

User Datagram Protocol Sockets Programming

Exercise 1: Using UDP Sockets and Primitives (Sender) [3pts]

In this exercise, you will gain a basic understanding of the different primitives of UDP Sockets for sending datagrams using the Java.net API.

1. Start the NetBeans IDE and create a new project. Set the name of the project as "UDP Test", set the project location to your desired path and uncheck the option to create the Main class then click Finish.
2. Create a new Java main class under the project. On the left pane, right click the project name, click on New ▢ Other... Under the Java category, select Java main class then click Next. Name the class as 'SenderTest' and click Finish.
3. At the start of the file, add the following code to import the Scanner class to allow console inputs, and the Java.net package which includes network programming libraries:

```
import java.util.Scanner;
import java.net.*;
```

4. Inside the main function, add the following code:

```
System.out.println("starting Sender program");
Scanner scanner = new Scanner(System.in);
DatagramSocket UDPSock;
int portnum = 0;

try
{
    UDPSock = new DatagramSocket(null);
    System.out.print("Socket created. Enter port for binding: ");
    portnum = Integer.parseInt(scanner.next());
    UDPSock.bind(new InetSocketAddress(portnum));
}
catch(Exception e)
{
    System.out.println("Error creating socket" + e.toString());
    return;
}
System.out.println("Socket bound to port "+ portnum);
scanner.next();
```

This code creates a new UDP socket and binds it to a port specified by a user input. When creating and binding sockets, a potential error may occur if the socket creation and binding fails. This will have to be caught and handled using a try-catch construct.

5. Run the code by right clicking on the class name on the left window pane and select 'Run file'
6. When prompted, enter '8000' for the port number. Do not exit the program yet.
7. Run the Windows command prompt by clicking on the Start button and searching for 'cmd'.
8. In the command prompt, enter the command "netstat -anp udp". This command is used to show all currently active UDP ports.

Observe the output. What indicates that the socket was successfully created and bound to port 8000?

What indicates that the socket was successfully created and bound to port 8000 was the socket 0.0.0.0 showing it was bound to port 8000 and as an active connection.

Lab – User Datagram Protocol Sockets Programming

9. Stop the program by entering any value.
10. We will now attempt to send data through the UDP socket. To do so, you will need to prepare a UDP Datagram with the data to be sent then transmit it using the send primitive. Remove the last “scanner.next();” code then add the following lines to the main function after your previous code:

```
DatagramPacket datagram;  
int rcvport;  
String mesg;  
  
System.out.print("Enter recipient port number: ");  
rcvport = scanner.nextInt();  
  
System.out.print("Type message for sending: ");  
mesg = scanner.next();  
  
try  
{  
    datagram= new DatagramPacket(mesg.getBytes(), mesg.length(),  
        InetAddress.getByName("192.168.1.1"), rcvport);  
  
    UDPSock.send(datagram);  
}  
catch (Exception e)  
{  
    System.out.println("Failed to send: " + e.toString());  
}
```

Creating the datagram requires a byte array containing the data payload, length of the array, the recipient IP address and the destination port UDP port number.

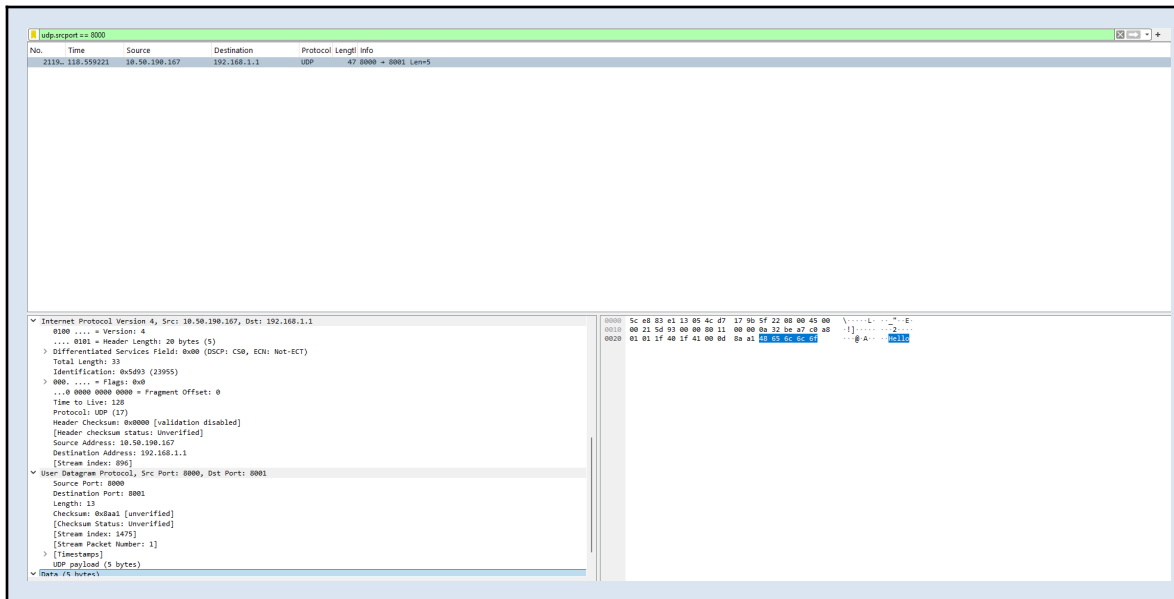
In the code above, the recipient is set to 192.168.1.1.

