
Writing UDP Server/Client Applications

Since TCP has had its moment of glory, it is time for UDP to get our attention. UDP is built on top of IP, and has a couple of important characteristics. For one, the packet sizes are limited to the amount that can be contained in a single IP packet—at most 65507 bytes; this is the 65535-byte IP packet size minus the minimum IP header of 20 bytes, and minus the 8-byte UDP header. Additionally, each packet is an individual, and is handled separately (no packet is aware of other packets). Moreover, the packets can arrive in any order, and some of them can be lost without the sender being informed, or they can arrive faster or slower than they can be processed—there's no guarantee of delivering/receiving data in a particular sequence and no guarantee that the delivered data will be received.

Since the sender can't track the packets' routes, each packet encapsulates the remote IP address and the port. If TCP is like a telephone, UDP is like a letter. The sender writes the receiver address (remote IP and port) and sender address (local IP and port) on the envelope (UDP packet), puts the letter (data to be sent) into the envelope, and sends the letter. He doesn't know if the letter will arrive to the receiver or not. Moreover, a more recent letter can arrive faster than and old one, and a letter might never arrive at all—the letters are not aware of one another. Keep in mind that TCP is for high-reliability data transmissions while UDP is for low-overhead transmissions. Typically, use UDP in applications in which reliability is not critical but speed is. UDP is good for sending messages from one system to another when the order isn't important and you don't need all of the messages to get to the other machine.

In the next sections, we will write a single-thread blocking client/server application based on UDP. We'll start with the server side.

Writing a UDP Server

To aid your understanding, we will split the developing process into discrete steps and bring to the front the features of NIO.2 meant to increase performance and ease of development. Again, we will write an echo server and a client that sends some text to it and receives it back.

Creating a Server Datagram–Oriented Socket Channel

The entire process of writing a client/server UDP application involves the `java.nio.channels.DatagramChannel` class, which represents a thread-safe selectable channel for datagram-oriented sockets. Therefore, we'll start our server by creating a new `DatagramChannel`, which can be accomplished by calling the NIO.2 `DatagramChannel.open()` method. This method gets a parameter known as a *protocol family* parameter, which is actually a `java.net.ProtocolFamily` object. This interface is new in NIO.2, and it represents a family of communication protocols—currently it has an implementation as `java.net.StandardProtocolFamily` and defines two enum constants:

- `StandardProtocolFamily.INET` : IP version 4 (IPv4)
- `StandardProtocolFamily.INET6` : IP version 6 (IPv6)

So, we can create a server datagram–oriented socket for IPv4 like this:

```
DatagramChannel datagramChannel = DatagramChannel.open(StandardProtocolFamily.INET);
```

The old NIO no-argument `DatagramChannel.open()` method is still available and can be used since it is not deprecated. But in this case, the `ProtocolFamily` of the channel's socket is platform (configuration) dependent and therefore unspecified.

You can check if a datagram-oriented socket channel is already open or has been successfully opened by calling the `DatagramChannel.isOpen()` method, which returns the corresponding `Boolean` value:

```
if (datagramChannel.isOpen()) {
   ...
}
```

A client datagram–oriented socket channel can be created and checked in the same manner.

Setting Datagram-Oriented Socket Channel Options

Datagram-oriented socket channels support the following options (although you can use the default values in most cases): `SO_REUSEADDR`, `SO_BROADCAST`, `IP_MULTICAST_LOOP`, `SO_SNDBUF`, `IP_MULTICAST_TTL`, `IP_TOS`, `IP_MULTICAST_IF`, and `SO_RCVBUF`. As an example, we can set the input and output buffers used by the networking implementation as follows:

```
datagramChannel.setOption(StandardSocketOptions.SO_RCVBUF, 4 * 1024);
datagramChannel.setOption(StandardSocketOptions.SO_SNDBUF, 4 * 1024);
```

Notice that you can find out the supported options for a datagram-oriented socket channel by calling the inherited method `supportedOptions()`:

```
Set<SocketOption<?>> options = datagramChannel.supportedOptions();
```

```
for(SocketOption<?> option : options) System.out.println(option);
```

Binding the Datagram-Oriented Socket Channel

At this point we can bind the channel's socket to a local address and configure the socket to listen for connections. For this we call the new `DatagramChannel.bind()` method (this method was introduced earlier in the "NetworkChannel Overview" section). Our server will wait for an incoming connection on localhost (127.0.0.1), port 5555 (arbitrarily chosen):

```
final int LOCAL_PORT = 5555;
final String LOCAL_IP = "127.0.0.1";
datagramChannel.bind(new InetSocketAddress(LOCAL_IP, LOCAL_PORT));
```

