Chapter: 10

UDP

1. UDP Protocol
2. UDP Client
3. UDP Servers
4. The DatagramPacket Class: The Constructor, The get Methods, setter Methods
5. The DatagramPacket Class: The Constructor, Sending and Receiving Datagrams, Managing Connections
6. Socket Options: SO_TIMEOUT, SORCVBUF, SO_SNDBUF, SO_RSUMEADDR, SO_BROADCAST and IP_TOS
7. UDP Appications: Simple UDP Clients, UDPServer and A UDP Echo Client
8. DatagramChannel: Using DatagramChannel

UDP Protocol:

- There are many kinds of applications in which raw speed is more important than getting every bit right.
- For example, in real-time audio or video, lost or swapped packets of data simply appear as static. Static is tolerable, but awkward pauses in the audio stream, when TCP requests retransmission or waits for a wayward packet to arrive, are unacceptable.
- In other applications, reliability tests can be implemented in the application layer.
- If a client sends a short UDP request to a server, it may assume that the packet is lost if no response is returned within an established period of time;
  This is one way the DomainName System (DNS) works (DNS can also operate over TCP.).
- In fact, you could implement a reliable file transfer protocol using UDP, and many people have:
  Network File System (NFS), Trivial FTP (TFTP), and FSP, a more distant relative of FTP, all use UDP.
  (The latest version of NFS can use either UDP or TCP.)
- In these protocols, the application is responsible for reliability; UDP doesn't take care of it (the application must handle missing or out-of-order packets)
- This is a lot of work, but there's no reason it can't be done although if you find yourself writing this code,

think carefully about whether you might be better off with TCP.

- The difference between TCP and UDP is often explained by analogy with the phone system and the post office.
- TCP is like the phone system. When you dial a number, the phone is answered and a connection is established between the two parties.
- As you talk, you know that the other party hears your words in the order in which you say them.
- If the phone is busy or no one answers, you find out right away.
- Java's implementation of UDP is split into two classes: DatagramPacket and Datagram Socket.
- The DatagramPacket class represents a data packet that can be sent or received over a UDP connection. It encapsulates the data to be sent or received, along with information about the destination or source address and port number.
- The DatagramSocket class represents a UDP socket that can be used to send and receive DatagramPackets. It provides methods for creating, binding, and closing sockets, as well as for sending and receiving DatagramPackets.

- To send a DatagramPacket, you first create a DatagramSocket object and bind it to a local address and port using the bind() method. Then you create a DatagramPacket object with the data you want to send and the destination address and port. Finally, you call the send() method on the DatagramSocket object, passing in the DatagramPacket object.

- To receive a DatagramPacket, you create a DatagramSocket object and bind it to a local address and port using the bind() method. Then you create a DatagramPacket object to hold the incoming data. Finally, you call the receive() method on the DatagramSocket object, passing in the DatagramPacket object.

UDP Client:

First, open a socket on port 0:

DatagramSocket socket = new DatagramSocket(0);

This is very different than a TCP socket.

You only specify a local port to connect to.

The socket does not know the remote host or address.

By specifying port 0 you ask Java to pick a random available port for you, much as with server sockets.

The **next step** is optional but highly recommended.

Set a timeout on the connection using the setSoTimeout() method.

Timeouts are measured in milliseconds, so this statement sets the socket to time out after 10 seconds of nonresponsiveness: *socket.setSoTimeout(10000);*

Next you need to set up the packets. You'll need two, one to send and one to receive.

```
InetAddress host = InetAddress.getByName("time.nist.gov");
DatagramPacket request = new DatagramPacket(new byte[1], 1, host, 13);
```

The packet that receives the server's response just contains an empty byte array.

This needs to be large enough to hold the entire response. If it's too small, it will be silently truncated 1k should be enough space:

```
byte[] data = new byte[1024];
DatagramPacket response = new DatagramPacket(data, data.length);
```

Now you're ready. First send the packet over the socket and then receive the response:

```
socket.send(request);

socket.receive(response);
```

Finally, extract the bytes from the response and convert them to a string you can show to the end user:

```
String daytime = new String(response.getData(), 0, response.getLength(),
"US-ASCII");

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

In Java 6 and earlier, you'll want to explicitly close the socket in a finally block to release resources the socket holds:

```
DatagramSocket socket = null;

try {

socket = new DatagramSocket(0);

// connect to the server...

