Client Server Application Programming

Week 4: User Datagram Protocol

By Elubu Joseph - MSc.IS

Lecturer

Department of Information Technology

Kumi University

Email: josebulinda@gmail.com

jose@kumiuniversity.ac.ug

Agenda

1. User Datagram Protocol-Overview
2. Datagram Packet Class,
3. DatagramSocket Class,
4. Listening for UDP Packets,
5. Sending UDP packets,
6. Building a UDP Client/Server,

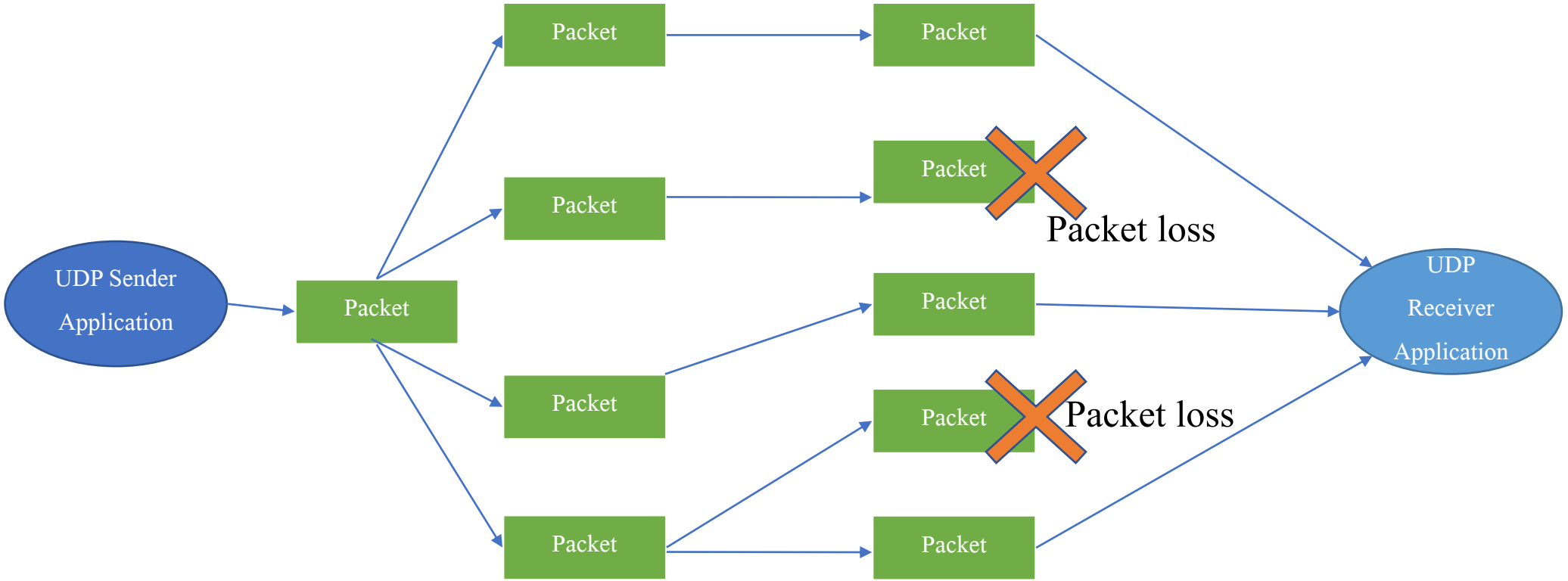
User Datagram Protocol-Overview

User Datagram Protocol (UDP) is a commonly used transport protocol employed by many types of applications.

UDP is a connectionless transport protocol, meaning that it doesn't guarantee either packet delivery or that packets arrive in sequential order. Rather than reading from, and writing to, an ordered sequence of bytes (using I/O streams, as discussed earlier), bytes of data are grouped together in discrete packets, which are sent over the network.

Although control over the ultimate destination of a UDP packet rests with the computer that sends it, how it reaches that destination is an arbitrary process as shown below.

UDP packet transport over a network can be unreliable



UDP packet transport over a network can be unreliable+

The packets may travel along different paths, as selected by the various network routers that distribute traffic flow seemingly imaginatively—depending on factors such as **network congestion**, **priority of routes**, and **cost of transmission**. (For example, for one packet a cheaper network route might be selected, even though it is slower, but another might travel along a superfast pipeline if the cheaper alternative becomes too congested.) This means that a packet can arrive out of sequence, if it encounters a faster route than the previous packet (or if the previous packet encounters some other form of delay).

No two packets are guaranteed the same route, and **if a particular route is heavily congested**, the packet may be discarded entirely.

Each packet has a time-to-live (TTL) counter, which is updated when the packet is routed along to the next point in the network. **When the timer expires**, it will be discarded, and the recipient of the packet will not be notified.

If a packet does arrive, however, it will always arrive intact. **Packets that are corrupt or only partially delivered** are discarded.

Merits of Using UDP

Given the potential for loss of data packets, it may seem odd that anyone would even consider using such an unreliable, seemingly anarchical system. In fact, there are many advantages of using UDP that may not be apparent at first glance.

1. UDP communication can be **more efficient** than guaranteed-delivery data streams. If the amount of data is small and the data is sent frequently (such as in the case of a counter whose previous value is irrelevant), it may make sense to avoid the overhead of guaranteed delivery.

2. Unlike TCP streams, which establish a connection, **UDP causes fewer overheads** amount of data being sent is small and the data is sent infrequently, the overhead of establishing a connection might not be worth it. UDP may be preferable in this case, particularly if data is being sent from a large number of machines to one central one, in which case the sum total of all these connections might cause significant overload.

Merits of Using UDP+

Given the potential for loss of data packets, it may seem odd that anyone would even consider using such an unreliable, seemingly anarchical system. In fact, there are many advantages to using UDP that may not be apparent at first glance.

3. Real-time applications that demand up-to-the-second or better performance may be candidates for UDP, as there are fewer delays due to the error checking and flow control of TCP.
4. UDP sockets can receive data from more than one host machine. If several machines must be communicated with, then UDP may be more convenient than other mechanisms such as TCP . Therefore Some network protocols specify UDP as the transport mechanism, requiring its use.