The wildcard address can also be used:

```
datagramChannel.bind(new InetSocketAddress(LOCAL_PORT));
```

The local address can also be automatically assigned if we pass null to the `bind()` method. You can also find out the bound local address by calling the `ServerSocketChannel.getLocalAddress()` method, which is inherited from the `NetworkChannel` interface. This returns null if the server socket channel has not been bound yet.

```
System.out.println(serverSocketChannel.getLocalAddress());
```

Transmitting Data Packets

At this point our server is ready to receive and send packets. Since UDP is a connectionless network protocol, you cannot just by default read and write to a `DatagramChannel` like you do from other channels—later, you will see how to set up a connection over UDP. Instead, you send and receive packets of data using the `DatagramChannel.send()` and `DatagramChannel.receive()` methods.

When you send a packet, you pass to the `send()` method a `ByteBuffer` that contains the precious data and the remote address (of the server or client, depending who is sending). Here's how this works according to the official documentation (see `http://download.oracle.com/javase/7/docs/api/)` :

> *If this channel is in non-blocking mode and there is sufficient room in the underlying output buffer, or if this channel is in blocking mode and sufficient room becomes available, then the remaining bytes in the given buffer are transmitted as a single datagram to the given target address. This method may be invoked at any time. If another thread has already initiated a write operation upon this channel, however, then an invocation of this method will block until the first operation is complete. If this channel's socket is not bound then this method will first cause the socket to be bound to an address that is assigned automatically, as if by invoking the `bind()` method with a parameter of `null` .*

The method will return the number of bytes sent.

When you receive a packet, you pass to the `receive()` method the buffer (`ByteBuffer`) into which the datagram is to be transferred. Again, here's how it works according to the documentation (see `http://download.oracle.com/javase/7/docs/api/`):

> *If a datagram is immediately available, or if this channel is in blocking mode and one eventually becomes available, then the datagram is copied into the given byte buffer and its source address is returned. If this channel is in non-blocking mode and a datagram is not immediately available then this method immediately returns null. This method may be invoked at any time. If another thread has already initiated a read operation upon this channel, however, then an invocation of this method will block until the first operation is complete. If this channel's socket is not bound then this method will first cause the socket to be bound to an address that is assigned automatically, as if by invoking the `bind()` method with a parameter of `null` .*

The method will return the datagram's source address, or `null` if this channel is in non-blocking mode and no datagram is immediately available. The remote address can be used to find out where to send an answer packet.

In addition, you can find out the remote address by calling the `DatagramChannel.getRemoteAddress()` method. This method is new in Java 7 (NIO.2), and it returns the remote address to which this channel's socket is connected—keep in mind that for a UDP connectionless case, this method returns `null` :

```
System.out.println("Connected to: " + datagramChannel.getRemoteAddress());
```

Our datagram echo server will listen for incoming packets in an infinite loop, in blocking mode (by default), and when a packet arrives, it will extract from it the remote address and data. The data is sent back based on the remote address:

```
final int MAX_PACKET_SIZE = 65507;
ByteBuffer echoText = ByteBuffer.allocateDirect(MAX_PACKET_SIZE);
...
while (true) {

        SocketAddress clientAddress = datagramChannel.receive(echoText);
```

```
        echoText.flip();
        System.out.println("I have received " + echoText.limit() + " bytes from " +
                                        clientAddress.toString() + "! Sending them back
...");
        datagramChannel.send(echoText, clientAddress);
        echoText.clear();
}
```

Closing the Datagram Channel

When a datagram channel becomes useless, it must be closed. For this, you can call the `DatagramChannel.close()` method:

```
datagramChannel.close();
```

Again, the Java 7 try-with-resources features can be used for automatic closing.

Putting All Together into the Server

Now we have everything we need for creating our server. Putting all of the previous information together will provide us the following server:

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.net.StandardProtocolFamily;
import java.nio.channels.DatagramChannel;
import java.net.StandardSocketOptions;
import java.nio.ByteBuffer;
import java.nio.channels.ClosedChannelException;

public class Main {

public static void main(String[] args) {
  final int LOCAL_PORT = 5555;
  final String LOCAL_IP = "127.0.0.1";  //modify this to your local IP
  final int MAX_PACKET_SIZE = 65507;

  ByteBuffer echoText = ByteBuffer.allocateDirect(MAX_PACKET_SIZE);

  //create a new datagram channel
  try (DatagramChannel datagramChannel = DatagramChannel.open(StandardProtocolFamily.INET))
{

        //check if the channel was successfully opened
        if (datagramChannel.isOpen()) {