} catch (IOException ex) {

System.err.println(ex);

} finally {

if (socket != null) {

try {
```

```
socket.close();
} catch (IOException ex) {
// ignore
}
}
}
```

Example: A daytime protocol client

```
import java.io.*;
import java.net.*;
public class DaytimeUDPClient {
private final static int PORT = 13;
private static final String HOSTNAME = "time.nist.gov";
public static void main(String[] args) {
try (DatagramSocket socket = new DatagramSocket(0)) {
socket.setSoTimeout(10000);
InetAddress host = InetAddress.getByName(HOSTNAME);
DatagramPacket request = new DatagramPacket(new byte[1], 1,
host , PORT);
```

```
DatagramPacket response = new DatagramPacket(new byte[1024], 1024);
socket.send(request);
socket.receive(response);
String result = new String(response.getData(), 0, response.getLength(),
"US-ASCII");
System.out.println(result);
} catch (IOException ex) {
ex.printStackTrace();
}
}
}
```

UDP Server:

A UDP server follows almost the same pattern as a UDP client, except that you usually receive before sending and don't choose an anonymous port to bind to. Unlike TCP, there's no separate DatagramServerSocket class.

For example, let's implement a daytime server over UDP. Begin by opening a datagram socket on a well-known port. For daytime, this port is 13:

DatagramSocket socket = new DatagramSocket(13);

Next, **create a packet** into which to receive a request. You need to supply both a byte array in which to store incoming data, the offset into the array, and the number of bytes to store. Here you set up a packet with space for 1,024 bytes starting at 0:

DatagramPacket request = new DatagramPacket(new byte[1024], 0, 1024);

Then receive it: socket.receive(request);

This call blocks indefinitely until a UDP packet arrives on port 13.

When it does, Java fills the byte array with data and the receive() method returns.

Next, **create a response packet**.

This has **four parts**: the raw data to send, the number of bytes of the raw data to send, the host to send to, and the port on that host to address.

String daytime = new Date().toString() + "\r\n";

byte[] data = daytime.getBytes("US-ASCII");

InetAddress host = request.getAddress();

int port = request.getPort();

DatagramPacket response = new DatagramPacket(data, data.length, host, port);

Finally, send the response back over the same socket that received it: socket.send(response);

Example 12-2. A daytime protocol server

```
import java.net.*;
import java.util.Date;
import java.util.logging.*;
import java.io.*;
public class DaytimeUDPServer {
private final static int PORT = 13;
private final static Logger audit = Logger.getLogger("requests");
private final static Logger errors = Logger.getLogger("errors");
public static void main(String[] args) {
try (DatagramSocket socket = new DatagramSocket(PORT)) {
while (true) {
try {
DatagramPacket request = new DatagramPacket(new byte[1024], 1024);
socket.receive(request);
```

