Unit-10 (UDP)

UDP (User Datagram Packet) Protocol:

- **Connection-less Protocol** faster than TCP(connection oriented protocol)
- Using User Datagram Protocol, Applications can send data/message to the other hosts without communications or channel or path.
- Even if the destination host is not available, application can send data
- There is **no guarantee** that the data is received in the other side
- Good for video streaming.

Java's Implementation of UDP

To implement UDP in Java, you can use the **DatagramSocket** and **DatagramPacket** classes.

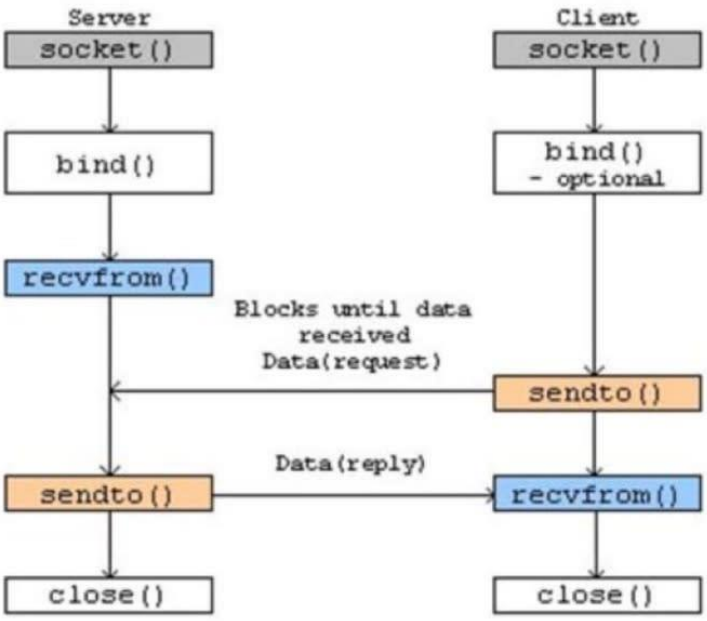
DatagramSocket

1. **DatagramSocket** is used to **send** and **receive** UDP packets.
2. It can be created with or without a specific port number.
3. **send(packet)** sends a packet to a specified address.
4. **receive(packet)** waits for a packet to receive.

DatagramPacket

1. **DatagramPacket** represents a **packet of data** to be sent or received.
2. It contains the **data**, **the length of the data**, **and the address/port** of the destination.
3. **DatagramPacket(byte[] buf, int length)** creates a packet to receive data.
4. **DatagramPacket(byte[] buf, int length, InetAddress address, int port)** creates a packet to send data.

The UDP socket communication between a server and a client consists of several phases as follows.



- **socket() -** Firstly a socket is defined in both server and client.
- **bind() –** In Server side (Bind the socket to a specific **port** to listen for incoming packets.). This is **optional** for the client socket.
- **recvfrom() -** Wait for and receive a packet from a client**.** And same for server
- **sendto() -** After connecting with a client, the server socket sends data to the client.
- **close() -** After successful data exchange, both sockets are closed i.e. system resources allocated for the sockets are released.

UDP Client and Server

UDP Client:

- The UDP client sends data to a UDP server.
- Unlike TCP, UDP does not establish a connection before sending data.
- The client creates a **DatagramSocket** to handle sending and receiving data packets.
- The client prepares the data, wraps it in a **DatagramPacket**, and sends it to the server's address and port.
- The client can use any available port for sending data, and the system assigns this port automatically.

Simple UDP Client:

- **Import Required Classes**: You need **DatagramSocket**, **DatagramPacket**, and **InetAddress**.
- **Create DatagramSocket**: This is UDP socket used to send data.
- **Prepare the Message**: Convert the message **string** to **bytes**.
- **Create DatagramPacket**: This packet contains the **message, destination address, and port.**
- **Send the Packet**: Use the **send** method of **DatagramSocket**.
- **Close the Socket**: Always **close** the socket to free up resources.