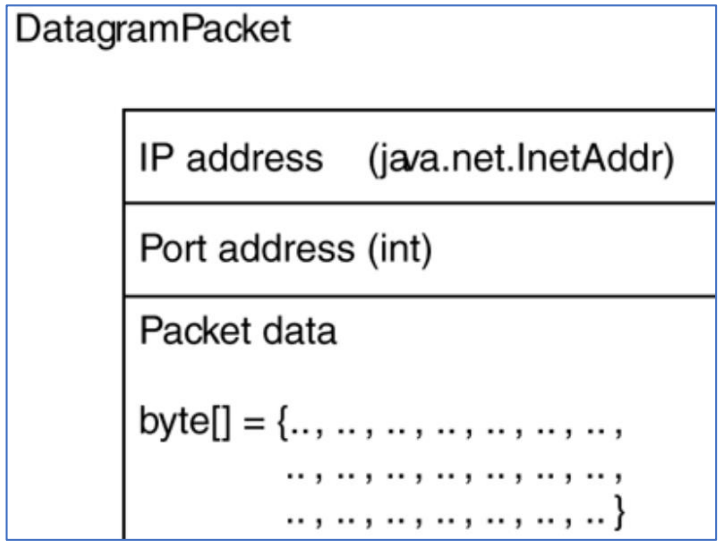
User Datagram Protocol is supported by Java in the form of two classes:

java.net.DatagramPacket

java.net.DatagramSocket

DatagramPacket Class

represents a data packet intended for transmission using the User Datagram Protocol. Packets are containers for a small sequence of bytes, and include addressing information such as an **IP address** and **a port**.



The meaning of the data stored in a DatagramPacket is determined by its context. When a DatagramPacket has been read from a UDP socket, the IP address of the packet represents the address of the sender (likewise with the port number).

However, when a DatagramPacket is used to send a UDP packet, the IP address stored in DatagramPacket represents the address of the recipient (likewise with the port number).

This reversal of meaning is important to remember—one wouldn't want to send a packet back to oneself!

Creating a DatagramPacket

There are two reasons to create a new DatagramPacket:

1. To send data to a remote machine using UDP
2. To receive data sent by a remote machine using UDP

Constructors

The choice of which DatagramPacket constructor to use is determined by its intended purpose. Either constructor requires the specification of a byte array, which will be used to store the UDP packet contents, and the length of the data packet or not.

1. To create a DatagramPacket for receiving incoming UDP packets, the following constructor should be used: DatagramPacket(byte[] buffer, int length). For example:

```
DatagramPacket packet = new DatagramPacket(new byte[256], 256);
```

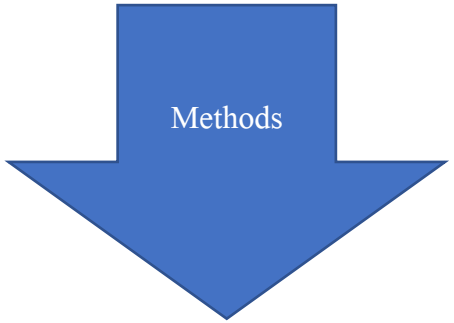
2. To send a DatagramPacket to a remote machine, it is preferable to use the following constructor: DatagramPacket(byte[] buffer, int length, InetAddress dest_addr, int dest_port). For example:

```
InetAddress addr = InetAddress.getByName("192.168.0.1"); DatagramPacket packet = new DatagramPacket ( new byte[128], 128, addr, 2000)
```

DatagramPacket methods

The DatagramPacket class provides some important methods that allow the remote address, remote port, data (as a byte array), and length of the packet to be retrieved. As of JDK1.1, there are also methods to modify these, via a corresponding set method. This means that a received **packet can be reused**.

For example, a packet's contents can be replaced and then sent back to the sender. This saves having to reset addressing information—the address and port of the packet are already set to those of the sender.



DatagramPacket class methods+

1. InetAddress getAddress()— returns the IP address from which a DatagramPacket was sent, or (if the packet is going to be sent to a remote machine), the destination IP address.
2. byte[] getData()— returns the contents of the DatagramPacket, represented as an array of bytes.
3. int getLength()— returns the length of the data stored in a DatagramPacket. This can be less than the actual size of the data buffer.
4. int getPort()— returns the port number from which a DatagramPacket was sent, or (if the packet is going to be sent to a remote machine), the destination port number.

DatagramPacket methods+

5. void setAddress(InetAddress addr)— assigns a new destination address to a DatagramPacket.
6. void setData(byte[] buffer)— assigns a new data buffer to the DatagramPacket. Remember to make the buffer long enough, to prevent data loss.
7. void setLength(int length)— assigns a new length to the DatagramPacket. Remember that the length must be less than or equal to the maximum size of the data buffer, or an IllegalArgumentException will be thrown.
8. void setPort(int port)— assigns a new destination port to a DatagramPacket.

DatagramSocket Class

provides access to a UDP socket, which allows UDP packets to be sent and received. A DatagramPacket is used to represent a UDP packet, and must be created prior to receiving any packets.

The same DatagramSocket can be used to receive packets as well as to send them. However, read operations are blocking, meaning that the application will continue to wait until a packet arrives. Since UDP packets do not guarantee delivery, this can cause an application to stall if the sender does not resubmit packets.

Creating a DatagramSocket

A DatagramSocket can be used to both send and receive packets. Each DatagramSocket binds to a port on the local machine, which is used for addressing packets. The port number need not match the port number of the remote machine, but if the application is a UDP server, it will usually choose a specific port number.

If the DatagramSocket is intended to be a client, and doesn't need to bind to a specific port number, a blank constructor can be specified.

Constructors(Three)

1. To create a client DatagramSocket, the following constructor is used: DatagramSocket() throws java.net.SocketException.
2. To create a server Datagram Socket, the following constructor is used, which takes as a parameter the port to which the UDP service will be bound: DatagramSocket(int port) throws java.net.SocketException.

Creating a DatagramSocket + Constructors(three)

3. DatagramSocket (int port, InetAddress addr) throws java.net.SocketException. Although rarely used, this constructor was introduced in JDK1.1 to deal with machines know by many ip addresses. If a machine is known by several IP addresses (referred to as multihomed), you can specify the IP address and port to which a UDP service should be bound. It takes as parameters the port to which the UDP service will be bound, as well as the InetAddress of the service. This constructor is

DatagramSocket class Methods

A number of methods are used in the DatagramSocket class to help in the sending and receipt of packets of data, they include the following:-

1. void close()— closes a socket, and unbinds it from the local port.
2. void connect(InetAddress remote_addr int remote_port)— restricts access to the specified remote address and port. The designation is misleading, as UDP doesn't actually create a "connection" between one machine and another. However, if this method is used, it causes exceptions to be thrown if an attempt is made to send packets to, or read packets from, any other host and port than those specified.
3. void disconnect()— disconnects the DatagramSocket and removes any restrictions imposed on it by an earlier connect operation.
4. InetAddress getInetAddress()— returns the remote address to which the socket is connected, or null if no such connection exists.
5. int getPort()— returns the remote port to which the socket is connected, or –1 if no such connection exists.

