Network Programming

Topics in this section include:

- What a socket is
- What you can do with a socket
- The difference between TCP/IP, UDP/IP and Multicast sockets
- How servers and clients communicate over sockets
- How to create a simple server
- How to create a simple client
- How to create a multithreaded server

Introduction

The Internet is all about connecting machines together. One of the most exciting aspects of Java is that it incorporates an easy-to-use, cross-platform model for network communications that makes it possible to learn network programming without years of study. This opens up a whole new class of applications to programmers.

What is a Socket?

Java's socket model is derived from BSD (UNIX) sockets, introduced in the early 1980s for interprocess communication using IP, the Internet Protocol.

The Internet Protocol breaks all communications into *packets*, finite-sized chunks of data which are separately and individually routed from source to destination. IP allows routers, bridges, etc. to drop packets--there is no delivery guarantee. Packet size is limited by the IP protocol to 65535 bytes. Of this, a minimum of 20 bytes is needed for the IP packet header, so there is a maximum of 65515 bytes available for user data in each packet.

Sockets are a means of using IP to communicate between machines, so sockets are one major feature that allows Java to interoperate with legacy systems by simply talking to existing servers using their pre-defined protocol.

Other common protocols are layered on top of the Internet protocol. The ones we discuss in this chapter are the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP). Applications can make use of these two protocols to communicate over the network, commonly by building additional protocols on top of the TCP or UDP base.

TELNET	HTTP	etc.	DNS	NFS	etc.
TCP			UDP		
IP					

Internet Addresses

Internet addresses are manipulated in Java by the use of the `InetAddress` class. `InetAddress` takes care of the Domain Name System (DNS) look-up and reverse look-up; IP addresses can be specified by either the host name or the raw IP address. `InetAddress` provides methods to `getByName()`, `getAllByName()`, `getLocalHost()`, `getAddress()`, etc.

IP Addresses

IP addresses are a 32-bit number, often represented as a "quad" of four 8-bit numbers separated by periods. They are organized into classes (A, B, C, D, and E) which are used to group varying numbers of hosts in a hierarchical numbering scheme.

Class A

1.0.0.0 to 126.255.255.255, inclusive. About 16 million IP addresses in a class A domain.

Class B

128.1.0.0 to 191.254.255.255, inclusive. About 64 thousand IP addresses in a class B domain.

Class C

192.0.1.0 to 223.255.254.255, inclusive. 256 IP addresses in a class C domain.

Class D

224.0.0.1 to 239.255.255.255, inclusive, denote multicast groups.

Class E

240.0.0.0 to 254.255.255.255, inclusive. Reserved for future use.

The IP address 127.0.0.1 is special, and is reserved to represent the loopback or "localhost" address.

Port number

The port number field of an IP packet is specified as a 16-bit unsigned integer. This means that valid port numbers range from 1 through 65535. (Port number 0 is reserved and can't be used).

Java does not have any unsigned data types; Java's `short` data type is 16 bits, but its range is -32768 to 32767 since it is a signed type. Thus, `short` is not large enough to hold a port number, so all classes which use or return a port number must represent the port number as an `int`. In the JDK 1.1, using an `int` with a value greater than 65535 will generate an `IllegalArgumentException`. In the JDK 1.0.2 and earlier, values greater than 65535 are truncated and only the low-order 16 bits are used.

Port numbers 1 through 255 are reserved by IP for well-known services. A well-known service is a service that is widely implemented which resides at a published, "well-known", port. If you connect to port 80 of a host, for instance, you may expect to find an HTTP server. On UNIX machines, ports less than 1024 are privileged and can only be bound by the root user. This is so an arbitrary user on a multi-user system can't impersonate well-known services like TELNET (port 23), creating a security problem. Windows has no such restrictions, but you should program as if it did so that your applications will work cross-platform.

Client/Server Computing

You can use the Java language to communicate with remote file systems using a client/server model. A server listens for connection requests from clients across the network or even from the same machine. Clients know how to connect to the server via an IP address and port number. Upon connection, the server reads the request sent by the client and responds appropriately. In this way, applications can be broken down into specific tasks that are accomplished in separate locations.

The data that is sent back and forth over a socket can be anything you like. Normally, the client sends a request for information or processing to the server, which performs a task or sends data back. You could, for example, place an `SQL` shell on the server and let people talk to it via a simple client "chat" program.