```
String daytime = new Date().toString();
byte[] data = daytime.getBytes("US-ASCII");
DatagramPacket response = new DatagramPacket(data, data.length,
request.getAddress(), request.getPort());
socket.send(response);
audit.info(daytime + " " + request.getAddress());
} catch (IOException | RuntimeException ex) {
errors.log(Level.SEVERE, ex.getMessage(), ex);
}
}
} catch (IOException ex) {
errors.log(Level.SEVERE, ex.getMessage(), ex);
}
}
}
```

The DatagramPacket Class:

The UDP header adds only eight bytes to the IP header. The UDP header includes source and destination port numbers, the length of everything that follows the IP header, and an optional checksum.

Because port numbers are given as two-byte unsigned integers, 65,536 different possible UDP ports are available per host.

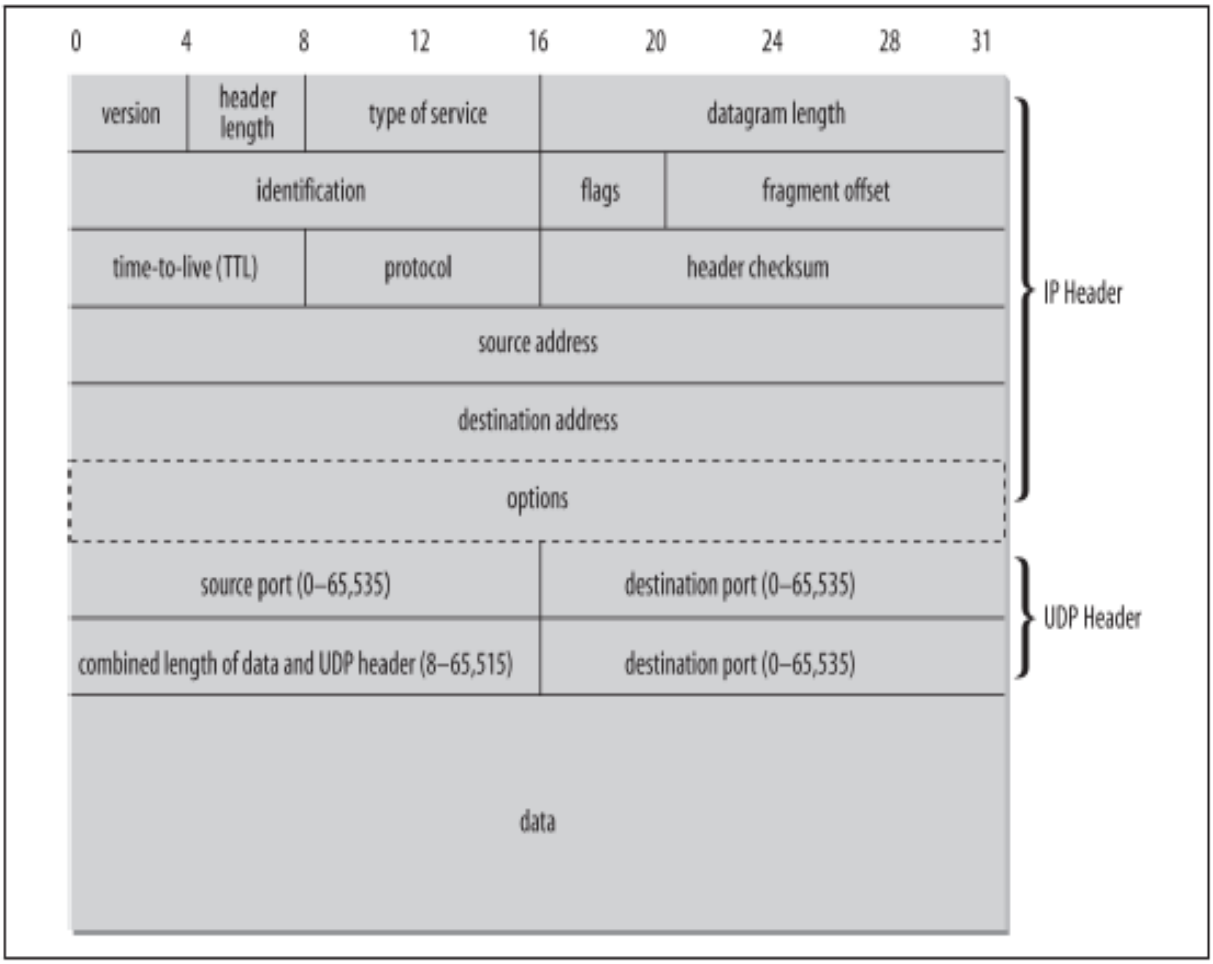
These are distinct from the 65,536 different TCP ports per host.

Because the length is also a two-byte unsigned integer, the number of bytes in a datagram is limited to 65,536 minus the eight bytes for the header.

However, this is redundant with the datagram length field of the IP header, which limits datagrams to between 65,467 and 65,507 bytes.

(The exact number depends on the size of the IP header.) The checksum field is optional and not used in or accessible from application layer programs.

If the checksum for the data fails, the native network software silently discards the datagram; neither the sender nor the receiver is notified. UDP is an unreliable protocol, after all.



In Java, a UDP datagram is represented by an instance of the DatagramPacket class:

public final class DatagramPacket extends Object

This class provides methods to get and set the source or destination address from the IP header, to get and set the source or destination port, to get and set the data, and to get and set the length of the data.

The Constructors:

DatagramPacket uses different constructors depending on whether the packet will be used to send data or to receive data.

1.Constructors for receiving datagrams:

These two constructors create new DatagramPacket objects for receiving data from the network:

public DatagramPacket(byte[] buffer, int length) public DatagramPacket(byte[] buffer, int offset, int length)

If the first constructor is used, when a socket receives a datagram, it stores the datagram's data part in buffer beginning at buffer[0] and continuing until the packet is completely stored or until length bytes have been written into the buffer.

If the second constructor is used, storage begins at buffer[offset] instead.

2.Constructors for sending datagrams:

These four constructors create new DatagramPacket objects used to send data across the network:

public *DatagramPacket(**byte**[] data, **int** length, InetAddress destination, **int** port)*
public *DatagramPacket(**byte**[] data, **int** offset, **int** length, InetAddress destination, **int** port)*
public *DatagramPacket(**byte**[] data, **int** length, SocketAddress destination)*
public *DatagramPacket(**byte**[] data, **int** offset, **int** length, SocketAddress destination)*

Each constructor creates a new DatagramPacket to be sent to another host. The packet is filled with length bytes of the data array starting at offset or 0 if offset is not used.

The get Methods:

DatagramPacket has six methods that retrieve different parts of a datagram;

public InetAddress getAddress()

The getAddress() method returns an InetAddress object containing the address of the remote host.

If the datagram was received from the Internet, the address returned is the address of the machine that sent it (the source address).

On the other hand, if the datagram was created locally to be sent to a remote machine, this method returns the address of the host to which the datagram is addressed (the destination address).

public int getPort()

The getPort() method returns an integer specifying the remote port.

If this datagram was received from the Internet, this is the port on the host that sent the packet.

If the datagram was created locally to be sent to a remote host, this is the port to which the packet is addressed on the remote machine.

public SocketAddress getSocketAddress()

The getSocketAddress() method returns a SocketAddress object containing the IP address and port of the remote host.

As is the case for getInetAddress(), if the datagram was received from the Internet, the address returned is the address of the machine that sent it (the source address).

On the other hand, if the datagram was created locally to be sent to a remote machine, this method returns the address of the host to which the datagram is addressed (the destination address).

The net effect is not noticeably different than calling getAddress() and getPort(). Also, if you're using nonblocking I/O, the DatagramChannel class accepts a SocketAd dress but not an InetAddress and port

public byte[] getData()

The getData() method returns a byte array containing the data from the datagram.

It's often necessary to convert the bytes into some other form of data before they'll be useful to your program.

One way to do this is to change the byte array into a String. For example, given a DatagramPacket dp received from the network, you can convert it to a UTF-8 String like this:

String s = new String(dp.getData(), "UTF-8");

the datagram does not contain text, One approach is to convert the byte array returned by getData() into a ByteArrayInput Stream

For example:

InputStream in = new ByteArrayInputStream(packet.getData(), packet.getOffset(), packet.getLength());

The ByteArrayInputStream can then be chained to a DataInputStream:

DataInputStream din = new DataInputStream(in);

The data can then be read using the DataInputStream's readInt(), readLong(), read Char(), and other methods.

public int getLength()

The getLength() method returns the number of bytes of data in the datagram

public int getOffset()

This method simply returns the point in the array returned by getData() where the data from the datagram begins.

Example: Construct a DatagramPacket to receive data

