A Guide to UDP In Java

Last updated: January 25, 2024

Written by: baeldung (https://www.baeldung.com/author/baeldung)

Reviewed by: Grzegorz Piwowarek (https://www.baeldung.com/editor/grzegorz-author)

Networking (https://www.baeldung.com/category/networking)

1. Overview

In this article, we will be exploring networking communication with Java, over the User Datagram Protocol (UDP (https://en.wikipedia.org/wiki/User_Datagram_Protocol)).

UDP is a communication protocol that **transmits independent packets over the network with no guarantee of arrival and no guarantee of the order of delivery**.

Most communication over the internet takes place over the Transmission Control Protocol (TCP), however, UDP has its place which we will be exploring in the next section.

2. Why Use UDP?

UDP is quite different (/cs/udp-vs-tcp) from the more common TCP. But before considering the surface level disadvantages of UDP, it's important to understand that the lack of overhead can make it significantly faster than TCP.

Apart from speed, we also need to remember that some kinds of communication do not require the reliability of TCP but value low latency instead. The video is a good example of an application that might benefit from running over UDP instead of TCP.

3. Building UDP Applications

Building UDP applications is very similar to building a TCP system; the only difference is that we don't establish a point to point connection between a client and a server.

The setup is very straightforward too. Java ships with built-in networking support for UDP – which is part of the *java.net* package. Therefore to perform networking operations over UDP, we only need to import the classes from the *java.net* package: *java.net.DatagramSocket* and *java.net.DatagramPacket*.

In the following sections, we will learn how to design applications that communicate over UDP; we'll use the popular echo protocol for this application.

First, we will build an echo server that sends back any message sent to it, then an echo client that just sends any arbitrary message to the server and finally, we will test the application to ensure everything is working fine.

4. The Server

In UDP communication, a single message is encapsulated in a *DatagramPacket* which is sent through a *DatagramSocket*.

Let's start by setting up a simple server:


```
public class EchoServer extends Thread {

    private DatagramSocket socket;
    private boolean running;
    private byte[] buf = new byte[256];

    public EchoServer() {
        socket = new DatagramSocket(4445);
    }

    public void run() {
        running = true;

        while (running) {
            DatagramPacket packet
              = new DatagramPacket(buf, buf.length);
            socket.receive(packet);

            InetAddress address = packet.getAddress();
            int port = packet.getPort();
            packet = new DatagramPacket(buf, buf.length, address, port);
            String received
              = new String(packet.getData(), 0, packet.getLength());

            if (received.equals("end")) {
                running = false;
                continue;
            }
            socket.send(packet);
        }
        socket.close();
    }
}
```

We create a global *DatagramSocket* which we will use throughout to send packets, a byte array to wrap our messages, and a status variable called *running*.

For simplicity, the server is extending *Thread*, so we can implement everything inside the *run* method.

Inside *run*, we create a while loop that just runs until *running* is changed to false by some error or a termination message from the client.

At the top of the loop, we instantiate a *DatagramPacket* to receive incoming messages.

Next, we call the *receive* method on the socket. This method blocks until a message arrives and it stores the message inside the byte array of the *DatagramPacket* passed to it.

After receiving the message, we retrieve the address and port of the client, since we are going to send the response
back.

Next, we create a *DatagramPacket* for sending a message to the client. Notice the difference in signature with the receiving packet. This one also requires address and port of the client we are sending the message to.

5. The Client

Now let's roll out a simple client for this new server:

```
public class EchoClient {
    private DatagramSocket socket;
    private InetAddress address;

    private byte[] buf;

    public EchoClient() {
        socket = new DatagramSocket();
        address = InetAddress.getByName("localhost");
    }

    public String sendEcho(String msg) {
        buf = msg.getBytes();
        DatagramPacket packet
          = new DatagramPacket(buf, buf.length, address, 4445);
        socket.send(packet);
        packet = new DatagramPacket(buf, buf.length);
        socket.receive(packet);
        String received = new String(
          packet.getData(), 0, packet.getLength());
        return received;
    }

    public void close() {
        socket.close();
    }
}
```

The code is not that different from the server's. We have our global *DatagramSocket* and address of the server. We instantiate these inside the constructor.

We have a separate method which sends messages to the server and returns the response.

We first convert the string message into a byte array, then create a *DatagramPacket* for sending messages.

Next – we send the message. We immediately convert the *DatagramPacket* into a receiving one.

When the echo arrives, we convert the bytes to a string and return the string.

6. The Test

In a class *UDPTest.java*, we simply create one test to check the echoing ability of our two applications:

```
public class UDPTest {
    EchoClient client;

    @Before
    public void setup(){
        new EchoServer().start();
        client = new EchoClient();
    }

    @Test
    public void whenCanSendAndReceivePacket_thenCorrect() {
        String echo = client.sendEcho("hello server");
        assertEquals("hello server", echo);
        echo = client.sendEcho("server is working");
        assertFalse(echo.equals("hello server"));
    }

    @After
    public void tearDown() {
        client.sendEcho("end");
        client.close();
    }
}
```

In *setup*, we start the server and also create the client. While in the *tearDown* method, we send a termination message to the server so that it can close and at the same time we close the client.

7. Conclusion

In this article, we have learned about the User Datagram Protocol and successfully built our own client-server applications that communicate over UDP.