



Linnæus University

Sweden

Report

Assignment 1

Socket programming



Author: Husamu Aldeen ALKHAFIJI -
ha223cz

Term: HT20

Course code: Introduction to networking -
1DV701



Contents

1	Problem 1	2
1.1	Discussion	2
2	Problem 2	2
2.1	Discussion	2
2.2	VG-task 1	5
2.3	VG-task 2	6
2.4	VG-task 3	9
3	Problem 3	9
3.1	Discussion	9
4	Problem 4	10
4.1	Discussion	10
5	Problem 5	13
5.1	Discussion	13



1 Problem 1

1.1 Discussion

We have set up the virtual environment just as specified by the assignment. The following figure shows a ping between the two machines.

```
64 bytes from 192.168.56.102: icmp_seq=10 ttl=64 time=0.336 ms
64 bytes from 192.168.56.102: icmp_seq=11 ttl=64 time=0.667 ms
^C64 bytes from 192.168.56.102: icmp_seq=12 ttl=64 time=0.270 ms
64 bytes from 192.168.56.102: icmp_seq=13 ttl=64 time=0.323 ms
64 bytes from 192.168.56.102: icmp_seq=14 ttl=64 time=0.262 ms
64 bytes from 192.168.56.102: icmp_seq=15 ttl=64 time=0.137 ms
^C
--- 192.168.56.102 ping statistics ---
15 packets transmitted, 15 received, 0% packet loss, time 14334ms
rtt min/avg/max/mdev = 0.137/0.364/0.815/0.187 ms
[sam66ish@parrot:~/Programming/UDPEcho]
$ping -c 5 192.168.56.102
PING 192.168.56.102 (192.168.56.102) 56(84) bytes of data:
64 bytes from 192.168.56.102: icmp_seq=1 ttl=64 time=0.241 ms
64 bytes from 192.168.56.102: icmp_seq=2 ttl=64 time=0.197 ms
64 bytes from 192.168.56.102: icmp_seq=3 ttl=64 time=0.270 ms
64 bytes from 192.168.56.102: icmp_seq=4 ttl=64 time=0.343 ms
64 bytes from 192.168.56.102: icmp_seq=5 ttl=64 time=0.368 ms
--- 192.168.56.102 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4082ms
rtt min/avg/max/mdev = 0.197/0.283/0.368/0.063 ms
[sam66ish@parrot:~/Programming/UDPEcho]
```

Figure 1. The main machine (foreground) pinging the virtual machine (background).

2 Problem 2

2.1 Discussion

In this problem, we first downloaded the provided UDP program. We made sure to run the server in the virtual machine and the client side in the main machine. The program runs as expected.

```
19  /* Create socket */
20  DatagramSocket socket= new DatagramSocket( bindaddr: null);
21
22  /* Create local endpoint using bind() */
23  SocketAddress localBindPoint= new InetSocketAddress(NIPORT);
24  socket.bind(localBindPoint);
25
26  /* Create remote endpoint */
27  SocketAddress remoteBindPoint=
28      new InetSocketAddress(args[0],
29          Integer.valueOf(args[1]));
30
31  /* Create datagram packet for sending message */
32  DatagramPacket sendPacket=
33      new DatagramPacket(msg.getBytes(),
34          msg.length(),
35          remoteBindPoint);
36
37  socket.send(sendPacket);
38
39  /* Receive message */
40  DatagramPacket receivePacket= new DatagramPacket(new byte[1024], 1024, socket);
41  socket.receive(receivePacket);
42  String msg= receivePacket.getData().toString();
43  System.out.println("Received: " + msg);
44
45  /* Close socket */
46  socket.close();
47
48  }
49
50  }
51
52  }
53
54  }
55
56  }
57
58  }
59
60  }
61
62  }
63
64  }
65
66  }
67
68  }
69
70  }
71
72  }
73
74  }
75
76  }
77
78  }
79
80  }
81
82  }
83
84  }
85
86  }
87
88  }
89
90  }
91
92  }
93
94  }
95
96  }
97
98  }
99
100 }
```

Run: UDPEchoClient

/usr/lib/jvm/java-11.0-openjdk-amd64/bin/java -javaagent:/home/sam66ish/Programming/idea-IC-203.6682.168/lib/idea_rt.jar-36915:/home/sam66ish/Programming/UDPEcho -UDPEchoClient

10 bytes sent and received

Process finished with exit code 0

Figure 2. The client program running as expected.

Next we are supposed to augment the functionality of the program as follows:
Add the following as arguments to the program:

- Buffer size.
- Msg transfer rate, if it's zero, transfer only once.
- Echo message.

We also updated the client to keep sending the packets at the transfer rate specified until the program is aborted manually. We used a transfer rate of 5 messages per second and we let the program run for 30 seconds before taking a screenshot of the result which is shown in the next figure.

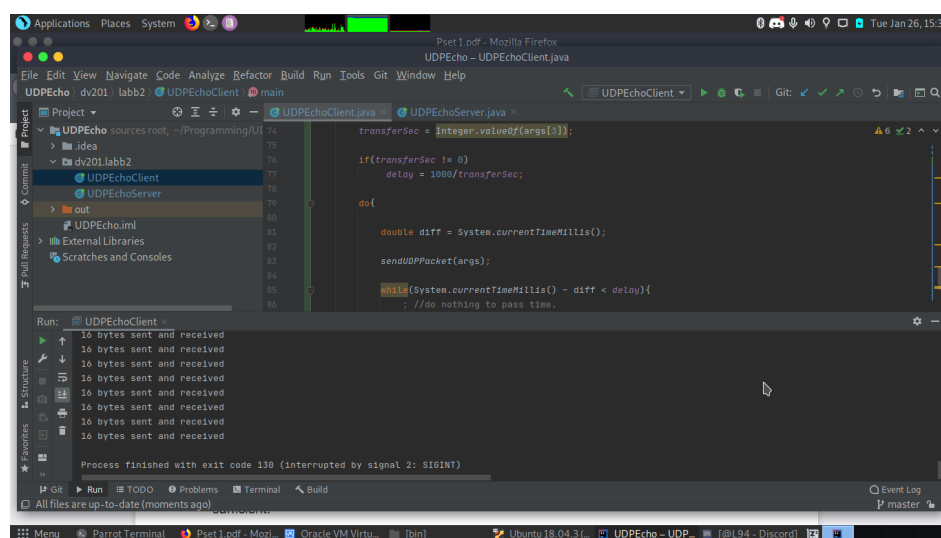


Figure 3. The client program running for 30 seconds sending 5 messages per second.