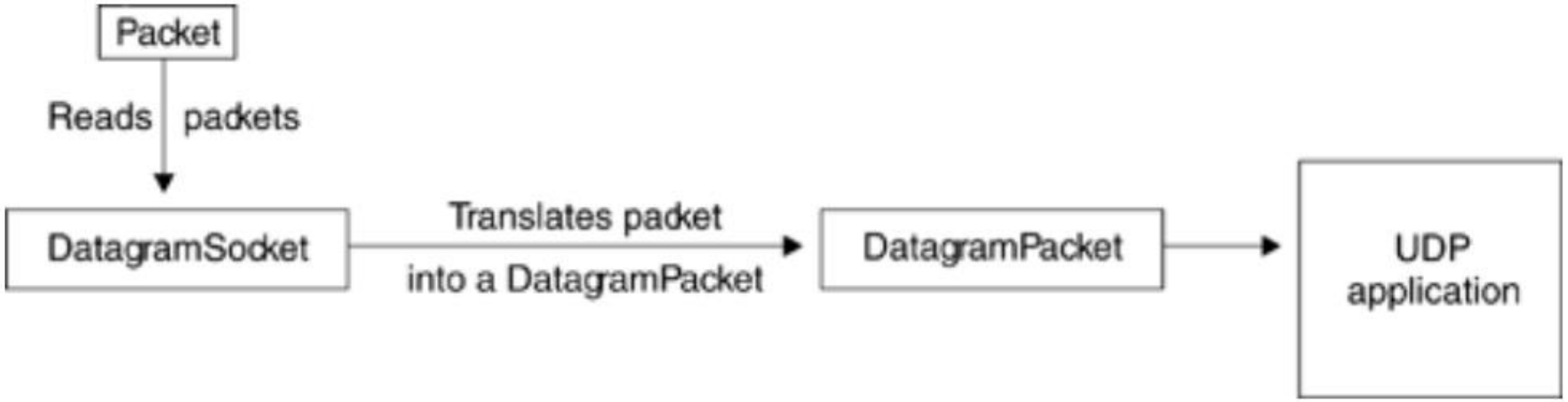
DatagramSocket class Methods

A number of methods are used in the DatagramSocket class to help in the sending and receipt of packets of data, they include the following:-

6. inetAddress getLocalAddress()— returns the local address to which the socketis bound.
7. int getLocalPort()— returns the local port to which the socket is bound.
8. int getReceiveBufferSize() throws java.net.SocketException— returns the maximum buffer size used for incoming UDP packets.
9. int getSendBufferSize() throws java.net.SocketException— returns the maximum buffer size used for outgoing UDP packets.
10. int getSoTimeout() throws java.net.SocketException— returns the value of the timeout socket option. This value is used to determine the number of milliseconds a read operation will block before throwing a java.io.InterruptedIOException. By default, this value will be zero, indicating that blocking I/O will be used.
11. void receive(DatagramPacket packet) throws java.io.IOException— reads a UDP packet and stores the contents in the specified packet. The address and port etc.

Listening for UDP Packets

Before an application can read UDP packets sent to it by remote machines, it must bind a socket to a local UDP port using DatagramSocket, and create a DatagramPacket that will act as a container for the UDP packet's data. Figure below shows the relationship between a UDP packet, the various Java classes used to process it, and the actual application.



When an application wishes to read UDP packets, it calls the DatagramSocket.receive method, which copies a UDP packet into the specified DatagramPacket. The contents of the DatagramPacket are processed, and the process is repeated as needed.

Listening for UDP Packets

The following code snippet illustrates this process:

```
import java.net.DatagramPacket; import java.net.DatagramSocket;
public class UDPServer {
    public static void main(String[] args) throws Exception {
        int port = 5005;
        byte[] buffer = new byte[1024];
        DatagramSocket socket = new DatagramSocket(port);
        DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
```

Listening for UDP Packets

The following code snippet illustrates this process:

```
while (true) {
        socket.receive(packet);
        String message = new String(packet.getData(), 0, packet.getLength());
        System.out.println("Received message: " + message);
    }
}
```

Listening for UDP Packets code explained

In the above code, we create a **DatagramSocket** using **new DatagramSocket(port)** and a **DatagramPacket** using **new DatagramPacket(buffer, buffer.length)**. The **port** variable specifies the UDP port number to listen on, and **buffer** is a byte array to store the incoming data.

We then enter a loop where we continuously receive data from the socket using **socket.receive(packet)**. The received data is stored in the **packet** variable, and we convert the bytes to a string using **new String(packet.getData(), 0, packet.getLength())**. The **getLength()** method returns the length of the received data in bytes.

Note that this code will also block until data is received on the socket, so it is typically run in a separate thread or process. Additionally, in order to receive packets from remote hosts, you may need to configure your firewall to allow incoming UDP traffic on the specified port.

