CHAPTER 6

Transmission Control Protocol

The Transmission Control Protocol (TCP) is a stream-based method of network communication that is far different from any discussed previously. This chapter discusses TCP streams and how they operate under Java.

6.1 Overview

TCP provides an interface to network communications that is radically different from the User Datagram Protocol (UDP) discussed in Chapter 5. The properties of TCP make it highly attractive to network programmers, as it simplifies network communication by removing many of the obstacles of UDP, such as ordering of packets and packet loss. While UDP is concerned with the transmission of packets of data, TCP focuses instead on establishing a network connection, through which a stream of bytes may be sent and received.

In Chapter 5 we saw that packets may be sent through a network using various paths and may arrive at different times. This benefits performance and robustness, as the loss of a single packet doesn't necessarily disrupt the transmission of other packets. Nonetheless, such a system creates extra work for programmers who need to guarantee delivery of data. TCP eliminates this extra work by guaranteeing delivery and order, providing for a reliable byte communication stream between client and server that supports two-way communication. It establishes a "virtual connection" between two machines, through which streams of data may be sent (see Figure 6-1).

TCP uses a lower-level communications protocol, the Internet Protocol (IP), to establish the connection between machines. This connection provides an interface that allows streams of bytes to be sent and received, and transparently converts the data into IP datagram packets. A common problem with

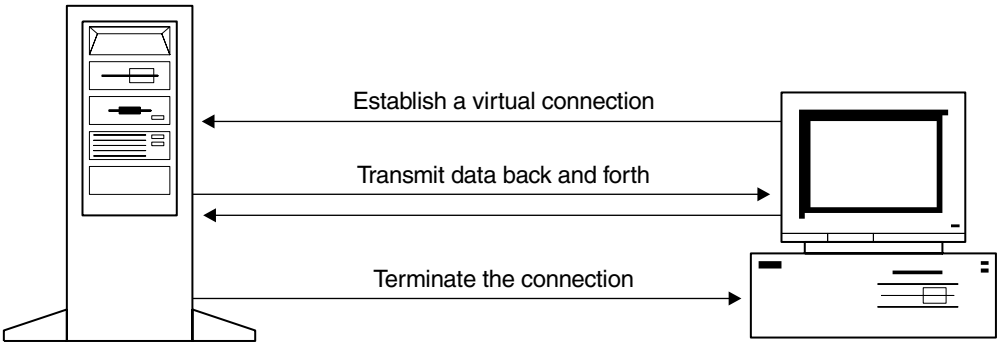

Figure 6-1 TCP establishes a virtual connection to transmit data.

datagrams, as we saw in Chapter 5, is that they do not guarantee that packets arrive at their destination. TCP takes care of this problem. It provides guaranteed delivery of bytes of data. Of course, it's always possible that network errors will prevent delivery, but TCP handles the implementation issues such as resending packets, and alerts the programmer only in serious cases such as if there is no route to a network host or if a connection is lost.

The virtual connection between two machines is represented by a socket. Sockets, introduced in Chapter 5, allow data to be sent and received; there are substantial differences between a UDP socket and a TCP socket, however. First, TCP sockets are connected to a single machine, whereas UDP sockets may transmit or receive data from multiple machines. Second, UDP sockets only send and receive packets of data, whereas TCP allows transmission of data through byte streams (represented as an `InputStream` and `OutputStream`). They are converted into datagram packets for transmission over the network, without requiring the programmer to intervene (as shown in Figure 6-2).

6.1.1 Advantages of TCP over UDP

The many advantages to using TCP over UDP are briefly summarized below.

6.1.1.1 Automatic Error Control

Data transmission over TCP streams is more dependable than transmission of packets of information via UDP. Under TCP, data packets sent through a virtual connection include a checksum to ensure that they have not been corrupted, just like UDP. However, delivery of data is guaranteed by the TCP—data packets lost in transit are retransmitted.