Finally we added error and exception handling to all the classes to make sure all the cases are covered. The things that could go wrong are by example.

- Incorrect IP address: inputting IPs that are invalid such as 0.0.0.0 and 255.255.255.255 and other reserved IPs such as the network signature which has the network subset of the IP set and the hosts subset all set to zero. The boundary between the network subset and the hosts subset depends on the subnet mask used. Other reserved IPs include the IPs used for local network broadcasting and specific network broadcasting. We have to cover all those reserved ones. It is necessary to handle errors relating to the IP address as an invalid IP address will cause the connection to fail as the target host will not be found.
- Incorrect port number: The port number for the socket cannot be 0. There are many reserved ports in TCP and UDP going from 1 to 1024 and sometimes beyond. It's not recommended to use a port that used for a known application however the program will not stop you from doing so. It is necessary to handle errors relating to the port number as an invalid port number



will cause the packets to be lost. Even worse using a port number that is already being used by another service might lead to the data going to that service instead which might cause grave errors.

- Negative transfer rate: We cannot have negative value to transfer a negative number of messages per second as it doesn't make sense.

We used a couple of Java exceptions to be able to handle such errors, including the following.

- `IOException`: This exception is handled mainly in the `contact()` method because it is thrown by the socket when it receives a bad IP or a bad port. The socket checks the validity of the IP and the port and then throws this exception in case anything goes wrong.
- `SocketException`: This exception is also thrown by the socket and it helps it to cover all the other arguments and more. In case the socket failed at any stage of the connection (or connection-less communication) it will throw this exception with a detailed explanation of what went wrong.
- `Exception`: This is the general exception class and is used to cover any error that was not handled by the two previous exceptions.
- Simple if check: We use a simple conditional if to handle to types of errors. One is to check the number of arguments is correct which means all the arguments are provided. The second is usage is we check that the transfer rate is not negative.

These exceptions allow us to cover all the errors possible from the sockets creation and operations. If we haven't added those exception it would be difficult to debug the problems as they give a good explanation of what is wrong. They also give a way to later create methods inside the catch statements to gracefully bounce back from such errors by asking the user to enter the arguments again for example. The next figure shows some of the arguments that the program accepts.

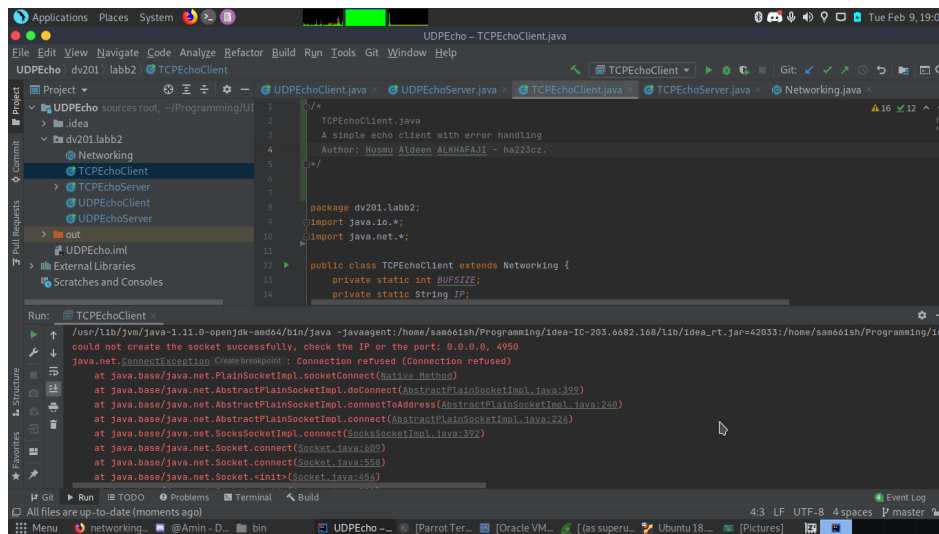


Figure 4. An error message because of a bad IP input (0.0.0.0).

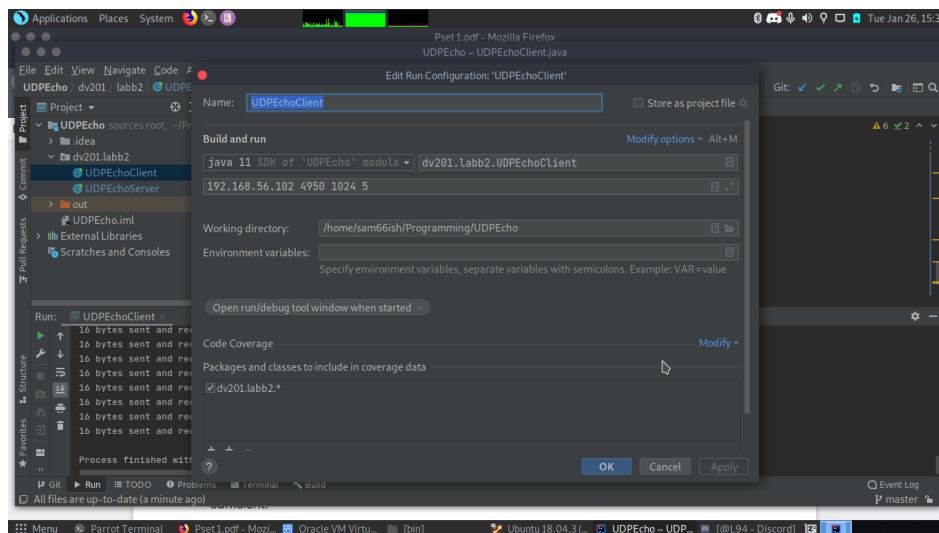


Figure 5. Some of the arguments accepted by the program.