            System.out.println("Echo server was successfully opened!");
            //set some options
            datagramChannel.setOption(StandardSocketOptions.SO_RCVBUF, 4 * 1024);
            datagramChannel.setOption(StandardSocketOptions.SO_SNDBUF, 4 * 1024);
            //bind the channel to local address
            datagramChannel.bind(new InetSocketAddress(LOCAL_IP, LOCAL_PORT));
            System.out.println("Echo server was binded
on:"+datagramChannel.getLocalAddress());
            System.out.println("Echo server is ready to echo ...");
```

```
            //transmitting data packets
            while (true) {

                  SocketAddress clientAddress = datagramChannel.receive(echoText);

                   echoText.flip();
                  System.out.println("I have received " + echoText.limit() + " bytes from "
+
                                          clientAddress.toString() + "! Sending them back
...");
                  datagramChannel.send(echoText, clientAddress);
                  echoText.clear();
            }
       } else {
         System.out.println("The channel cannot be opened!");
       }
  } catch (Exception ex) {
            if (ex instanceof ClosedChannelException) {
                System.err.println("The channel was unexpected closed ...");
            }
            if (ex instanceof SecurityException) {
                System.err.println("A security exception occured ...");
            }
            if (ex instanceof IOException) {
                System.err.println("An I/O error occured ...");
            }

            System.err.println("\n" + ex);
  }
 }
}
```

Writing a Connectionless UDP Client

Writing a connectionless UDP client is similar to writing a UDP server. After creating a new `DatagramChannel` in the same manner as shown previously, and setting whatever options you need, you can start sending and receiving data packets. A client datagram–oriented socket channel doesn't have to be bound to a local address, since the server will extract the IP address and port from each received data packet—in other words, it knows where the client lives. Moreover, if this channel's socket is not bound, then the `send()` and `receive()` methods will first cause the socket (client or server) to be bound to an address that is assigned automatically, as if by invoking the `bind()` method with a parameter of `null`. But keep in mind that if the server side is automatically bound (not explicitly), then the client should be aware of the chosen address (or more precisely, of the chosen IP address and port). The opposite is also true if the server sends the first data packet.

Our client knows that the server lives at the address 127.0.0.1, port 5555; therefore, it sends the first data packet and receives the answer from it. Here it is the code:

```
 import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.StandardProtocolFamily;
import java.nio.channels.DatagramChannel;
import java.net.StandardSocketOptions;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.ClosedChannelException;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
```

```
public class Main {

public static void main(String[] args) throws IOException {

  final int REMOTE_PORT = 5555;
  final String REMOTE_IP = "127.0.0.1"; //modify this accordingly if you want to test
remote
  final int MAX_PACKET_SIZE = 65507;

  CharBuffer charBuffer = null;
  Charset charset = Charset.defaultCharset();
  CharsetDecoder decoder = charset.newDecoder();
  ByteBuffer textToEcho = ByteBuffer.wrap("Echo this: I'm a big and ugly
server!".getBytes());
  ByteBuffer echoedText = ByteBuffer.allocateDirect(MAX_PACKET_SIZE);

  //create a new datagram channel
  try (DatagramChannel datagramChannel = DatagramChannel.open(StandardProtocolFamily.INET))
{

       //check if the channel was successfully opened
       if (datagramChannel.isOpen()) {

           //set some options
           datagramChannel.setOption(StandardSocketOptions.SO_RCVBUF, 4 * 1024);
           datagramChannel.setOption(StandardSocketOptions.SO_SNDBUF, 4 * 1024);

           //transmitting data packets
           int sent = datagramChannel.send(textToEcho,
                                           new InetSocketAddress(REMOTE_IP, REMOTE_PORT));
           System.out.println("I have successfully sent "+sent+ " bytes to the Echo
Server!");

           datagramChannel.receive(echoedText);

           echoedText.flip();
           charBuffer = decoder.decode(echoedText);
           System.out.println(charBuffer.toString());
           echoedText.clear();

       } else {
         System.out.println("The channel cannot be opened!");
        }
  } catch (Exception ex) {
    if (ex instanceof ClosedChannelException) {
        System.err.println("The channel was unexpected closed ...");
    }
    if (ex instanceof SecurityException) {
        System.err.println("A security exception occured ...");
```

```
    }
    if (ex instanceof IOException) {
        System.err.println("An I/O error occured ...");
    }

    System.err.println("\n" + ex);
  }
 }
}
```

Testing the UDP Connectionless Echo Application

Testing the application is a simple task. First, start the server and wait until you see the following message:

```
 Echo server was successfully opened!
Echo server was binded on: /127.0.0.1:5555
Echo server is ready to echo ...
```

Then start the client and check out the output. Here is some possible output from the UDP server:

```
 Echo server was successfully opened!

