NETWORK DISTRIBUTED SYSTEMS

Lab2: Datagram Socket Programming

DatagramSocket's are Java's mechanism for network communication via UDP instead of TCP. UDP is still layered on top of IP. You can use Java's `DatagramSocket` both for sending and receiving UPD datagrams.

UDP VS TCP

UDP works a bit differently from TCP. When you send data via TCP you first create a connection. Once the TCP connection is established TCP guarantees that your data arrives at the other end, or it will tell you that an error occurred.

With UDP you just send packets of data (datagrams) to some IP address on the network. You have no guarantee that the data will arrive. You also have no guarantee about the order which UDP packets arrive in at the receiver. This means that UDP has less protocol overhead (no stream integrity checking) than TCP.

UDP VS TCP

UDP is appropriate for data transfers where it doesn't matter if a packet is lost in transition. For instance, imagine a transfer of a live TV-signal over the internet. You want the signal to arrive at the clients as close to live as possible. Therefore, if a frame or two are lost, you don't really care.

You don't want the live broadcast to be delayed just to make sure all frames are shown at the client. You'd rather skip the missed frames, and move directly to the newest frames at all times.

UDP VS TCP

This could also be the case with a surveillance camera broadcasting over the internet. Who cares what happened in the past, when you are trying to monitor the present. You don't want to end up being 30 seconds behind reality, just because you want to show all frames to the person monitoring the camera. It is a bit different with the storage of the camera recordings.

You may not want to lose a single frame when recording the images from the camera to disk. You may rather want a little delay, than not have those frames to go back and examine, if something important occurs.

SENDING DATA VIA A DATAGRAMSOCKET

To send data via Java's **`DatagramSocket`** you must first create a **`DatagramPacket`**. Here is how that is done:

```
byte[] buffer = new byte[65508];
InetAddress address = InetAddress.getByName("jenkov.com");

DatagramPacket packet = new DatagramPacket(
    buffer, buffer.length, address, 9000);
```

SENDING DATA VIA A DATAGRAMSOCKET

The byte buffer (the byte array) is the data that is to be sent in the UDP datagram. The length of the above buffer, 65508 bytes, is the maximum amount of data you can send in a single UDP packet.

The length given to the `DatagramPacket` constructor is the length of the data in the buffer to send. All data in the buffer after that amount of data is ignored.

The `InetAddress` instance contains the address of the node (e.g. server) to send the UDP packet to. The `InetAddress` class represents an IP address (Internet Address). The `getByName()` method returns an `InetAddress` instance with the IP address matching the given host name.

The port parameter is the UDP port the server to receiver the data is listeing on. UDP and TCP ports are not the same. A computer can have different processes listening on e.g. port 80 in UDP and in TCP at the same time.

To send the `DatagramPacket` you must create a `DatagramSocket` targeted at sending data. Here is how that is done:

```
DatagramSocket datagramSocket = new DatagramSocket();
```

SENDING DATA VIA A DATAGRAMSOCKET

To send data you call the **`send()`** method, like this:

```
datagramSocket.send(packet);
```

Here is a full example:

```
DatagramSocket datagramSocket = new DatagramSocket();

byte[] buffer = "0123456789".getBytes();
InetAddress receiverAddress = InetAddress.getLocalHost();

DatagramPacket packet = new DatagramPacket(
        buffer, buffer.length, receiverAddress, 80);
datagramSocket.send(packet);
```

RECEIVING DATA VIA A DATAGRAMSOCKET

Receiving data via a **`DatagramSocket`** is done by first creating a **`DatagramPacket`** and then receiving data into it via the **`DatagramSocket`**'s **`receive()`** method. Here is an example:

```
DatagramSocket datagramSocket = new DatagramSocket(80);

byte[] buffer = new byte[10];
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

datagramSocket.receive(packet);
```

Notice how the **`DatagramSocket`** is instantiated with the parameter value 80 passed to its constructor. This parameter is the UDP port the **`DatagramSocket`** is to receive UDP packets on. As mentioned earlier, TCP and UDP ports are not the same, and thus do not overlap. You can have two different processes listening on both TCP and UDP port 80, without any conflict.

RECEIVING DATA VIA A DATAGRAMSOCKET

Second, a byte buffer and a **`DatagramPacket`** is created. Notice how the **`DatagramPacket`** has no information about the node to send data to, as it does when creating a **`DatagramPacket`** for sending data. This is because we are going to use the **`DatagramPacket`** for receiving data, not sending it. Thus no destination address is needed.

Finally the **`DatagramSocket`**'s **`receive()`** method is called. This method blocks until a **`DatagramPacket`** is received.

The data received is located in the **`DatagramPacket`**'s byte buffer. This buffer can be obtained by calling:

```
byte[] buffer = packet.getData();
```

RECEIVING DATA VIA A DATAGRAMSOCKET

How much data was received in the buffer is up to you to find out. The protocol you are using should specify either how much data is sent per UDP packet, or specify an end-of-data marker you can look for instead.

A real server program would probably call the `receive()` method in a loop, and pass all received `DatagramPacket`'s to a pool of worker threads, just like a TCP server does with incoming connections (see multithreading for more details).

EXERCISE

- Write an instant messaging application using UDP.
- Message sequence
- A -> B
- B -> A
- A -> B
- and so on
- Do not worry about multithreading