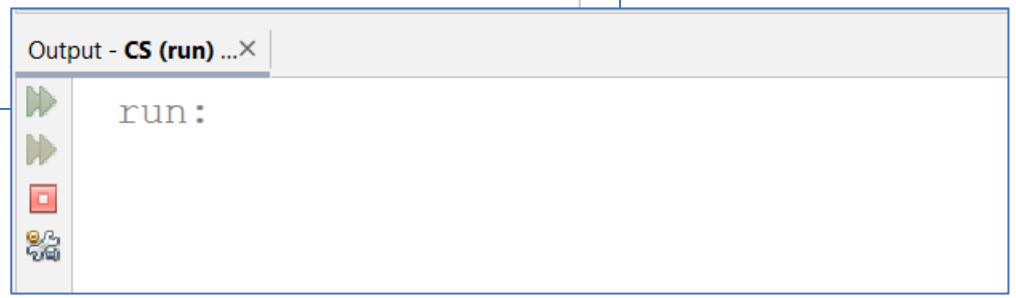
Listening for UDP Packets code in NetBeans-Output

```java
package cs;
import java.net.DatagramPacket;
import java.net.DatagramSocket;

public class UDPServer {
    public static void main(String[] args) throws Exception {
        int port = 5005;
        byte[] buffer = new byte[1024];
        DatagramSocket socket = new DatagramSocket(port);
        DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

        while (true) {
            socket.receive(packet);
            String message = new String(packet.getData(), 0, packet.getLength());
            System.out.println("Received message: " + message);
        }
    }
}
```

No output until the client sends a packet



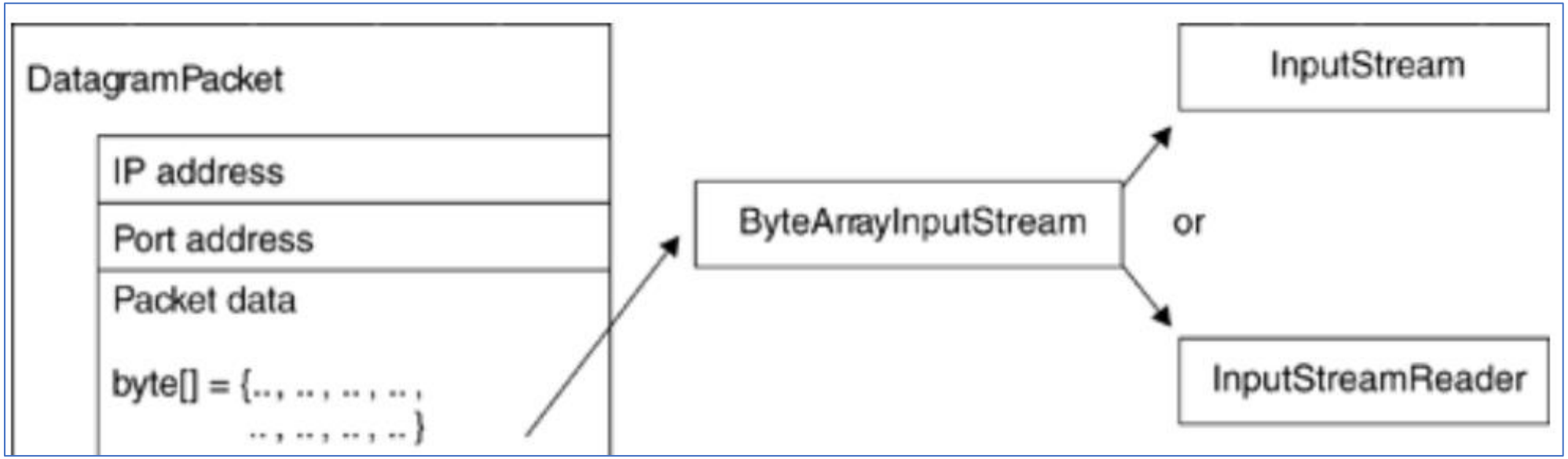
Reading from a UDP packet is simplified by applying input streams.

When processing the packet, the application must work directly with an array of bytes. If, however, your application is better suited to reading text, you can use classes from the Java I/O package to convert between a byte array and another type of stream or reader.

By hooking a ByteArrayInputStream to the contents of a datagram and then to another type of InputStream or an InputStreamReader, you can access the contents of UDP packets relatively easily (see Figure in the next slide).

Reading from a UDP packet is simplified by applying input streams.

Many developers prefer to use Java I/O streams to process data, using a DataInputStream or a BufferedReader to access the contents of byte arrays.

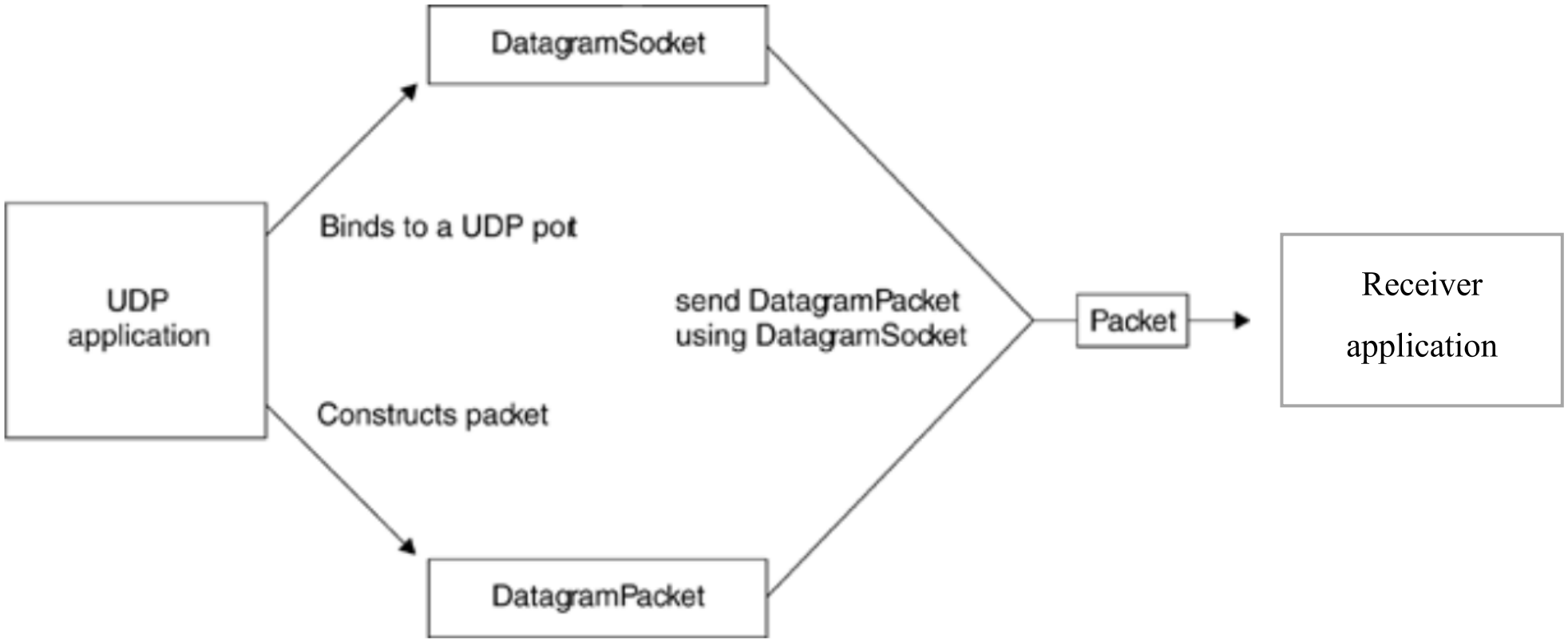


Sending UDP packets

The same interface (DatagramSocket) employed to receive UDP packets is also used to send them. When sending a packet, the application must create a DatagramPacket, set the address and port information, and write the data intended for transmission to its byte array.