Echo server was binded on: /127.0.0.1:5555

Echo server is ready to echo ...

I have received 37 bytes from /127.0.0.1:49155! Sending them back ...
```

And here is some possible UDP client output:

```
 I have successfully sent 37 bytes to the Echo Server!

Echo this: I'm a big and ugly server!
```

Caution Don't forget to manually stop the UDP server after finishing tests!

Writing a Connected UDP Client

If you want to use the `DatagramChannel.read()` and `DatagramChannel.write()` methods (based on `ByteBuffer` s), rather then `send()` and `receive()` , you need to write a connected UDP client. In a connected-client scenario, the channel's socket is configured so that it only receives/sends datagrams from/to the given remote *peer* address. After the connection is established, data packets may not be received/sent from/to any other address. A datagram-oriented socket remains connected until it is explicitly disconnected or until it is closed.

This type of client must explicitly call the `DatagramChannel.connect()` method and pass to it the server-side remote address, as follows:

```
 final int REMOTE_PORT = 5555;
final String REMOTE_IP = "127.0.0.1";
datagramChannel.connect(new InetSocketAddress(REMOTE_IP, REMOTE_PORT));
```

Notice that, unlike the `SocketChannel.connect()` method, this method does not actually send/receive any packets across the network, since UDP is a connectionless protocol—this method returns pretty quickly, and does not block the application in a concrete sense. There is no need here for a `finishConnect()` or `isConnectionPending()` method. This method may be invoked at any time, because it will not affect read/write operations that are already in progress at the moment that it is invoked. If this channel's socket is not bound, then this method will first cause the socket to be bound to

an address that is assigned automatically, as if invoking the `bind()` method with a parameter of `null`.

You can check out a connection status by calling the `DatagramChannel.isConnected()` method. A corresponding `boolean` value will be returned (`true` if this channel's socket is open and connected):

```
if (datagramChannel.isConnected()) {
   ...
}
```

The following application is a UDP connected client for our UDP echo server. It connects to the remote address and uses `read()` / `write()` methods for transmitting data:

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.StandardProtocolFamily;
import java.nio.channels.DatagramChannel;
import java.net.StandardSocketOptions;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.ClosedChannelException;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;

public class Main {

public static void main(String[] args) throws IOException {

  final int REMOTE_PORT = 5555;
  final String REMOTE_IP = "127.0.0.1"; //modify this accordingly if you want to test
remote
  final int MAX_PACKET_SIZE = 65507;

  CharBuffer charBuffer = null;
  Charset charset = Charset.defaultCharset();
    CharsetDecoder decoder = charset.newDecoder();
  ByteBuffer textToEcho = ByteBuffer.wrap("Echo this: I'm a big and ugly
server!".getBytes());
  ByteBuffer echoedText = ByteBuffer.allocateDirect(MAX_PACKET_SIZE);

  //create a new datagram channel
  try (DatagramChannel datagramChannel = DatagramChannel.open(StandardProtocolFamily.INET))
{

       //set some options
       datagramChannel.setOption(StandardSocketOptions.SO_RCVBUF, 4 * 1024);
       datagramChannel.setOption(StandardSocketOptions.SO_SNDBUF, 4 * 1024);

       //check if the channel was successfully opened
       if (datagramChannel.isOpen()) {

           //connect to remote address
           datagramChannel.connect(new InetSocketAddress(REMOTE_IP, REMOTE_PORT));
```

```
            //check if the channel was successfully connected
            if (datagramChannel.isConnected()) {

                //transmitting data packets
                int sent = datagramChannel.write(textToEcho);
                System.out.println("I have successfully sent "+sent
                                                          +" bytes to the Echo
    Server!");

                datagramChannel.read(echoedText);

                echoedText.flip();
                charBuffer = decoder.decode(echoedText);
                System.out.println(charBuffer.toString());
                echoedText.clear();

            } else {
              System.out.println("The channel cannot be connected!");
            }
          } else {
            System.out.println("The channel cannot be opened!");
          }
      } catch (Exception ex) {
        if (ex instanceof ClosedChannelException) {
            System.err.println("The channel was unexpected closed ...");
        }
        if (ex instanceof SecurityException) {
            System.err.println("A security exception occured ...");
        }
        if (ex instanceof IOException) {
            System.err.println("An I/O error occured ...");
        }

        System.err.println("\n" + ex);
      }
     }
    }
```

