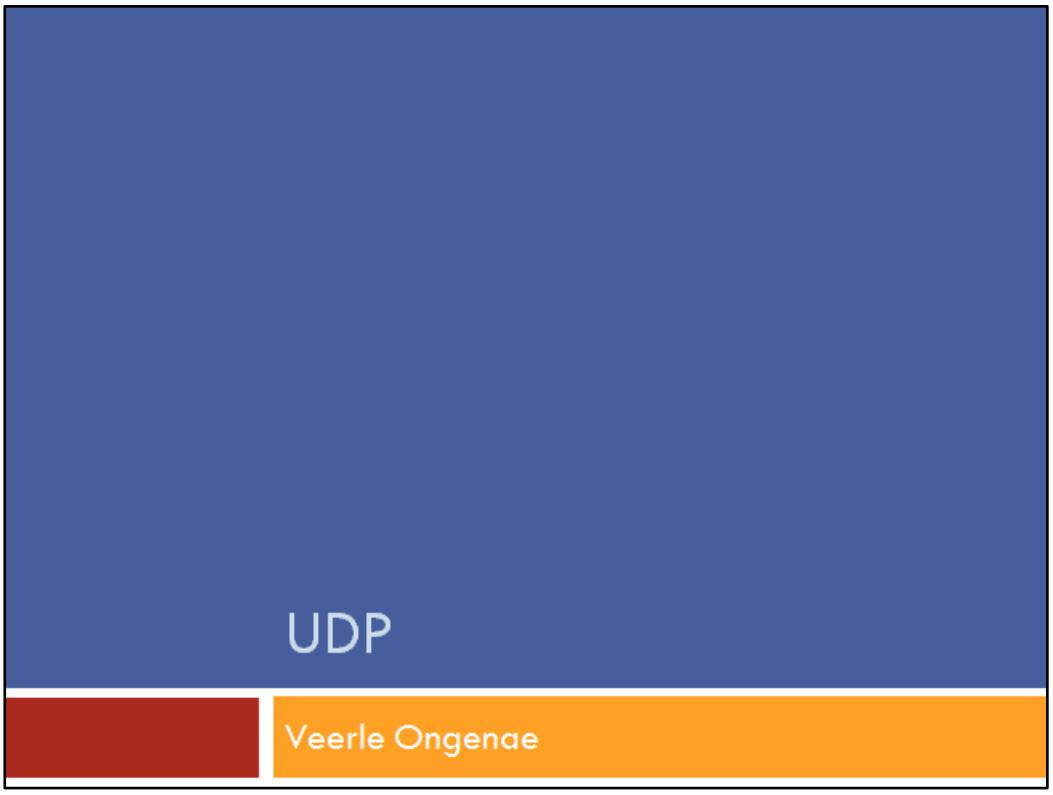
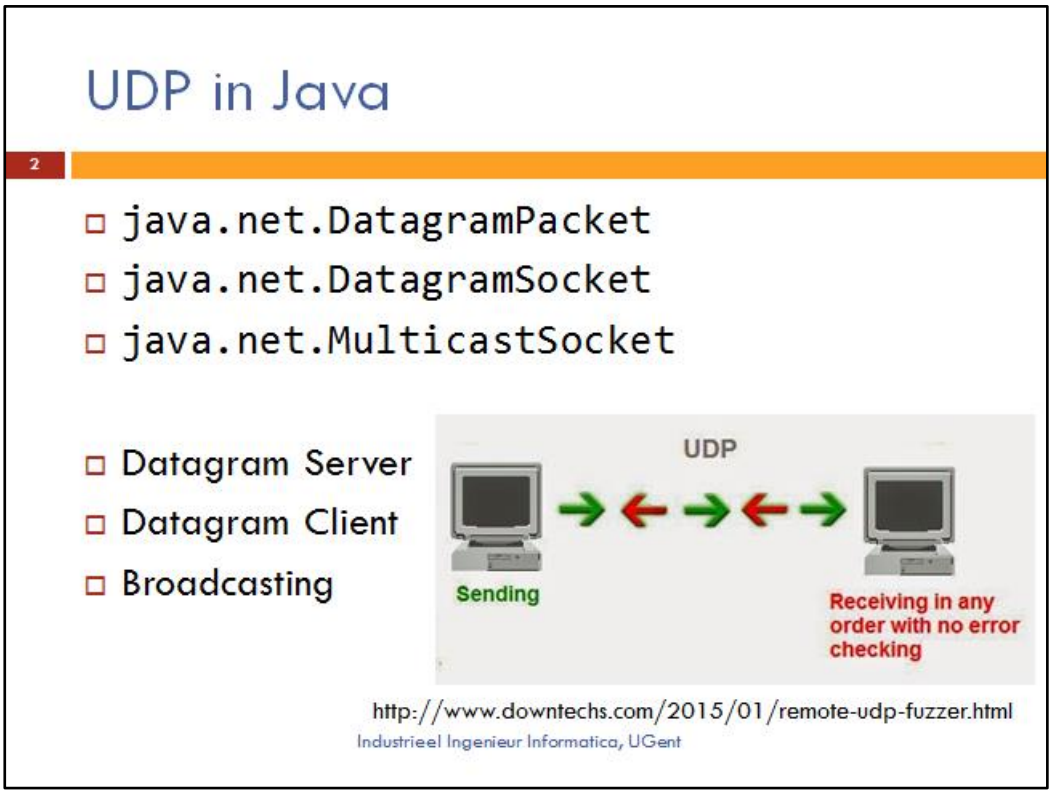