You may be wondering just how this is achieved—after all, IP and UDP do not guarantee delivery; neither do they give any warning when datagram packets are dropped. Whenever a collection of data is sent by TCP using data-

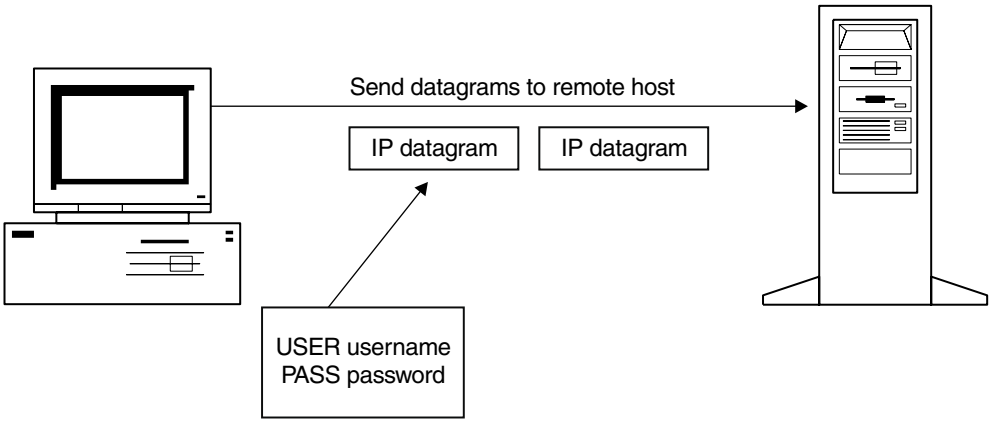

Figure 6-2 TCP deals with streams of data such as protocol commands, but converts streams into IP datagrams for transport over the network.

grams, a timer is started. Recall our UDP examples from Chapter 5, in which the `DatagramSocket.setSoTimeout` method was used to start a timer for a `receive()` operation. In TCP, if the recipient sends an acknowledgment, the timer is disabled. But if an acknowledgment isn't received before the time runs out, the packet is retransmitted. This means that any data written to a TCP socket will reach the other side without the need for further intervention by programmers (barring some catastrophe that causes an entire network to go down). All of the code for error control is handled by TCP.

6.1.1.2 Reliability

Since the data sent between two machines participating in a TCP connection is transmitted by IP datagrams, the datagram packets will frequently arrive out of order. This would throw for a loop any program reading information from a TCP socket, as the order of the byte stream would be disrupted and frequently unreliable. Fortunately, issues such as ordering are handled by TCP—each datagram packet contains a sequence number that is used to order data. Later packets arriving before earlier packets will be held in a queue until an ordered sequence of data is available. The data will then be passed to the application through the interface of the socket.

6.1.1.3 Ease of Use

While storing information in datagram packets is certainly not beyond the reach of programmers, it doesn't lead to the most efficient way of communication between computers. There's added complexity, and it can be argued that the task of designing and creating software within a deadline provides complexity

enough for programmers. Developers typically welcome anything that can reduce the complexity of software development, and the TCP does just this. TCP allows the programmer to think in a completely different way, one that is much more streamlined. Rather than being packaged into discrete units (datagram packets), the data is instead treated as a continuous stream, like the I/O streams the reader is by now familiar with. TCP sockets continue the tradition of Unix programming, in which communication is treated in the same way as file input and output. The mechanism is the same whether the developer is writing to a network socket, a communications pipe, a data structure, the user console, or a file. This also applies, of course, to reading information. This makes communicating via TCP sockets far simpler than communicating via datagram packets.

6.1.2 Communication between Applications Using Ports

It is clear that there are significant differences between TCP and UDP, but there is also an important similarity between these two protocols. Both share the concept of a communications port, which distinguishes one application from another. Many services and clients run on the same port, and it would be impossible to sort out which one was which without distributing them by port number. When a TCP socket establishes a connection to another machine, it requires two very important pieces of information to connect to the remote end—the IP address of the machine and the port number. In addition, a local IP address and port number will be bound to it, so that the remote machine can identify which application established the connection (as illustrated in Figure 6–3). After all, you wouldn't want your e-mail to be accessible by another user running software on the same system.