The well-known read()/write() methods are available in DatagramChannel:

- Reading sequence of bytes from this channel into the given buffer. These methods return the number of bytes read (it can be zero) or –1 if the channel has reached the end of the stream:

```
public abstract int read(ByteBuffer dst) throws IOException
public final long read(ByteBuffer[] dsts) throws IOException
public abstract long read(ByteBuffer[] dsts, int offset, int length) throws
IOException
```

- Writing a sequence of bytes to this channel from the given buffer. These methods return the number of bytes written; it can be zero:

```
public abstract int write(ByteBuffer src) throws IOException
public final long write(ByteBuffer[] srcs) throws IOException
public abstract long write(ByteBuffer[] srcs, int offset, int length) throws
IOException
```

Testing the UDP Connected Echo Application

Testing the application is a simple task. First, start the server and wait until you see this message:

```
Echo server was successfully opened!

Echo server was binded on: /127.0.0.1:5555

Echo server is ready to echo ...
```

Then start the client and check out the output. The UDP server output is shown here:

```
Echo server was successfully opened!

Echo server was binded on: /127.0.0.1:5555

Echo server is ready to echo ...

I have received 37 bytes from /127.0.0.1:57374! Sending them back ...
```

Here's the UDP client output:

```
I have successfully sent 37 bytes to the Echo Server!

Echo this: I'm a big and ugly server!
```

Multicasting

You are probably already familiar with the term *multicasting*. But, if you are not, let's have a short overview of this concept. Without academic descriptions and definitions, think of multicasting as the Internet's version of broadcasting. For example, a television station broadcasts its signal from one source, but the signal can reach everyone that lives in the signal area—only the ones that do not have the right equipment or refuse to catch the signal will fail to receive the transmission.

In the computer world, the TV station can be translated to a main node, or machine, that spreads datagrams to a group of destination hosts. This is possible thanks to the *multicast transport service*, which sends datagrams from a source to multiple receivers in a single call—this opposed to the *unicast transport service*, which is specific to high-level network protocols that are based on point-to-point connections and requires a replicated unicast for sending the same data to multiple points (actually, it sends a copy of the data to each point).

Multicasting introduces the notion of a group for representing the receivers of the datagrams. A group is identified by a class D IP address (a multicast group IPv4 address is between 224.0.0.1 and 239.255.255.255). When a new receiver (client) wants to join a multicast group, it needs to connect to the group through the corresponding IP address and listen for the incoming datagrams.

Many real-life cases can be programmed based on multicasting, such as online conferencing, news distribution, advertising, e-mail groups, and data-sharing management.

Next, we'll discuss NIO.2's contribution to multicasting.

MulticastChannel Overview

NIO.2 comes with a new interface for mapping a network channel that supports IP multicasting. This is the `java.nio.channels.MulticastChannel` interface. At the API level, this is a subinterface of the `NetworkChannel` interface presented earlier in this chapter, and it is implemented by a single class: the `DatagramChannel` class.

Basically, it defines two `join()` methods and a `close()` method. Focusing on the `join()` methods, here it is a short overview:

- The first `join()` method is called by a client who wants to join a multicast group for receiving the incoming datagrams. We need to pass the IP address of the group and the network interface on which to join the group (you will see shortly how to check if your machine has a network interface capable of multicasting). If the indicated group is successfully joined, this method returns a `MembershipKey` instance. This is new in NIO.2, and it is a token representing the membership of an IP multicast group (see the next section).

```
MembershipKey join(InetAddress g, NetworkInterface i) throws IOException
```

- The second `join()` method is also used for joining a multicast group. In this case, however, we indicate a source address from which group members can begin receiving datagrams. Membership is *cumulative*, which means that this method may be invoked again with the same group and interface for receiving datagrams sent by other source addresses to the group.

```
MembershipKey join(InetAddress g, NetworkInterface i, InetAddress s) throws
IOException
```

Note A multicast channel may join several multicast groups, including the same group on more than one interface.

The `close()` method is used to drop the membership (if any group was joined) and close the channel.

MembershipKey Overview

When you join a multicast group, you get a membership key that can be used to perform different kinds of actions inside that group. The most common are presented here:

- *Block/unblock*: You can block the sent datagrams from a specific source by calling the `block()` method and passing the source address. Moreover, you can unblock the blocked source by calling the `unblock()` method with the same address.
- *Get group*: You can get the source address of the multicast group for which this membership key was created by calling the no-argument `group()` method. This method returns an `InetAddress` object.
- *Get channel*: You can get the channel for which this membership key was created by calling the no-argument method `channel()`. This method returns a `MulticastChannel` object.
- *Get source address*: If the membership key is source specific (receives only datagrams from a specific source address), you can get the source address by calling the no-argument `sourceAddress()` method. This method returns an `InetAddress` object.
- *Get network interface*: You can get the network interface for which this membership key was created by calling the no-argument `networkInterface()` method. This method returns a `NetworkInterface` object.
- *Check validity*: You can check if a membership is valid by calling the `isValid()` method. This method returns a `boolean` value.
- *Drop*: You can drop membership (the channel will no longer receive any datagrams sent to the group) by calling the no-argument `drop()` method.

