Table of Contents

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Chapter 13. UDP Datagrams and Sockets

Previous chapters discussed network applications that use the TCP protocol. TCP is designed for reliable transmission of data. If data is lost or damaged in transmission, TCP ensures that the data is resent; if packets of data arrive out of order, TCP puts them back in the correct order; if the data is coming too fast for the connection, TCP throttles the speed back so that packets won't be lost. A program never needs to worry about receiving data that is out of order or incorrect. However, this reliability comes at a price. That price is speed. Establishing and tearing down TCP connections can take a fair amount of time, particularly for protocols such as HTTP, which tend to require many short transmissions.

The User Datagram Protocol (UDP) is an alternative protocol for sending data over IP that is very quick, but not reliable. That is, when you send UDP data, you have no way of knowing whether it arrived, much less whether different pieces of data arrived in the order in which you sent them. However, the pieces that do arrive generally arrive quickly.

13.1. The UDP Protocol

The obvious question to ask is why anyone would ever use an unreliable protocol. Surely, if you have data worth sending, you care about whether the data arrives correctly? Clearly, UDP isn't a good match for applications like FTP that require reliable transmission of data over potentially unreliable networks. However, there are many kinds of applications in which raw speed is more important than getting every bit right. For example, in real-time audio or video, lost or swapped packets of data simply appear as static. Static is tolerable, but awkward pauses in the audio stream, when TCP requests a retransmission or waits for a wayward packet to arrive, are unacceptable. In other applications, reliability tests can be implemented in the application layer. For example, if a client sends a short UDP request to a server, it may assume that the packet is lost if no response is returned within an established period of time; this is one way the Domain Name System (DNS) works. (DNS can also operate over TCP.) In fact, you could implement a reliable file transfer protocol using UDP, and many people have: Network File System (NFS), Trivial FTP (TFTP), and

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

FSP, a more distant relative of FTP, all use UDP. (The latest version of NFS can use either UDP or TCP.) In these protocols, the application is responsible for reliability; UDP doesn't take care of it. That is, the application must handle missing or out-of-order packets. This is a lot of work, but there's no reason it can't be done—although if you find yourself writing this code, think carefully about whether you might be better off with TCP.

The difference between TCP and UDP is often explained by analogy with the phone system and the post office. TCP is like the phone system. When you dial a number, the phone is answered and a connection is established between the two parties. As you talk, you know that the other party hears your words in the order in which you say them. If the phone is busy or no one answers, you find out right away. UDP, by contrast, is like the postal system. You send packets of mail to an address. Most of the letters arrive, but some may be lost on the way. The letters probably arrive in the order in which you sent them, but that's not guaranteed. The farther away you are from your recipient, the more likely it is that mail will be lost on the way or arrive out of order. If this is a problem, you can write sequential numbers on the envelopes, then ask the recipients to arrange them in the correct order and send you mail telling you which letters arrived so that you can resend any that didn't get there the first time. However, you and your correspondent need to agree on this protocol in advance. The post office will not do it for you.

Both the phone system and the post office have their uses. Although either one could be used for almost any communication, in some cases one is definitely superior to the other. The same is true of UDP and TCP. The last several chapters have all focused on TCP applications, which are more common than UDP applications. However, UDP also has its place; in this chapter, we'll look at what you can do with UDP in Java. If you want to go further, look at Chapter 14. Multicasting relies on UDP; a multicast socket is a fairly simple variation on a UDP socket.

Java's implementation of UDP is split into two classes: `DatagramPacket` and `DatagramSocket`. The `DatagramPacket` class stuffs bytes of data into UDP packets called *datagrams* and lets you unstuff datagrams that you receive. A `DatagramSocket` sends as well as receives UDP datagrams. To send data, you put the data in a `DatagramPacket` and send the packet using a `DatagramSocket`. To receive data, you receive a `DatagramPacket` object from a `DatagramSocket` and then read the contents of the packet. The sockets themselves are very simple creatures. In UDP, everything about a datagram, including the address to which it is directed, is included in the packet itself; the socket only needs to know the local port on which to listen or send.

This division of labor contrasts with the `Socket` and `ServerSocket` classes used by TCP. First, UDP doesn't have any notion of a unique connection between two hosts. One socket sends and receives all data directed to or from a port without any concern for whom the remote host is. A single `DatagramSocket` can send data to and receive data from many independent hosts. The socket isn't dedicated to a single connection, as it is in TCP. In fact, UDP doesn't have any concept of a connection between two hosts; it only knows about individual datagrams. Figuring out who

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

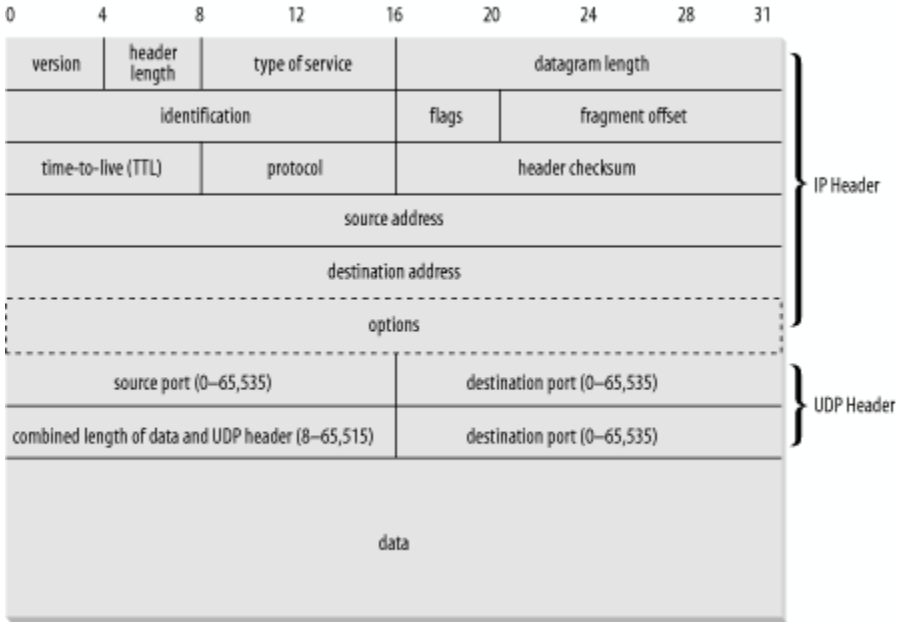
sent what data is the application's responsibility. Second, TCP sockets treat a network connection as a stream: you send and receive data with input and output streams that you get from the socket. UDP doesn't allow this; you always work with individual datagram packets. All the data you stuff into a single datagram is sent as a single packet and is either received or lost as a group. One packet is not necessarily related to the next. Given two packets, there is no way to determine which packet was sent first and which was sent second. Instead of the orderly queue of data that's necessary for a stream, datagrams try to crowd into the recipient as quickly as possible, like a crowd of people pushing their way onto a bus. And occasionally, if the bus is crowded enough, a few packets, like people, may not squeeze on and will be left waiting at the bus stop.

13.2. The DatagramPacket Class

UDP datagrams add very little to the IP datagrams they sit on top of. Figure 13-1 shows a typical UDP datagram. The UDP header adds only eight bytes to the IP header. The UDP header includes source and destination port numbers, the length of everything that follows the IP header, and an optional checksum. Since port numbers are given as 2-byte unsigned integers, 65,536 different possible UDP ports are available per host. These are distinct from the 65,536 different TCP ports per host. Since the length is also a 2-byte unsigned integer, the number of bytes in a datagram is limited to 65,536 minus the 8 bytes for the header. However, this is redundant with the datagram length field of the IP header, which limits datagrams to between 65,467 and 65,507 bytes. (The exact number depends on the size of the IP header.) The checksum field is optional and not used in or accessible from application layer programs. If the checksum for the data fails, the native network software silently discards the datagram; neither the sender nor the receiver is notified. UDP is an unreliable protocol, after all.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Figure 13-1. The structure of a UDP datagram



Although the theoretical maximum amount of data in a UDP datagram is 65,507 bytes, in practice there is almost always much less. On many platforms, the actual limit is more likely to be 8,192 bytes (8K). And implementations are not required to accept datagrams with more than 576 total bytes, including data and headers. Consequently, you should be extremely wary of any program that depends on sending or receiving UDP packets with more than 8K of data. Most of the time, larger packets are simply truncated to 8K of data. For maximum safety, the data portion of a UDP packet should be kept to 512 bytes or less, although this limit can negatively affect performance compared to larger packet sizes. (This is a problem for TCP datagrams too, but the stream-based API provided by `Socket` and `ServerSocket` completely shields programmers from these details.)

In Java, a UDP datagram is represented by an instance of the `DatagramPacket` class:

```
public final class DatagramPacket extends Object
```

This class provides methods to get and set the source or destination address from the IP header, to get and set the source or destination port, to get and set the data, and to get and set the length of the data. The remaining header fields are inaccessible from pure Java code.

13.2.1. The Constructors

`DatagramPacket` uses different constructors depending on whether the packet will be used to send data or to receive data. This is a little unusual. Normally, constructors are overloaded to let

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

you provide different kinds of information when you create an object, not to create objects of the same class that will be used in different contexts. In this case, all six constructors take as arguments a `byte` array that holds the datagram's data and the number of bytes in that array to use for the datagram's data. When you want to receive a datagram, these are the only arguments you provide; in addition, the array should be empty. When the socket receives a datagram from the network, it stores the datagram's data in the `DatagramPacket` object's buffer array, up to the length you specified.

The second set of `DatagramPacket` constructors is used to create datagrams you will send over the network. Like the first, these constructors require a buffer array and a length, but they also require the `InetAddress` and port to which the packet is to be sent. In this case, you will pass to the constructor a byte array containing the data you want to send and the destination address and port to which the packet is to be sent. The `DatagramSocket` reads the destination address and port from the packet; the address and port aren't stored within the socket, as they are in TCP.