Some applications that you write to communicate over the network will not require the reliable, point-to-point channel provided by TCP. Rather, your applications might benefit from a mode of communication that delivers independent packages of information whose arrival and order of arrival are not guaranteed.



The UDP protocol provides a mode of network communication whereby applications send packets of data, called datagrams, to one another. A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed. The DatagramPacket and DatagramSocket classes in the java.net package implement system-independent datagram communication using UDP.

Clients and servers that communicate via a reliable channel, such as a TCP socket, have a dedicated point-to-point channel between themselves, or at least the illusion of one. To communicate, they establish a connection, transmit the data, and then close the connection. All data sent over the channel is received in the same order in which it was sent. This is guaranteed by the channel.

In contrast, applications that communicate via datagrams send and receive completely independent packets of information. These clients and servers do not have and do not need a dedicated point-to-point channel. The delivery of datagrams to their destinations is not guaranteed. Nor is the order of their arrival.

What Is a Datagram?

Definition: A *datagram* is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.

The java.net package contains three classes to help you write Java programs that use datagrams to send and receive packets over the network: DatagramSocket, DatagramPacket, and MulticastSocket. An application can send and receive DatagramPackets through a DatagramSocket. In addition, DatagramPackets can be broadcast to multiple recipients all listening to a MulticastSocket.

Datagram Server

- Maak `DatagramSocket`
- Lus
  - Maak leeg `DatagramPacket`
  - Wacht op pakket
  - Bepaal poort en adres van de client
  - Maak `DatagramPacket` met antwoord
  - Verstuur antwoord
- Sluit socket

Datagram Server

4

```
DatagramSocket socket = new DatagramSocket(4445);
boolean moreQuotes = true;
while (moreQuotes) {
  byte[] buf = new byte[256];
  DatagramPacket packet = new DatagramPacket(buf, buf.length);
  socket.receive(packet); // receive request

  String dString = …; // figure out response
  buf = dString.getBytes();

  InetAddress address = packet.getAddress();
  int port = packet.getPort();
  packet = new DatagramPacket(buf, buf.length, address, port);
  socket.send(packet); // send the response to the client at "address" and "port"
}
socket.close();
```

Industrieel Ingenieur Informatica, UGent

The server continuously receives datagram packets over a datagram socket. Each datagram packet received by the server indicates a client request for a quotation. When the server receives a datagram, it replies by sending a datagram packet that contains a one-line "quote of the moment" back to the client.

When created, the server creates a DatagramSocket on port 4445 (arbitrarily chosen). This is the DatagramSocket through which the server communicates with all of its clients.
Remember that certain ports are dedicated to well-known services and you cannot use them. If you specify a port that is in use, the creation of the DatagramSocket will fail.
The constructor also opens a BufferedReader on a file named one-liners.txt which contains a list of quotes. Each quote in the file is on a line by itself.