2.2 VG-task 1

In this section, we fixed the implementation so that when high message rates are given. Now, when the rate is too high that it is impossible to send all the messages in one second, the program aborts when the second is reached and prints the number of packets that were sent and how many that could not be sent. The following figure showcases this new functionality.

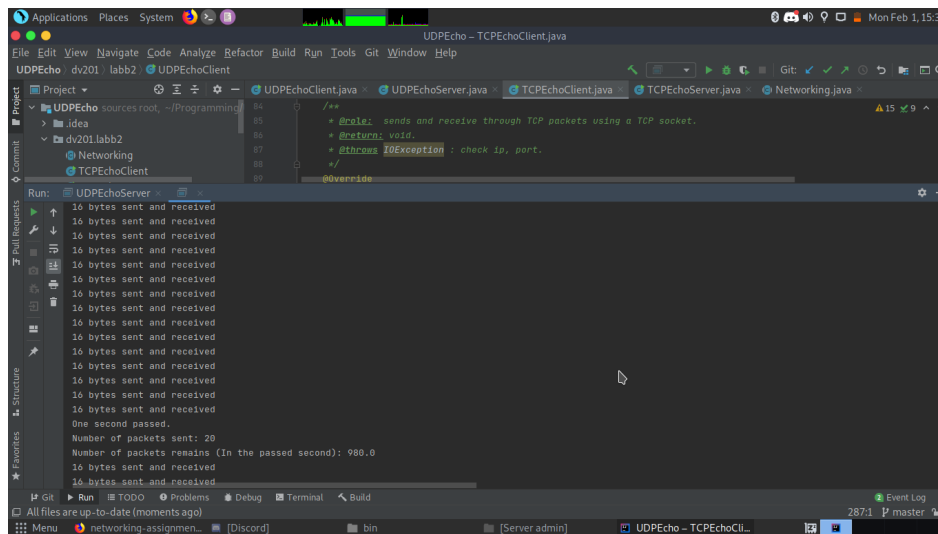


Figure 6. The program showing how many packets sent and how many are left.

We can see that the number of packets sent in one second slightly vary from one second to another. This could be the cause of many things. The packets might be getting lost in the way or the delay might be coming from the current machine CPU because of other programs taking priority. In any case, while it is impossible to send a very high number of packets in one second, the variation between each second is rather small which means there's a maximum amount of messages that we cannot surpass.

2.3 VG-task 2

In this section, we have created an abstract class called `networking`. This class is implemented by the UDP program for both the client and the server.

We have decided to create a shared abstract method called `contact()`. This method handles the creation of the packet, sending the packet and receiving a packet as a response. The exact implementation will be different depending on which class implements the method with which protocol.

Having this one method that handles everything makes it easier to use the program and is the best way to use an abstract interface for the program. The following figure shows the abstract class created for this program.

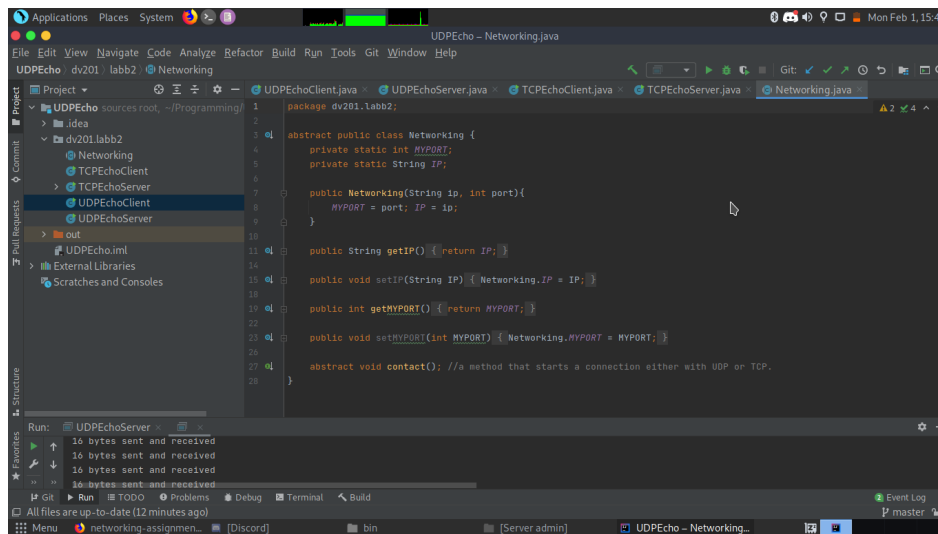


Figure 7. The abstract class used by the program.

As seen by the figure above, the class has the **abstract method contact()** which encapsulates the creation and the operations of a TCP or a UDP socket. This method will be implemented by the TCP and the UDP program and the implementation of the method will differ according to their needs. The motivation to separate this method is to streamline the creation and the operations of the different socket. All you have to do is to call the contact method and it will handle each connection according to the service used.

The following is the different implementations of the contact method for both UDP and TCP clients and servers.