The IP and port number of the server is generally well-known and advertised so the client knows where to find the service. In contrast, the port number on client the side is generally allocated automatically by the kernel.

Many protocols used on the Internet (HTTP for example) are designed to be driven from the command line. They send their requests and responses across the net in plain text. One of the easiest ways to become familiar with network programming and/or specific protocols is to use the TELNET application to "talk" directly to a server from the command line.

User Datagram Protocol (UDP)

UDP provides an unreliable packet delivery system built on top of the IP protocol. As with IP, each packet is an individual, and is handled separately. Because of this, the amount of data that can be sent in a UDP packet is limited to the amount that can be contained in a single IP packet. Thus, a UDP packet can contain at most 65507 bytes (this is the 65535-byte IP packet size minus the minimum IP header of 20 bytes and minus the 8-byte UDP header).

UDP packets can arrive out of order or not at all. No packet has any knowledge of the preceding or following packet. The recipient does not acknowledge packets, so the sender does not know that the transmission was successful. UDP has no provisions for flow control--packets can be received faster than they can be used. We call this type of communication *connectionless* because the packets have no relationship to each other and because there is no state maintained.

The destination IP address and port number is encapsulated in each UDP packet. These two numbers together uniquely identify the recipient and are used by the underlying operating system to deliver the packet to a specific process (application).

One way to think of UDP is by analogy to communications via a letter. You write the letter (this is the data you are sending); put the letter inside an envelope (the UDP packet); address the envelope (using an IP address and a port number); put your return address on the envelope (your local IP address and port number); and then you send the letter.

Like a real letter, you have no way of knowing whether a UDP packet was received. If you send a second letter one day after the first, the second one may be received *before* the first. Or, the second one may never be received.

So why use UDP if it unreliable? Two reasons: speed and overhead. UDP packets have almost no overhead--you simply send them then forget about them. And they are fast, since there is no acknowledgement required for each packet. Keep in mind the degree of unreliability we are talking about. For all practical purposes, an Ethernet breaks down if more than about 2 percent of all packets are lost. So, when we say unreliable, the worst-case loss is very small.

UDP is appropriate for the many network services that do not require guaranteed delivery. An example of this is a network time service. Consider a time daemon that issues a UDP packet every second so computers on the LAN can synchronize their clocks. If a packet is lost, it's no big deal--the next one will be by in another second and will contain all necessary information to accomplish the task.

Another common use of UDP is in networked, multi-user games, where a player's position is sent periodically. Again, if one position update is lost, the next one will contain all the required information.

A broad class of applications is built on top of UDP using streaming protocols. With streaming protocols, receiving data in real-time is far more important than guaranteeing delivery. Examples of real-time streaming protocols are RealAudio and RealVideo which respectively deliver real-time streaming audio and video over the Internet. The reason a streaming protocol is desired in these cases is because if an audio or video packet is lost, it is much better for the client to see this as noise or "drop-out" in the sound or picture rather than having a long pause while the client software stops the playback, requests the missing data from the server. That would result in a very choppy, bursty playback which most people find unacceptable, and which would place a heavy demand on the server.

Creating UDP Servers

To create a server with UDP, do the following:

1. Create a `DatagramSocket` attached to a port.

```
int port = 1234;
DatagramSocket socket = new DatagramSocket(port);
```

2. Allocate space to hold the incoming packet, and create an instance of `DatagramPacket` to hold the incoming data.

```
byte[] buffer = new byte[1024];
DatagramPacket packet =
       new DatagramPacket(buffer, buffer.length);
```

3. Block until a packet is received, then extract the information you need from the packet.

```
// Block on receive()
socket.receive(packet);

// Find out where packet came from
// so we can reply to the same host/port
InetAddress remoteHost = packet.getAddress();
int         remotePort = packet.getPort();

// Extract the packet data
byte[]      data       = packet.getData();
```

The server can now process the data it has received from the client, and issue an appropriate reply in response to the client's request.

Creating UDP Clients

Writing code for a UDP client is similar to what we did for a server. Again, we need a `DatagramSocket` and a `DatagramPacket`. The only real difference is that we must specify the destination address with each packet, so the form of the `DatagramPacket` constructor used here specifies the destination host and port number. Then, of course, we initially send packets instead of receiving.