The run method contains a while loop that continues as long as there are more quotes in the file. During each iteration of the loop, the thread waits for a DatagramPacket to arrive over the DatagramSocket. The packet indicates a request from a client. In response to the client's request, the QuoteServerThread gets a quote from the file, puts it in a DatagramPacket and sends it over the DatagramSocket to the client that asked for it.

Let's look first at the section that receives the requests from clients:
byte[] buf = new byte[256];

```
DatagramPacket packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);
```

The first statement creates an array of bytes which is then used to create a **DatagramPacket**. The DatagramPacket will be used to receive a datagram from the socket because of the constructor used to create it. This constructor requires only two arguments: a byte array that contains client-specific data and the length of the byte array. When constructing a DatagramPacket to send over the DatagramSocket, you also must supply the Internet address and port number of the packet's destination. You'll see this later when we discuss how the server responds to a client request.

The last statement in the previous code snippet receives a datagram from the socket (the information received from the client gets copied into the packet). The **receive** method waits forever until a packet is received. If no packet is received, the server makes no further progress and just waits.
Now assume that, the server has received a request from a client for a quote. Now the server must respond. This section of code in the run method constructs the response:

```
String dString = null;
if (in == null) dString = new Date().toString();
else dString = getNextQuote();
buf = dString.getBytes();
```

If the quote file did not get opened for some reason, then in equals null. If this is the case, the quote server serves up the time of day instead. Otherwise, the quote server gets the next quote from the already opened file. Finally, the code converts the string to an array of bytes.

Now, the run method sends the response to the client over the DatagramSocket with this code:

```
InetAddress address = packet.getAddress();
int port = packet.getPort();
packet = new DatagramPacket(buf, buf.length, address, port);
socket.send(packet);
```

The first two statements in this code segment get the Internet address and the port number, respectively, from the datagram packet received from the client. The Internet address and port number indicate where the datagram packet came from. This is where the server must send its response. In this example, the byte array of the datagram packet contains no relevant information. The arrival of the packet itself indicates a request from a client that can be found at the Internet address and port number indicated in the datagram packet.

The third statement creates a new DatagramPacket object intended for sending a

datagram message over the datagram socket. You can tell that the new DatagramPacket is intended to send data over the socket because of the constructor used to create it. This constructor requires four arguments. The first two arguments are the same required by the constructor used to create receiving datagrams: a byte array containing the message from the sender to the receiver and the length of this array. The next two arguments are different: an Internet address and a port number. These two arguments are the complete address of the destination of the datagram packet and must be supplied by the sender of the datagram. The last line of code sends the DatagramPacket on its way.

When the server has read all the quotes from the quote file, the while loop terminates and the run method cleans up:

```
socket.close();
```

Datagram Client

- Maak `DatagramSocket`
- Bepaal poort en adres van de server
- Maak `DatagramPacket`
- Verstuur vraag
- Maak leeg `DatagramPacket`
- Wacht op antwoord
- Sluit socket

Datagram Client

6

```
try (DatagramSocket socket = new DatagramSocket()) {
  byte[] buf = new byte[256];
  InetAddress address = InetAddress.getByName(HOST);
  DatagramPacket packet
    = new DatagramPacket(buf, buf.length, address, 4445);
  socket.send(packet); // send request

  packet = new DatagramPacket(buf, buf.length);
  socket.receive(packet); // get response
  // display response
  String received
    = new String(packet.getData(), 0, packet.getLength());
  System.out.println("Quote of the Moment: " + received);
} catch (… ex) { … }
```

Industrieel Ingenieur Informatica, UGent

The client application in this example is fairly simple. It sends a single datagram packet to the server indicating that the client would like to receive a quote of the moment. The client then waits for the server to send a datagram packet in response.

The QuoteClient class implements a client application for the QuoteServer. This application sends a request to the QuoteServer, waits for the response, and, when the response is received, displays it to the standard output. Let's look at the code in detail.

The QuoteClient class contains one method, the main method for the client application. The top of the main method declares several local variables for its use:

```
InetAddress address;
DatagramSocket socket = null;
DatagramPacket packet;
byte[] sendBuf = new byte[256];
```

Next, the main method creates a DatagramSocket:

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

The client uses a constructor that does not require a port number. This constructor just binds the DatagramSocket to any available local port. It doesn't matter what port the client is bound to because the DatagramPackets contain the addressing information. The server gets the port number from the DatagramPackets and send

its response to that port.