If replying to a received packet, the **address and port** information will **already be stored**, and only the data need be overwritten. Once the packet is ready for transmission, the send method of DatagramSocket is invoked, and a UDP packet is sent.

See the figure below


DatagramSocket
UDP application
Binds to a UDP pot
send DatagramPacket using DatagramSocket
Packet
Receiver application
Constructs packet
DatagramPacket

Sending UDP Packets code

The following code snippet illustrates this process:

```
import java.net.DatagramPacket; import java.net.DatagramSocket; import
java.net.InetAddress;
public class UDPClient {
    public static void main(String[] args) throws Exception {
        int port = 5005;
        String message = “Hello, Server!";
        InetAddress address = InetAddress.getByName( "localhost“ );
        byte[] buffer = message.getBytes();
```

Sending UDP Packets+code

The following code snippet illustrates this process:

```
DatagramSocket socket = new DatagramSocket();

        DatagramPacket packet = new DatagramPacket(buffer, buffer.length, address, port);

        socket.send(packet);

        System.out.println("Sent message: " + message);

        socket.close();

    }

}
```

Sending UDP Packets+Code explained

In this code, we create a **DatagramSocket** using **new DatagramSocket()**, and then create a **DatagramPacket** using **new DatagramPacket(buffer, buffer.length, address, port)**. The **buffer** variable is a byte array that contains the data to be sent, and **address** and **port** specify the destination IP address and port number.

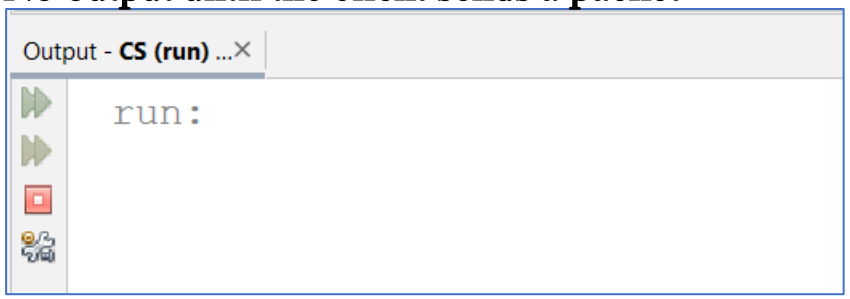
We then send the packet using **socket.send(packet)** and close the socket using **socket.close()**.

Note that the destination IP address and port number must be reachable from the sending host, and any firewalls or network configurations must allow outgoing UDP traffic on the specified port.

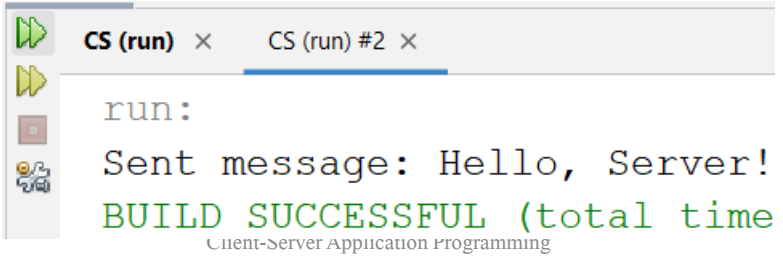
Sending UDP Packets code in NetBeans-Output

```java
package cs; import java.net.DatagramPacket; import java.net.DatagramSocket;
import java.net.InetAddress;
public class UDPClient {
    public static void main(String[] args) throws Exception {
        int port = 5005;
        String message = "Hello, Server!";
        InetAddress address = InetAddress.getByName("localhost");
        byte[] buffer = message.getBytes();
        DatagramSocket socket = new DatagramSocket();
        DatagramPacket packet = new DatagramPacket(buffer, buffer.length, address, port);
        socket.send(packet);
        System.out.println("Sent message: " + message);
        socket.close();
    }
}
```

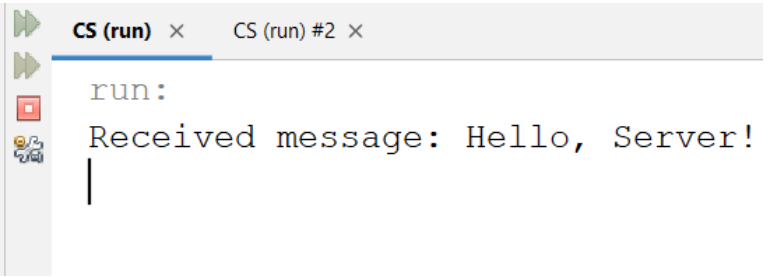
No output until the client sends a packet



Client sent Packets



Server received packets, but still waiting...



Question

Modify the above program to send packets based on user input.

Building a UDP Client/Server

The previous example illustrates the technical details of how an individual packet may be sent and received. But applications need a series of packets, not just one. The next example shows how to build a UDP server, a long-running system that is capable of serving many requests during its lifetime.

The type of service that is provided is an echo service, which echoes back the contents of a packet. The echo service runs on a well-known port, port 7, and if it is known that a system has an echo server installed, the server may be accessed by clients to see if a system is up and running (similar to the ping application). The example below demonstrates how to write an echo server that will receive packets from the client read then send results back to the client.

Server side code

Server Side code

The following example involves building an echo service, which transmits any packet it receives straight back to the sender. The code uses no new networking classes or methods, but employs a special technique. It loops continuously to serve one client after another.