13.2.1.1. Constructors for receiving datagrams

These two constructors create new `DatagramPacket` objects for receiving data from the network:

```
public DatagramPacket(byte[] buffer, int length)
public DatagramPacket(byte[] buffer, int offset, int length) // Java 1.2
```

When a socket receives a datagram, it stores the datagram's data part in `buffer` beginning at `buffer[0]` and continuing until the packet is completely stored or until `length` bytes have been written into the `buffer`. If the second constructor is used, storage begins at `buffer[offset]` instead. Otherwise, these two constructors are identical. `length` must be less than or equal to `buffer.length-offset`. If you try to construct a `DatagramPacket` with a length that will overflow the `buffer`, the constructor throws an `IllegalArgumentException`. This is a `RuntimeException`, so your code is not required to catch it. It is okay to construct a `DatagramPacket` with a length less than `buffer.length-offset`. In this case, at most the first `length` bytes of `buffer` will be filled when the datagram is received. For example, this code fragment creates a new `DatagramPacket` for receiving a datagram of up to 8,192 bytes:

```
byte[] buffer = new byte[8192];
DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
```

The constructor doesn't care how large the buffer is and would happily let you create a `DatagramPacket` with megabytes of data. However, the underlying native network software is less forgiving, and most native UDP implementations don't support more than 8,192 bytes of data per datagram. The theoretical limit for an IPv4 datagram is 65,507 bytes of data, and a

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

`DatagramPacket` with a 65,507-byte buffer can receive any possible IPv4 datagram without losing data. IPv6 datagrams raise the theoretical limit to 65,536 bytes. In practice, however, many UDP-based protocols such as DNS and TFTP use packets with 512 bytes of data per datagram or fewer. The largest data size in common usage is 8,192 bytes for NFS. Almost all UDP datagrams you're likely to encounter will have 8K of data or fewer. In fact, many operating systems don't support UDP datagrams with more than 8K of data and either truncate, split, or discard larger datagrams. If a large datagram is too big and as a result the network truncates or drops it, your Java program won't be notified of the problem. (UDP is an unreliable protocol, after all.) Consequently, you shouldn't create `DatagramPacket` objects with more than 8,192 bytes of data.

13.2.1.2. Constructors for sending datagrams

These four constructors create new `DatagramPacket` objects for sending data across the network:

```
public DatagramPacket(byte[] data, int length,
 InetAddress destination, int port)
public DatagramPacket(byte[] data, int offset, int length,
 InetAddress destination, int port) // Java 1.2
public DatagramPacket(byte[] data, int length,
 SocketAddress destination, int port) // Java 1.4
public DatagramPacket(byte[] data, int offset, int length,
 SocketAddress destination, int port) // Java 1.4
```

Each constructor creates a new `DatagramPacket` to be sent to another host. The packet is filled with `length` bytes of the `data` array starting at `offset` or 0 if `offset` is not used. If you try to construct a `DatagramPacket` with a length that is greater than `data.length`, the constructor throws an `IllegalArgumentException`. It's okay to construct a `DatagramPacket` object with an `offset` and a `length` that will leave extra, unused space at the end of the `data` array. In this case, only `length` bytes of `data` will be sent over the network. The `InetAddress` or `SocketAddress` object `destination` points to the host you want the packet delivered to; the `int` argument `port` is the port on that host.

Choosing a Datagram Size

The correct amount of data to stuff into one packet depends on the situation. Some protocols dictate the size of the packet. For example, *rlogin* transmits each character to the remote system almost as soon as the user types it. Therefore, packets tend to be

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

short: a single byte of data, plus a few bytes of headers. Other applications aren't so picky. For example, file transfer is more efficient with large buffers; the only requirement is that you split files into packets no larger than the maximum allowable packet size.

Several factors are involved in choosing the optimal packet size. If the network is highly unreliable, such as a packet radio network, smaller packets are preferable since they're less likely to be corrupted in transit. On the other hand, very fast and reliable LANs should use the largest packet size possible. Eight kilobytes—that is, 8,192 bytes—is a good compromise for many types of networks.

It's customary to convert the data to a `byte` array and place it in `data` *before* creating the `DatagramPacket`, but it's not absolutely necessary. Changing `data` *after* the datagram has been constructed and *before* it has been sent changes the data in the datagram; the data isn't copied into a private buffer. In some applications, you can take advantage of this. For example, you could store data that changes over time in `data` and send out the current datagram (with the most recent data) every minute. However, it's more important to make sure that the data doesn't change when you don't want it to. This is especially true if your program is multithreaded, and different threads may write into the data buffer. If this is the case, synchronize the `data` variable or copy the data into a temporary buffer before you construct the `DatagramPacket`.

For instance, this code fragment creates a new `DatagramPacket` filled with the data "This is a test" in ASCII. The packet is directed at port 7 (the echo port) of the host www.ibiblio.org:

```
String s = "This is a test";
byte[] data = s.getBytes("ASCII");

try {
  InetAddress ia = InetAddress.getByName("www.ibiblio.org");
  int port = 7;
  DatagramPacket dp = new DatagramPacket(data, data.length, ia, port);
  // send the packet...
}
catch (IOException ex)
}
```

Most of the time, the hardest part of creating a new `DatagramPacket` is translating the data into a `byte` array. Since this code fragment wants to send an ASCII string, it uses the `getBytes ( )` method of `java.lang.String`. The `java.io.ByteArrayOutputStream` class can also be very useful for preparing data for inclusion in datagrams.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

13.2.2. The get Methods

`DatagramPacket` has six methods that retrieve different parts of a datagram: the actual data plus several fields from its header. These methods are mostly used for datagrams received from the network.

13.2.2.1. public InetAddress getAddress( )

The `getAddress( )` method returns an `InetAddress` object containing the address of the remote host. If the datagram was received from the Internet, the address returned is the address of the machine that sent it (the source address). On the other hand, if the datagram was created locally to be sent to a remote machine, this method returns the address of the host to which the datagram is addressed (the destination address). This method is most commonly used to determine the address of the host that sent a UDP datagram, so that the recipient can reply.

13.2.2.2. public int getPort( )

The `getPort( )` method returns an integer specifying the remote port. If this datagram was received from the Internet, this is the port on the host that sent the packet. If the datagram was created locally to be sent to a remote host, this is the port to which the packet is addressed on the remote machine.

13.2.2.3. public SocketAddress getSocketAddress( ) // Java 1.4

The `getSocketAddress()` method returns a `SocketAddress` object containing the IP address and port of the remote host. As is the case for `getInetAddress( )`, if the datagram was received from the Internet, the address returned is the address of the machine that sent it (the source address). On the other hand, if the datagram was created locally to be sent to a remote machine, this method returns the address of the host to which the datagram is addressed (the destination address). You typically invoke this method to determine the address and port of the host that sent a UDP datagram before you reply. The net effect is not noticeably different than calling `getAddress( )` and `getPort( )`, but if you're using Java 1.4 this saves one method call. Also, if you're using non-blocking I/O, the `DatagramChannel` class accepts a `SocketAddress` but not an `InetAddress` and port.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

13.2.2.4. public byte[] getData( )

The `getData( )` method returns a byte array containing the data from the datagram. It's often necessary to convert the bytes into some other form of data before they'll be useful to your program. One way to do this is to change the byte array into a `String` using the following `String` constructor:

```
public String(byte[] buffer, String encoding)
```

The first argument, `buffer`, is the array of bytes that contains the data from the datagram. The second argument contains the name of the encoding used for this string, such as ASCII or ISO-8859-1. Thus, given a `DatagramPacket` `dp` received from the network, you can convert it to a `String` like this:

```
String s = new String(dp.getData( ), "ASCII");
```

If the datagram does not contain text, converting it to Java data is more difficult. One approach is to convert the `byte` array returned by `getData()` into a `ByteArrayInputStream` using this constructor:

```
public ByteArrayInputStream(byte[] buffer, int offset, int length)
```

`buffer` is the `byte` array to be used as an `InputStream`. It's important to specify the portion of the `buffer` that you want to use as an `InputStream` using the `offset` and `length` arguments. When converting datagram data into `InputStream` objects, `offset` is either 0 (Java 1.1) or given by the `DatagramPacket` object's `getOffset( )` method (Java 2), and `length` is given by the `DatagramPacket` object's `getLength( )` method. For example:

```
InputStream in = new ByteArrayInputStream(packet.getData( ),
 packet.getOffset( ), packet.getLength( ));
```

You *must* specify the `offset` and the `length` when constructing the `ByteArrayInputStream`. Do not use the `ByteArrayInputStream( )` constructor that takes only an array as an argument. The array returned by `packet.getData( )` probably has extra space in it that was not filled with data from the network. This space will contain whatever random values those components of the array had when the `DatagramPacket` was constructed.

The `ByteArrayInputStream` can then be chained to a `DataInputStream`:

```
DataInputStream din = new DataInputStream(in);
```

The data can then be read using the `DataInputStream`'s `readInt( )`, `readLong( )`, `readChar( )`, and other methods. Of course, this assumes that the datagram's sender uses the

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

same data formats as Java; it's probably the case when the sender is written in Java, and is often (though not necessarily) the case otherwise. (Most modern computers use the same floating point format as Java, and most network protocols specify two complement integers in network byte order, which also matches Java's formats.)

13.2.2.5. public int getLength( )

The `getLength( )` method returns the number of bytes of data in the datagram. This is *not* necessarily the same as the length of the array returned by `getData( )`, i.e., `getData( ).length`. The `int` returned by `getLength( )` may be less than the length of the array returned by `getData( )`.

13.2.2.6. public int getOffset( ) // Java 1.2

This method simply returns the point in the array returned by `getData( )` where the data from the datagram begins.

Example 13-1 uses all the methods covered in this section to print the information in the `DatagramPacket`. This example is a little artificial; because the program creates a `DatagramPacket`, it already knows what's in it. More often, you'll use these methods on a `DatagramPacket` received from the network, but that will have to wait for the introduction of the `DatagramSocket` class in the next section.

Example 13-1. Construct a DatagramPacket to receive data

```
import java.net.*;

public class DatagramExample {

  public static void main(String[] args) {

    String s = "This is a test.";

    byte[] data = s.getBytes( );
    try {
      InetAddress ia = InetAddress.getByName("www.ibiblio.org");
      int port = 7;
      DatagramPacket dp
       = new DatagramPacket(data, data.length, ia, port);
      System.out.println("This packet is addressed to "
       + dp.getAddress( ) + " on port " + dp.getPort( ));
      System.out.println("There are " + dp.getLength( )
       + " bytes of data in the packet");
      System.out.println(
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
            new String(dp.getData( ), dp.getOffset( ), dp.getLength( )));
        }
        catch (UnknownHostException e) {
          System.err.println(e);
        }

      }

    }
```

Here's the output:

```
% java DatagramExample
This packet is addressed to www.ibiblio.org/152.2.254.81 on port 7
There are 15 bytes of data in the packet
This is a test.
```

13.2.3. The set Methods

Most of the time, the six constructors are sufficient for creating datagrams. However, Java also provides several methods for changing the data, remote address, and remote port after the datagram has been created. These methods might be important in a situation where the time to create and garbage collect new `DatagramPacket` objects is a significant performance hit. In some situations, reusing objects can be significantly faster than constructing new ones: for example, in a networked twitch game like Quake that sends a datagram for every bullet fired or every centimeter of movement. However, you would have to use a very speedy connection for the improvement to be noticeable relative to the slowness of the network itself.