Ports in TCP are just like ports in UDP—they are represented by a number in the range 1–65535. Ports below 1024 are restricted to use by well-known services such as HTTP, FTP, SMTP, POP3, and telnet. Table 6-1 lists a few of the well-known services and their associated port numbers.

6.1.3 Socket Operations

TCP sockets can perform a variety of operations. They can:

- Establish a connection to a remote host
- Send data to a remote host
- Receive data from a remote host
- Close a connection

In addition, there is a special type of socket that provides a service that will bind to a specific port number. This type of socket is normally used only in servers, and can perform the following operations:

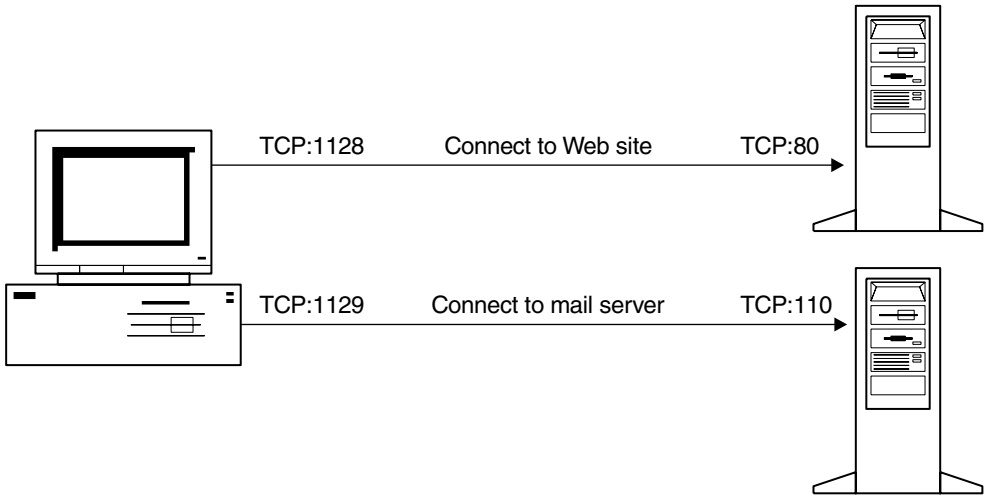

Figure 6-3 Local ports identify the application establishing a connection from other programs, allowing multiple TCP applications to run on the same machine.

Table 6-1 Protocols and Their Associated Ports

Well-Known Services	Service Port
Telnet	23
Simple Mail Transfer Protocol	25
HyperText Transfer Protocol	80
Post Office Protocol 3	110

- Bind to a local port
- Accept incoming connections from remote hosts
- Unbind from a local port

These two sockets are grouped into different categories, and are used by either a client or a server (since some clients may also be acting as servers, and some servers as clients). However, it is normal practice for the role of client and server to be separate.

6.2 TCP and the Client/Server Paradigm

In network programming (and often in other forms of communication, such as database programming), applications that use sockets are divided into two categories, the client and the server. You are probably familiar with the phrase

client/server programming, although the exact meaning of the phrase may be unclear to you. This paradigm is the subject of the discussion below.

6.2.1 The Client/Server Paradigm

The client/server paradigm divides software into two categories, clients and servers. A client is software that initiates a connection and sends requests, whereas a server is software that listens for connections and processes requests. In the context of UDP programming, no actual connection is established, and UDP applications may both initiate and receive requests on the same socket. In the context of TCP, where connections are established between machines, the client/server paradigm is much more relevant.