A membership key is valid when you create it and remains valid until the membership is dropped by using the `drop()` method or the channel is closed.

NetworkInterface Overview

The `NetworkInterface` class represents a *network interface*, which is made up of a name and a list of IP addresses assigned to this interface. It is used to identify the local interface to which a multicast group is joined. For example, the following code will return information about all the network interfaces found on your machine:

```
 import java.net.InetAddress;
import java.net.NetworkInterface;
import java.util.Enumeration;

public class Main {

public static void main(String argv[]) throws Exception {

  Enumeration enumInterfaces = NetworkInterface.getNetworkInterfaces();
  while (enumInterfaces.hasMoreElements()) {
     NetworkInterface net = (NetworkInterface) enumInterfaces.nextElement();
     System.out.println("Network Interface Display Name: " + net.getDisplayName());
     System.out.println(net.getDisplayName() + " is up and running ?" + net.isUp());
     System.out.println(net.getDisplayName()+" Supports Multicast:
"+net.supportsMulticast());
     System.out.println(net.getDisplayName() + " Name: " + net.getName());
     System.out.println(net.getDisplayName() + " Is Virtual:  " + net.isVirtual());
```

```
        System.out.println("IP addresses:");
        Enumeration enumIP = net.getInetAddresses();
        while (enumIP.hasMoreElements()) {
           InetAddress ip = (InetAddress) enumIP.nextElement();
           System.out.println("IP address:" + ip);
        }
     }
   }
}
```

This application will return all the network interfaces found on your machine, and for each one will render its *display name* (a human-readable `String` describing the network device) and *name* (the real name used to identify a network interface). Moreover, each network interface is checked to see if it supports multicast, if it is virtual (a subinterface), and if it is up and running.

Figure 8-4 shows a fragment of output on my machine. The framed interface is the one used for testing multicast applications—its name is eth3 and will be used later in the client/server multicast application for indicating this interface.