13.2.3.1. public void setData(byte[] data)

The `setData( )` method changes the payload of the UDP datagram. You might use this method if you are sending a large file (where large is defined as "bigger than can comfortably fit in one datagram") to a remote host. You could repeatedly send the same `DatagramPacket` object, just changing the data each time.

13.2.3.2. public void setData(byte[] data, int offset, int length) // Java 1.2

This overloaded variant of the `setData()` method provides an alternative approach to sending a large quantity of data. Instead of sending lots of new arrays, you can put all the data in one array and send it a piece at a time. For instance, this loop sends a large array in 512-byte chunks:

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
int offset = 0;
DatagramPacket dp = new DatagramPacket(bigarray, offset, 512);
int bytesSent = 0;
while (bytesSent < bigarray.length) {
     socket.send(dp);
     bytesSent += dp.getLength( );
     int bytesToSend = bigarray.length - bytesSent;
     int size = (bytesToSend > 512) ? 512 : bytesToSend;
     dp.setData(bigarray, bytesSent, 512);
}
```

On the other hand, this strategy requires either a lot of confidence that the data will in fact arrive or, alternatively, a disregard for the consequences of its not arriving. It's relatively difficult to attach sequence numbers or other reliability tags to individual packets when you take this approach.

13.2.3.3. public void setAddress(InetAddress remote)

The `setAddress( )` method changes the address a datagram packet is sent to. This might allow you to send the same datagram to many different recipients. For example:

```
String s = "Really Important Message";
byte[] data = s.getBytes("ASCII");
DatagramPacket dp = new DatagramPacket(data, data.length);
dp.setPort(2000);
int network = "128.238.5.";
for (int host = 1; host < 255; host++) {
  try {
    InetAddress remote = InetAddress.getByName(network + host);
    dp.setAddress(remote);
    socket.send(dp);
  }
  catch (IOException ex) {
    // slip it; continue with the next host
  }
}
```

Whether this is a sensible choice depends on the application. If you're trying to send to all the stations on a network segment, as in this fragment, you'd probably be better off using the local broadcast address and letting the network do the work. The local broadcast address is determined by setting all bits of the IP address after the network and subnet IDs to 1. For example, Polytechnic University's network address is 128.238.0.0. Consequently, its broadcast address is 128.238.255.255. Sending a datagram to 128.238.255.255 copies it to every host on that network (although some routers and firewalls may block it, depending on its origin).

For more widely separated hosts, you're probably better off using multicasting. Multicasting actually uses the same `DatagramPacket` class described here. However, it uses different IP addresses and a `MulticastSocket` instead of a `DatagramSocket`. We'll discuss this further in Chapter 14.

13.2.3.4. public void setPort(int port)

The `setPort( )` method changes the port a datagram is addressed to. I honestly can't think of many uses for this method. It could be used in a port scanner application that tried to find open ports running particular UDP-based services such as FSP. Another possibility might be some sort of networked game or conferencing server where the clients that need to receive the same information are all running on different ports as well as different hosts. In this case, `setPort ( )` could be used in conjunction with `setAddress( )` to change destinations before sending the same datagram out again.

13.2.3.5. public void setAddress(SocketAddress remote) // Java 1.4

The `setSocketAddress()` method changes the address and port a datagram packet is sent to. You can use this when replying. For example, this code fragment receives a datagram packet and responds to the same address with a packet containing the ASCII string "Hello there":

```
DatagramPacket  input = newDatagramPacket(new byte[8192], 8192);
socket.receive(input);
SocketAddress address = input.getSocketAddress( );
DatagramPacket output = new DatagramPacket("Hello there"
                                           .getBytes("ASCII"), 11);
output.setAddress(address);
socket.send(output);
```

You could certainly write the same code using `InetAddress` objects and ports instead of a `SocketAddress`. Indeed, in Java 1.3 and earlier, you have to. The code would be just a few lines longer:

```
DatagramPacket  input = newDatagramPacket(new byte[8192], 8192);
socket.receive(input);
InetAddress address = input.getAddress( );
int port = input.getPort( );
DatagramPacket output = new DatagramPacket("Hello there".getBytes("ASCII"), 11);
output.setAddress(address);
output.setPort(port);
socket.send(output);
```

13.2.3.6. public void setLength(int length)

The `setLength( )` method changes the number of bytes of data in the internal buffer that are considered to be part of the datagram's data as opposed to merely unfilled space. This method is useful when receiving datagrams, as we'll explore later in this chapter. When a datagram is received, its length is set to the length of the incoming data. This means that if you try to receive another datagram into the same `DatagramPacket`, it's limited to no more than the number of bytes in the first. That is, once you've received a 10-byte datagram, all subsequent datagrams will be

truncated to 10 bytes; once you've received a 9-byte datagram, all subsequent datagrams will be truncated to 9 bytes; and so on. This method lets you reset the length of the buffer so that subsequent datagrams aren't truncated.

13.3. The DatagramSocket Class

To send or receive a `DatagramPacket`, you must open a datagram socket. In Java, a datagram socket is created and accessed through the `DatagramSocket` class:

```
public class DatagramSocket extends Object
```

All datagram sockets are bound to a local port, on which they listen for incoming data and which they place in the header of outgoing datagrams. 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). This port number is placed in any outgoing datagrams and will be used by the server to address any response datagrams. If you're writing a server, clients need to know on which port the server is listening for incoming datagrams; therefore, when a server constructs a `DatagramSocket`, it specifies the local port on which it will listen. However, the sockets used by clients and servers are otherwise identical: they differ only in whether they use an anonymous (system-assigned) or a well-known port. There's no distinction between client sockets and server sockets, as there is with TCP; there's no such thing as a `DatagramServerSocket`.

13.3.1. The Constructors

The `DatagramSocket` constructors are used in different situations, much like the `DatagramPacket` constructors. The first constructor opens a datagram socket on an anonymous local port. The second constructor opens a datagram socket on a well-known local port that listens to all local network interfaces. The third constructor opens a datagram socket on a well-known local port on a specific network interface. Java 1.4 adds a constructor that allows this network interface and port to be specified with a `SocketAddress`. Java 1.4 also adds a protected constructor that allows you to change the implementation class. All five constructors deal only with the local address and port. The remote address and port are stored in the `DatagramPacket`, not the `DatagramSocket`. Indeed, one `DatagramSocket` can send and receive datagrams from multiple remote hosts and ports.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

13.3.1.1. public DatagramSocket( ) throws SocketException

This constructor creates a socket that is bound to an anonymous port. For example:

```
try {
  DatagramSocket client = new DatagramSocket( );
  // send packets...
}
catch (SocketException ex) {
  System.err.println(ex);
}
```

You would use this constructor in a client that initiates a conversation with a server. In this scenario, you don't care what port the socket is bound to, because the server will send its response to the port from which the datagram originated. Letting the system assign a port means that you don't have to worry about finding an unused port. If for some reason you need to know the local port, you can find out with the `getLocalPort( )` method described later in this chapter.

The same socket can receive the datagrams that a server sends back to it. A `SocketException` is thrown if the socket can't be created. It's unusual for this constructor to throw an exception; it's hard to imagine situations in which the socket could not be opened, since the system gets to choose the local port.

13.3.1.2. public DatagramSocket(int port) throws SocketException

This constructor creates a socket that listens for incoming datagrams on a particular port, specified by the `port` argument. Use this constructor to write a server that listens on a well-known port; if servers listened on anonymous ports, clients would not be able to contact them. A `SocketException` is thrown if the socket can't be created. There are two common reasons for the constructor to fail: the specified port is already occupied, or you are trying to connect to a port below 1,024 and you don't have sufficient privileges (i.e., you are not root on a Unix system; for better or worse, other platforms allow anyone to connect to low-numbered ports).

TCP ports and UDP ports are not related. Two unrelated servers or clients can use the same port number if one uses UDP and the other uses TCP. Example 13-2 is a port scanner that looks for UDP ports in use on the local host. It decides that the port is in use if the `DatagramSocket` constructor throws an exception. As written, it looks at ports from 1,024 and up to avoid Unix's requirement that it run as root to bind to ports below 1,024. You can easily extend it to check ports below 1,024, however, if you have root access or are running it on Windows.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Example 13-2. Look for local UDP ports

```
import java.net.*;

public class UDPPortScanner {

  public static void main(String[] args) {

    for (int port = 1024; port <= 65535; port++) {
      try {
        // the next line will fail and drop into the catch block if
        // there is already a server running on port i
        DatagramSocket server = new DatagramSocket(port);
        server.close( );
      }
      catch (SocketException ex) {
        System.out.println("There is a server on port " + port + ".");
      } // end try
    } // end for

  }

}
```

The speed at which `UDPPortScanner` runs depends strongly on the speed of your machine and its UDP implementation. I've clocked Example 13-2 at as little as two minutes on a moderately powered SPARCstation, under 12 seconds on a 1Ghz TiBook, about 7 seconds on a 1.4GHz Athlon system running Linux, and as long as an hour on a PowerBook 5300 running MacOS 8. Here are the results from the Linux workstation on which much of the code in this book was written:

```
% java UDPPortScanner
There is a server on port 2049.
There is a server on port 32768.
There is a server on port 32770.
There is a server on port 32771.
```

The first port, 2049, is an NFS server. The high-numbered ports in the 30,000 range are Remote Procedure Call (RPC) services. Along with RPC, common protocols that use UDP include NFS, TFTP, and FSP.

It's much harder to scan UDP ports on a remote system than to scan for remote TCP ports. Whereas there's always some indication that a listening port, regardless of application layer protocol, has received your TCP packet, UDP provides no such guarantees. To determine that a UDP server is listening, you have to send it a packet it will recognize and respond to.

13.3.1.3. public DatagramSocket(int port, InetAddress interface) throws SocketException

This constructor is primarily used on multihomed hosts; it creates a socket that listens for incoming datagrams on a specific port and network interface. The `port` argument is the port on which this socket listens for datagrams. As with TCP sockets, you need to be root on a Unix system to create

a `DatagramSocket` on a port below 1,024. The `address` argument is an `InetAddress` object matching one of the host's network addresses. A `SocketException` is thrown if the socket can't be created. There are three common reasons for this constructor to fail: the specified port is already occupied, you are trying to connect to a port below 1,024 and you're not root on a Unix system, or `address` is not the address of one of the system's network interfaces.

13.3.1.4. public DatagramSocket(SocketAddress interface) throws SocketException // Java 1.4

This constructor is similar to the previous one except that the network interface address and port are read from a `SocketAddress`. For example, this code fragment creates a socket that only listens on the local loopback address:

```
SocketAddress address = new InetSocketAddress("127.0.0.1", 9999);
DatagramSocket socket = new DatagramSocket(address);
```

13.3.1.5. protected DatagramSocket(DatagramSocketImpl impl) throws SocketException // Java 1.4

This constructor enables subclasses to provide their own implementation of the UDP protocol, rather than blindly accepting the default. Unlike sockets created by the other four constructors, this socket is not initially bound to a port. Before using it you have to bind it to a `SocketAddress` using the `bind( )` method, which is also new in Java 1.4:

```
public void bind(SocketAddress addr) throws SocketException
```

You can pass null to this method, binding the socket to any available address and port.

13.3.2. Sending and Receiving Datagrams

The primary task of the `DatagramSocket` class is to send and receive UDP datagrams. One socket can both send and receive. Indeed, it can send and receive to and from multiple hosts at the same time.

13.3.2.1. public void send(DatagramPacket dp) throws IOException

Once a `DatagramPacket` is created and a `DatagramSocket` is constructed, send the packet by passing it to the socket's `send()` method. For example, if `theSocket` is a

DatagramSocket object and theOutput is a DatagramPacket object, send theOutput using theSocket like this:

```
theSocket.send(theOutput);
```

If there's a problem sending the data, an IOException may be thrown. However, this is less common with DatagramSocket than Socket or ServerSocket, since the unreliable nature of UDP means you won't get an exception just because the packet doesn't arrive at its destination. You may get an IOException if you're trying to send a larger datagram than the host's native networking software supports, but then again you may not. This depends heavily on the native UDP software in the OS and the native code that interfaces between this and Java's DatagramSocketImpl class. This method may also throw a SecurityException if the SecurityManager won't let you communicate with the host to which the packet is addressed. This is primarily a problem for applets and other remotely loaded code.

Example 13-3 is a UDP-based discard client. It reads lines of user input from System.in and sends them to a discard server, which simply discards all the data. Each line is stuffed in a DatagramPacket. Many of the simpler Internet protocols, such as discard, have both TCP and UDP implementations.

Example 13-3. A UDP discard client

```
import java.net.*;
import java.io.*;

public class UDPDiscardClient {

  public final static int DEFAULT_PORT = 9;

  public static void main(String[] args) {

    String hostname;
    int port = DEFAULT_PORT;

    if (args.length > 0) {
      hostname = args[0];
      try {
              port = Integer.parseInt(args[1]);
      }
      catch (Exception ex) {
        // use default port
      }
    }
    else {
      hostname = "localhost";
    }

    try {
      InetAddress server = InetAddress.getByName(hostname);
      BufferedReader userInput
       = new BufferedReader(new InputStreamReader(System.in));
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
        DatagramSocket theSocket = new DatagramSocket( );
        while (true) {
          String theLine = userInput.readLine( );
          if (theLine.equals(".")) break;
          byte[] data = theLine.getBytes( );
          DatagramPacket theOutput
           = new DatagramPacket(data, data.length, server, port);
          theSocket.send(theOutput);
        }  // end while
      }  // end try
      catch (UnknownHostException uhex) {
        System.err.println(uhex);
      }
      catch (SocketException sex) {
        System.err.println(sex);
      }
      catch (IOException ioex) {
        System.err.println(ioex);
      }

    }  // end main

  }
```

The `UDPDiscardClient` class should look familiar. It has a single static field, `DEFAULT_PORT`, which is set to the standard port for the discard protocol (port 9), and a single method, `main( )`. The `main()` method reads a hostname from the command line and converts that hostname to the `InetAddress` object called `server`. A `BufferedReader` is chained to `System.in` to read user input from the keyboard. Next, a `DatagramSocket` object called `theSocket` is constructed. After creating the socket, the program enters an infinite `while` loop that reads user input line by line using `readLine()`. We are careful, however, to use only `readLine( )` to read data from the console, the one place where it is guaranteed to work as advertised. Since the discard protocol deals only with raw bytes, we can ignore character encoding issues.

In the `while` loop, each line is converted to a byte array using the `getBytes( )` method, and the bytes are stuffed in a new `DatagramPacket`, `theOutput`. Finally, `theOutput` is sent over `theSocket`, and the loop continues. If at any point the user types a period on a line by itself, the program exits. The `DatagramSocket` constructor may throw a `SocketException`, so that needs to be caught. Because this is a discard client, we don't need to worry about data coming back from the server.

13.3.2.2. public void receive(DatagramPacket dp) throws IOException

This method receives a single UDP datagram from the network and stores it in the preexisting `DatagramPacket` object `dp`. Like the `accept( )` method in the `ServerSocket` class, this method blocks the calling thread until a datagram arrives. If your program does anything besides wait for datagrams, you should call `receive()` in a separate thread.

The datagram's buffer should be large enough to hold the data received. If not, `receive( )` places as much data in the buffer as it can hold; the rest is lost. It may be useful to remember that the maximum size of the data portion of a UDP datagram is 65,507 bytes. (That's the 65,536-byte maximum size of an IP datagram minus the 20-byte size of the IP header and the 8-byte size of the UDP header.) Some application protocols that use UDP further restrict the maximum number of bytes in a packet; for instance, NFS uses a maximum packet size of 8,192 bytes.

If there's a problem receiving the data, an `IOException` may be thrown. In practice, this is rare. Unlike `send( )`, this method does not throw a `SecurityException` if an applet receives a datagram from other than the applet host. However, it will silently discard all such packets. (This behavior prevents a denial-of-service attack against applets that receive UDP datagrams.)

Example 13-4 shows a UDP discard server that receives incoming datagrams. Just for fun, it logs the data in each datagram to `System.out` so that you can see who's sending what to your discard server.

Example 13-4. The UDPDiscardServer

```
import java.net.*;
import java.io.*;

public class UDPDiscardServer {

  public final static int DEFAULT_PORT = 9;
  public final static int MAX_PACKET_SIZE = 65507;

  public static void main(String[] args) {

    int port = DEFAULT_PORT;
    byte[] buffer = new byte[MAX_PACKET_SIZE];

    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception ex) {
      // use default port
    }

    try {
      DatagramSocket server = new DatagramSocket(port);
      DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
      while (true) {
        try {
          server.receive(packet);
          String s = new String(packet.getData( ), 0, packet.getLength( ));
          System.out.println(packet.getAddress( ) + " at port "
           + packet.getPort( ) + " says " + s);
          // reset the length for the next packet
          packet.setLength(buffer.length);
        }
        catch (IOException ex) {
          System.err.println(ex);
        }
       } // end while
```

```
      }  // end try
      catch (SocketException  ex) {
        System.err.println(ex);
      }  // end catch

    }  // end main

  }
```

This is a simple class with a single method, `main()`. It reads the port the server listens to from the command line. If the port is not specified on the command line, it listens on port 9. It then opens a `DatagramSocket` on that port and creates a `DatagramPacket` with a 65,507-byte buffer—large enough to receive any possible packet. Then the server enters an infinite loop that receives packets and prints the contents and the originating host on the console. A high-performance discard server would skip this step. As each datagram is received, the length of `packet` is set to the length of the data in that datagram. Consequently, as the last step of the loop, the length of the packet is reset to the maximum possible value. Otherwise, the incoming packets would be limited to the minimum size of all previous packets. You can run the discard client on one machine and connect to the discard server on a second machine to verify that the network is working.

13.3.2.3. public void close( )

Calling a `DatagramSocket` object's `close( )` method frees the port occupied by that socket. For example:

```
try {
  DatagramSocket server = new DatagramSocket( );
  server.close( );
}
catch (SocketException ex) {
  System.err.println(ex);
}
```

It's never a bad idea to close a `DatagramSocket` when you're through with it; it's particularly important to close an unneeded socket if the program will continue to run for a significant amount of time. For example, the `close()` method was essential in Example 13-2, `UDPPortScanner`: if this program did not close the sockets it opened, it would tie up every UDP port on the system for a significant amount of time. On the other hand, if the program ends as soon as you're through with the `DatagramSocket`, you don't need to close the socket explicitly; the socket is automatically closed upon garbage collection. However, Java won't run the garbage collector just because you've run out of ports or sockets, unless by lucky happenstance you run out of memory at the same time. Closing unneeded sockets never hurts and is good programming practice.

13.3.2.4. public int getLocalPort( )

A `DatagramSocket`'s `getLocalPort( )` method returns an `int` that represents the local port on which the socket is listening. Use this method if you created a `DatagramSocket` with an anonymous port and want to find out what port the socket has been assigned. For example:

```
try {
  DatagramSocket ds = new DatagramSocket( );
  System.out.println("The socket is using port " + ds.getLocalPort( ));
}
catch (SocketException ex) {
  ex.printStackTrace( );
}
```

13.3.2.5. public InetAddress getLocalAddress( )

A `DatagramSocket`'s `getLocalAddress( )` method returns an `InetAddress` object that represents the local address to which the socket is bound. It's rarely needed in practice. Normally, you either know or don't care which address a socket is listening to.

13.3.2.6. public SocketAddress getLocalSocketAddress( ) // Java 1.4

The `getLocalSocketAddress()` method returns a `SocketAddress` object that wraps the local interface and port to which the socket is bound. Like `getLocalAddress( )`, it's a little hard to imagine a realistic use case here. This method probably exists mostly for parallelism with `setLocalSocketAddress()`.

13.3.3. Managing Connections

Unlike TCP sockets, datagram sockets aren't very picky about whom they'll talk to. In fact, by default they'll talk to anyone, but this is often not what you want. For instance, applets are only allowed to send datagrams to and receive datagrams from the applet host. An NFS or FSP client should accept packets only from the server it's talking to. A networked game should listen to datagrams only from the people playing the game. In Java 1.1, programs must manually check the source addresses and ports of the hosts sending them data to make sure they're who they should be. However, Java 1.2 adds four methods that let you choose which host you can send datagrams to and receive datagrams from, while rejecting all others' packets.

13.3.3.1. public void connect(InetAddress host, int port) // Java 1.2

The `connect( )` method doesn't really establish a connection in the TCP sense. However, it does specify that the `DatagramSocket` will send packets to and receive packets from only the specified remote host on the specified remote port. Attempts to send packets to a different host or port will throw an `IllegalArgumentException`. Packets received from a different host or a different port will be discarded without an exception or other notification.

A security check is made when the `connect( )` method is invoked. If the VM is allowed to send data to that host and port, the check passes silently. Otherwise, a `SecurityException` is thrown. However, once the connection has been made, `send( )` and `receive( )` on that `DatagramSocket` no longer make the security checks they'd normally make.

13.3.3.2. public void disconnect( ) // Java 1.2

The `disconnect( )` method breaks the "connection" of a connected `DatagramSocket` so that it can once again send packets to and receive packets from any host and port.

13.3.3.3. public int getPort( ) // Java 1.2

If and only if a `DatagramSocket` is connected, the `getPort( )` method returns the remote port to which it is connected. Otherwise, it returns -1.

13.3.3.4. public InetAddress getInetAddress( ) // Java 1.2

If and only if a `DatagramSocket` is connected, the `getInetAddress( )` method returns the address of the remote host to which it is connected. Otherwise, it returns null.

13.3.3.5. public InetAddress getRemoteSocketAddress( ) // Java 1.4

If a `DatagramSocket` is connected, the `getRemoteSocketAddress( )` method returns the address of the remote host to which it is connected. Otherwise, it returns null.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

13.3.4. Socket Options

The only socket option supported for datagram sockets in Java 1.1 is SO_TIMEOUT. Java 1.2 adds SO_SNDBUF and SO_RCVBUF. Java 1.4 adds SO_REUSEADDR and SO_BROADCAST and enables the specification of the traffic class.

13.3.4.1. SO_TIMEOUT

SO_TIMEOUT is the amount of time, in milliseconds, that `receive( )` waits for an incoming datagram before throwing an `InterruptedIOException` (a subclass of `IOException`). Its value must be nonnegative. If SO_TIMEOUT is 0, `receive( )` never times out. This value can be changed with the `setSoTimeout( )` method and inspected with the `getSoTimeout( )` method:

```
public synchronized void setSoTimeout(int timeout)
 throws SocketException
public synchronized int getSoTimeout( ) throws IOException
```

The default is to never time out, and indeed there are few situations in which you would need to set SO_TIMEOUT. You might need it if you were implementing a secure protocol that required responses to occur within a fixed amount of time. You might also decide that the host you're communicating with is dead (unreachable or not responding) if you don't receive a response within a certain amount of time.

The `setSoTimeout( )` method sets the SO_TIMEOUT field for a datagram socket. When the timeout expires, an `InterruptedIOException` is thrown. (In Java 1.4 and later, `SocketTimeoutException`, a subclass of `InterruptedIOException`, is thrown instead.) Set this option *before* you call `receive( )`. You cannot change it while `receive ( )` is waiting for a datagram. The `timeout` argument must be greater than or equal to zero; if it is not, `setSoTimeout( )` throws a `SocketException`. For example:

```
try {
  buffer = new byte[2056];
  DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
  DatagramSocket ds = new DatagramSocket(2048);
  ds.setSoTimeout(30000); // block for no more than 30 seconds
  try {
   ds.receive(dp);
    // process the packet...
  }
  catch (InterruptedIOException ex) {
    ss.close( );
    System.err.println("No connection within 30 seconds");
  }
catch (SocketException ex) {
  System.err.println(ex);
}
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
catch (IOException ex) {
  System.err.println("Unexpected IOException: " + ex);
}
```

The `getSoTimeout( )` method returns the current value of this `DatagramSocket` object's SO_TIMEOUT field. For example:

```
public void printSoTimeout(DatagramSocket ds) {

  int timeout = ds.getSoTimeOut( );
  if (timeout > 0) {
    System.out.println(ds + " will time out after "
     + timeout + "milliseconds.");
  }
  else if (timeout == 0) {
    System.out.println(ds + " will never time out.");
  }
  else {
    System.out.println("Something is seriously wrong with " + ds);
  }

}
```

13.3.4.2. SO_RCVBUF

The SO_RCVBUF option of `DatagramSocket` is closely related to the SO_RCVBUF option of `Socket`. It determines the size of the buffer used for network I/O. Larger buffers tend to improve performance for reasonably fast (say, Ethernet-speed) connections because they can store more incoming datagrams before overflowing. Sufficiently large receive buffers are even more important for UDP than for TCP, since a UDP datagram that arrives when the buffer is full will be lost, whereas a TCP datagram that arrives at a full buffer will eventually be retransmitted. Furthermore, SO_RCVBUF sets the maximum size of datagram packets that can be received by the application. Packets that won't fit in the receive buffer are silently discarded.

`DatagramSocket` has methods to get and set the suggested receive buffer size used for network input:

```
public void setReceiveBufferSize(int size) throws SocketException // Java 1.2
public int getReceiveBufferSize( ) throws SocketException         // Java 1.2
```

The `setReceiveBufferSize( )` method suggests a number of bytes to use for buffering input from this socket. However, the underlying implementation is free to ignore this suggestion. For instance, many 4.3 BSD-derived systems have a maximum receive buffer size of about 52K and won't let you set a limit higher than this. My Linux box was limited to 64K. Other systems raise this to about 240K. The details are highly platform-dependent. Consequently, you may wish to check the actual size of the receive buffer with `getReceiveBufferSize( )` after setting it. The `getReceiveBufferSize( )` method returns the number of bytes in the buffer used for input from this socket.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Both methods throw a `SocketException` if the underlying socket implementation does not recognize the SO_RCVBUF option. This might happen on a non-POSIX operating system. The `setReceiveBufferSize( )` method throws an `IllegalArgumentException` if its argument is less than or equal to zero.

13.3.4.3. SO_SNDBUF

`DatagramSocket` has methods to get and set the suggested send buffer size used for network output:

```
public void setSendBufferSize(int size)  throws SocketException // Java 1.2
public int getSendBufferSize( ) throws SocketException          // Java 1.2
```

The `setSendBufferSize( )` method suggests a number of bytes to use for buffering output on this socket. Once again, however, the operating system is free to ignore this suggestion. Consequently, you'll want to check the result of `setSendBufferSize( )` by immediately following it with a call to `getSend BufferSize( )` to find out the real the buffer size.

Both methods throw a `SocketException` if the underlying native network software doesn't understand the `SO_SNDBUF` option. The `setSendBufferSize( )` method also throws an `IllegalArgumentException` if its argument is less than or equal to zero.

13.3.4.4. SO_REUSEADDR

The SO_REUSEADDR option does not mean the same thing for UDP sockets as it does for TCP sockets. For UDP, SO_REUSEADDR can control whether multiple datagram sockets can bind to the same port and address *at the same time*. If multiple sockets are bound to the same port, received packets will be copied to all bound sockets. This option is controlled by these two methods:

```
public void setReuseAddress(boolean on) throws SocketException  // Java 1.4
public boolean getReuseAddress( ) throws SocketException        // Java 1.4
```

For this to work reliably, `setReuseAddress( )` must be called *before* the new socket binds to the port. This means the socket must be created in an unconnected state using the protected constructor that takes a `DatagramImpl` as an argument. In other words, it won't work with a plain vanilla `DatagramSocket`. Reusable ports are most commonly used for multicast sockets, which will be discussed in the next chapter. Datagram channels also create unconnected datagram sockets that can be configured to reuse ports, as you'll see later in this chapter.

13.3.4.5. SO_BROADCAST

The SO_BROADCAST option controls whether a socket is allowed to send packets to and receive packets from broadcast addresses such as 192.168.254.255, the local network broadcast address for the network with the local address 192.168.254.*. UDP broadcasting is often used for protocols like the JXTA Peer Discovery Protocol and the Service Location Protocol that need to communicate with servers on the local net whose addresses are not known in advance. This option is controlled with these two methods:

```
public void setBroadcast(boolean on) throws SocketException  // Java 1.4
public boolean getBroadcast( ) throws SocketException         // Java 1.4
```

Routers and gateways do not normally forward broadcast messages, but they can still kick up a lot of traffic on the local network. This option is turned on by default, but if you like you can disable it thusly:

```
socket.setBroadcast(false);
```

This option can be changed after the socket has been bound.

On some implementations, sockets bound to a specific address do not receive broadcast packets. In other words, use the `DatagramPacket(int port)` constructor, not the `DatagramPacket(InetAddress address, int port)` constructor to listen to broadcasts. This is necessary in addition to setting the SO_BROADCAST option to true.

13.3.4.6. Traffic class

Traffic class is essentially the same for UDP as it is for TCP. After all, packets are actually routed and prioritized according to IP, which both TCP and UDP sit on top of. There's really no difference between the `setTrafficClass( )` and `getTrafficClass( )` methods in `DatagramSocket` and those in `Socket`. They just have to be repeated here because `DatagramSocket` and `Socket` don't have a common superclass. These two methods let you inspect and set the class of service for a socket using these two methods:

```
public int getTrafficClass( ) throws SocketException // Java 1.4
public void setTrafficClass(int trafficClass) throws SocketException
                                                     // Java 1.4
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

The traffic class is given as an int between 0 and 255. (Values outside this range cause `IllegalArgumentExceptions`.) This int is a combination of bit-flags. Specifically:

- 0x02: Low cost
- 0x04: High reliability
- 0x08: Maximum throughput
- 0x10: Minimum delay

Java always sets the lowest order, ones bit to zero, even if you try to set it to one. The three high-order bits are not yet used. For example, this code fragment requests a low cost connection:

```
DatagramSocket s = new DatagramSocket ( );
s.setTrafficClass(0x02);
```

This code fragment requests a connection with maximum throughput and minimum delay:

```
DatagramSocket s = new DatagramSocket ( );
s.setTrafficClass(0x08 | 0x10);
```

The underlying socket implementation is not required to respect any of these requests. They hint at the policy that is desired. Probably most current implementations will ignore these values completely. If the local network stack is unable to provide the requested class of service, it may throw a `SocketException`, but it's not required to and truth be told, it probably won't.

13.4. Some Useful Applications

In this section, you'll see several Internet servers and clients that use `DatagramPacket` and `DatagramSocket`. Some of these will be familiar from previous chapters because many Internet protocols have both TCP and UDP implementations. When an IP packet is received by a host, the host determines whether the packet is a TCP packet or a UDP datagram by inspecting the IP header. As I said earlier, there's no connection between UDP and TCP ports; TCP and UDP servers can share the same port number without problems. By convention, if a service has both TCP and UDP implementations, it uses the same port for both, although there's no technical reason this has to be the case.

13.4.1. Simple UDP Clients

Several Internet services need to know only the client's address and port; they ignore any data the client sends in its datagrams. Daytime, quote of the day, time, and chargen are four such protocols.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Each of these responds the same way, regardless of the data contained in the datagram, or indeed regardless of whether there actually is any data in the datagram. Clients for these protocols simply send a UDP datagram to the server and read the response that comes back. Therefore, let's begin with a simple client called `UDPPoke`, shown in Example 13-5, which sends an empty UDP packet to a specified host and port and reads a response packet from the same host.