When software acts as a client, or as a server, it has a rigidly defined role that fits easily into a familiar mental model. Either the software is initiating requests, or it is processing them. Switching between these roles makes for a more complex system. Even if switching is permitted, at any given time one software program must be the client and one software program must be the server. If they both try to be clients at the same time, no server exists to process the requests!

The client/server paradigm is an important theoretical concept that is widely used in practical applications. There are other communications models as well, such as peer to peer, in which either party may initiate communication. However, the client/server concept is a popular choice due to its simplicity and is used in most network programming.

6.2.2 Network Clients

Network clients initiate connections and usually take charge of network transactions. The server is there to fulfill the requests of the client—a client does not fulfill the requests of a server. Although the client is in control, some power still resides in the server, of course. A client can tell a server to delete all files on the local file system, but the server isn't necessarily compelled to carry out that action (thankfully!).

The network client speaks to the server using an agreed-upon standard for communication, the network protocol. For example, an HTTP client uses a set of commands different from a mail client, and has a completely different purpose. Connecting an HTTP client to a mail server, or a mail client to an HTTP server, will result not only in an error message but in an error message that the client will not understand. For this reason, as part of the protocol specification, a port number is used so that the client can locate the server. A Web server typically runs on port 80, and while some servers can run on nonstandard

ports, the convention for a URL is not to list a port, as it is assumed that port 80 is used. For more information on ports, see Section 6.1.2.

6.2.3 Network Servers

The role of the network server is to bind to a specific port (which is used by the client to locate the server), and to listen for new connections. While the client is temporary, and runs only when the user chooses, the server must run continually (even if no clients are actually connected) in the hope that someone, at some time, will want its services. The server is often referred to as a daemon process, to use Unix parlance. It runs indefinitely, and is normally automatically started when the host computer of the server is started. So the server waits, and waits, and waits, until a client establishes a connection to the server port. Some servers can handle only a single connection at a time, while others can handle many connections concurrently, through the use of threads. Multi-threaded programming is discussed in depth in Chapter 7.

When a connection is being processed, the server is submissive. It waits for the client to send requests, and dutifully processes them (though the server is free to respond with an error message, particularly if the request violates some important precept of the protocol or presents a security risk). Some protocols, like HTTP/1.0, normally allow only one request per connection, whereas others, such as POP3, support a sequence of requests. Servers will answer the client request by sending either a response or an error message—the format of which varies from protocol to protocol. Learning a network protocol (when writing either a client or a server) is a little like learning a new language, as the syntax changes. Typically, though, the number of commands is much smaller, making things a little easier. The behavior of the server is determined in part by the protocol and in part by the developer. (Some commands may be optional, and are not always supported by server implementations.)

6.3 TCP Sockets and Java

Java offers good support for TCP sockets, in the form of two socket classes, `java.net.Socket` and `java.net.ServerSocket`. When writing client software that connects to an existing service, the `Socket` class should be used. When writing server software that binds to a local port in order to provide a service, the `ServerSocket` class should be employed. This is different from the way a `DatagramSocket` works with UDP—the function of connecting to servers, and the function of accepting data from clients, is split into a separate class under TCP.

6.4 Socket Class

The `Socket` class represents client sockets, and is a communication channel between two TCP communications ports belonging to one or two machines. A socket may connect to a port on the local system, avoiding the need for a second machine, but most network software will usually involve two machines. TCP sockets can't communicate with more than two machines, however. If this functionality is required, a client application should establish multiple socket connections, one for each machine.

Constructors

There are several constructors for the `java.net.Socket` class. Two constructors, which allowed a boolean parameter to specify whether UDP or TCP sockets were to be used, have been deprecated. These constructors should not be used and are not listed here—if UDP functionality is required, use a `DatagramSocket` (covered in Chapter 5).

The easiest way to create a socket is to specify the hostname of the machine and the port of the service. For example, to connect to a Web server on port 80, the following code might be used:

```
try
{
       // Connect to the specified host and port
       Socket mySocket = new Socket ( "www.awl.com", 80);

       // ......
}
catch (Exception e)
{
       System.err.println ("Err - " + e);
}
```

However, a wide range of constructors is available, for different situations. Unless otherwise specified, all constructors are public.

