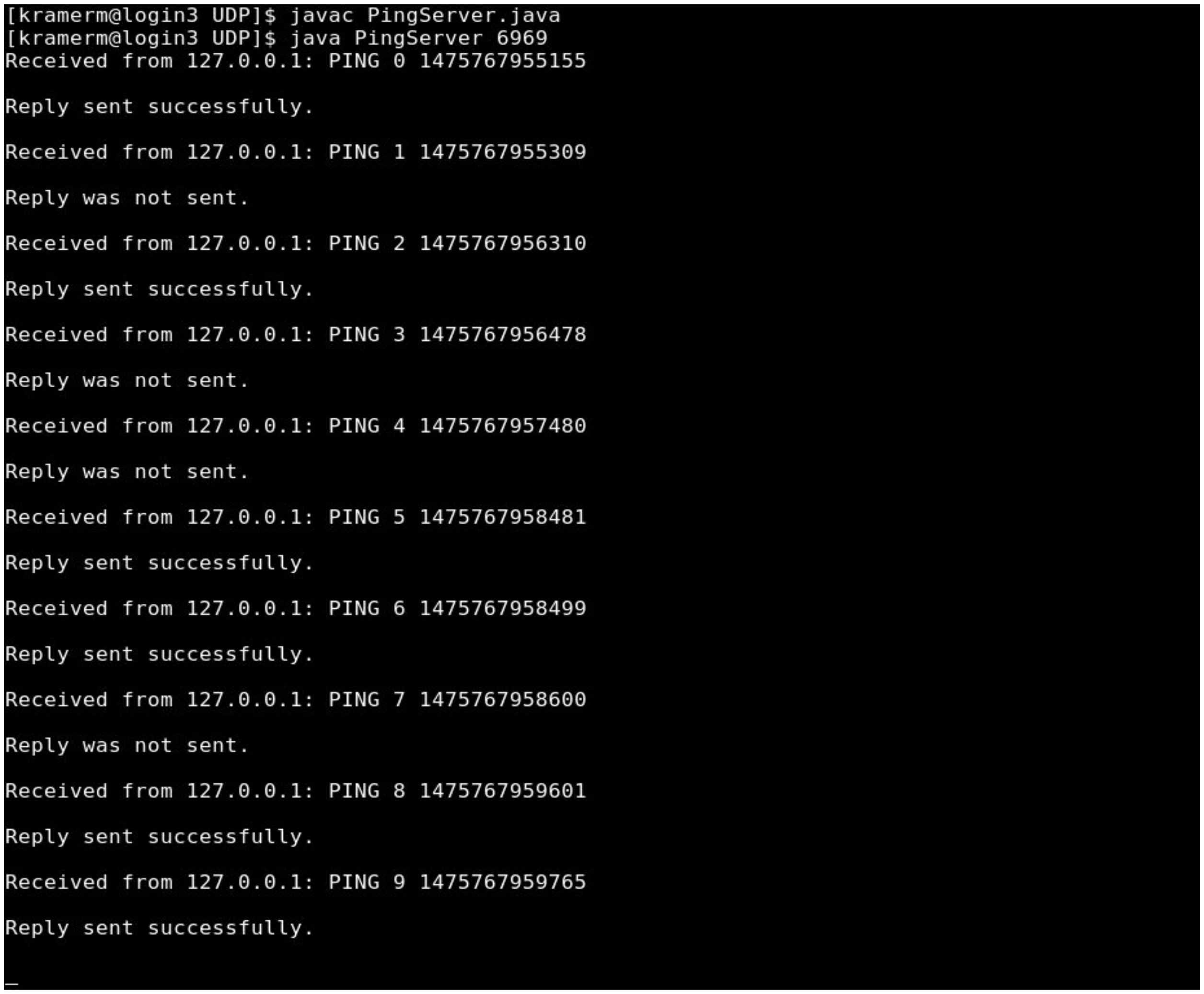
```
javac PingServer.java

javac PingClient.java
```

In order to execute this program, the **`PingServer`** program must be started prior to running the **`PingClient`** program. This is achieved by running the commands below.

```
java PingServer [PORT]

java PingClient [HOST] [PORT]
```