Potential errors caused by attempting to send through a non-existent or unbound socket need to be caught using a try-catch construct.

11. Run Wireshark to view the UDP datagram being sent. Set the capture interface to your Ethernet connection, then click start.
12. On the capturing screen, add the following filter to view only UDP packets sent from port 8000.
udp.srcport == 8000
then click Apply.
13. Run your program again. Set your bind port to 8000 and the recipient port to 8001. Type in the message “Hello”.
14. Check your Wireshark output. A copy of the UDP datagram you sent should appear on the capture pane. Examine its content by clicking on the packet then viewing its details on the lower pane.

What are the source and destination IP addresses and ports of the packet? (Attach screenshot)

Lab – User Datagram Protocol Sockets Programming



The address 192.168.1.1 is a non-existent host in the network. Despite this, your socket allows you to send a UDP datagram to this destination. Why is this so?

The socket allows this action due to the nature of the User Datagram Protocol (UDP) and how it operates.

15. At this point, the OS automatically closes the port when the program is halted. However, it is always good practice to properly close ports when no longer needed. Add the following inside the main function after your previous code:

```
UDPSock.close();
```

16. Modify your code such that the recipient of the send packets is your own computer then test your program using Wireshark to verify that it works as expected

Exercise 2: Using UDP Sockets and Primitives (Receiver) [2pts]

In this exercise, you will gain a basic understanding of the different primitives of UDP Sockets for receiving datagrams using the Java.net API.

1. Create a new main class under your project and name it ReceiverTest.
2. In the main function, add the necessary code to create a socket and bind it to a port number provided by user input.
3. To receive UDP datagrams, a Datagram object with a byte array buffer must be created. This will be used to hold any packets received by the socket.

Add the following lines after your previous code in the ReceiverTest file:

```
byte[] packetbuffer = new byte[100];
DatagramPacket datagram = new DatagramPacket(packetbuffer, 100);

String msg;

try
{
    UDPSock.receive(datagram);
    msg = new String(datagram.getData());
}
```

Lab – User Datagram Protocol Sockets Programming

```
        System.out.println(msg);

    }
    catch (Exception e)
    {
        System.out.println(e.toString());
    }

    UDPSock.close();
```

The code above prepares a receive buffer of 100 bytes then accesses the socket once to receive a packet. Potential errors for trying to receive from an unbound socket needs to be handled using a try-catch construct. As with the sender code, the socket must be closed when no longer needed.

4. Test your programs:

- i. Run the sender by binding its socket to port 8000 and setting the receiver port to 8001. Do not send any message yet.
- ii. Run the recipient by binding its socket to port 8001. Check netstat to verify that both UDP 8000 and 8001 ports are open.
- iii. On the sender program, send the message "Hello". This should be received by the receiver program.

Did it work?

Yes, the program did work.

Exercise 3: Programming Blocking and Non-blocking Sockets [8pts]

In this exercise, you will gain an understanding of the differences in using blocking and non-blocking sockets in a program.

1. In Java, whether a socket behaves in blocking or non-blocking mode is controlled by setting its socket timeout property. By default, a socket will be running in blocking mode.