1. First allocate space to hold the data we are sending and create an instance of `DatagramPacket` to hold the data.

```
byte[] buffer = new byte[1024];
int port = 1234;
InetAddress host =
  InetAddress.getByName("magelang.com");
DatagramPacket packet =
  new DatagramPacket(buffer, buffer.length,
                     host,   port);
```

2. Create a `DatagramSocket` and send the packet using this socket.

```
DatagramSocket socket = new DatagramSocket();
socket.send(packet);
```

The `DatagramSocket` constructor that takes no arguments will allocate a free local port to use. You can find out what local port number has been allocated for your socket, along with other information about your socket if needed.

```
// Find out where we are sending from
InetAddress localHostname = socket.getLocalAddress();
int         localPort     = socket.getLocalPort();
```

The client then waits for a reply from the server. Many protocols require the server to reply to the host and port number that the client used, so the client can now invoke `socket.receive()` to wait for information from the server.

Transmission Control Protocol (TCP)

We saw in the previous section that UDP provides an unreliable packet delivery system--each packet is an individual, and is handled separately. Packets can arrive out of order or not at all. The recipient does not acknowledge them, so the sender does not know that the transmission was successful. There are no provisions for flow control--packets can be received faster than they can be used. Packet size was limited by the underlying IP protocol.

TCP, Transmission Control Protocol, was designed to address these problems. TCP packets are lost occasionally, just like UDP packets. The difference is that the TCP protocol takes care of requesting retransmits to ensure that all packets show up, and tracks packet sequence numbers to be sure that they are delivered in the correct order. While UDP required us to send packets of byte arrays, with TCP we can use streams along with the standard Java file I/O mechanism.

Unlike UDP, the Transmission Control Protocol, TCP, establishes a *connection* between the two endpoints. Negotiation is performed to establish a socket, and the socket remains open throughout the duration of the communications. The recipient acknowledges each packet, and packet retransmissions are performed by the protocol if packets are missed or arrive out of order. In this way TCP can allow an application to send as much data as it desires and not be subject to the IP packet size limit. TCP is responsible for breaking the data into packets, buffering the data, resending lost or out of order packets, acknowledging receipt, and controlling rate of data flow by telling the sender to speed up or slow down so that the application never receives more than it can handle.

Again, unlike UDP, the destination host and port number is not sufficient to identify a recipient of a TCP connection. There are five distinct elements that make a TCP connection unique:

- IP address of the server
- IP address of the client
- Port number of the server
- Port number of the client

- Protocol (UDP, TCP/IP, etc...)

where each requested client socket is assigned a unique port number whereas the server port number is always the same. If *any* of these numbers is different, the socket is different. A server can thus listen to one and only one port, and talk to multiple clients at the same time.

So a TCP connection is more like a telephone connection than a letter; you need to know not only the phone number (IP address), but since the phone may be shared by many people at that location, you also need the name of the user you want to talk to at the other end (port number). The analogy can be taken a little further. If you don't hear what the other person has said, a simple request ("What?") will prompt the other end to resend or repeat the phrase. And, the connection remains open until someone hangs up.

Creating TCP Servers

To create a TCP server, do the following:

1. Create a `ServerSocket` attached to a port number.

   ```
   ServerSocket server = new ServerSocket(port);
   ```

2. Wait for connections from clients requesting connections to that port.

   ```
   // Block on accept()
   Socket channel = server.accept();
   ```

   You'll get a `Socket` object as a result of the connection.

3. Get input and output streams associated with the socket.

   ```
   out    = new PrintWriter
       (channel.getOutputStream());
   reader = new InputStreamReader
       (channel.getInputStream());
   in     = new BufferedReader (reader);
   ```

   Now you can read and write to the socket, thus, communicating with the client.

   ```
   String data = in.readLine();
   out.println("Hey! I heard you over this socket!");
   ```

When a server invokes the `accept()` method of the `ServerSocket` instance, the main server thread blocks until a client connects to the server; it is then prevented from accepting further client connections until the server has processed the client's request. This is known as an *iterative* server, since the main server method handles each client request in its entirety before moving on to the next request. Iterative servers are good when the requests take a known, short period of time. For example, requesting the current day and time from a time-of-day server.

Creating TCP Clients

To create a TCP client, do the following:

1. Create a `Socket` object attached to a remote host, port.

   ```
   Socket client = new Socket(host, port);
   ```

   When the constructor returns, you have a connection.