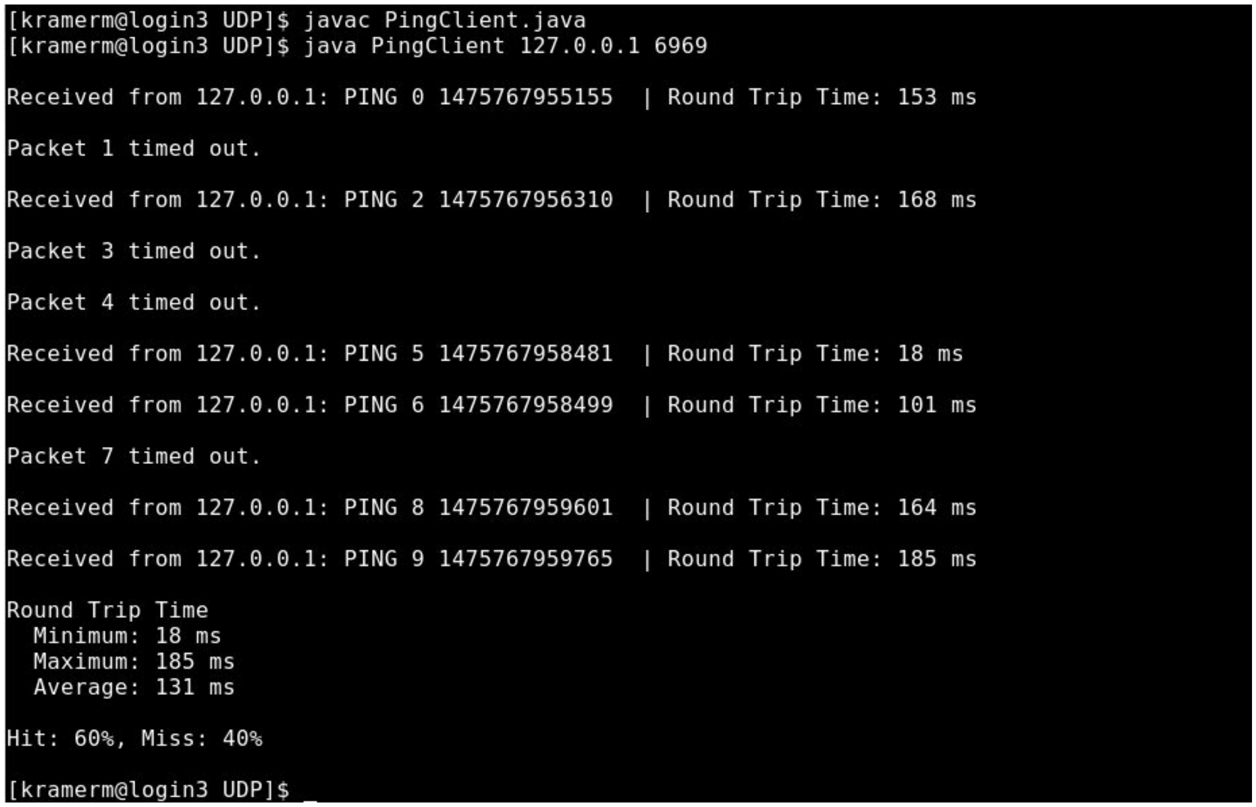
Output

The expected output from the UDP pinger program can be found below.



```
[kramerm@login3 UDP]$ javac PingServer.java
[kramerm@login3 UDP]$ java PingServer 6969
Received from 127.0.0.1: PING 0 1475767955155

Reply sent successfully.

Received from 127.0.0.1: PING 1 1475767955309

Reply was not sent.

Received from 127.0.0.1: PING 2 1475767956310

Reply sent successfully.

Received from 127.0.0.1: PING 3 1475767956478

Reply was not sent.

Received from 127.0.0.1: PING 4 1475767957480

Reply was not sent.

Received from 127.0.0.1: PING 5 1475767958481

Reply sent successfully.

Received from 127.0.0.1: PING 6 1475767958499

Reply sent successfully.

Received from 127.0.0.1: PING 7 1475767958600

Reply was not sent.

Received from 127.0.0.1: PING 8 1475767959601

Reply sent successfully.

Received from 127.0.0.1: PING 9 1475767959765

Reply sent successfully.

_
```

[Figure 15]: *Sample output from the PingServer.java program.*



```
[kramerm@login3 UDP]$ javac PingClient.java
[kramerm@login3 UDP]$ java PingClient 127.0.0.1 6969

Received from 127.0.0.1: PING 0 1475767955155  | Round Trip Time: 153 ms

Packet 1 timed out.

Received from 127.0.0.1: PING 2 1475767956310  | Round Trip Time: 168 ms

Packet 3 timed out.

Packet 4 timed out.

Received from 127.0.0.1: PING 5 1475767958481  | Round Trip Time: 18 ms

Received from 127.0.0.1: PING 6 1475767958499  | Round Trip Time: 101 ms

Packet 7 timed out.

Received from 127.0.0.1: PING 8 1475767959601  | Round Trip Time: 164 ms

Received from 127.0.0.1: PING 9 1475767959765  | Round Trip Time: 185 ms

Round Trip Time
  Minimum: 18 ms
  Maximum: 185 ms
  Average: 131 ms

Hit: 60%, Miss: 40%

[kramerm@login3 UDP]$ _
```

[Figure 16]: *Sample output from the PingClient.java program.*

Analysis

UDP is inherently a connectionless protocol. As a result, data is transferred quickly but is sometimes lost due to network delay or. Our sample output (Fig. 16), for example, saw 6 successful responses out of the 10 total pings, or a 60% hit rate. Further execution resulted in varied results in this regard, with rates as low as 30% and as high as 90%. This large variance reinforces the notion that UDP is an unreliable protocol.

Ending Remarks

All in all, this project proved to be both challenging and enlightening. Through it, we gained a deeper understanding of these two protocols and experience with socket programming using Java. TCP and UDP are very different protocols, but they each have their purpose. While TCP is a far more reliable protocol, it is slower and more resource intensive due to the measures that it takes to maintain the integrity of the data. On the other hand, despite being an exceptionally fast protocol, UDP consequently suffers far more packet loss. Because of the stark differences between them, it becomes clear that there are situations whether each is unrivaled in its abilities.