In the receiver program, modify the section of the code that is used to receive datagrams from the socket. Add the following lines in **blue** to the parts of your code:

```
byte[] packetbuffer = new byte[100];
DatagramPacket datagram = new DatagramPacket(packetbuffer, 100);

String msg;

int timeout = UDPSock.getSoTimeout();
System.out.println("Socket will receive with timeout = " + timeout + "msec");

try
{
    UDPSock.receive(datagram);
    msg = new String(datagram.getData());
    System.out.println(msg);
}
catch (Exception e)
{
    System.out.println(e.toString());
}

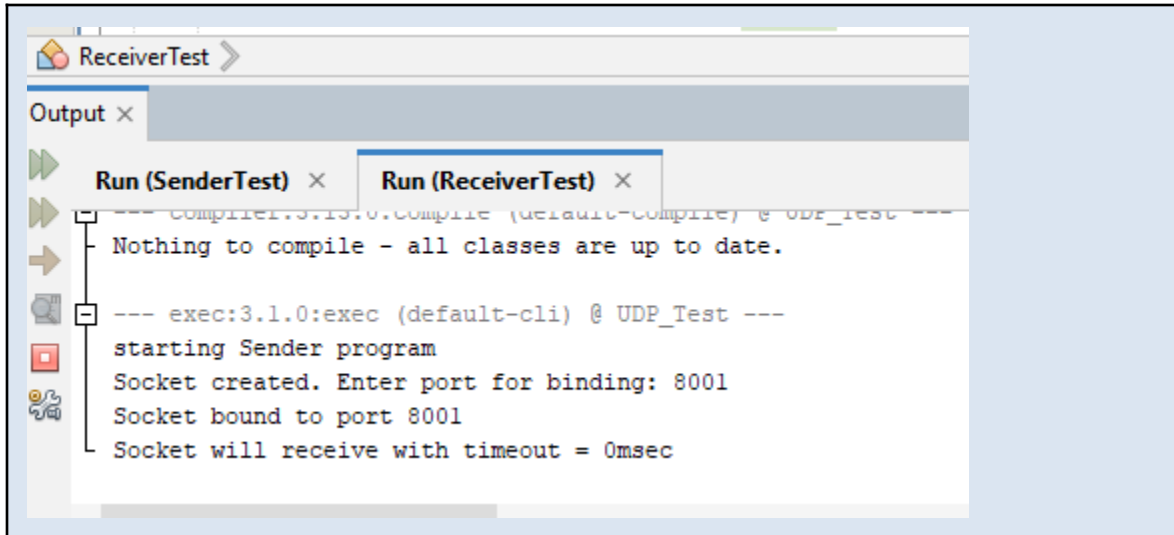
System.out.println("Done receiving. Closing socket now. ");
```

Lab – User Datagram Protocol Sockets Programming

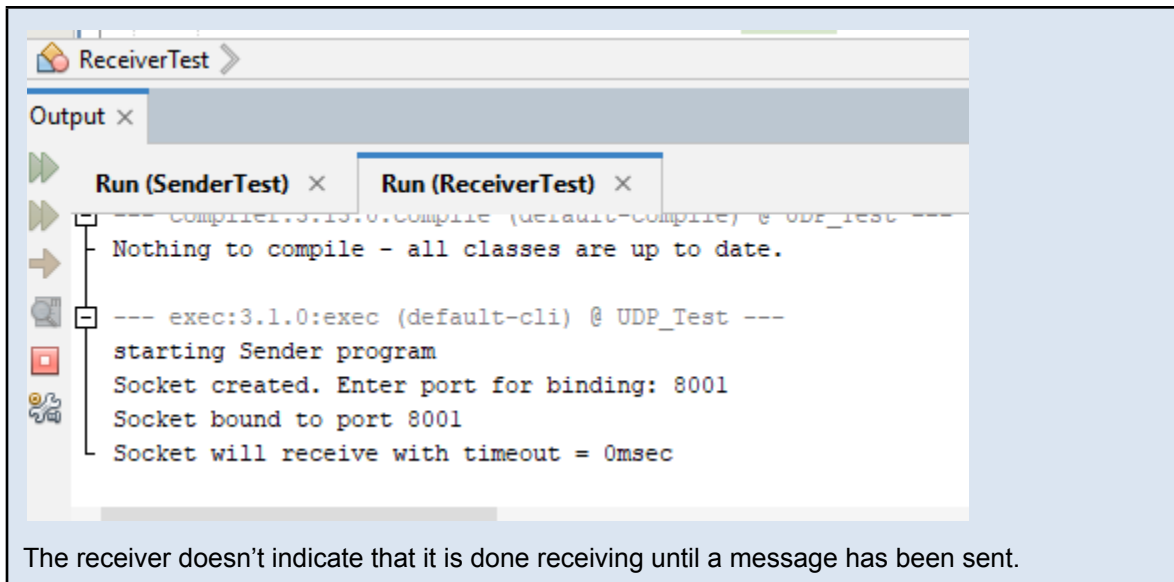
```
UDPSock.close();
```

2. Run the sender (UDP 8000) and receiver program (8001). Provide the necessary inputs to bind ports and specify the receiving port, but do not send any message yet. (Attach screenshots)

Based on the output of the receiver, what is the timeout setting of its socket?



Wait 10 seconds then check the receiver program. Does it indicate that it is done receiving?