2. Get input and output streams associated with the socket.

   ```
   out    = new PrintWriter
       (client.getOutputStream());
   reader = new InputStreamReader
       (client.getInputStream());
   in     = new BufferedReader (reader);
   ```

   Now you can read and write to the socket, thus, communicating with the server.

   ```
   out.println("Watson!" + "Come here...I need you!");
   String data = in.readLine();
   ```

Notice that servers *must* be launched before clients

Servers Handling Multiple Clients

While iterative servers are simple, they are severely limited in their performance since only one client at a time can be handled. In particular, when the amount of processing needed to handle a client request is unknown *a priori* (i.e., depends on the request itself), iterative servers are unacceptable. A TELNET session, for instance, can take an unbounded amount of time, so the server should not wait until that session is over before accepting new clients.

To overcome this problem with iterative servers, a separate thread can handle each client session, allowing the server to deal with multiple clients simultaneously. This is known as a *concurrent* server--the main server method launches a thread to handle each client request, then continues to listen for additional clients.

Here is an example server method called `acceptClients()` that listens for connections and handles multiple clients, each in a separate thread:

```
public void acceptClients() {
  Socket socket = null;
  while (true) {
    // For each connection, start a new handler
    // in its own thread and keep on listening
    try {
      socket = server.accept();
      PrintWriter out =
        new PrintWriter(socket.getOutputStream());
      InputStreamReader reader =
        new InputStreamReader(socket.getInputStream());
      BufferedReader in = new BufferedReader(reader);
      // Create new client handler
      // and launch thread on it.
      Handler h = new Handler(in, out);
      (new Thread(h)).start();
    }
    catch (IOException e) {
      System.err.println
        ("Error creating socket connection");
      System.exit(1);
    }
  }
}
```

An enhancement to this server could be to pre-allocate a collection of `Thread` objects to be used for handling client requests. This will allow the server to handle the requests quicker by utilizing a thread from the pool rather than dynamically allocating one. In addition, it will allow the server to manage the client connections for the purpose of limiting number of clients or setting timeout values for the client connections.

Multithreaded Clients

In the previous section we learned why we needed to use threads in our server if we wanted to simultaneously deal with multiple clients. It is not so clear why we would need more than one thread for our *client*, but we generally do. The reason for this is that the I/O operations we perform on a socket generally block. So the client can read or write to the socket, but not both at the same time. This poses a problem when the client does not know ahead of time how much data the server is going to send back in response to the client's request. How many times should the client call `readLine()`?

Having one thread to read and one thread to write, we can ensure that we never enter into a situation where the client fails to read the entire server response, or where the client is stuck waiting for too much data from the server.

Instead of creating two threads within your client, a more common scenario would be to take advantage of the fact that the AWT handles user input in a separate thread; you can receive user input and send data over the socket from an AWT event handling method, while your client class receives data in its `run()` method.

Sending Objects

TCP is a stream protocol; we can do anything with socket streams that we can with file streams. One way we can greatly simplify communications between clients and servers written in Java is to use an `ObjectInputStream` and `ObjectOutputStream` combination to read and write objects across the socket.

This is a higher level of abstraction--we don't have to write `int` or `float` data types, but we can send a whole object. For example:

1. First, define an object to send. We'll define a class called `Message` to encapsulate our communications.

```
public class Message implements Serializable {
  private int    senderID;
  private String messageText;

  public Message(int id, String text) {
      senderID    = id;
      messageText = text;
  }
  public String getText() {
    return messageText;
  }
}
```

2. Next, instantiate the object; wrap the socket's streams in object streams; and then send the message across the socket.

```
Message sayhey = new Message("123456789", "Hello");

out = new ObjectOutputStream(socket.getOutputStream());
in  = new ObjectInputStream(socket.getInputStream());

out.writeObject(sayhey);
```

3. On the other side of the socket, the message can be retrieved and used by invoking methods on the returned object.

```
Message messageObject = (Message) in.readObject();
String messageText = messageObject.getText();
```

Multicast Protocol

TCP and UDP are both *unicast* protocols; there is one sender and one receiver. Multicast packets are a special type of UDP packets. But while UDP packets have only one destination and only one receiver, multicast packets can have an *arbitrary* number of receivers.

Multicast is quite distinct from broadcast; with broadcast packets, *every* host on the network receives the packet. With multicast, only those hosts that have registered an interest in receiving the packet get it.