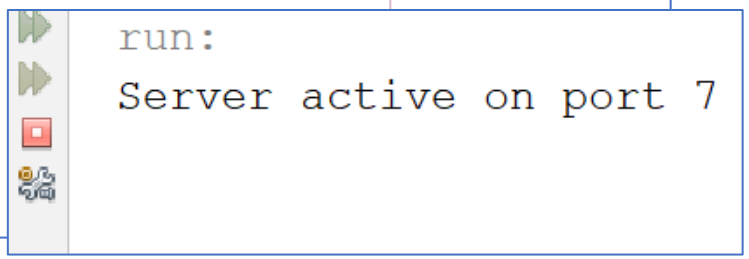
Though only one UDP packet will be processed at a time, the delay between receiving a packet and dispatching it again is negligible, resulting in the illusion of concurrent processing.

EchoServerPro code

```
1  package cs; import java.net.*; import java.io.*;
2  public class EchoServerPro{
3      // UDP port to which service is bound
4      public static final int SERVICE_PORT = 7;
5      // Max size of packet, large enough for almost any client
6      public static final int BUFSIZE = 4096;
7      // Socket used for reading and writing UDP packets
8      private DatagramSocket socket;
9      public EchoServerPro(){
10     try{
11     // Bind to the specified UDP port, to listen for incoming data packets
12     socket = new DatagramSocket( SERVICE_PORT );
13     System.out.println ("Server active on port " + socket.getLocalPort() );
14     }
       catch (Exception e){
16     System.err.println ("Unable to bind port");
17     }
18     }
```

EchoServerPro code+

```
19  public void serviceClients(){
20      // Create a buffer large enough for incoming packets
21      byte[] buffer = new byte[BUFSIZE];
22      for (;;){
23      try{
24      // Create a DatagramPacket for reading UDP packets
25      DatagramPacket packet = new DatagramPacket( buffer, BUFSIZE );
26      // Receive incoming packets
27      socket.receive(packet);
28          System.out.println ("Packet received from " +
29          packet.getAddress() + ":" +packet.getPort() +" of length " + packet.getLength() );
30          // Echo the packet back - address and port are already set for us !
31          socket.send(packet);
32      }
33      catch (IOException ioe){
34      System.err.println ("Error : " + ioe);
35      }}
36  }
37      public static void main(String args[]){
38          EchoServerPro server = new EchoServerPro();
39          server.serviceClients();
40      }
41  }
```



Test the code on your on NetBeans

For testing purposes, I have included for you the entire code in the following slides.

EchoServerPro Code

```
package cs; import java.net.*; import java.io.*;

public class EchoServerPro{

    // UDP port to which service is bound

    public static final int SERVICE_PORT = 7;

    // Max size of packet, large enough for almost any client

    public static final int BUFSIZE = 4096;

    // Socket used for reading and writing UDP packets

    private DatagramSocket socket;
```

EchoClientPro Code

```
public EchoServerPro(){
    try{
    // Bind to the specified UDP port, to listen for incoming data packets
    socket = new DatagramSocket( SERVICE_PORT );
    System.out.println ("Server active on port " + socket.getLocalPort() );
    }
    catch (Exception e){
    System.err.println ("Unable to bind port");
    }
    }
```

EchoClientPro Code

```
public void serviceClients(){
    // Create a buffer large enough for incoming packets
    byte[] buffer = new byte[BUFSIZE];
    for (;;){
    try{
    // Create a DatagramPacket for reading UDP packets
    DatagramPacket packet = new DatagramPacket( buffer, BUFSIZE );
    // Receive incoming packets
    socket.receive(packet);
        System.out.println ("Packet received from " +
        packet.getAddress() + ":" +packet.getPort() +" of length " + packet.getLength() );
```

EchoClientPro Code

```
// Echo the packet back - address and port are already set for us !
        socket.send(packet);
    }
    catch (IOException ioe){
    System.err.println ("Error : " + ioe);
    } }}
    public static void main(String args[]){
        EchoServerPro server = new EchoServerPro();
        server.serviceClients();
    }}
```

Client side code

EchoClientPro Code

The following client can be used with the echo service and can easily be adapted to support other services. Repeated packets are sent to the echo service, and a timeout is caught to prevent the service from stalling if a packet becomes lost, and the client then waits to receive it. Remember that packet loss in an intranet environment is unlikely, but with slow network connections on the Internet it is quite possible.

EchoClientCode

```
1  package cs; import java.net.*; import java.io.*;
2  public class EchoClientPro{
3      // UDP port to which service is bound
4      public static final int SERVICE_PORT = 7;
5      // Max size of packet
6      public static final int BUFSIZE = 256;
7  public static void main(String args[]){
8          String hostname = "localhost";
9          // Get an InetAddress for the specified hostname
           InetAddress addr = null;
11         try{
12             // Resolve the hostname to an InetAddr
13             addr = InetAddress.getByName(hostname);
14         }
15         catch (UnknownHostException uhe){
16         System.err.println ("Unable to resolve host");
17         return;
18         }
19         try{
20             // Bind to any free port
21             DatagramSocket socket = new DatagramSocket();
22             // Set a timeout value of two seconds
23             socket.setSoTimeout (2 * 1000);
24             for (int i = 1 ; i <= 6; i++){
```

EchoClientCode+

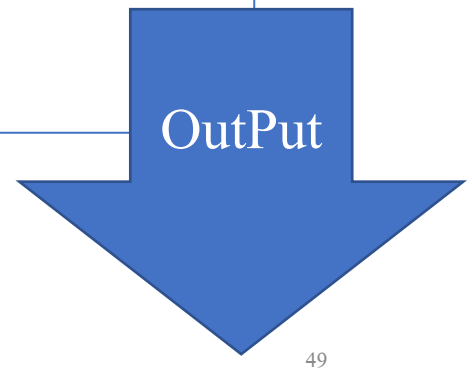
```
25                  // Copy some data to our packet
26                  String message = "Packet number " + i ;
27                  char[] cArray = message.toCharArray();
28                  byte[] sendbuf = new byte[cArray.length];
29                  for (int offset = 0; offset < cArray.length ; offset++){
30                  sendbuf[offset] = (byte)
31                  cArray[offset];
32                  }
33          // Create a packet to send to the UDP  server
34          DatagramPacket sendPacket = new DatagramPacket(sendbuf, cArray.length, addr, SERVICE_PORT);
35          System.out.println ("Sending packet to " + hostname);
36          // Send the packet
37          socket.send(sendPacket);
38          System.out.print ("Waiting for packet.... ");
39          // Create a small packet for receiving UDP packets
40          byte[] recbuf = new byte[BUFSIZE];
41          DatagramPacket receivePacket = new
42          DatagramPacket(recbuf,BUFSIZE);
43          // Declare a timeout flag
44          boolean timeout = false;
45          // Catch any InterruptedIOException that is thrown
46          // while waiting to receive a UDP packet
47          try{
48              socket.receive (receivePacket);
```