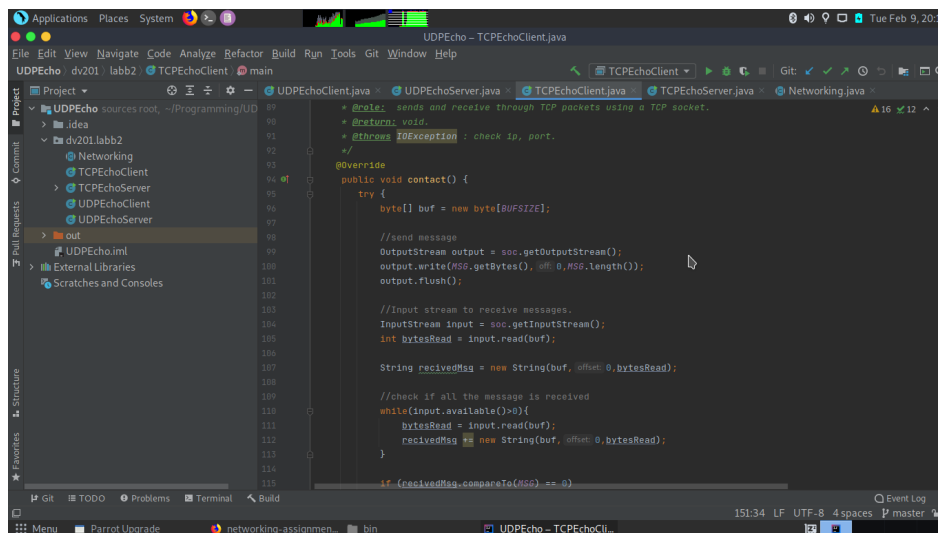


Figure 8. TCP client implementation of the contact() method.



```
public TCPEchoServer(String ip, int port) { super(ip, port); }

/**
 * @role: Creates serverSocket and accepts client sockets and passes them to a handler thread.
 * @return: void.
 * @throws IOException: check ip, port.
 */
@Override
void contact() {
    try {
        //create a new server socket.
        ServerSocket ss = new ServerSocket(MYPORT);

        //start listening from here, creating a new socket using serverSocket everytime a new client contacts.
        while(true) {
            Socket s = ss.accept(); //connect with next client.
            Handler h = new Handler(s); //hand it over to client thread.
            h.run(); //thread starts.
        }
    } catch (IOException e) {
        System.err.println("could not create the socket successfully, check the IP or the port: " + getIP() + ", " + e.getMessage());
        System.exit(2);
    }
}
```

Figure 9. TCP server implementation of the contact() method.

```
public void contact() {
    try {
        byte[] buf = new byte[getBUFSIZE()];

        /* Create socket */
        DatagramSocket socket = new DatagramSocket(bindaddr: null);

        /* Create local endpoint using bind() */
        SocketAddress localBindPoint = new InetSocketAddress(getMYPORT());
        socket.bind(localBindPoint);

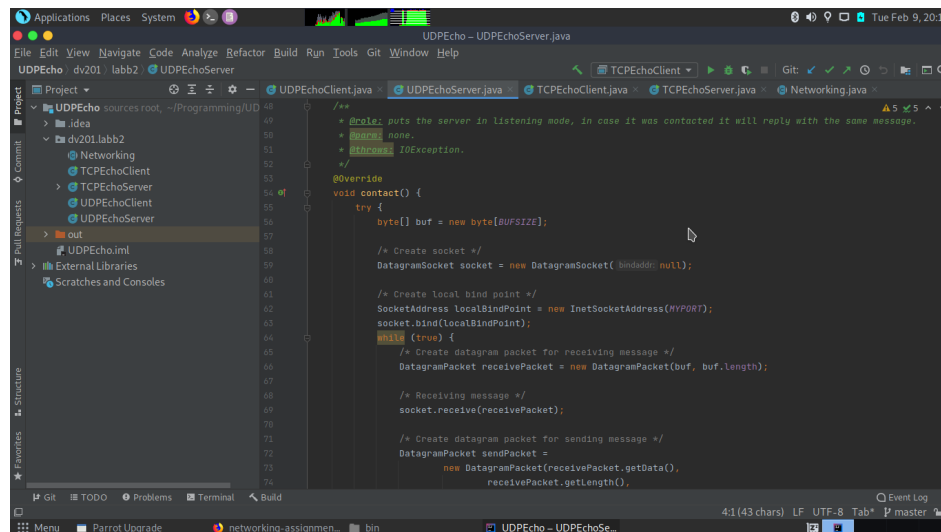
        /* Create remote endpoint */
        SocketAddress remoteBindPoint =
            new InetSocketAddress(getIP(),
                getMYPORT());

        /* Create datagram packet for sending message */
        DatagramPacket sendPacket =
            new DatagramPacket(MSG.getBytes(),
                MSG.length(),
                remoteBindPoint);

        /* Create datagram packet for receiving echoed message */
        DatagramPacket receivePacket = new DatagramPacket(buf, buf.length);

        /* Send and receive messages */
    }
}
```

Figure 10. UDP client implementation of the contact() method.



```

48  /**
49   * @author puts the server in listening mode, in case it was contacted it will reply with the same message.
50   * @author none.
51   * @author IOException.
52   */
53  @Override
54  void contact() {
55      try {
56          byte[] buf = new byte[BUFSIZE];
57
58          /* Create socket */
59          DatagramSocket socket = new DatagramSocket( bindAddr: null);
60
61          /* Create local bind point */
62          SocketAddress localBindPoint = new InetSocketAddress(MYPORT);
63          socket.bind(localBindPoint);
64          while (true) {
65              /* Create datagram packet for receiving message */
66              DatagramPacket receivePacket = new DatagramPacket(buf, buf.length);
67
68              /* Receiving message */
69              socket.receive(receivePacket);
70
71              /* Create datagram packet for sending message */
72              DatagramPacket sendPacket =
73                  new DatagramPacket(receivePacket.getData(),
74                      receivePacket.getLength());

```

Figure 11. UDP server implementation of the contact() method.