This is similar to the way an `AWTEvent` and it's listeners behave in the AWT. In the same way that an `AWTEvent` is sent only to registered listeners, a multicast packet is sent only to members of the *multicast group*. `AWTEvents`, however, are unicast, and must be sent individually to each listener--if there are two listeners, two events are sent. With a `MulticastSocket`, only one is sent and it is received by many.

As you might guess, `MulticastSocket` is a subclass of `DatagramSocket` which has the extended ability to join and leave multicast groups. A multicast group consists of both a multicast address and a port number. The only difference between UDP and multicast in this respect is that multicast groups are represented by Class D internet addresses. Just as there are well-know ports for network services, there are reserved, well-known multicast groups for multicast network services.

When an application subscribes to a multicast group (host/port), it receives datagrams sent by other hosts to that group, as do all other members of the group. Multiple applications may subscribe to a multicast group and port concurrently, and they will all receive group datagrams.

When an application sends a message to a multicast group, all subscribing recipients to that host and port receive the message (within the time-to-live range of the packet, see below). The application needn't be a member of the multicast group to send messages to it.

A Multicast Application

Because of the nature of multicast, it doesn't fit well into the client/server model. A "server" doesn't know how many clients (if any!) are subscribed to the multicast group. And the whole idea of multicast is to send only one packet, *not* to send packets tailored to each individual client. If the server needs to respond to individual client requests, then it probably is not appropriate to use multicast. In practice, multicast applications are either one-way, as with streaming audio, or peer-to-peer, as with a multi-user network game. Because there are multiple recipients, a "client" can't use an arbitrary port number--the server can't respond to each client individually at the client's preferred port. Rather, the client must bind to a well-known port in much the same way that the server binds to a well-known port.

To use a `MulticastSocket` to communicate, we perform the following steps:

1. First, create a new instance of `MulticastSocket`. (If we wish to receive multicast packets over this socket, we must be sure to join a group and set the port to be a know port number for communication within that group.)

```
InetAddress group = InetAddress.getByName
    ("224.1.2.3");
// Binds port number on our side
MulticastSocket socket = new MulticastSocket(port);
// Binds multicast address on our side
// (Only needed to receive packets, not to send)
socket.joinGroup(group);
```

2. Next, we create a `DatagramPacket` and send it over the socket, just as we did in the UDP section. This packet will be sent to every host which has joined this group.

```
byte[] buffer = new byte[1024];
// Addressed to the group, port
DatagramPacket packet =
    new DatagramPacket(buffer, buffer.length,
                       group,  port);
socket.send(packet);
```

3. If we wish to receive multicast packets over this same socket,

```
byte[] response = new byte[1024];
DatagramPacket packet =
    new DatagramPacket(response, response.length);
socket.receive(packet);
```

4. When we no longer wish to receive packets from a group, we must leave the group.

```
socket.leaveGroup(group);
```

Time-to-live, or TTL, is an 8-bit unsigned byte that is assigned to each multicast packet. For each route, the TTL value stored in the packet is decremented. When it reaches zero, the packet is discarded. So setting the TTL value of a packet (via `setTTL()`) controls how many hops it can travel. This is especially important when programming multicast applications, because you don't want to send your packets to every host in the world--you want to restrict your packets to certain places. For example, you might want to make sure that the packets are restricted to you local LAN, so you set TTL to be one or two. Or, if you are writing a multicast application for a WAN, you might need to set TTL to be 10 or more.

All hosts do not support multicast packets. Multicast-capable routers needed in order to properly route multicast packets over the network, and your host's TCP protocol must recognize multicast addresses. Currently, there is only limited support for multicast across the Internet, but you can use multicast on most LANs.

Further Reading and References

There is really only one book you need if you wish to learn more about the Internet Protocol, TCP, UDP, and network programming in general:

[Com91] Internetworking with TCP/IP (Volumes I-III) by Douglas E. Comer, Prentice Hall, Englewood Cliffs, NJ, 1991.

If you want to focus more on the applications issues, especially in a UNIX environment, you should also look at:

[Ste97] UNIX Network Programming by W. Richard Stevens, Prentice Hall, Englewood Cliffs, NJ, 1997.

Neither of these books has anything to do with Java, but either presents a solid base of understanding for network programming. Since Java's socket model derives directly from BSD UNIX, if you read the material in these books, the `java.net` package will be easily understandable.

[*MML: 0.995a*]
[*Version: $Id: //depot/main/src/edu/modules/Sockets/sockets.mml#3 $*]