EchoClientCode++

```
49          }
50          catch (InterruptedIOException ioe){
51              timeout = true;
52          }
53      if (!timeout){
54          System.out.println ("packet received!");
55          System.out.println ("Details : " + receivePacket.getAddress());
56          // Obtain a byte input stream to read the UDP packet
57          ByteArrayInputStream bin = new  ByteArrayInputStream (receivePacket.getData(), 0,
58          receivePacket.getLength() );
59          // Connect a reader for easier access
60          BufferedReader reader = new BufferedReader ( new InputStreamReader ( bin ) );
61          // Loop indefinitely
62          for (;;){
63              String line = reader.readLine();
64              // Check for end of data
65              if (line == null)
66              break;
67              else
68              System.out.println (line);
69          }
70      }
71      else{
72          System.out.println ("packet lost!");
```

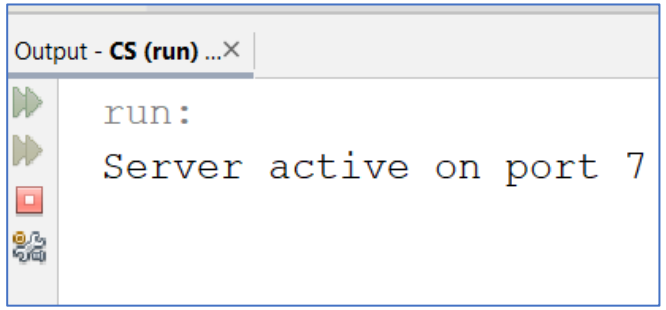
EchoClientCode+++

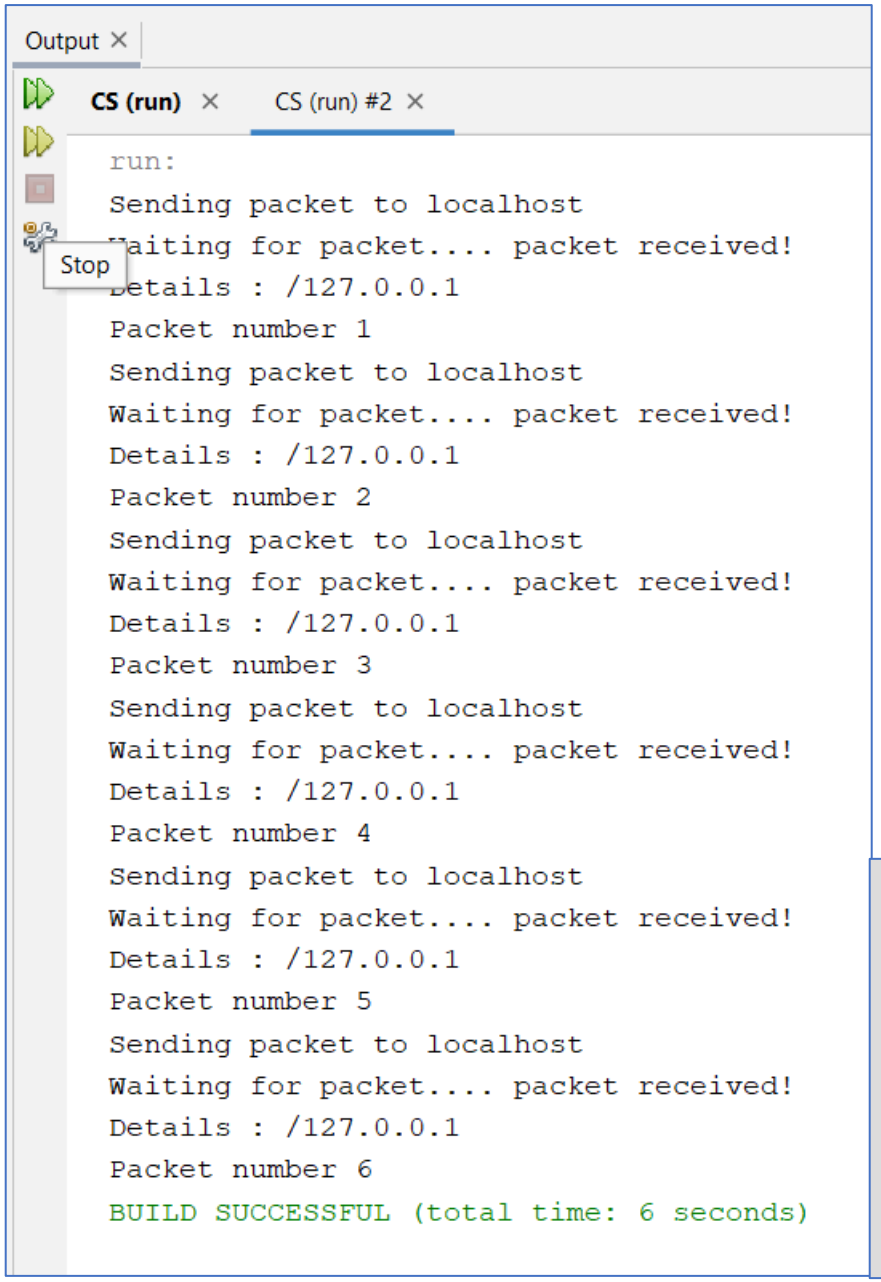
```
73      }
74              // Sleep for a second, to allow user to see packet
75          try{
            Thread.sleep(1000);
77          }
78          catch (InterruptedException ie) { }
79          }
80      }
81          catch (IOException ioe){
82          System.err.println ("Socket error " + ioe);
83          }
84          }
85      }
```