2.4 VG-task 3

In this section we have simply cleaned up the code and documented everything appropriately. We also made sure to use encapsulation whenever possible making use of Java's setters and getters.

3 Problem 3

3.1 Discussion

In this problem, we are tasked to re-implement this whole program using the TCP protocol with the addition that the server should create a thread for each connection and use it to handle each client separately.

We will reuse most of the UDP program to this however we will be changing from the data-gram socket used by UDP to the socket class which is used for TCP. We will also be implementing the abstract class that we created earlier and we will have the contact method handle the creation, sending and receiving of the packets.

The following figure shows the contact() method of the TCP server. As we can see we create a handler object which implements the runnable interface. In it's run method, we use the Input-stream and Output-stream as well as the buffered reader to handle the default functionality of the program for each user.

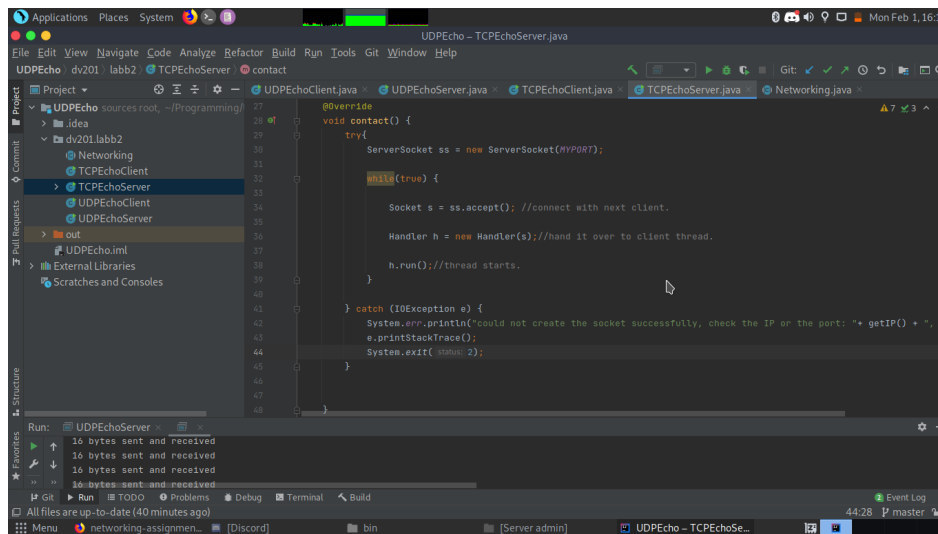


Figure 12. Contact method implementation of the TCP server.

4 Problem 4

4.1 Discussion

First, let us see how the TCP packet look like,

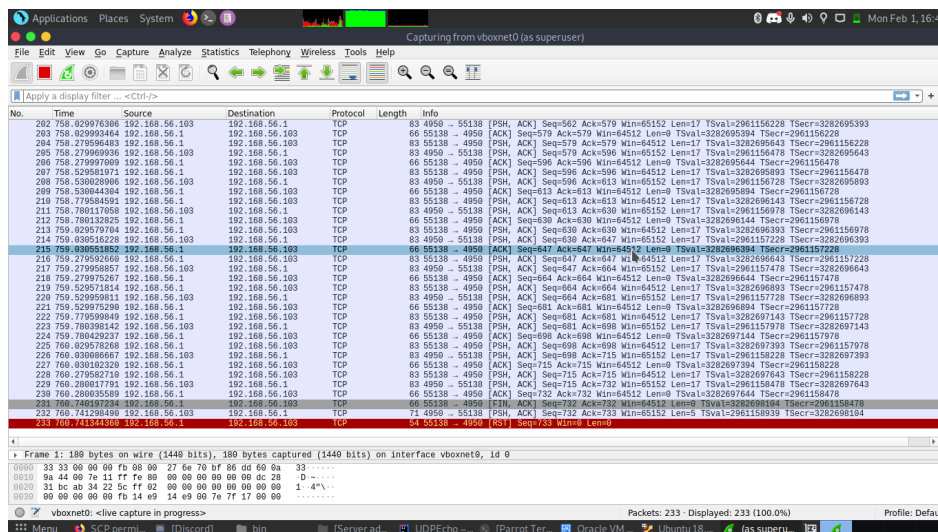


Figure 13. The packets between the client and the server (TCP).

As we can see from the figure, a lot of packets are going around. After carefully examining them, we can see they're TCP packets because of their distinctive header. We can also see that there are some special packets going around. Let's examine them.

- SYN: Used to initiate and establish a connection. It also helps you to synchronize sequence numbers between devices which is what TCP uses as a countermeasure against duplicate packets.



- ACK: Helps to confirm to the other side that it has received the SYN.
- SYN-ACK: SYN message from local device and ACK of the earlier packet.
- FIN: Used to terminate a connection.
- RST: Aborts a connection in response to an error. We can see this packet when we manually terminate the client.
- PSH: push flag has two jobs, the first for the sender it informs TCP that data should be sent immediately the second is to inform the receiving host that the data should be pushed up to the receiving application immediately. Hence PSH can be used to facilitate real-time communication via TCP.

We can also see from the figure that the TCP did a three way handshake before it started the communication. First the client sent a SYN packet to the server to inform it that it wants to start the connection, then the server sent back a set of SYN-ACK the SYN bit is to specify which segment number the client should start from and the ACK tells the client that the server received its request. The client then sends an ACK to the server letting it know that it received its instructions.