Next, the QuoteClient program sends a request to the server:

```
byte[] buf = new byte[256];
InetAddress address = InetAddress.getByName(HOST);
DatagramPacket packet = new DatagramPacket(buf, buf.length, address,
PORT);
socket.send(packet);
```

The code segment gets the Internet address for the host (presumably the name of the machine on which the server is running). This InetAddress and the port number (the port number that the server used to create its DatagramSocket) are then used to create DatagramPacket destined for that Internet address and port number. Therefore the DatagramPacket will be delivered to the quote server.
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.

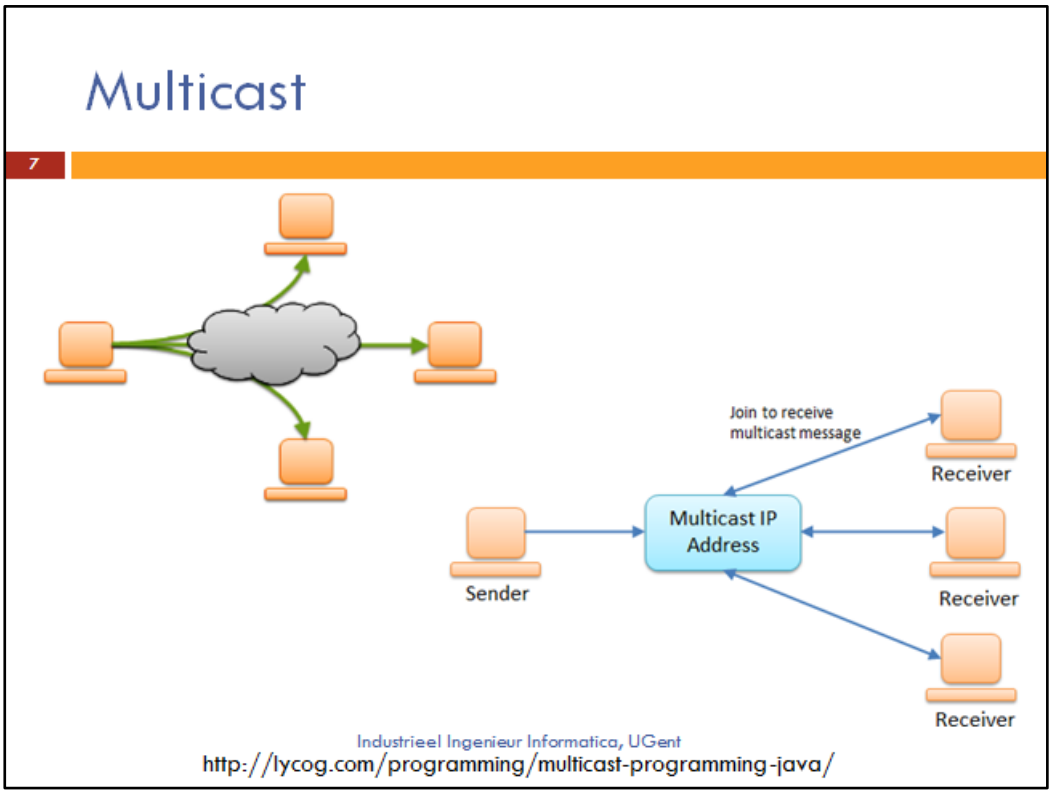
Note that the code creates a DatagramPacket with an empty byte array. The byte array is empty because this datagram packet is simply a request to the server for information. All the server needs to know to send a response--the address and port number to which reply--is automatically part of the packet.

Next, the client gets a response from the server and displays it:

```
packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);
String received = new String(packet.getData(), 0, packet.getLength());
System.out.println("Quote of the Moment: " + received);
```

To get a response from the server, the client creates a "receive" packet and uses the DatagramSocket receive method to receive the reply from the server. The receive method waits until a datagram packet destined for the client comes through the socket. Note that if the server's reply is somehow lost, the client will **wait forever** because of the no-guarantee policy of the datagram model. Normally, a client sets a **timer** so that it doesn't wait forever for a reply; if no reply arrives, the timer goes off and the client retransmits.

When the client receives a reply from the server, the client uses the getData method to retrieve that data from the packet. The client then converts the data to a string and displays it.