```
import java.io.*;
import java.net.*;
public class DatagramExample {
public static void main(String[] args) {
String s = "This is a test.";
try {
byte[] data = s.getBytes("UTF-8");
InetAddress ia = InetAddress.getByName("www.ibiblio.org");
int port = 7;
DatagramPacket dp
= new DatagramPacket(data, data.length, ia, port);
System.out.println("This packet is addressed to "
+ dp.getAddress() + " on port " + dp.getPort());
System.out.println("There are " + dp.getLength()
+ " bytes of data in the packet");
System.out.println(
new String(dp.getData(), dp.getOffset(), dp.getLength(), "UTF-8"));
} catch (UnknownHostException | UnsupportedEncodingException ex) {
```

System.err.println(ex);

}

}

}

The setter Methods:

Most of the time, the six constructors are sufficient for creating datagrams.

However, Java also provides several methods for changing the data, remote address, and remote port after the datagram has been created

public void setData(byte[] data)

The setData() method changes the payload of the UDP datagram

public void setData(byte[] data, int offset, int length)

This overloaded variant of the setData() method provides an alternative approach to sending a large quantity of data

public void setAddress(InetAddress remote)

The setAddress() method changes the address a datagram packet is sent to. This might allow you to send the same datagram to many different recipients.

public void setPort(int port)

The setPort() method changes the port a datagram is addressed to.

public void setAddress(SocketAddress remote)

The setSocketAddress() method changes the address and port a datagram packet is sent to. You can use this when replying.

public void setLength(int length)

The setLength() method changes the number of bytes of data in the internal buffer that are considered to be part of the datagram's data as opposed to merely unfilled space.

The DatagramSocket Class

To send or receive a DatagramPacket, you must open a datagram socket. In Java, a datagram socket is created and accessed through the DatagramSocket class:

public class DatagramSocket extends Object

There's no distinction between client sockets and server sockets, as there is with TCP. There's no such thing as a **DatagramServerSocket**.

The Constructors

There are 5 different constructors for creating socket in UDP connections

1. public **DatagramSocket()** throws SocketException

- This constructor creates a socket that is bound to an anonymous port. For example:

```
try {
DatagramSocket client = new DatagramSocket();
// similar to DatagramSocket(0)
// send packets...
} catch (SocketException ex) {
System.err.println(ex);
}
```

- Pick this constructor for a client that initiates a conversation with a server
- The constructor throws a SocketException if the socket can't bind to a port.

2. public **DatagramSocket(int port)** throws SocketException

- This constructor creates a socket that listens for incoming datagrams on a particular port, specified by the port argument.
- Use this constructor to write a server that listens on a well-known port.
- A SocketException is thrown if the socket can't be created.

Q. Scan the all available ports on the system using UDP connection. (or, WAP to scan UDP ports)

```
import java.net.*;
public class UDPPortScanner {
public static void main(String[] args) {
for (int port = 1024; port <= 65535; port++) {
try {
DatagramSocket server = new
DatagramSocket(port);
server.close();
} catch (SocketException ex) {
System.out.println("There is a server on port
```

" + port + ".");

}

}

}

}

3. public **DatagramSocket(int port, InetAddress interface)** throws SocketException

- It creates a socket that listens for incoming datagrams on a **specific port** and **network interface**
- The port argument is the port on which this socket listens for datagrams
- The address argument is an InetAddress object matching one of the host's network addresses
- A SocketException is thrown if the socket can't be created

4. public **DatagramSocket(SocketAddress interface)** throws SocketException

- This constructor is similar to the previous one except that the **network interface address and port are read from a SocketAddress**.

- For example, this code fragment creates a socket that only listens on the local loopback address:

  SocketAddress *address = new*

  InetSocketAddress("127.0.0.1", 9999);

  DatagramSocket socket = new

  DatagramSocket(address);

5. protected **DatagramSocket(DatagramSocketImpl)** throws SocketException

- Unlike sockets created by the other four constructors, this socket is not initially bound to a port.
- Before using it, you have to bind it to a SocketAddress using the bind() method:

  public void **bind**(SocketAddress addr) throws

  SocketException

  - You can pass null to this method, binding the socket to any available address and port.

Sending and Receiving Datagrams

The primary task of the DatagramSocket class is to send and receive UDP datagrams. One socket can both send and receive.

public void send(DatagramPacket dp) throws

IOException

- Once a DatagramPacket is created and a DatagramSocket is constructed, send the packet by passing it to the socket's send() method.
- For example, if theSocket is a Datagram Socket object and theOutput is a DatagramPacket object, send theOutput using theSocket like this:

  theSocket.send(theOutput);

- If there's a problem sending the data, this method may throw an **IOException.**

public void receive(DatagramPacket dp) throws

IOException

- This method receives a single UDP datagram from the network and stores it in the preexisting DatagramPacket object dp.

- Like the accept() method in the ServerSocket class, this method blocks the calling thread until a datagram arrives.

theServer.receive(dp);

public void close()

- Calling a DatagramSocket object's close() method frees the port occupied by that socket. Normally, you'd close socket in a final block of try-catch.

.