Now let's see what the UDP program is doing,

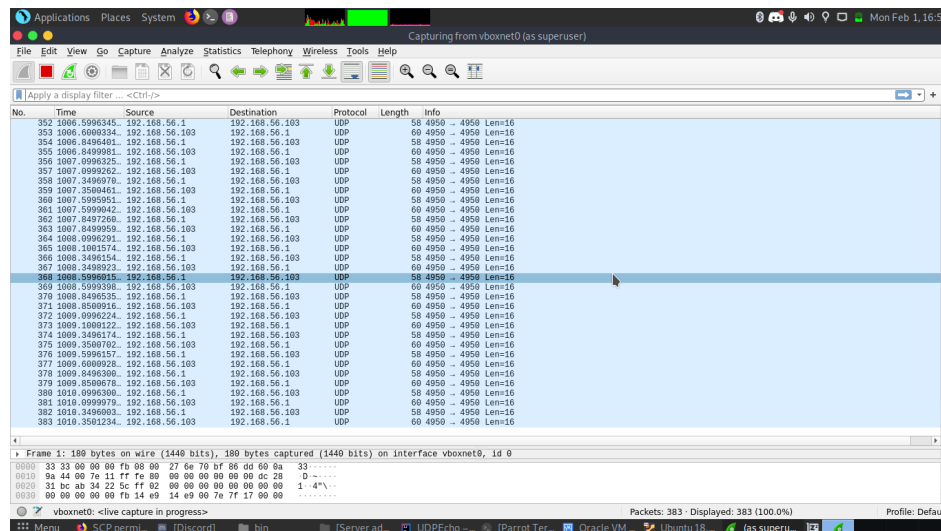


Figure 14. The packets between the client and the server (UDP).

As we can see from the figure, there isn't any handshaking going on. As we know, UDP protocol is connection-less and follows the "best-effort" approach just like the IP suite. Therefore, there aren't any guarantees for the packet sent. The packet could be lost, duplicated, received out of order or worse. This means we don't see any SYN packet which is used for sequencing. We also don't see any ACK packet as there's nothing to be acknowledged. The whole communication is considered one way as UDP is not a full duplex-ed connection protocol as it is connection-less. Another difference between UDP and TCP is in their header,



if we pick one packet and look closely, we can see that the UDP header is much smaller and takes way less space than the TCP header. However this comes with the price of UDP being unreliable as discussed earlier.



5 Problem 5

5.1 Discussion

In this problem, we will be varying the buffer size for both of the TCP and UDP protocols. We will observe the effects of this in wireshark and then discuss the differences between TCP and UDP. We will be using a buffer size of 1 and then a 100 to see the difference behaviors at the two extremes.

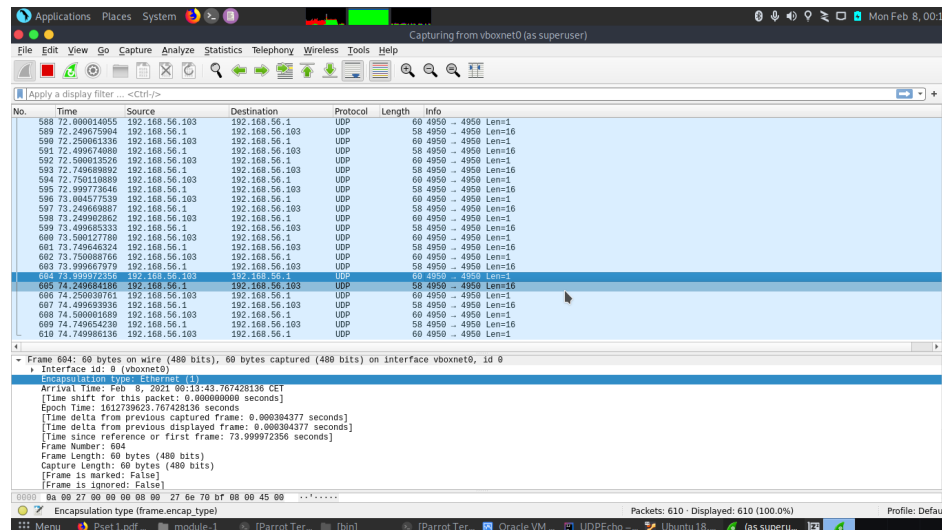


Figure 15. Buffer size 1 (UDP)

Looking at the previous figure, we can see that it's what wireshark captures when we launch the UDP program using a buffer size of 1. We can see the datagram size is as small as 1. Which is because of the buffer sized we specified. We also notice the best-effort behavior that the UDP exhibits as we see no 3-way handshake nor any control messages going back and forth. Another thing we notice is that sometimes the program reports the messages are not equal, this is because the UDP packets might be delivered out of order which causes the program to compare two things that are not the same. It is also because the client doesn't bother to assemble data in case it came in segments as UDP is supposed to send everything together in one message. This is also something particular to the UDP protocol. We can also observe that the UDP packets are not going as a stream but as messages that are being sent individually by the protocol. We can see the UDP header being small which causes a smaller overhead when sending data.

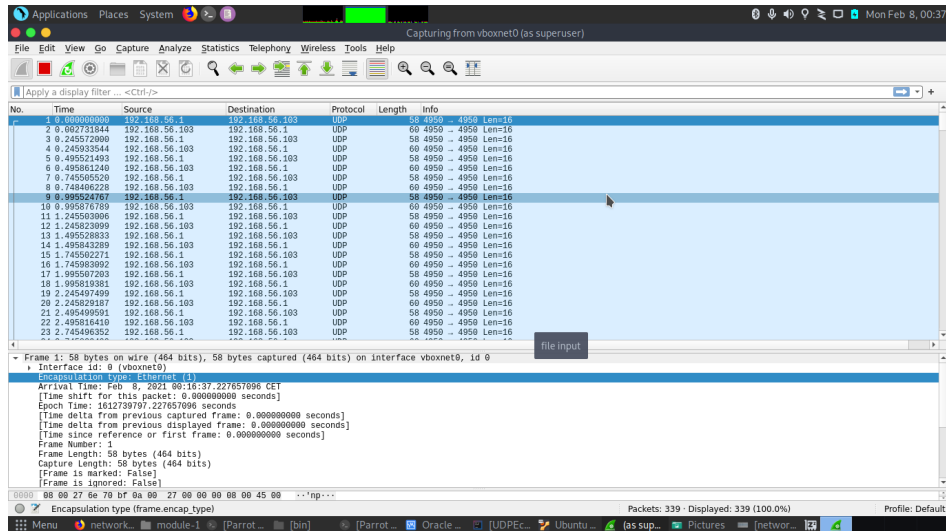


Figure 16. Buffer size 100 (UDP).