3. Using the sender program, send any message to the receiver then check the output of receiver program. Does the receiver indicate that it is done receiving this time?

Yes, the receiver does indicate that it has received the message.

Based on your observation, how does a blocking socket behave when the receive primitive is invoked

The blocking socket's receive operation (i.e. `recv`) will pause the program until data is available, which can be beneficial for certain applications but may lead to inefficiencies in scenarios requiring high responsiveness or concurrent processing.

4. The receiver code will now be modified so that the socket will operate in non-blocking mode.

Lab – User Datagram Protocol Sockets Programming

Add the following lines in **blue** to the parts of your code:

```
byte[] packetbuffer = new byte[100];
DatagramPacket datagram = new DatagramPacket(packetbuffer, 100);

String msg;

UDPSock.setSoTimeout(5000);

int timeout = UDPSock.getSoTimeout();
System.out.println("Socket will receive with timeout = " + timeout + "msec");

try
{
    UDPSock.receive(datagram);
    msg = new String(datagram.getData());
    System.out.println(msg);
}
catch(SocketTimeoutException e)
{
    System.out.println("Receive timeout");
}
catch(Exception e)
{
    System.out.println(e.toString());
}

System.out.println("Done receiving. Closing socket now. ");

UDPSock.close();
```

The additional code above sets the UDP socket timeout to 5 seconds and adds the necessary procedure to handle a timeout.

5. Run the sender and receiver again using the same port settings as in step 2 of this exercise. Do not send a message yet.

Wait 10 seconds then check the receiver program. Does it indicate that it is done receiving?

Yes, it does indicate that it is done receiving without a message being sent.

6. The code will now be modified to add the necessary procedures so that a receive can be attempted up again up to 10 times if a non-blocking socket times out.

In a real application however, a non-blocking socket should instead be checked by implementing a timer that calls the receive primitive at a regular interval. DO NOT use a loop to redo the receive as this would just make the socket behave as if it were in blocking mode.

```
DatagramPacket datagram = new DatagramPacket(packetbuffer, 100);

String msg;

UDPSock.setSoTimeout(5000);
int timeout = UDPSock.getSoTimeout();
System.out.println("Socket will receive with timeout = " + timeout + "msec");

int i;
for (i = 0; i < 10; i++)
{
    try
```

Lab – User Datagram Protocol Sockets Programming

```
{
    UDPSock.receive(datagram);
    mesg = new String(datagram.getData());
    System.out.println(mesg);
    i = 10;

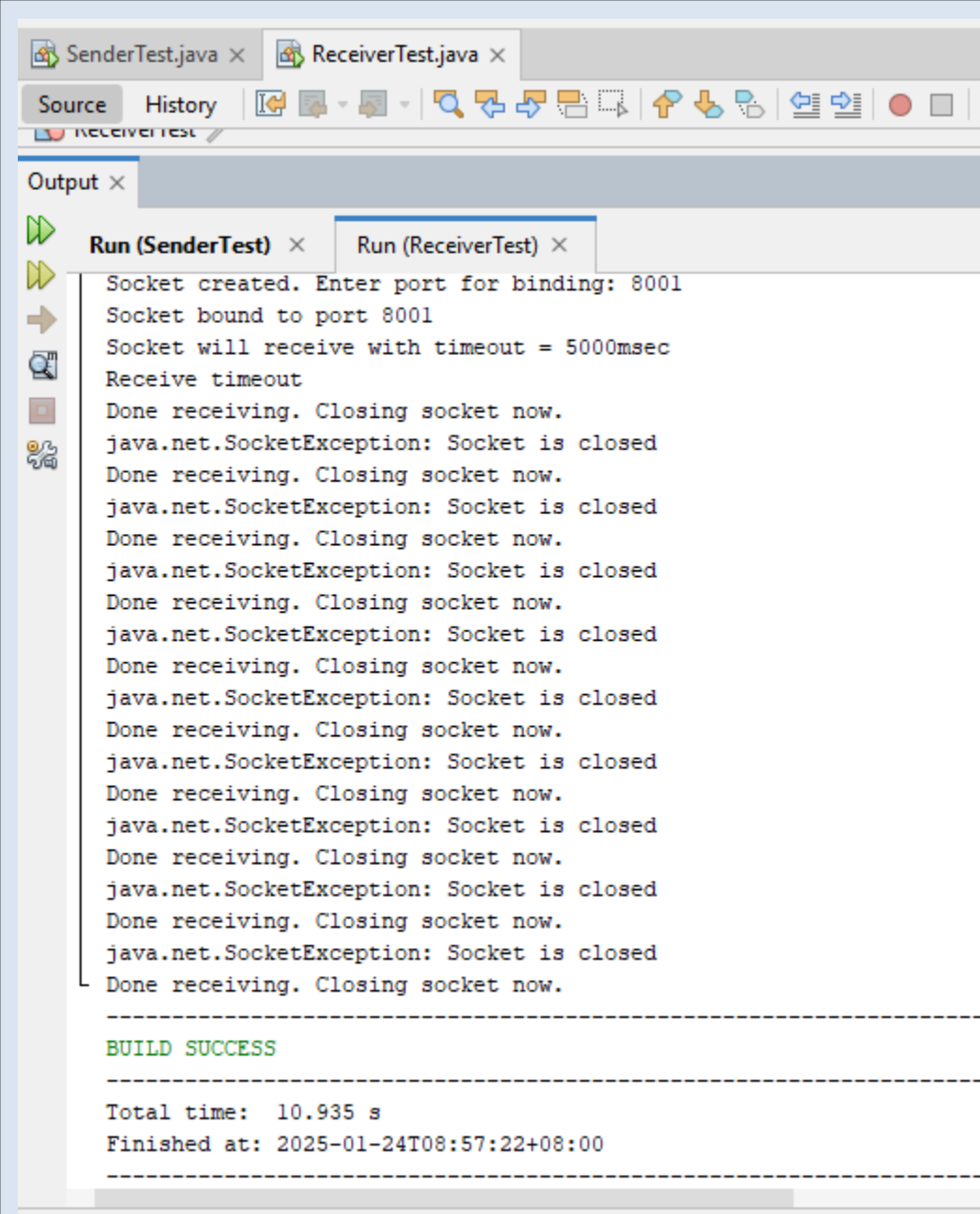
}
catch(SocketTimeoutException e)
{
    System.out.println("Receive timeout");
}
catch(Exception e)
{
    System.out.println(e.toString());
}
}

System.out.println("Done receiving. Closing socket now. ");

UDPSock.close();
```