Multicast is a special feature of UDP protocol that enable programmer to send message to a group of receivers on a specific multicast IP address and port. Multicast has advantage in this scenario.

Let us say I want to send "Hello" message to 100 computers on my network. Perhaps, my first solution is to send the "Hello" message to each of them via UDP or TCP.
What a problem is this scenario? There are 3 generic problems:

- Consume a lot of processing power on sender as it needs to send to every receiver
- Bandwidth flooding
- The arrival time is not the same for every receiver

Seeing this problem, I propose my second solution by employing Multicast. Multicast runs over UDP protocol.

Multicast Characteristics
Multicast is using UDP under the hood. So sending and receiving data are much the same as UDP
The big noticeable from UDP

- Sender should address packages to an IP number in the range between **224.0.0.1** and **239.255.255.254**. Please see the full range of Multicast IP Address
- Receivers must join multicast group to receive packet

Several multicast sockets can be bound simultaneously to the same port (Contrary to UDP and TCP)
Multicast is viable for video conference, service discovery application, etc.

Multicast Server

- Maak `DatagramSocket`
- Lus
  - Bepaal poort en adres van de multicast
  - Maak `DatagramPacket` met bericht
  - Verstuur pakket
  - Slaap even
- Sluit socket

Multicast Server

9

```
DatagramSocket socket = new DatagramSocket(4445);
boolean moreQuotes = true;
while (moreQuotes) {
  String dString = …; // multicast bericht
  byte[] buf = dString.getBytes();

  InetAddress group = InetAddress.getByName("230.0.0.1");
  DatagramPacket packet
    = new DatagramPacket(buf, buf.length, group, 4446);
  socket.send(packet);

  try { sleep((long) (Math.random() * FIVE_SECONDS));
   } catch (InterruptedException e) { }
}
socket.close();
```

Industrieel Ingenieur Informatica, UGent

In addition to DatagramSocket, which lets programs send packets to one another, java.net includes a class called MulticastSocket. This kind of socket is used on the client-side to listen for packets that the server broadcasts to multiple clients.

Let's rewrite the quote server so that it broadcasts DatagramPackets to multiple recipients. Instead of sending quotes to a specific client that makes a request, the new server now needs to broadcast quotes at a regular interval. The client needs to be modified so that it passively listens for quotes and does so on a MulticastSocket.

This example is comprised of three classes which are modifications of the three classes from the previous example: MulticastServer, MulticastServerThread, and MulticastClient. This discussion highlights the interesting parts of these classes.

Here is the new version of the server's main program. The differences between this code and the previous version, QuoteServer, are shown in bold:

```
import java.io.*;
public class MulticastServer {
        public static void main(String[] args) throws IOException {
                new MulticastServerThread().start();
        }
}
```

Basically, the server got a new name and creates a MulticastServerThread instead of

a QuoteServerThread. Now let's look at the MulticastServerThread which contains the heart of the server. Here's its class declaration:
public class MulticastServerThread extends QuoteServerThread { // ... }

We've made this class a subclass of QuoteServerThread so that it can use the constructor, and inherit some member variable and the getNextQuote method. Recall that QuoteServerThread creates a DatagramSocket bound to port 4445 and opens the quote file. The DatagramSocket's port number doesn't actually matter in this example because the client never send anything to the server.
The only method explicitly implemented in MulticastServerThread is its run method. The differences between this run method and the one in QuoteServerThread are shown in bold:

```
public void run() {
        while (moreQuotes) {
                try {
                        byte[] buf = new byte[256];
                        // don't wait for request...just send a quote
                        String dString = null;
                        if (in == null) dString = new Date().toString();
                        dString = getNextQuote();
                        buf = dString.getBytes();
                        InetAddress group = InetAddress.getByName("203.0.113.0");
                        DatagramPacket packet; packet = new DatagramPacket(buf,
                        buf.length, group, 4446);
                        socket.send(packet);
                        try {
                                sleep((long)Math.random() * FIVE_SECONDS);
                        } catch (InterruptedException e) { }
                } catch (IOException e) {
                                e.printStackTrace();
                                moreQuotes = false;
                }
        }
        socket.close();
}
```