In the previous figure, we see the same UDP protocol exhibiting its best-effort behaviors again. This time however, we see that the size of data is bigger. Since the buffer size is now bigger than the datagram of UDP, any data past the datagram size will not be transported. It will be lost and since the UDP protocol doesn't do any error handling or retransmission, the data will be lost forever.

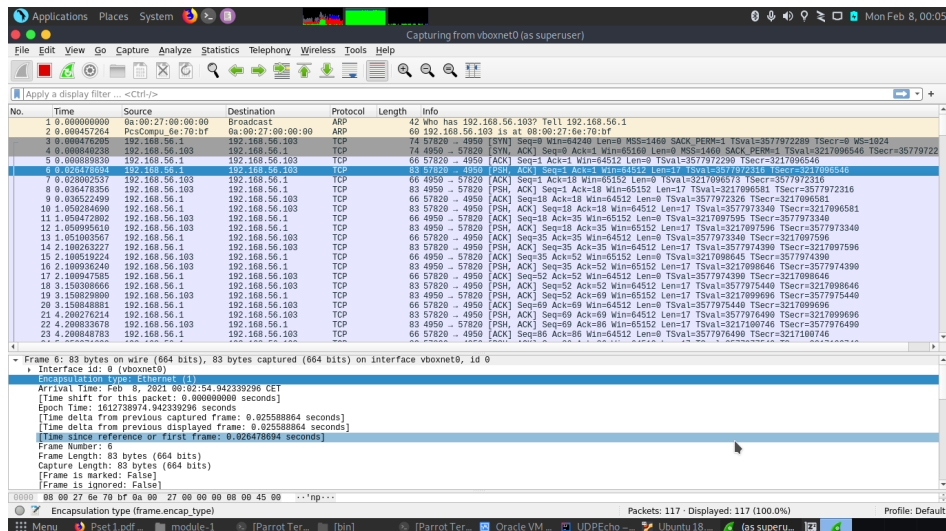


Figure 17. Buffer size 1 (TCP).

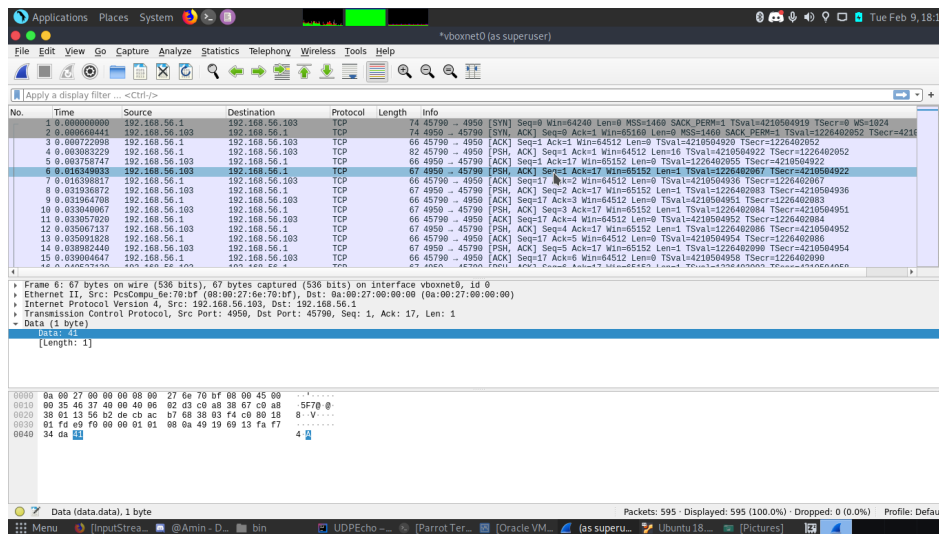


Figure 18. Buffer size 1 (TCP).

This time around we have the transmission between the client and server of the program that uses the TCP transport protocol. We can notice typical behavior of this transport layer such as the 3-way handshake that happened in the beginning as the host contacted the machine, the machine replied with an ACK and the host replied again that it received its reply. This handshake allows this layer to start a connection reliably. A second thing we can notice is that the TCP protocol is connection oriented protocol so we see a lot of control messages going back and forth to handle the communication in the best way possible. As this is a TCP protocol, we can see that the messages are being sent as a stream of data. The TCP packets have a fixed size as we can see. The size of the buffer affects the packet by having the TCP protocol needing less packets to deliver the message. From the second figure above we can see that the payload for the TCP packet is 1. Therefore it only transfers one character at a time as shown. When this is the case for the client, the server receives each character and echos it back to the client, the client receives each character and reassembles it back to the full message before comparing and verifying everything is correct. When the server has a buffer size of 1, the client gives it a full message and the server sends the message one character at a time. The client reassembles the message as usual. The client is always supposed to reassemble the message even if the buffer size for the server or the client itself is not enough for the whole message. This is the reason why we chose a buffer size of 1.