Simple Program:

```
import java.net.*;
public class App {
    public static void main(String[] args) throws Exception {
      // Create a DatagramSocket
      DatagramSocket clientSocket = new DatagramSocket();
      // Prepare the message to be sent
      String message = "Hello, UDP Server!";
      byte[] sbuffer = message.getBytes();
      // Create a DatagramPacket with the server's address and port
      InetAddress serverAddress = InetAddress.getByName("localhost");
      DatagramPacket sendPacket = new DatagramPacket(sbuffer, sbuffer.length,
serverAddress, 5000);
      // Send the packet
      clientSocket.send(sendPacket);
      // Close the socket
      clientSocket.close();
    }
```

```
}
```

Step to create UDP Client Socket

1. Creates a **DatagramSocket**.
2. Converts the message to **bytes**.
3. Creates a **DatagramPacket** with the message, **server address**, and **server port**.
4. Send the packet using the **send** method.
5. Wait for a response from the server if needed
6. Closes the socket to free resources.

UDP Server:

- The UDP server receives data from UDP clients.
- The server needs to listen on a specific port, so clients know where to send the data.
- The server creates a **DatagramSocket** bound to the specific port to receive incoming packets.
- The server waits for incoming packets, processes the received data, and optionally sends a response back to the client.
- The server uses a **blocking call** to wait for data, meaning it will pause and wait until a packet arrives.

UDPServer.java

```
import java.net.*;
public class UDPServer {
     public static void main(String[] args) {
        try {
            // Create a DatagramSocket bound to a specific port
            DatagramSocket serverSocket = new DatagramSocket(5000);
            // Prepare a buffer to hold incoming data
            byte[] receiveBuffer = new byte[1024];
            DatagramPacket receivePacket = new DatagramPacket(receiveBuffer,
receiveBuffer.length);
            // Wait for and receive a packet (blocking call)
            System.out.println("Server is waiting for a packet...");
            serverSocket.receive(receivePacket);
            // Process the received data
            String receivedMessage = new String(receivePacket.getData(), 0,
receivePacket.getLength());
            System.out.println("Received: " + receivedMessage);
            // Close the socket
            serverSocket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Lab: write a simple UDP java program to send data from client to server.

Steps to Create UDPServer Socket

1. Creates a DatagramSocket bound to a specific port.
2. Waits for incoming packets using the **receive** method (blocking call).
3. Processes the received data (e.g., converting bytes to a string).
4. Can send a response back to the client if needed.
5. **Closes** the socket to free resources.

The DatagramSocket Class:

- To Send or receive a **DatagramPacket**, you must open a datagram socket. A datagram socket is created and accessed through the **DatagramSocket** class.
- **DatagramSocket** class extends from **Object** class
- If you're writing a **client**, you don't care what the **local port** is, so you call a constructor that lets the system assign an **unused port** (an anonymous port).
- If you're writing a server, client need to know on which port the server is listening for incoming datagrams

Datagram Socket Constructor:

```
DatagramSocket socket = new DatagramSocket();
DatagramSocket socket = new DatagramSocket(int port);
DatagramSocket socket = new DatagramSocket(int port, InetAddress laddr);
DatagramSocket socket = new DatagramSocket(SocketAddress bindaddr);
```

Default Constructor:

```
DatagramSocket socket = new DatagramSocket();
```

- Creates a socket that is bound to any available port on the local machine.
- Typically used by clients who don't care which port they use to send data.

Constructor with Port

```
DatagramSocket socket = new DatagramSocket(int port);
```

- Creates a socket and binds it to a specific port on the local machine.
- Used by servers that need to listen for incoming datagrams on a specific port.

Constructor with Port and InetAddress

```
DatagramSocket socket = new DatagramSocket(int port, InetAddress laddr);
```

- Creates a socket and binds it to a specific port and local address.
- Used when you need to specify both the local port and local address for the socket.

Constructor with SocketAddress

```
DatagramSocket socket = new DatagramSocket(SocketAddress bindaddr);
```

- A **SocketAddress** object representing the local address (IP address and port) to which the socket will be bound.

Datagram Packet class Constructor:

The **DatagramPacket** class is used to create a packet for sending or receiving data via a datagram socket.

Sending Data:

```
DatagramPacket(byte[] buf, int length)
```

- **buf** is the buffer to store the incoming data.
- **length** is the length of the buffer.

Receiving Data:

```
DatagramPacket(byte[] buf, int length, InetAddress address, int port)
```

- **buf** is the data to be sent.
- **length** is the length of the data.
- **address** is the destination IP address.
- **port** is the destination port number.

Receiving Data with offset

```
DatagramPacket(byte[] buf, int offset, int length)
```

- **buf** is the buffer to store the incoming data.
- **offset** is the starting position in the buffer.
- **length** is the length of the buffer.

Sending Data with offset

```
DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)
```

- **buf** is the data to be sent.
- **offset** is the starting position in the data buffer.
- **length** is the length of the data.
- **address** is the destination IP address.
- **port** is the destination port number.

Datagram Packet Class Getter and Setter Methods:

public byte[] getData():returns byte array containing the data from datagram.

public int getLength(): Returns the length of the data to be sent or the length of the data received.

public int getOffset(): Returns the starting point of the data in the buffer.

public InetAddress getAddress(): Returns the IP address of the machine to which this datagram is being sent or from which it was received. Used to to the source or destination IP address

public int getPort(): Used to know the source or destination port.

Sending and receiving data

Sending Data:

1. Create a **DatagramSocket**:

   ```
   DatagramSocket socket = new DatagramSocket();
   ```
2. Prepare the **Data**:
   ```
   String message = "Hello, World!";
   byte[] buffer = message.getBytes();
   ```
3. Create a **DatagramPacket**:
   ```
   InetAddress address = InetAddress.getByName("localhost");
   DatagramPacket packet = new DatagramPacket(buffer, buffer.length, address,
   1234);
   ```
4. Send the **Packet**:
   ```
   socket.send(packet);
   ```
5. Close the **Socket**:
   ```
   socket.close();
   ```

Receiving Data:

1. Create a **DatagramSocket**:
   ```
   DatagramSocket socket = new DatagramSocket(1234);
   ```
2. Prepare a **Buffer**:
   ```
   byte[] buffer = new byte[1024];
   ```
3. Create a **DatagramPacket**:
   ```
   DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
   ```
4. **Receive** the Packet:
   ```
   socket.receive(packet);
   ```
5. **Process** the Data:
   ```
   String receivedMessage = new String(packet.getData(), 0,
   packet.getLength());
   System.out.println("Received: " + receivedMessage);
   ```
6. **Close** the Socket:
   socket.close();

Socket Options

SO_TIMEOUT: Sets a timeout for blocking receive calls.
```
socket.setSoTimeout(2000);
```
If no packet is received within the specified timeout, a **SocketTimeoutException** is thrown.
SO_RCVBUF: sets the receive buffer size for the socket.
```
socket.setReceiveBufferSize(1024);
```

Configures the buffer size used for incoming data.
SO_SNDBUF: Sets the send buffer size for the socket.

```
socket.setSendBufferSize(1024);
```

Configures the buffer size used for outgoing data.
SO_REUSEADDR: Enables or disables the ability to reuse an address.

```
socket.setReuseAddress(true);  // Enable address reuse
```

Allows multiple sockets to bind to the same address and port.
SO_BROADCAST: Enables or disables the ability to send broadcast packets.

```
socket.setBroadcast(true);  // Enable broadca
```

Allows the socket to send packets to the broadcast address.
Example:

```
        DatagramSocket socket = new DatagramSocket();
          // Set socket options
          socket.setSoTimeout(2000);  // Set receive timeout to 2 seconds
          socket.setReceiveBufferSize(1024);  // Set receive buffer size to 1024
bytes
          socket.setSendBufferSize(1024);  // Set send buffer size to 1024 bytes
          socket.setReuseAddress(true);  // Enable address reuse
          socket.setBroadcast(true);  // Enable broadcast
          // Get and print socket options
          System.out.println("SO_TIMEOUT: " + socket.getSoTimeout());
          System.out.println("SO_RCVBUF: " + socket.getReceiveBufferSize());
          System.out.println("SO_SNDBUF: " + socket.getSendBufferSize());
          System.out.println("SO_REUSEADDR: " + socket.getReuseAddress());
          System.out.println("SO_BROADCAST: " + socket.getBroadcast());
```

IP_TOS

Because the traffic class is determined by the value of the IP_TOS field in each IP packet header, it is essentially the same for UDP as it is for TCP

These two methods let you inspect and set the **class of service** for a socket using these two methods:

Public int getTrafficeClass() throws SocketException

Public void setTrafficClass(int trafficClass) throws SocketException

Example:

DatagramSocket s=new DatagramSocket();

s.setTrafficeClass(0xB8); //10111000 in binary

UDP Echo Client

A UDP Echo Client is a program that sends a message to a server and waits for the same message to be sent back (echoed) by the server.

- **Create a socket:**
  Initialize a **DatagramSocket** to send and receive data.
- **Prepare the message:**
  Convert the message you want to send into a **byte array**.
- **Send the message:**
  Create a packet with the **byte array, server address, and port**, then send it through the socket.
- **Receive the echoed message:**
  Wait for the server to send back the same message and store it in a buffer.
- **Display the message:**
  Convert the received byte array back into a string and print it.
- **Close the socket:**
  Close the **DatagramSocket** to free up resources.

DatagramChannel:

- **DatagramChannel** in Java allows for **non-blocking I/O** operations with UDP.
- **DatagramChannel** can be used with a **Selector**, which allows you to manage multiple channels using a **single thread**. This helps in efficiently handling a **large number of connections**.
- **DatagramChannel** works efficiently with **direct buffers**, which can improve I/O performance

Server Code

The server will:

1. Open a **DatagramChannel**.
2. **Bind** it to a specific port.
3. Wait to receive a message.
4. Echo the message back to the sender.

Client Code

The client will:

1. Open a **DatagramChannel**.
2. Send a message to the server.
3. Wait for the echoed message from the server.
4. Print the echoed message.

Simple Example:

Server.java

```java
import java.net.*;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;

public class Server {
    public static void main(String[] args) throws Exception{
        // Open and bind the DatagramChannel to port 1234
            DatagramChannel serverChannel = DatagramChannel.open();
            serverChannel.bind(new InetSocketAddress(1234));
            System.out.println("Server listening on port 1234...");
            // Buffer to receive data
            ByteBuffer buffer = ByteBuffer.allocate(1024);
            // Receive a single message from the client
            buffer.clear();
            InetSocketAddress clientAddress = (InetSocketAddress)
serverChannel.receive(buffer);
            buffer.flip();
            // Print the received message
            String receivedMessage = new String(buffer.array(), 0,
buffer.limit());
            System.out.println("Received from client: " + receivedMessage);
            // Echo the message back to the client
            serverChannel.send(buffer, clientAddress);
            System.out.println("Echoed message back to client.");
            // Close the channel
            serverChannel.close();
    }
}
```

Client.java

```java
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;

public class App {
    public static void main(String[] args) throws Exception {
        // Open a DatagramChannel
            DatagramChannel clientChannel = DatagramChannel.open();
            // Prepare the message to send
            String message = "Hello, Server!";
```

```
            ByteBuffer buffer = ByteBuffer.wrap(message.getBytes());
            // Send the message to the server at localhost on port 1234
            InetSocketAddress serverAddress = new InetSocketAddress("127.0.0.1",
1234);
            clientChannel.send(buffer, serverAddress);
            System.out.println("Message sent to server.");
            // Prepare to receive the echoed message
            ByteBuffer receiveBuffer = ByteBuffer.allocate(1024);
            // Receive the echoed message from the server
            clientChannel.receive(receiveBuffer);
            receiveBuffer.flip();
            // Process and print the received message
            String receivedMessage = new String(receiveBuffer.array(), 0,
receiveBuffer.limit());
            System.out.println("Received from server: " + receivedMessage);
            // Close the channel
            clientChannel.close();
    }
}
```

Output in server:

```
Server listening on port 1234...

Received from client: Hello, Server!

Echoed message back to client.
```

Output in client:

```
Message sent to server.

Received from server: Hello, Server!
```