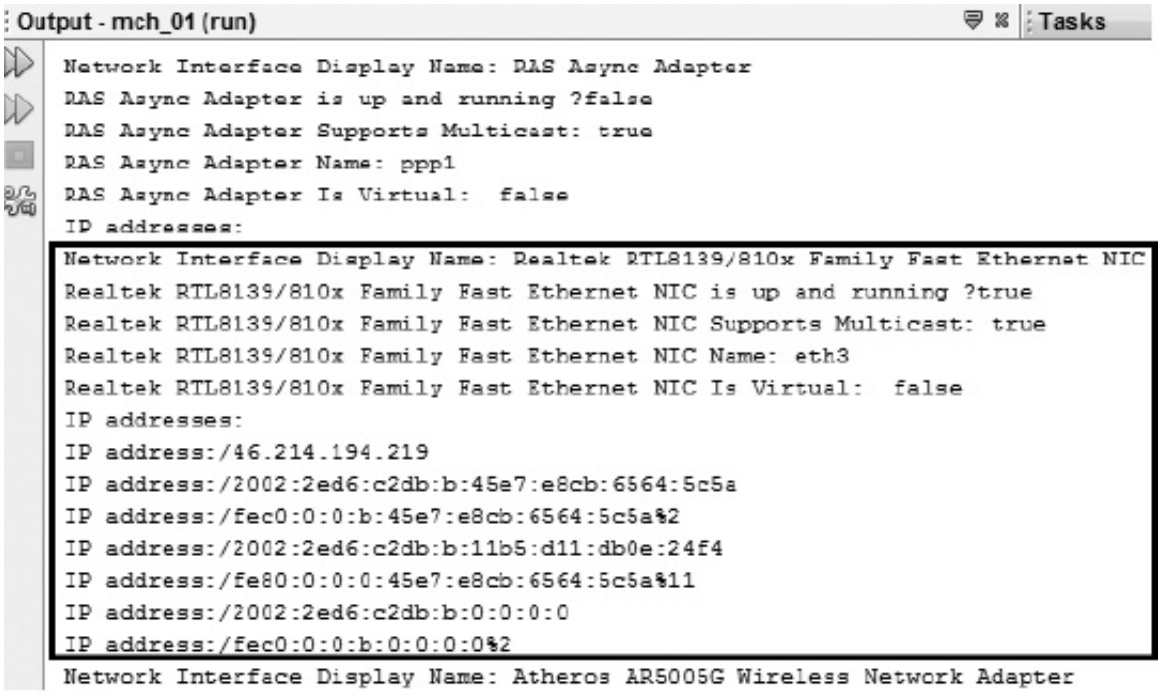

Figure 8-4. *Find out local interfaces.*

Writing a UDP Multicast Server

In this section, we will write a UDP multicast server that sends to the group datagrams containing the current date and time on the server. This will be repeated every 10 seconds. Now that we have some experience with writing UDP client/server applications, there is no need to repeat the entire process step by step. We'll just point out the main differences that transform a usual UDP client/server application into a UDP multicast client/server application.

We start the developing process by creating a new `DatagramChannel` object by calling the `open()` method. Next, we set two important options, `IP_MULTICAST_IF` and `SO_REUSEADDR`. The first one will indicate the network interface for IP multicast datagrams used in this case, and the second one should be enabled prior to binding the socket—this is required to allow multiple members of the group to bind to the same address:

```
NetworkInterface networkInterface = NetworkInterface.getByName("eth3");
...
datagramChannel.setOption(StandardSocketOptions.IP_MULTICAST_IF, networkInterface);
datagramChannel.setOption(StandardSocketOptions.SO_REUSEADDR, true);
```

Next, we bind the channel's socket to the local address by calling the `bind()` method:

```
final int DEFAULT_PORT = 5555;
datagramChannel.bind(new InetSocketAddress(DEFAULT_PORT));
```

Finally, we prepare the datagram-transmitting code. Since we send to the group the server date and time every 10 seconds, we need an infinite loop containing a sleep duration of 10 seconds and a call to the `send()` method. The multicast group IP address was arbitrarily chosen as 225.4.5.6, and it is mapped by an

`InetAddress` object:

```
final int DEFAULT_PORT = 5555;
final String GROUP = "225.4.5.6";
ByteBuffer datetime;
...
while (true) {

       //sleep for 10 seconds
       try {
           Thread.sleep(10000);
       } catch (InterruptedException ex) {}

       System.out.println("Sending data ...");
       datetime = ByteBuffer.wrap(new Date().toString().getBytes());
       datagramChannel.send(datetime, new
                            InetSocketAddress(InetAddress.getByName(GROUP), DEFAULT_PORT));
       datetime.flip();
}
```

Putting everything together will result in the following application:

```
import java.io.IOException;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.NetworkInterface;
import java.net.StandardProtocolFamily;
import java.nio.channels.DatagramChannel;
import java.net.StandardSocketOptions;
import java.nio.ByteBuffer;
import java.util.Date;

public class Main {

 public static void main(String[] args) {

  final int DEFAULT_PORT = 5555;
  final String GROUP = "225.4.5.6";
  ByteBuffer datetime;

  //create a new channel
  try (DatagramChannel datagramChannel = DatagramChannel.open(StandardProtocolFamily.INET))
{

       //check if the channel was successfully created
       if (datagramChannel.isOpen()) {

           //get the network interface used for multicast
           NetworkInterface networkInterface = NetworkInterface.getByName("eth3");

           //set some options
            datagramChannel.setOption(StandardSocketOptions.IP_MULTICAST_IF,
```

```
networkInterface);
            datagramChannel.setOption(StandardSocketOptions.SO_REUSEADDR,
true);

            //bind the channel to the local address
            datagramChannel.bind(new InetSocketAddress(DEFAULT_PORT));
            System.out.println("Date-time server is ready ... shortly I'll start sending
...");

            //transmitting datagrams
            while (true) {

                  //sleep for 10 seconds
                  try {
                      Thread.sleep(10000);
                  } catch (InterruptedException ex) {}

                  System.out.println("Sending data ...");
                  datetime = ByteBuffer.wrap(new Date().toString().getBytes());
                  datagramChannel.send(datetime, new
                              InetSocketAddress(InetAddress.getByName(GROUP),
DEFAULT_PORT));
                  datetime.flip();
            }
          } else {
            System.out.println("The channel cannot be opened!");
          }
    } catch (IOException ex) {
      System.err.println(ex);
    }
  }
}
```