The `UDPPoke` class has three private fields. The `bufferSize` field specifies how large a return packet is expected. An 8,192-byte buffer is large enough for most of the protocols that `UDPPoke` is useful for, but it can be increased by passing a different value to the constructor. The `DatagramSocket` object `socket` will be used to both send and receive datagrams. Finally, the `DatagramPacket` object `outgoing` is the message sent to the individual servers.

The constructors initialize all three fields using an `InetAddress` for the host and `int`s for the port, the buffer length, and the number of milliseconds to wait before timing out. These last three become part of the `DatagramSocket` field `socket`. If the buffer length is not specified, 8,192 bytes is used. If the timeout is not given, 30 seconds (30,000 milliseconds) is used. The host, port, and buffer size are also used to construct the `outgoing` `DatagramPacket`. Although in theory you should be able to send a datagram with no data at all, bugs in some Java implementations require that you add at least one byte of data to the datagram. The simple servers we're currently considering ignore this data.

Once a `UDPPoke` object has been constructed, clients will call its `poke( )` method to send an empty `outgoing` datagram to the target and read its response. The response is initially set to null. When the expected datagram appears, its data is copied into the `response` field. This method returns null if the response doesn't come quickly enough or never comes at all.

The `main( )` method merely reads the host and port to connect to from the command line, constructs a `UDPPoke` object, and pokes it. Most of the simple protocols that this client suits will return ASCII text, so we'll attempt to convert the response to an ASCII string and print it. Not all VMs support the ASCII character encoding, so we'll provide the possibility of using the ASCII superset Latin-1 (8859-1) as a backup.

Example 13-5. The UDPPoke class

```
import java.net.*;
import java.io.*;

public class UDPPoke {

  private int            bufferSize; // in bytes
  private DatagramSocket socket;
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
  private DatagramPacket outgoing;

  public UDPPoke(InetAddress host, int port, int bufferSize,
   int timeout) throws SocketException {

    outgoing = new DatagramPacket(new byte[1], 1, host, port);
    this.bufferSize = bufferSize;
    socket = new DatagramSocket(0);
    socket .connect(host, port); // requires Java 2
    socket .setSoTimeout(timeout);

  }

  public UDPPoke(InetAddress host, int port, int bufferSize)
   throws SocketException {
    this(host, port, bufferSize, 30000);
  }

  public UDPPoke(InetAddress host, int port)
   throws SocketException {
    this(host, port, 8192, 30000);
  }

  public byte[] poke( ) throws IOException {

    byte[] response = null;
    try {
      socket .send(outgoing);
      DatagramPacket incoming
       = new DatagramPacket(new byte[bufferSize], bufferSize);
      // next line blocks until the response is received
      socket .receive(incoming);
      int numBytes = incoming.getLength( );
      response = new byte[numBytes];
      System.arraycopy(incoming.getData( ), 0, response, 0, numBytes);
    }
    catch (IOException ex) {
      // response will be null
    }

    // may return null
    return response;
  }

  public static void main(String[] args) {

    InetAddress host;
    int port = 0;

    try {
      host = InetAddress.getByName(args[0]);
      port = Integer.parseInt(args[1]);
      if (port < 1 || port > 65535) throw new Exception( );
    }
    catch (Exception ex) {
      System.out.println("Usage: java UDPPoke host port");
      return;
    }

    try {
      UDPPoke poker = new UDPPoke(host, port);
      byte[] response = poker.poke( );
      if (response == null) {
              System.out.println("No response within allotted time");
              return;
      }
      String result = "";
      try {
        result = new String(response, "ASCII");
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
        }
        catch (UnsupportedEncodingException e) {
                // try a different encoding
                result = new String(response, "8859_1");
        }
        System.out.println(result);
      }
      catch (Exception ex) {
        System.err.println(ex);
        ex.printStackTrace( );
      }

    }  // end main

  }
```

For example, this connects to a daytime server over UDP:

```
D:\JAVA\JNP3\examples\13>java UDPPoke rama.poly.edu 13
Sun Oct  3 13:04:22 1999
```

This connects to a chargen server:

```
D:\JAVA\JNP3\examples\13>java UDPPoke rama.poly.edu 19
123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuv
```

Given this class, UDP daytime, time, chargen, and quote of the day clients are almost trivial. Example 13-6 demonstrates a time client. The most complicated part is converting the four raw bytes returned by the server to a `java.util.Date` object. The same algorithm as in Example 10-5 is used here, so I won't repeat that discussion. The other protocols are left as exercises for the reader.

Example 13-6. A UDP time client

```
import java.net.*;
import java.util.*;

public class UDPTimeClient {

  public final static int DEFAULT_PORT = 37;
  public final static String DEFAULT_HOST = "time-a.nist.gov";

  public static void main(String[] args) {

    InetAddress host;
    int port = DEFAULT_PORT;

    try {
      if (args.length > 0) {
        host = InetAddress.getByName(args[0]);
      }
      else {
        host = InetAddress.getByName(DEFAULT_HOST);
      }
    }
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
      catch (Exception ex) {
        System.out.println("Usage: java UDPTimeClient host port");
        return;
      }

      if (args.length > 1) {
        try {
          port = Integer.parseInt(args[1]);
          if (port <= 0 || port > 65535) port = DEFAULT_PORT;;
        }
        catch (Exception ex){
        }
      }

      try {
        UDPPoke poker = new UDPPoke(host, port);
        byte[] response = poker.poke( );
          if (response == null) {
          System.out.println("No response within allotted time");
          return;
          }
          else if (response.length != 4) {
          System.out.println("Unrecognized response format");
          return;
          }


        // The time protocol sets the epoch at 1900,
        // the Java Date class at 1970. This number
        // converts between them.

        long differenceBetweenEpochs = 2208988800L;

        long secondsSince1900 = 0;
        for (int i = 0; i < 4; i++) {
          secondsSince1900
           = (secondsSince1900 << 8) | (response[i] & 0x000000FF);
        }

        long secondsSince1970
         = secondsSince1900 - differenceBetweenEpochs;
        long msSince1970 = secondsSince1970 * 1000;
        Date time = new Date(msSince1970);

        System.out.println(time);
      }
      catch (Exception ex) {
        System.err.println(ex);
        ex.printStackTrace( );
      }

    }

  }
```

13.4.2. UDPServer

Clients aren't the only programs that benefit from a reusable implementation. The servers for these protocols are very similar. They all wait for UDP datagrams on a specified port and reply to each datagram with another datagram. The servers differ only in the content of the datagram that they return. Example 13-7 is a simple `UDPServer` class that can be subclassed to provide specific servers for different protocols.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

The `UDPServer` class has two fields, the `int bufferSize` and the `DatagramSocket socket`, the latter of which is protected so it can be used by subclasses. The constructor opens the `DatagramSocket socket` on a specified local port to receive datagrams of no more than `bufferSize` bytes.

`UDPServer` extends `Thread` so that multiple instances can run in parallel. Its `run( )` method contains an infinite loop that repeatedly receives an incoming datagram and responds by passing it to the abstract `respond( )` method. This method will be overridden by particular subclasses in order to implement different kinds of servers.

`UDPServer` is a very flexible class. Subclasses can send zero, one, or many datagrams in response to each incoming datagram. If a lot of processing is required to respond to a packet, the `respond( )` method can spawn a thread to do it. However, UDP servers tend not to have extended interactions with a client. Each incoming packet is treated independently of other packets, so the response can usually be handled directly in the `respond( )` method without spawning a thread.

Example 13-7. The UDPServer class

```
import java.net.*;
import java.io.*;

public abstract class UDPServer extends Thread {

  private   int             bufferSize; // in bytes
  protected DatagramSocket socket;

  public UDPServer(int port, int bufferSize)
   throws SocketException {
    this.bufferSize = bufferSize;
    this.socket = new DatagramSocket(port);
  }

  public UDPServer(int port) throws SocketException {
    this(port, 8192);
  }

  public void run( ) {

    byte[] buffer = new byte[bufferSize];
    while (true) {
      DatagramPacket incoming = new DatagramPacket(buffer, buffer.length);
      try {
        socket.receive(incoming);
        this.respond(incoming);
      }
      catch (IOException ex) {
        System.err.println(ex);
      }
    } // end while

  }  // end run
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
  public abstract void respond(DatagramPacket request);

}
```

The easiest protocol to handle is discard. All we need to do is write a `main( )` method that sets the port and start the thread. `respond( )` is a do-nothing method. Example 13-8 is a high-performance UDP discard server that does nothing with incoming packets.

Example 13-8. A high-performance UDP discard server

```
import java.net.*;

public class FastUDPDiscardServer extends UDPServer {

  public final static int DEFAULT_PORT = 9;

  public FastUDPDiscardServer( ) throws SocketException {
    super(DEFAULT_PORT);
  }

  public void respond(DatagramPacket packet) {}

  public static void main(String[] args) {

   try {
     UDPServer server = new FastUDPDiscardServer( );
     server.start( );
   }
   catch (SocketException ex) {
     System.err.println(ex);
   }

  }

}
```

Example 13-9 is a slightly more interesting discard server that prints the incoming packets on `System.out`.

Example 13-9. A UDP discard server

```
import java.net.*;

public class LoggingUDPDiscardServer extends UDPServer {

  public final static int DEFAULT_PORT = 9999;

  public LoggingUDPDiscardServer( ) throws SocketException {
    super(DEFAULT_PORT);
  }
```

```
    public void respond(DatagramPacket packet) {

      byte[] data = new byte[packet.getLength( )];
      System.arraycopy(packet.getData( ), 0, data, 0, packet.getLength( ));
      try {
        String s = new String(data, "8859_1");
        System.out.println(packet.getAddress( ) + " at port "
         + packet.getPort( ) + " says " + s);
      }
      catch (java.io.UnsupportedEncodingException ex) {
        // This shouldn't happen
      }

    }

    public static void main(String[] args) {

     try {
       UDPServer erver = new LoggingUDPDiscardServer( );
       server.start( );
     }
     catch (SocketException ex) {
       System.err.println(ex);
     }

    }

  }
```

It isn't much harder to implement an echo server, as Example 13-10 shows. Unlike a stream-based TCP echo server, multiple threads are not required to handle multiple clients.