```
finally {
try {
if (server != null) server.close();
} catch (IOException ex) {
}
```

..

- In Java 7, DatagramSocket implements AutoCloseable so you can use try-with-resources:

```
try (DatagramSocket server = new
DatagramSocket()) {
// use the socket...}
```

public int getLocalPort()

A DatagramSocket's getLocalPort() method returns an int that represents the local port on which the socket is listening.

For example:

```
DatagramSocket ds = new DatagramSocket();
System.out.println("The socket is using port " +
ds.getLocalPort());
```

public InetAddress getLocalAddress()

A DatagramSocket's getLocalAddress() method returns an InetAddress object that represents the local address to which the socket is bound.

public SocketAddress getLocalSocketAddress()

The getLocalSocketAddress() method returns a SocketAddress object that wraps the local interface and port to which the socket is bound.

Managing Connections

The next **five methods** let you choose which host you can send datagrams to and receive datagrams from, while rejecting all others' packets.

public void connect(InetAddress host, int port)

The connect() method doesn't really establish a connection in the TCP sense. However, it does specify that the DatagramSocket will only send packets to and receive packets from the specified remote host on the specified remote port.

Attempts to send packets to a different host or port will throw an **IllegalArgumentException**. Packets received from a different host or a different port will be discarded without an exception or other notification.

public void disconnect()

The disconnect() method breaks the "connection" of a connected DatagramSocket so that it can once again send packets to and receive packets from any host and port.

public int getPort()

If and only if a DatagramSocket is connected, the getPort() method returns the remote port to which it is connected. Otherwise, it returns –1.

public InetAddress getInetAddress()

If and only if a DatagramSocket is connected, the getInetAddress() method returns the address of the remote host to which it is connected. Otherwise, it returns null.

public InetAddress getRemoteSocketAddress()

If a DatagramSocket is connected, the getRemoteSocketAddress() method returns the address of the remote host to which it is connected.

Otherwise, it returns null.

Socket Options

Java supports six socket options for UDP:

1. SO_TIMEOUT
2. SO_RCVBUF
3. SO_SNDBUF
4. SO_REUSEADDR
5. SO_BROADCAST
6. IP_TOS

SO_TIMEOUT

public void setSoTimeout(int timeout) throws SocketException

public int getSoTimeout() throws IOException

If SO_TIMEOUT is 0, receive() never times out

SO_RCVBUF

It **determines the size of the buffer** used for network I/O. Larger buffers tend to improve performance for reasonably fast. DatagramSocket has methods to set and get the suggested receive buffer size used for network input:

*public void **setReceiveBufferSize**(int size) throws SocketException*

*public int **getReceiveBufferSize**() throws SocketException*

SO_SNDBUF

DatagramSocket has methods to get and set the suggested send buffer size used for network output:

*public void **setSendBufferSize**(int size) throws SocketException*

*public int **getSendBufferSize**() throws SocketException*

however, the operating system is free to ignore this suggestion

SO_REUSEADDR

The SO_REUSEADDR option **does not mean the same thing for UDP** sockets as it does for TCP sockets.

For UDP, SO_REUSEADDR controls whether multiple datagram sockets can bind to the same port and address at the same time.

If multiple sockets are bound to the same port, received packets will be copied to all bound sockets. This option is controlled by these two methods:

*public void **setReuseAddress**(boolean on) throws SocketException*

*public boolean **getReuseAddress**() throws SocketException*

SO_BROADCAST

The SO_BROADCAST option controls whether a socket is allowed to send packets to and receive packets from broadcast addresses such as 192.168.254.255 (in this case, 192.168.254.*) This option is turned on by default, but if you like you can disable it thusly:

socket.setBroadcast(false);

IP_TOS (Type of Service) Class of Service

Exactly same as TCP IP_TOS.

*public int **getTrafficClass**() throws SocketException*

*public void **setTrafficClass**(int trafficClass)*

throws SocketException.

The traffic class is given as an int between 0 and 255. Because this value is copied to an eight-bit field in the TCP header, only the low-order byte of this int is used; and values outside this range cause

IllegalArgumentExceptions.

Some Useful Applications

Echo Client-Server in UDP

Echo Server:

```
import java.io.*;
import java.net.*;
public class UDPEchoServer extends UDPServer {
public final static int DEFAULT_PORT = 7;
public UDPEchoServer() {
super(DEFAULT_PORT);
}
@Override
public void respond(DatagramSocket socket,
DatagramPacket packet)
throws IOException {
DatagramPacket outgoing = new
DatagramPacket(packet.getData(),
packet.getLength(), packet.getAddress(),
packet.getPort());
socket.send(outgoing);
```