Writing a UDP Multicast Client

The code for a UDP multicast client is almost the same as for a server, with a few differences. First, you may want to check if the remote address is actually a multicast address—this is possible by calling the `InetAddress.isMulticastAddress()` method, which returns a `boolean`. And second, since this is a client, it must join the group by calling one of the two `join()` methods. The datagram-transmitting code is adapted only for receiving datagrams from the UDP multicast server. The following application is a possible client implementation:

```
 import java.io.IOException;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.NetworkInterface;
import java.net.StandardProtocolFamily;
import java.nio.channels.DatagramChannel;
import java.net.StandardSocketOptions;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.MembershipKey;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
```

```
 public class Main {

public static void main(String[] args) {

 final int DEFAULT_PORT = 5555;
 final int MAX_PACKET_SIZE = 65507;
 final String GROUP = "225.4.5.6";

 CharBuffer charBuffer = null;
 Charset charset = Charset.defaultCharset();
 CharsetDecoder decoder = charset.newDecoder();
 ByteBuffer datetime = ByteBuffer.allocateDirect(MAX_PACKET_SIZE);

 //create a new channel
 try (DatagramChannel datagramChannel = DatagramChannel.open(StandardProtocolFamily.INET))
{

      InetAddress group = InetAddress.getByName(GROUP);
      //check if the group address is multicast
      if (group.isMulticastAddress()) {
          //check if the channel was successfully created
          if (datagramChannel.isOpen()) {
              //get the network interface used for multicast
              NetworkInterface networkInterface = NetworkInterface.getByName("eth3");

              //set some options
              datagramChannel.setOption(StandardSocketOptions.SO_REUSEADDR, true);
              //bind the channel to the local address
              datagramChannel.bind(new InetSocketAddress(DEFAULT_PORT));
              //join the multicast group and get ready to receive datagrams
              MembershipKey key = datagramChannel.join(group, networkInterface);

              //wait for datagrams
              while (true) {

                     if (key.isValid()) {

                         datagramChannel.receive(datetime);
                         datetime.flip();
                         charBuffer = decoder.decode(datetime);
                         System.out.println(charBuffer.toString());
                         datetime.clear();
                     } else {
                       break;
                     }
              }

          } else {
            System.out.println("The channel cannot be opened!");
          }
```

```
        } else {
          System.out.println("This is not  multicast address!");
         }

  } catch (IOException ex) {
    System.err.println(ex);
  }
 }
}
```

Blocking and Unblocking Datagrams

Sometimes joining multicast groups can bring to you undesired datagrams (the reasons are not relevant here). You can block receiving a datagram from a sender by calling the `MembershipKey.block()` method and passing to it the `InetAddress` of that sender. In addition, you can unblock the same sender, and start receiving datagrams from it again, by calling the `MembershipKey.unblock()` method and passing it the same `InetAddress`. Usually, you'll be in one of the following two scenarios:

- You have a list of senders' addresses that you'd like to join. Supposing that the addresses are stored in a `List`, you can loop it and join each address separately, as shown here:

```
 List<InetAddress> like = ...;
DatagramChannel datagramChannel =...;

if(!like.isEmpty()){
    for(InetAddress source: like){
        datagramChannel.join(group, network_interface, source);
    }
}
```

- You have a list of senders' addresses that you don't want to join. Supposing that the addresses are stored in a `List`, then you can loop it and block each address separately, as shown here:

```
 List<InetAddress> dislike = ...;
DatagramChannel datagramChannel =...;

MembershipKey key = datagramChannel.join(group, network_interface);

if(!dislike.isEmpty()){
   for(InetAddress source: dislike){
       key.block(source);
   }
}
```

Testing the UDP Multicast Application

Testing the application is a simple task. First, start the multicast server and wait until you see this message:

```
Date-time server is ready ... shortly I'll start sending ..
```

Then start the client and check out the output. Here is some example output for the UDP multicast server:

```
Date-time server is ready ... shortly I'll start sending ...
```

```
Sending data ...

Sending data ...

Sending data ...

Sending data ...

Sending data ...
```

Here is the UDP client output (the client is started after a few minutes):

```
 Sat Oct 08 09:40:09 GMT+02:00 2011

Sat Oct 08 09:40:19 GMT+02:00 2011
```

Performing some tests on this example will reveal some issues. When the server is started, it sends datagrams without being aware of whether any client is listening for those datagrams. Also, it is not aware of when clients join or leave the group. On the opposite side, the client starts receiving datagrams when it joins the group, but is not aware of whether the server stops sending because of any causes. If the server goes offline, the client is still waiting, and it will receive again when the server is online again and begins sending. It can be an interesting exercise to try solving these issues if your case requires more control. Also, you may want to experiment with threads, blocking /non-blocking modes, and connectionless/connected features to add more flexibly and performance to your multicasting applications.