The interesting change is how the DatagramPacket is constructed, in particular, the InetAddress and port used to construct the DatagramPacket. Recall that the previous example retrieved the InetAddress and port number from the packet sent to the server from the client. This was because the server needed to reply directly to the client. Now, the server needs to address multiple clients. So this time both the InetAddress and the port number are hard-coded.
The hard-coded port number is 4446 (the client must have a MulticastSocket bound to this port). The hard-coded InetAddress of the DatagramPacket is "203.0.113.0" and is a group identifier (rather than the Internet address of the machine on which a single client is running). This particular address was arbitrarily chosen from the

reserved for this purpose.
Created in this way, the DatagramPacket is destined for all clients listening to port number 4446 who are member of the "203.0.113.0" group.

Multicast Client

- Bepaal poort
- Maak `MulticastSocket`
- Bepaal multicastadres
- Sluit aan bij multicastgroep
- Maak lege `DatagramPacket`
- Wacht op bericht
- Verlaat multicastgroep
- Sluit socket

Multicast Client

```
11
try (MulticastSocket socket = new MulticastSocket(4446)) {
  InetAddress address = InetAddress.getByName("230.0.0.1");
  socket.joinGroup(address);

  DatagramPacket packet;
  for (int i = 0; i < 5; i++) { // get a few quotes
    byte[] buf = new byte[256];
    packet = new DatagramPacket(buf, buf.length);
    socket.receive(packet);

    String received = new String(packet.getData(), 0, packet.getLength()
    System.out.println("Quote of the Moment: " + received);
  }
  socket.leaveGroup(address);
} catch (…) { … }
```


The client needs to be modified so that it passively listens for quotes and does so on a MulticastSocket.

To listen to port number 4446, the new client program just created its MulticastSocket with that port number. To become a member of the "203.0.113.0" group, the client calls the MulticastSocket's joinGroup method with the InetAddress that identifies the group. Now, the client is set up to receive DatagramPackets destined for the port and group specified. Here's the relevant code from the new client program (which was also rewritten to passively receive quotes rather than actively request them). The bold statements are the ones that interact with the MulticastSocket:

MulticastSocket socket = new MulticastSocket(4446);
InetAddress group = InetAddress.getByName("203.0.113.0");
socket.joinGroup(group);
DatagramPacket packet;
for (int i = 0; i < 5; i++) {
    byte[] buf = new byte[256];
    packet = new DatagramPacket(buf, buf.length);
    socket.receive(packet);
    String received = new String(packet.getData());
    System.out.println("Quote of the Moment: " + received);
}
socket.leaveGroup(group);

```
socket.close();
```

Notice that the server uses a DatagramSocket to broadcast packet received by the client over a MulticastSocket. Alternatively, it could have used a MulticastSocket. The socket used by the server to send the DatagramPacket is not important. What's important when broadcasting packets is the addressing information contained in the DatagramPacket, and the socket used by the client to listen for it.

MulticastSocket extends DatagramSocket.

DatagramChannel - server

12

```
// maak channel
DatagramChannel in = DatagramChannel.open();
// verbind met poort
in.socket().bind(new InetSocketAddress(PORT));
// maak buffer
ByteBuffer bufIn = ByteBuffer.allocate(48);
bufIn.clear(); // bij volgende request
// ontvang request
in.receive(bufIn);
// lees buffer
bufIn.flip();
```

Industrieel Ingenieur Informatica, UGent

A Java NIO DatagramChannel is a channel that can send and receive UDP packets. Since UDP is a connection-less network protocol, you cannot just by default read and write to a DatagramChannel like you do from other channels. Instead you send and receive packets of data.

Opening a DatagramChannel

Here is how you open a DatagramChannel:

```
DatagramChannel channel = DatagramChannel.open();
channel.socket().bind(new InetSocketAddress(9999));
```

This example opens a DatagramChannel which can receive packets on UDP port 9999.

Receiving Data

You receive data from a DatagramChannel by calling its receive() method, like this:

```
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
channel.receive(buf);
```

The receive() method will copy the content of a received packet of data into the given Buffer. If the received packet contains more data than the Buffer can contain, the remaining data is discarded silently.