```
}
public static void main(String[] args) {
UDPServer server = new UDPEchoServer();
Thread t = new Thread(server);
t.start();
}
}
```

Echo Client:

```
import java.net.*;
public class UDPEchoClient {
public final static int PORT = 7;
public static void main(String[] args) {
String hostname = "localhost";
if (args.length > 0) {
hostname = args[0];
}
try {
InetAddress ia =
InetAddress.getByName(hostname);
```

```
DatagramSocket socket = new DatagramSocket();
SenderThread sender = new SenderThread(socket,
ia, PORT);
sender.start();
Thread receiver = new ReceiverThread(socket);
receiver.start();
} catch (UnknownHostException ex) {
System.err.println(ex);
} catch (SocketException ex) {
System.err.println(ex);
}
}
}
```

DatagramChannel

By default datagram channel is blocking while it can be use in non blocking mode. In order to make it nonblocking we can use the **configureBlocking(false)** method. Datagram channel can be open by calling its one of the static method named as **open()**.

Like SocketChannel and ServerSocketChannel, DatagramChannel is a subclass of SelectableChannel that can be registered with a Selector.

Using DatagramChannel

Opening a socket

open() and open(ProtocolFamily family) − Open method is used to open a datagram channel for single address while parametrized open method open channel for multiple addresses represented as protocol family.

For example:

DatagramChannel channel = DatagramChannel.open();

Receiving

receive(ByteBuffer dst) − This method is used to receive datagram via this channel.

For example:

public SocketAddress receive(ByteBuffer dst) throws IOException

Sending

send(ByteBuffer src, SocketAddress target) – This method is used to send datagram via this channel.

For Example:

public int send(ByteBuffer src, SocketAddress target) throws IOException

Connecting

connect(SocketAddress remote) − This method is used to connect the socket to the remote address.

For example:

SocketAddress remote = new InetSocketAddress("time.nist.gov", 37);

channel.connect(remote);

isConnected() − As already mentioned this method returns the status of whether it is connected or not.

Finally, the **disconnect()** method breaks the connection:

public DatagramChannel disconnect() throws IOException

Reading

DatagramChannel has the usual three read() methods:

1. *public **int** read(ByteBuffer dst) throws IOException*
2. *public **long** read(ByteBuffer[] dsts) throws IOException*
3. *public **long** read(ByteBuffer[] dsts, int offset, int length) throws IOException*

Writing

Naturally, DatagramChannel has the **three write methods** common to all writable, scattering channels, which can be used instead of the send() method:

1. *public int write(ByteBuffer src) throws IOException*
2. *public long write(ByteBuffer[] dsts) throws IOException*
3. *public long write(ByteBuffer[] dsts, int offset, int length) throws IOException*

Q. Echo Client-Server based on UDP Channels.

Echo Server

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;
public class DatagramChannelServer {
public static void main(String[] args) throws
IOException {
DatagramChannel server = DatagramChannel.open();
InetSocketAddress iAdd = new
InetSocketAddress("localhost", 8989);
server.bind(iAdd);
System.out.println("Server Started: " + iAdd);
ByteBuffer buffer = ByteBuffer.allocate(1024);
//receive buffer from client.
SocketAddress remoteAdd = server.receive(buffer);
//change mode of buffer
buffer.flip();
```

```
int limits = buffer.limit();
byte bytes[] = new byte[limits];
buffer.get(bytes, 0, limits);
String msg = new String(bytes);
System.out.println("Client at " + remoteAdd + "
sent: " + msg);
server.send(buffer,remoteAdd);
server.close();
}
}
```

Echo-Client

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;
public class DatagramChannelClient {
public static void main(String[] args) throws
IOException {
```

```
DatagramChannel client = null;
client = DatagramChannel.open();
client.bind(null);
String msg = "Hello World!";
ByteBuffer buffer =
ByteBuffer.wrap(msg.getBytes());
InetSocketAddress serverAddress = new
InetSocketAddress("localhost",
8989);
client.send(buffer, serverAddress);
buffer.clear();
client.receive(buffer);
buffer.flip();
client.close();
}
}
```