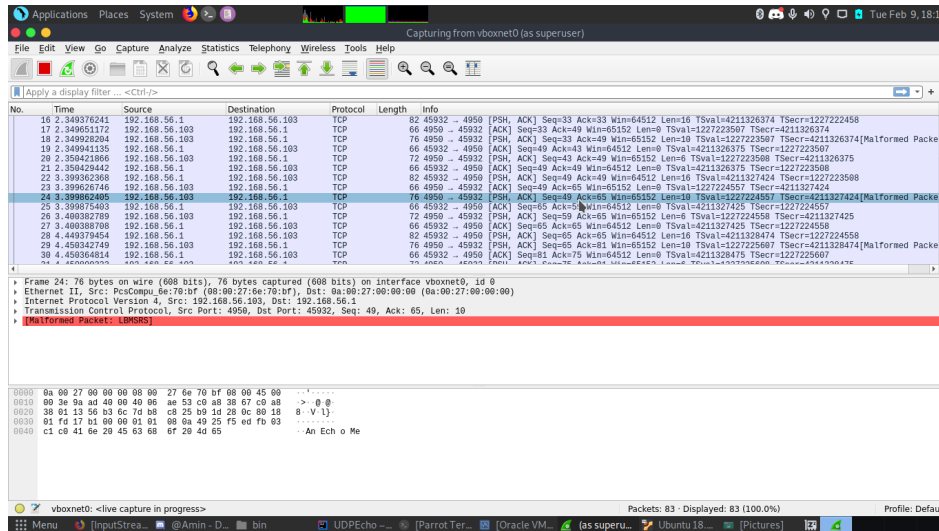


Figure 19. A server with buffersize 10 and a client with buffersize 1. (1)

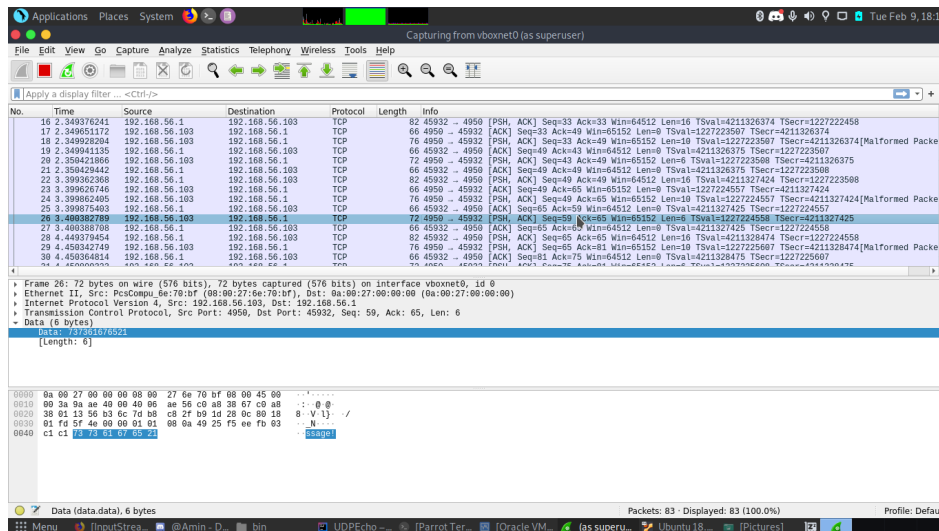


Figure 20. A server with buffersize 10 and a client with buffersize 1. (2)

As we can see from the two previous figures, the server sends 10 bytes of the message first, then sends the remaining 6. The client handles reassembling them and comparing them. This means the smaller the buffer size the more TCP packets are needed to send the message.

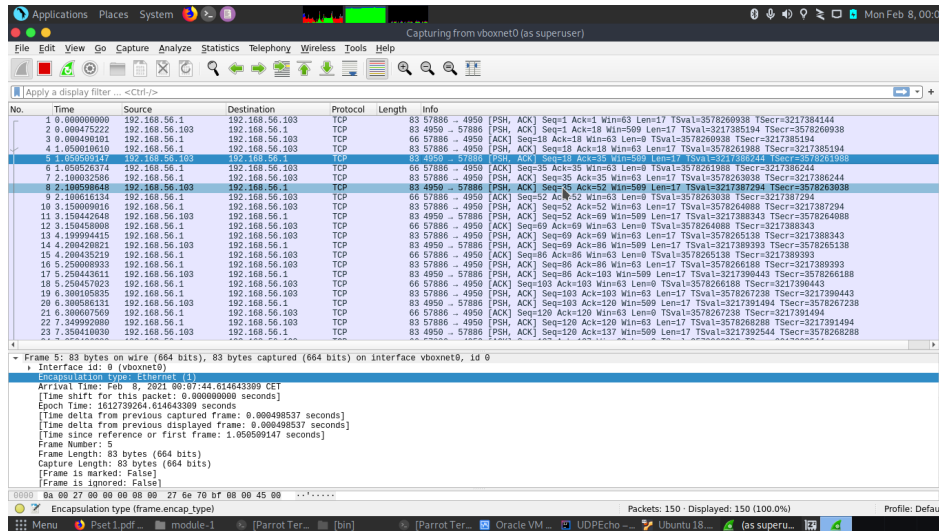


Figure 21. Buffer size 100 (TCP).

From the previous figure, we can see that having a bigger buffer size and a message that is bigger than the buffer size meant that the TCP protocol needed more TCP packets to transmit the data back and forth, as the data is being broken down into segments. We can see that the program never reports that the messages are not the same, this was also the case when the buffer size was 1 for TCP. This is because TCP uses sequencing to make sure the packets are ordered and reassembled by the receiver. So bigger buffer size meant that the TCP had to segment the data further while a bigger buffer size for UDP meant that the UDP sent bigger messages individually and if this buffer size is bigger than the maximum size of the datagram, data is lost. If those messages are bigger than the MTU of a network, the routers will handle the fragmentation if IPv4 is used or the host if IPv6 is used.