Complete client/Server output

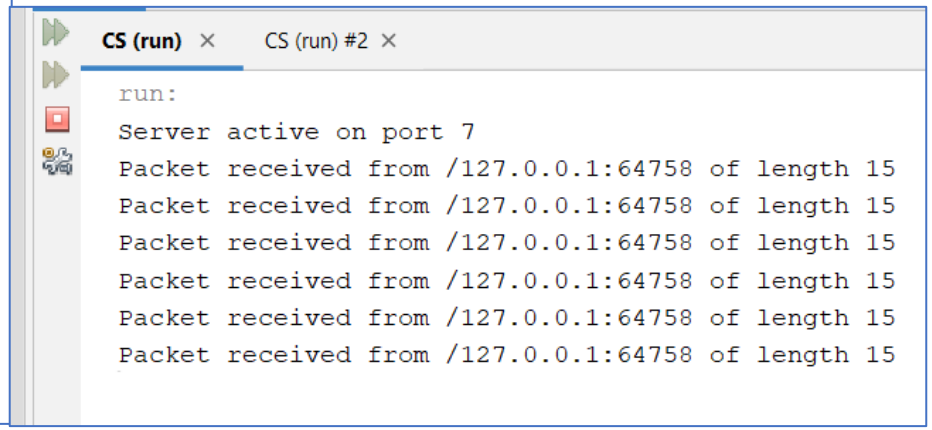
Server waiting for the client to send packets





EchoClientPro Sending and receiving packets

Server After EchoClientPro has sent the packets



Test the Code

For purposes of tests, I have include for you the EchoClientPro code in the slides below.

EchoClientPro Code

```
package cs; import java.net.*; import java.io.*;

public class EchoClientPro{

        // UDP port to which service is bound

        public static final int SERVICE_PORT = 7;

        // Max size of packet

        public static final int BUFSIZE = 256;

        public static void main(String args[]){

                String hostname = "localhost";

                // Get an InetAddress for the specified hostname

                InetAddress addr = null;
```

EchoClientPro Code

```
    try{
    // Resolve the hostname to an InetAddr
    addr = InetAddress.getByName(hostname);
}
catch (UnknownHostException uhe){
System.err.println ("Unable to resolve host");
return;
}
```

EchoClientPro Code

```
try{
  // Bind to any free port
   DatagramSocket socket = new DatagramSocket();
   // Set a timeout value of two seconds
socket.setSoTimeout (2 * 1000);
 for (int i = 1 ; i <= 6; i++){
                    // Copy some data to our packet
                    String message = "Packet number " + i ;
                    char[] cArray = message.toCharArray();
                    byte[] sendbuf = new byte[cArray.length];
                    for (int offset = 0; offset < cArray.length ; offset++){
                    sendbuf[offset] = (byte)
                     cArray[offset];
 }
```

EchoClientPro Code

```
// Create a packet to send to the UDP  server
DatagramPacket sendPacket = new DatagramPacket(sendbuf, cArray.length, addr, SERVICE_PORT);
 System.out.println ("Sending packet to " + hostname);
 // Send the packet
 socket.send(sendPacket);
    System.out.print ("Waiting for packet.... ");
    // Create a small packet for receiving UDP packets
    byte[] recbuf = new byte[BUFSIZE];
    DatagramPacket receivePacket = new
    DatagramPacket(recbuf,BUFSIZE);
```

EchoClientPro Code

```
// Declare a timeout flag
boolean timeout = false;
 // Catch any InterruptedIOException that is thrown
// while waiting to receive a UDP packet
try{
  socket.receive (receivePacket);
}
catch (InterruptedIOException ioe){
  timeout = true;
}
```

EchoClientPro Code

```
if (!timeout){
System.out.println ("packet received!");
System.out.println ("Details : " + receivePacket.getAddress());
 // Obtain a byte input stream to read the UDP packet
ByteArrayInputStream bin = new  ByteArrayInputStream
(receivePacket.getData(), 0,receivePacket.getLength() );
 // Connect a reader for easier access
 BufferedReader reader = new BufferedReader ( new InputStreamReader ( bin ) );
```

EchoClientPro Code

```
// Loop indefinitely
    for (;;){
        String line = reader.readLine();
        // Check for end of data
        if (line == null)
        break;
        else
        System.out.println (line);
    }}
else{
    System.out.println ("packet lost!");
}
```

EchoClientPro Code

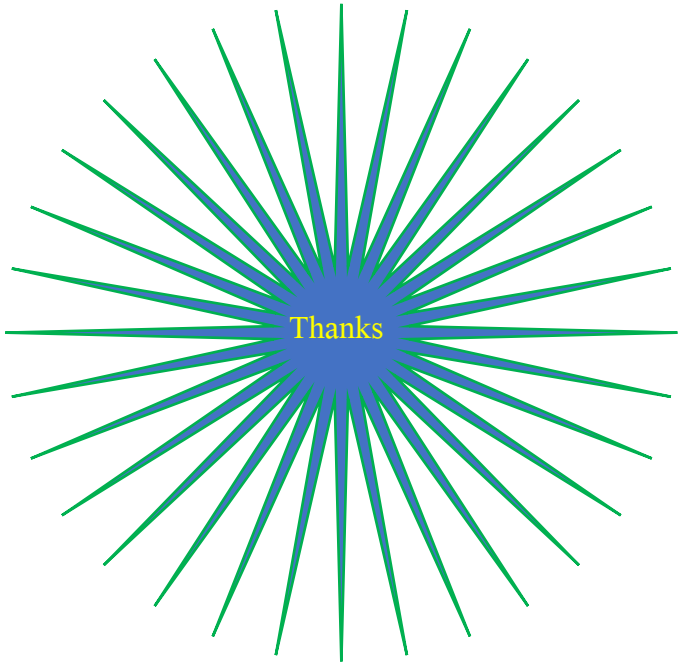
```
// Sleep for a second, to allow user to see packet
   try{
   Thread.sleep(1000);
   }
   catch (InterruptedException ie) { }
   }}
   catch (IOException ioe){
   System.err.println ("Socket error " + ioe);
   } }}
```

Summary

Summary

1. User Datagram Protocol(Overview
2. Datagram Packet Class,
3. DatagramSocket Class,
4. Listening for UDP Packets,
5. Sending UDP packets,
6. Building a UDP Client/Server,

Thank you for Listening



References

Java Network Programming and Distributed Computing, David R, Michael R (2002), Publisher: Addison Wesley, ISBN: 0-201-71037-4

PramatarovHi, M. (2022, November 18). *What is domain name resolution?* ClouDNS Blog. Retrieved March 30, 2023, from https://www.cloudns.net/blog/domain-name-resolution/

What is domain name resolution. BleepingComputer. (2004, April 9). Retrieved March 30, 2023, from https://www.bleepingcomputer.com/tutorials/what-is-domain-name-resolution/