- `protected Socket ()`—creates an unconnected socket using the default implementation provided by the current socket factory. Developers should not normally use this method, as it does not allow a hostname or port to be specified.
- `Socket (InetAddress address, int port)` throws `java.io.IOException`, `java.lang.SecurityException`—creates a socket connected to the specified IP address and port. If a connection cannot be established, or if connecting to that host violates a security restriction (such as when an applet tries to connect to a machine other than the machine from which it was loaded), an exception is thrown.

- `Socket (InetAddress address, int port, InetAddress localAddress, int localPort)` throws `java.io.IOException, java.lang.SecurityException`—creates a socket connected to the specified address and port, and is bound to the specified local address and local port. By default, a free port is used, but this method allows you to specify a specific port number, as well as a specific address, in the case of multihomed hosts (i.e., a machine where the localhost is known by two or more IP addresses).
- `protected Socket (SocketImpl implementation)`—creates an unconnected socket using the specified socket implementation. Developers should not normally use this method, as it does not allow a hostname or port to be specified.
- `Socket (String host, int port)` throws `java.net.UnknownHostException, java.io.IOException, java.lang.SecurityException`—creates a socket connected to the specified host and port. This method allows a string to be specified, rather than an `InetAddress`. If the hostname could not be resolved, a connection could not be established, or a security restriction is violated, an exception is thrown.
- `Socket (String host, int port, InetAddress localAddress, int localPort)` throws `java.net.UnknownHostException, java.io.IOException, java.lang.SecurityException`—creates a socket connected to the specified host and port, and bound to the specified local port and address. This allows a hostname to be specified as a string, and not an `InetAddress` instance, as well as allowing a specific local address and port to be bound to. These local parameters are useful for multihomed hosts (i.e., a machine where the localhost is known by two or more IP addresses). If the hostname can't be resolved, a connection cannot be established, or a security restriction is violated, an exception is thrown.

6.4.1 Creating a Socket

Under normal circumstances, a socket is connected to a machine and port when it is created. Although there is a blank constructor that does not require a hostname or port, it is protected and can't be called from normal applications. Furthermore, there isn't a `connect()` method that allows you to specify these details at a later point in time, so under normal circumstances the socket will be connected when created. If the network is fine, the call to a socket constructor will return as soon as a connection is established, but if the remote machine is not responding, the constructor method may block for an indefinite amount of time. This varies from system to system, depending on a variety of

factors such as the operating system being used and the default network timeout (some machines on a local intranet, for example, seem to respond faster than some Internet machines, depending on network settings). You can't ever guarantee how long a socket may block for, but this is abnormal behavior and won't happen frequently. Nonetheless, in mission-critical systems it may be appropriate to place such calls in a second thread, to prevent an application from stalling.

NOTE: At a lower level, sockets are produced by a socket factory, which is a special class responsible for creating the appropriate socket implementation. Under normal circumstances, a standard `java.net.Socket` will be produced, but in special situations, such as special networking environments in which custom sockets are used (for example, to break through a firewall by using a special proxy server), socket factories may actually return a socket subclass. The details of socket factories are best left to experienced developers who are familiar with the intricacies of Java networking and have a definite purpose for creating custom sockets and socket factories. For more information on this topic, consult the Java API documentation for the `java.net.SocketFactory` and `java.net.SocketImplFactory` class.