7. Redo the test for send and receive. Send a message to the receiver between 20-25 seconds after you execute both programs and input the necessary parameters for port numbers. (Attach screenshots)

At which read attempt did the receiver finally get the message?



```
SenderTest.java x ReceiverTest.java x
Source History
ReceiverTest
Output x
Run (SenderTest) x Run (ReceiverTest) x
Socket created. Enter port for binding: 8001
Socket bound to port 8001
Socket will receive with timeout = 5000msec
Receive timeout
Done receiving. Closing socket now.
java.net.SocketException: Socket is closed
Done receiving. Closing socket now.
java.net.SocketException: Socket is closed
Done receiving. Closing socket now.
java.net.SocketException: Socket is closed
Done receiving. Closing socket now.
java.net.SocketException: Socket is closed
Done receiving. Closing socket now.
java.net.SocketException: Socket is closed
Done receiving. Closing socket now.
java.net.SocketException: Socket is closed
Done receiving. Closing socket now.
java.net.SocketException: Socket is closed
Done receiving. Closing socket now.
java.net.SocketException: Socket is closed
Done receiving. Closing socket now.
java.net.SocketException: Socket is closed
Done receiving. Closing socket now.
-----
BUILD SUCCESS
-----
Total time: 10.935 s
Finished at: 2025-01-24T08:57:22+08:00
-----
```

The receiver does not receive the message due to receiver timing out at 10.935 seconds.

Based on your observation, how does a non-blocking socket behave when the receive primitive is invoked

When a non-blocking socket's receive primitive is invoked, it allows the program to continue executing without waiting for data, returning immediately with either the received data or an indication that no data is currently available. This behavior is essential for building responsive and efficient network applications.

Why would a programmer resort to using a blocking socket? How about a nonblocking socket?

The choice between resorting to a blocking or a non-blocking sockets depends on the specific requirements of the application, including the need for simplicity versus responsiveness and concurrency.

Challenge: Modify your code for both programs to achieve the following: [7 pts] (Attach screenshots)

- The sender is able to repeatedly get user input then send it to the receiver. User messages is assumed to be 90 characters or less.
- When the user types in 'exit', the sender will send it as a message to the receiver , close its socket, then stop executing.
- The receiver is able to continuously receive messages from the sender and print it out. It will close its socket and stop executing only when it receives an "exit" message from the sender.

Hint:

- The receive buffer will be overwritten only up the length of the message received each time a packet arrives. Hence if you reuse the buffer to hold each message received, parts of previous messages may still be in the buffer if the current received message is shorter than the ones that came before it. You will need to devise a way by which the receiver can determine the actual length of the message received and print only that part of the buffer.