DatagramChannel - client

13

```
try (DatagramChannel channel = DatagramChannel.open()) {
  ByteBuffer buf = ByteBuffer.allocate(48);
  for (int i = 0; i < 10; i++) {
    buf.clear();
    String newData = "logbericht " + System.currentTimeMillis();
    buf.put(newData.getBytes());
    buf.flip();

    int bytesSent
      = channel.send(buf, new InetSocketAddress("localhost", 4445));
    System.out.println(bytesSent + " verzonden");
  }
}
```

Industrieel Ingenieur Informatica, UGent

A Java NIO DatagramChannel is a channel that can send and receive UDP packets. Since UDP is a connection-less network protocol, you cannot just by default read and write to a DatagramChannel like you do from other channels. Instead you send and receive packets of data.

Opening a DatagramChannel

Here is how you open a DatagramChannel:

```
DatagramChannel channel = DatagramChannel.open();
```

Sending Data

You can send data via a DatagramChannel by calling its send() method, like this:

```
String newData = "New String to write to file..." + System.currentTimeMillis();
ByteBuffer buf = ByteBuffer.allocate(48);
buf.clear();
buf.put(newData.getBytes());
buf.flip();
int bytesSent = channel.send(buf, new InetSocketAddress("jenkov.com",
80));
```

This example sends the string to the "jenkov.com" server on UDP port 80. Nothing is listening on that port though, so nothing will happen. You will not be notified of whether the send packet was received or not, since UDP does not make any guarantees about delivery of data.

Connecting to a Specific Address

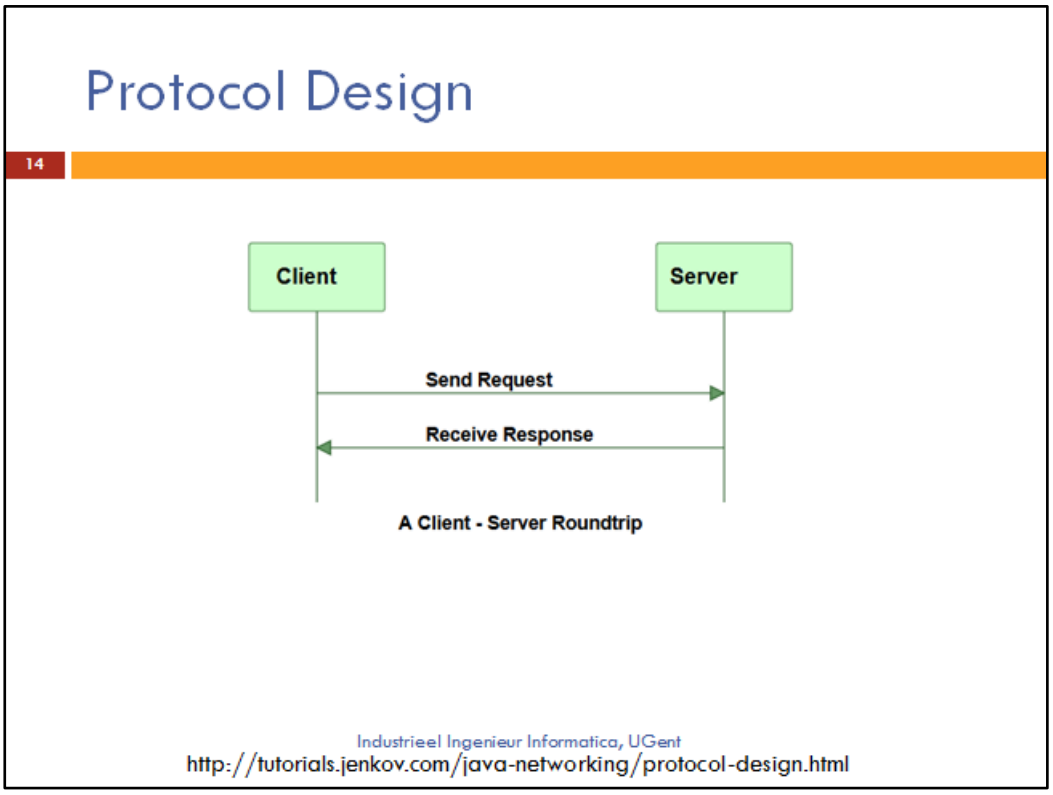
It is possible to "connect" a DatagramChannel to a specific address on the network. Since UDP is connection-less, this way of connecting to an address does not create a real connection, like with a TCP channel. Rather, it locks your DatagramChannel so you can only send and receive data packets from one specific address.