Example 13-10. A UDP echo server

```
import java.net.*;
import java.io.*;

public class UDPEchoServer extends UDPServer {

  public final static int DEFAULT_PORT = 7;

  public UDPEchoServer( ) throws SocketException {
    super(DEFAULT_PORT);
  }

  public void respond(DatagramPacket packet) {

    try {
      DatagramPacket outgoing = new DatagramPacket(packet.getData( ),
       packet.getLength( ), packet.getAddress( ), packet.getPort( ));
      socket.send(outgoing);
    }
    catch (IOException ex) {
      System.err.println(ex);
    }

  }

  public static void main(String[] args) {

   try {
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
        UDPServer server = new UDPEchoServer( );
        server.start( );
      }
      catch (SocketException ex) {
        System.err.println(ex);
      }

    }

  }
```

A daytime server is only slightly more complex. The server listens for incoming UDP datagrams on port 13. When it detects an incoming datagram, it returns the current date and time at the server as a one-line ASCII string. Example 13-11 demonstrates this.

Example 13-11. The UDP daytime server

```
import java.net.*;
import java.io.*;
import java.util.*;


public class UDPDaytimeServer extends UDPServer {

  public final static int DEFAULT_PORT = 13;

  public UDPDaytimeServer( ) throws SocketException {
    super(DEFAULT_PORT);
  }

  public void respond(DatagramPacket packet) {

    try {
      Date now = new Date( );
      String response = now.toString( ) + "\r\n";
      byte[] data = response.getBytes("ASCII");
      DatagramPacket outgoing = new DatagramPacket(data,
       data.length, packet.getAddress( ), packet.getPort( ));
      socket.send(outgoing);
    }
    catch (IOException ex) {
      System.err.println(ex);
    }

  }

  public static void main(String[] args) {

   try {
      UDPServer server = new UDPDaytimeServer( );
      server.start( );
    }
    catch (SocketException ex) {
      System.err.println(ex);
    }

  }

}
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

13.4.3. A UDP Echo Client

The `UDPPoke` class implemented earlier isn't suitable for all protocols. In particular, protocols that require multiple datagrams require a different implementation. The echo protocol has both TCP and UDP implementations. Implementing the echo protocol with TCP is simple; it's more complex with UDP because you don't have I/O streams or the concept of a connection to work with. A TCP-based echo client can send a message and wait for a response on the same connection. However, a UDP-based echo client has no guarantee that the message it sent was received. Therefore, it cannot simply wait for the response; it needs to be prepared to send and receive data asynchronously.

This behavior is fairly simple to implement using threads, however. One thread can process user input and send it to the echo server, while a second thread accepts input from the server and displays it to the user. The client is divided into three classes: the main `UDPEchoClient` class, the `SenderThread` class, and the `ReceiverThread` class.

The `UDPEchoClient` class should look familiar. It reads a hostname from the command line and converts it to an `InetAddress` object. `UDPEchoClient` uses this object and the default echo port to construct a `SenderThread` object. This constructor can throw a `SocketException`, so the exception must be caught. Then the `SenderThread` starts. The same `DatagramSocket` that the `SenderThread` uses is used to construct a `ReceiverThread`, which is then started. It's important to use the same `DatagramSocket` for both sending and receiving data because the echo server will send the response back to the port the data was sent from. Example 13-12 shows the code for the `UDPEchoClient`.

Example 13-12. The UDPEchoClient class

```
import java.net.*;
import java.io.*;

public class UDPEchoClient {

  public final static int DEFAULT_PORT = 7;

  public static void main(String[] args) {

    String hostname = "localhost";
    int port = DEFAULT_PORT;

    if (args.length > 0) {
      hostname = args[0];
    }

    try {
      InetAddress ia = InetAddress.getByName(hostname);
      Thread sender = new SenderThread(ia, DEFAULT_PORT);
      sender.start( );
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
        Thread receiver = new ReceiverThread(sender.getSocket( ));
        receiver.start( );
      }
      catch (UnknownHostException ex) {
        System.err.println(ex);
      }
      catch (SocketException ex) {
        System.err.println(ex);
      }

    }  // end main

  }
```

The `SenderThread` class reads input from the console a line at a time and sends it to the echo server. It's shown in Example 13-13. The input is provided by `System.in`, but a different client could include an option to read input from a different stream—perhaps opening a `FileInputStream` to read from a file. The three fields of this class define the server to which it sends data, the port on that server, and the `DatagramSocket` that does the sending, all set in the single constructor. The `DatagramSocket` is connected to the remote server to make sure all datagrams received were in fact sent by the right server. It's rather unlikely that some other server on the Internet is going to bombard this particular port with extraneous data, so this is not a big flaw. However, it's a good habit to make sure that the packets you receive come from the right place, especially if security is a concern.

The `run( )` method processes user input a line at a time. To do this, the `BufferedReader userInput` is chained to `System.in`. An infinite loop reads lines of user input. Each line is stored in `theLine`. A period on a line by itself signals the end of user input and breaks out of the loop. Otherwise, the bytes of data are stored in the data array using the `getBytes( )` method from `java.lang.String`. Next, the data array is placed in the payload part of the `DatagramPacket output`, along with information about the server, the port, and the data length. This packet is then sent to its destination by `socket`. This thread then yields to give other threads an opportunity to run.

Example 13-13. The SenderThread class

```
import java.net.*;
import java.io.*;

public class SenderThread extends Thread {

  private InetAddress server;
  private DatagramSocket socket;
  private boolean stopped = false;
  private int port;

  public SenderThread(InetAddress address, int port)
   throws SocketException {
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
        this.server = address;
        this.port = port;
        this.socket = new DatagramSocket( );
        this.socket.connect(server, port);
      }

      public void halt( ) {
        this.stopped = true;
      }

      public DatagramSocket getSocket( ) {
        return this.socket;
      }

      public void run( ) {

        try {
          BufferedReader userInput
           = new BufferedReader(new InputStreamReader(System.in));
          while (true) {
            if (stopped) return;
            String theLine = userInput.readLine( );
            if (theLine.equals(".")) break;
            byte[] data = theLine.getBytes( );
            DatagramPacket output
             = new DatagramPacket(data, data.length, server, port);
            socket.send(output);
            Thread.yield( );
          }
        }  // end try
        catch (IOException ex) {
          System.err.println(ex);
        }

      }  // end run

    }
```

The `ReceiverThread` class shown in Example 13-14 waits for datagrams to arrive from the network. When a datagram is received, it is converted to a `String` and printed on `System.out` for display to the user. A more advanced `EchoClient` could include an option to send the output elsewhere.

This class has two fields. The more important is the `DatagramSocket`, `theSocket`, which must be the same `DatagramSocket` used by the `SenderThread`. Data arrives on the port used by that `DatagramSocket`; any other `DatagramSocket` would not be allowed to connect to the same port. The second field, `stopped`, is a boolean used to halt this thread without invoking the deprecated `stop( )` method.

The `run( )` method is an infinite loop that uses `socket`'s `receive( )` method to wait for incoming datagrams. When an incoming datagram appears, it is converted into a `String` with the same length as the incoming data and printed on `System.out`. As in the input thread, this thread then yields to give other threads an opportunity to execute.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Example 13-14. The ReceiverThread class

```
import java.net.*;
import java.io.*;

class ReceiverThread extends Thread {

  DatagramSocket socket;
  private boolean stopped = false;

  public ReceiverThread(DatagramSocket ds) throws SocketException {
    this.socket = ds;
  }

  public void halt( ) {
    this.stopped = true;
  }

  public void run( ) {

    byte[] buffer = new byte[65507];
    while (true) {
      if (stopped) return;
      DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
      try {
        socket.receive(dp);
        String s = new String(dp.getData( ), 0, dp.getLength( ));
        System.out.println(s);
        Thread.yield( );
      }
      catch (IOException ex) {
        System.err.println(ex);
      }

    }

  }

}
```

You can run the echo client on one machine and connect to the echo server on a second machine to verify that the network is functioning properly between them.

13.5. DatagramChannel

Java 1.4 adds a `DatagramChannel` class for use in non-blocking UDP applications, just as it adds `SocketChannel` and `ServerSocketChannel` for use in non-blocking TCP applications. Like `SocketChannel` and `ServerSocketChannel`, `DatagramChannel` is a subclass of `SelectableChannel` that can be registered with a `Selector`. This is useful in servers where one thread can manage communications with multiple different clients. However, UDP is by its nature much more asynchronous than TCP so the net effect is smaller. In UDP it's always been the case that a single datagram socket can process requests from multiple clients for both input and output. What the `DatagramChannel` class adds is the ability to do this in a non-

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

blocking fashion, so methods return quickly if the network isn't immediately ready to receive or send data.

13.5.1. Using DatagramChannel

`DatagramChannel` is a near-complete alternate abstraction for UDP I/O. You still need to use the `DatagramSocket` class to bind a channel to a port. However, you do not have to use it thereafter, nor do you ever use `DatagramPacket`. Instead, you read and write `ByteBuffers`, just as you do with a `SocketChannel`.

13.5.1.1. Opening a socket

The `java.nio.channels.DatagramChannel` class does not have any public constructors. Instead, you create a new `DatagramChannel` object using the static `open( )` method:

```
public static DatagramChannel open( ) throws IOException
```

For example:

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

This channel is not initially bound to any port. To bind it, you need to access the channel's peer `DatagramSocket` object using the `socket()` method:

```
public abstract DatagramSocket socket( )
```

For example, this binds a channel to port 3141:

```
SocketAddress address = new InetSocketAddress(3141);
DatagramSocket socket = channel.socket( );
socket.bind(address);
```

13.5.1.2. Connecting

Like `DatagramSocket`, a `DatagramChannel` can be connected; that is, it can be configured to only receive datagrams from and send datagrams to one host. This is accomplished with the `connect()` method:

```
public abstract DatagramChannel connect(SocketAddress remote)
                                                        throws IOException
```

However, unlike the `connect( )` method of `SocketChannel`, this method does not actually send or receive any packets across the network because UDP is a connectionless protocol. Thus this method returns fairly quickly, and doesn't block in any meaningful sense. There's no need here for a `finishConnect()` or `isConnectionPending( )` method. There is an `isConnected()` method that returns true if and only if the `DatagramSocket` is connected:

```
public abstract boolean isConnected( )
```

This tells you whether the `DatagramChannel` is limited to one host. Unlike `SocketChannel`, a `DatagramChannel` doesn't have to be connected to transmit or receive data.

Finally, there is a `disconnect()` method that breaks the connection:

```
public abstract DatagramChannel disconnect( ) throws IOException
```

This doesn't really close anything because nothing was really open in the first place. It just allows the channel to once again send and receive data from multiple hosts.

Connected channels may be marginally faster than unconnected channels in sandbox environments such as applets because the virtual machine only needs to check whether the connection is allowed on the initial call to the `connect( )` method, not every time a packet is sent or received. As always, only concern yourself with this if profiling indicates it is a bottleneck.

13.5.1.3. Receiving

The `receive( )` method reads one datagram packet from the channel into a `ByteBuffer`. It returns the address of the host that sent the packet:

```
public abstract SocketAddress receive(ByteBuffer dst) throws IOException
```

If the channel is blocking (the default) this method will not return until a packet has been read. If the channel is non-blocking, this method will immediately return null if no packet is available to read.

If the datagram packet has more data than the buffer can hold, *the extra data is thrown away with no notification of the problem*. You do not receive a `BufferOverflowException` or anything similar. UDP is unreliable, after all. This behavior introduces an additional layer of unreliability into the system. The data can arrive safely from the network and still be lost inside your own program.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

Using this method, we can reimplement the discard server to log the host sending the data as well as the data sent. Example 13-15 demonstrates. It avoids the potential loss of data by using a buffer that's big enough to hold any UDP packet and clearing it before it's used again.

Example 13-15. A UDPDiscardServer based on channels

```
import java.net.*;
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class UDPDiscardServerWithChannels {

  public final static int DEFAULT_PORT = 9;
  public final static int MAX_PACKET_SIZE = 65507;

  public static void main(String[] args) {

    int port = DEFAULT_PORT;
    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception ex) {
    }

    try {
      DatagramChannel channel = DatagramChannel.open( );
      DatagramSocket socket = channel.socket( );
      SocketAddress address = new InetSocketAddress(port);
      socket.bind(address);
      ByteBuffer buffer = ByteBuffer.allocateDirect(MAX_PACKET_SIZE);
      while (true) {
        SocketAddress client = channel.receive(buffer);
        buffer.flip( );
        System.out.print(client + " says ");
        while (buffer.hasRemaining( )) System.out.write(buffer.get( ));
        System.out.println( );
        buffer.clear( );
      } // end while
    }  // end try
    catch (IOException ex) {
      System.err.println(ex);
    }  // end catch

  }  // end main

}
```

13.5.1.4. Sending

The `send( )` method writes one datagram packet into the channel from a `ByteBuffer` to the address specified as the second argument:

```
public abstract int send(ByteBuffer src, SocketAddress target) throws
                                         IOException
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

The source `ByteBuffer` can be reused if you want to send the same data to multiple clients. Just don't forget to rewind it first.

The `send( )` method returns the number of bytes written. This will either be the number of bytes remaining in the output buffer or zero. It is zero if there's not enough room in the network interface's output buffer for the amount of data you're trying to send. Don't overstuff the buffer. If you put more data in the buffer than the network interface can handle, it will never send anything. This method will not fragment the data into multiple packets. It writes everything or nothing.

Example 13-16 demonstrates with a simple echo server based on channels. The `receive( )` method reads a packet, much as it did in Example 13-15. However, this time, rather than logging the packet on `System.out`, it returns the same data to the client that sent it.

Example 13-16. A UDPEchoServer based on channels

```
import java.net.*;
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class UDPEchoServerWithChannels {

  public final static int DEFAULT_PORT = 7;
  public final static int MAX_PACKET_SIZE = 65507;

  public static void main(String[] args) {

    int port = DEFAULT_PORT;
    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception ex) {
    }

    try {
      DatagramChannel channel = DatagramChannel.open( );
      DatagramSocket socket = channel.socket( );
      SocketAddress address = new InetSocketAddress(port);
      socket.bind(address);
      ByteBuffer buffer = ByteBuffer.allocateDirect(MAX_PACKET_SIZE);
      while (true) {
        SocketAddress client = channel.receive(buffer);
        buffer.flip( );
        channel.send(buffer, client);
        buffer.clear( );
      } // end while
    }  // end try
    catch (IOException ex) {
      System.err.println(ex);
    }  // end catch

  }  // end main

}
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

This program is blocking and synchronous. This is much less of a problem for UDP-based protocols than for TCP protocols. The unreliable, packet-based, connectionless nature of UDP means that the server at most has to wait for the local buffer to clear. It does not have to and does not wait for the client to be ready to receive data. There's much less opportunity for one client to get held up behind a slower client.

13.5.1.5. Reading

Besides the special purpose `receive( )` method, `DatagramChannel` has the usual three `read( )` methods:

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

However, these methods can only be used on connected channels. That is, before invoking one of these methods, you must invoke `connect( )` to glue the channel to a particular remote host. This makes them more suitable for use with clients that know who they'll be talking to than for servers that must accept input from multiple hosts at the same time that are normally not known prior to the arrival of the first packet.

Each of these three methods only reads a single datagram packet from the network. As much data from that datagram as possible is stored in the argument `ByteBuffer`(s). Each method returns the number of bytes read or -1 if the channel has been closed. This method may return 0 for any of several reasons, including:

- The channel is non-blocking and no packet was ready.
- A datagram packet contained no data.
- The buffer is full.

As with the `receive( )` method, if the datagram packet has more data than the `ByteBuffer`(s) can hold, *the extra data is thrown away with no notification of the problem*. You do not receive a `BufferOverflowException` or anything similar.

13.5.1.6. Writing

Naturally, `DatagramChannel` has the three write methods common to all writable, scattering channels, which can be used instead of the `send()` method:

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

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

However, these methods can only be used on connected channels; otherwise they don't know where to send the packet. Each of these methods sends a single datagram packet over the connection. None of these methods are guaranteed to write the complete contents of the buffer(s). However, the cursor-based nature of buffers enables you to easily call this method again and again until the buffer is fully drained and the data has been completely sent, possibly using multiple datagram packets. For example:

```
while (buffer.hasRemaining( ) && channel.write(buffer) != -1) ;
```

We can use the read and write methods to implement a simple UDP echo client. On the client side, it's easy to connect before sending. Because packets may be lost in transit (always remember UDP is unreliable), we don't want to tie up the sending while waiting to receive a packet. Thus, we can take advantage of selectors and non-blocking I/O. These work for UDP pretty much exactly like they worked for TCP in Chapter 12. This time, though, rather than sending text data, let's send one hundred ints from 0 to 99. We'll print out the values returned so it will be easy to figure out if any packets are being lost. Example 13-17 demonstrates.

Example 13-17. A UDP echo client based on channels

```
import java.net.*;
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.*;

public class UDPEchoClientWithChannels {

  public  final static int DEFAULT_PORT = 7;
  private final static int LIMIT = 100;

  public static void main(String[] args) {

    int port = DEFAULT_PORT;
    try {
      port = Integer.parseInt(args[1]);
    }
    catch (Exception ex) {
    }

    SocketAddress remote;
    try {
      remote = new InetSocketAddress(args[0], port);
    }
    catch (Exception ex) {
      System.err.println("Usage: java UDPEchoClientWithChannels host [port]");
      return;
    }

    try {
```


Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

```
        DatagramChannel channel = DatagramChannel.open( );
        channel.configureBlocking(false);
        channel.connect(remote);

        Selector selector = Selector.open( );
        channel.register(selector, SelectionKey.OP_READ | SelectionKey.OP_WRITE);

        ByteBuffer buffer = ByteBuffer.allocate(4);
        int n = 0;
        int numbersRead = 0;
        while (true) {
          // wait one minute for a connection
          selector.select(60000);
          Set readyKeys = selector.selectedKeys( );
          if (readyKeys.isEmpty( ) && n == LIMIT) {
            // All packets have been written and it doesn't look like any
            // more are will arrive from the network
            break;
          }
          else {
            Iterator iterator = readyKeys.iterator( );
            while (iterator.hasNext( )) {
              SelectionKey key = (SelectionKey) iterator.next( );
              iterator.remove( );
              if (key.isReadable( )) {
                buffer.clear( );
                channel.read(buffer);
                buffer.flip( );
                int echo = buffer.getInt( );
                System.out.println("Read: " + echo);
                numbersRead++;
              }
              if (key.isWritable( )) {
                buffer.clear( );
                buffer.putInt(n);
                buffer.flip( );
                channel.write(buffer);
                System.out.println("Wrote: " + n);
                n++;
               if (n == LIMIT) {
                  // All packets have been written; switch to read-only mode
                  key.interestOps(SelectionKey.OP_READ);
                } // end if
              }  // end while
            } // end else
          }  // end while

        }  // end while
        System.out.println("Echoed " + numbersRead + " out of " + LIMIT +
                                                                 " sent");
        System.out.println("Success rate: " + 100.0 * numbersRead / LIMIT +
                                                                 "%");

      }  // end try
      catch (IOException ex) {
        System.err.println(ex);
      }  // end catch

    }  // end main

  }
```

There is one major difference between selecting TCP channels and selecting datagram channels. Because datagram channels are truly connectionless (despite the `connect( )` method), you need to notice when the data transfer is complete and shut down. In this example, we assume the data is finished when all packets have been sent and one minute has passed since the last packet was

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004

received. Any expected packets that have not been received by this point are assumed to be lost in the ether.

A typical run produced output like this:

```
Wrote: 0
Read: 0
Wrote: 1
Wrote: 2
Read: 1
Wrote: 3
Read: 2
Wrote: 4
Wrote: 5
Wrote: 6
Wrote: 7
Wrote: 8
Wrote: 9
Wrote: 10
Wrote: 11
Wrote: 12
Wrote: 13
Wrote: 14
Wrote: 15
Wrote: 16
Wrote: 17
Wrote: 18
Wrote: 19
Wrote: 20
Wrote: 21
Wrote: 22
Read: 3
Wrote: 23
...
Wrote: 97
Read: 72
Wrote: 98
Read: 73
Wrote: 99
Read: 75
Read: 76
...
Read: 97
Read: 98
Read: 99
Echoed 92 out of 100 sent
Success rate: 92.0%
```

Connecting to a remote server a couple of miles and seven hops away (according to traceroute), I saw between 90% and 98% of the packets make the round trip.

13.5.1.7. Closing

Just as with regular datagram sockets, a channel should be closed when you're done with it to free up the port and any other resources it may be using:

```
public void close( ) throws IOException
```

Closing an already closed channel has no effect. Attempting to write data to or read data from a closed channel throws an exception. If you're uncertain whether a channel has been closed, check with `isOpen( )`:

```
public boolean isOpen( )
```

This returns `false` if the channel is closed, `true` if it's open.

Java Network Programming, 3rd Edition By Elliotte Rusty Harold ISBN: 0596007213 Publisher: O'Reilly
Print Publication Date: 10/1/2004