Here is an example:

```
channel.connect(new InetSocketAddress("jenkov.com", 80));
```

When connected you can also use the read() and write() method, as if you were using a traditional channel. You just don't have any guarantees about delivery of the sent data. Here are a few examples:

```
int bytesRead = channel.read(buf);
int bytesWritten = channel.write(buf);
```



If you are designing a client-server system you may also have to design a communication protocol between the client and the server. Of course, sometimes this protocol is already have been decided for you, e.g. HTTP, XML-RPC (XML over HTTP), or SOAP (also XML over HTTP). But once in a while the protocol decision is open, so let's look at a few issued you may want to think about when designing your client - server protocol:

Client - Server Roundtrips
When a client and server communicates to perform some operation they exchange information. For instance, the client will ask for a service to be performed, and the server will attempt to perform it, and send back a response telling the client of the result. Such an exchange of information between the client and server is called a **roundtrip**.

When a computer (client or server) sends data to another computer over the internet it takes some time from the time the data is sent, to the data is received at the other end. This is the time it takes the data to travel over the internet. This time is called **latency**.

The more roundtrips you have in your protocol, the slower the protocol becomes, especially if latency is high. The HTTP protocol consists of only a single request and a single response to perform its service. A single roundtrip in other words. The SMTP protocol on the other hand, consists of several roundtrips between the client and

the server before an email is sent.

The only reason to break your protocol up into multiple roundtrips is, if you have a large amount of data to send from the client to the server. You have two options in this case:

Send the header information in a separate roundtrip.
Break the message body up into smaller chunks.

Sending the header in a separate roundtrip (the first) can be smart if the server can do some initial pre-validation of e.g. header information. If that header information is invalid, sending the large body of data would have been a waste anyways.

If the network connection fails while you are transfering a large amount of data, you may have to resend all that data from scratch. By breaking the data up into smaller chunks you only have to resend the chunks from the chunk where the network connection failed and onwards. The successfully transfered chunks do not have be resent.

Demarcating the End of Requests and Responses

If your protocol allows multiple requests to be send over the same connection, you need some way for the server to know when one request ends, and a new begins. The client also needs to know when one response ends, and another begins.

You have two options for demarcating the end of a request:

Send the length in bytes of the request in the beginning of the request.
Send an end-of-request marker after the request data.

HTTP uses the first mechanism. In one of the request headers the "Content-Length" is sent. This header tells how many bytes after the headers that belongs to the request.
The advantage of this model is that you don't have the overhead of the end-of-request marker. Nor do you have to encode the body of the data to avoid the data looking like the end-of-request marker.

The disadvantage of the first method is that the sender must know how many bytes are transfered before the data is transfered. If the data is generated dynamically you will first have to buffer all the data before sending it, to count the number of bytes.

By using an end-of-request marker you don't have to know how many bytes you are sending. You just need to send an end-of-request marker at the end of the data. You do, however, have to make sure that the data sent does not contain any data that can be mistaken for the end-of-request marker. Here one way to do that:

Lets say the end-of-request marker is the byte value 255. Of course the data can contain the value 255 too. So, for each byte in the data that contains the value 255 you add an extra byte, also with the value 255. The end-of-request marker is

changed from the byte value 255 to 255 followed by the value 0. Here are the encodings summarized:

255 in data --> 255, 255
end-of-request --> 255, 0

The sequence 255, 0 can never occur in the data, since you are changing all 255's to 255,255. And, a 255,255,0 will not be mistaken for a 255,0. The first 255's will be interpreted together, and the last 0 by itself.

Penetrating Firewalls

Most firewalls block all other traffic than the HTTP protocol. Therefore it can be a good idea to layer your protocol on top of HTTP, like XML-RPC, SOAP and REST does. To layer your protocol on top of HTTP you send your data forth and back between client and server inside HTTP requests and responses. Remember, an HTTP request and response can contain more than just text or HTML. You can send binary data in there too.

The only thing that can be a little weird by layering your request on top of the HTTP protocol is that an HTTP request **must** contain a "Host" header field. If you are designing a P2P protocol on top of HTTP, your peers most likely won't be running multiple "Hosts". This required header field is in that situation an unnecessary overhead (but a small one).