6.4.2 Using a Socket

Sockets can perform a variety of tasks, such as reading information, sending data, closing a connection, and setting socket options. In addition, the following methods are provided to obtain information about a socket, such as address and port locations:

Methods

- `void close()` throws `java.io.IOException`—closes the socket connection. Closing a connect may or may not allow remaining data to be sent, depending on the value of the `SO_LINGER` socket option. Developers are advised to flush any output streams before closing a socket connection.
- `InetAddress getInetAddress()`—returns the address of the remote machine that is connected to the socket.
- `InputStream getInputStream()` throws `java.io.IOException`—returns an input stream, which reads from the application this socket is connected to.
- `OutputStream getOutputStream()` throws `java.io.IOException`—returns an output stream, which writes to the application that this socket is connected to.

- `boolean getKeepAlive()` throws `java.net.SocketException`—returns the state of the `SO_KEEPALIVE` socket option.
- `InetAddress getLocalAddress()`—returns the local address associated with the socket (useful in the case of multihomed machines).
- `int getLocalPort()`—returns the port number that the socket is bound to on the local machine.
- `int getPort()`—returns the port number of the remote service to which the socket is connected.
- `int getReceiveBufferSize()` throws `java.net.SocketException`—returns the receive buffer size used by the socket, determined by the value of the `SO_RCVBUF` socket option.
- `int getSendBufferSize()` throws `java.net.SocketException`—returns the send buffer size used by the socket, determined by the value of the `SO_SNDBUF` socket option.
- `int getSoLinger()` throws `java.net.SocketException`—returns the value of the `SO_LINGER` socket option, which controls how long unsent data will be queued when a connection is terminated.
- `int getSoTimeout()` throws `java.net.SocketException`—returns the value of the `SO_TIMEOUT` socket option, which controls how many milliseconds a read operation will block for. If a value of 0 is returned, the timer is disabled and a thread will block indefinitely (until data is available or the stream is terminated).
- `boolean getTcpNoDelay()` throws `java.net.SocketException`—returns "true" if the `TCP_NODELAY` socket option is set, which controls whether Nagle's algorithm (discussed in Section 6.4.4.5) is enabled.
- `void setKeepAlive(boolean onFlag)` throws `java.net.SocketException`—enables or disables the `SO_KEEPALIVE` socket option.
- `void setReceiveBufferSize(int size)` throws `java.net.SocketException`—modifies the value of the `SO_RCVBUF` socket option, which recommends a buffer size for the operating system's network code to use for receiving incoming data. Not every system will support this functionality or allows absolute control over this feature. If you want to buffer incoming data, you're advised to instead use a `BufferedInputStream` or a `BufferedReader`.
- `void setSendBufferSize(int size)` throws `java.net.SocketException`—modifies the value of the `SO_SNDBUF` socket option, which recommends a buffer size for the operating system's network code to use for sending incoming data. Not every system will support this functionality or allows absolute control over this feature. If you want to buffer incoming data, you're advised to instead use a `BufferedOutputStream` or a `BufferedWriter`.

- `static void setSocketImplFactory (SocketImplFactory factory)` throws `java.net.SocketException, java.io.IOException, java.lang.SecurityException` —assigns a socket implementation factory for the JVM, which may already exist, or may violate security restrictions, either of which causes an exception to be thrown. Only one factory can be specified, and this factory will be used whenever a socket is created.
- `void setSoLinger(boolean onFlag, int duration)` throws `java.net.SocketException, java.lang.IllegalArgumentException`—enables or disables the `SO_LINGER` socket option (according to the value of the `onFlag` boolean parameter), and specifies a duration in seconds. If a negative value is specified, an exception is thrown.
- `void setSoTimeout(int duration)` throws `java.net.SocketException`—modifies the value of the `SO_TIMEOUT` socket option, which controls how long (in milliseconds) a read operation will block. A value of zero disables timeouts, and blocks indefinitely. If a timeout does occur, a `java.io.IOInterruptedException` is thrown whenever a read operation occurs on the socket's input stream. This is distinct from the internal TCP timer, which triggers a resend of unacknowledged datagram packets (see Section 6.1.1.1 on error control).
- `void setTcpNoDelay(boolean onFlag)` throws `java.net.SocketException`—enables or disables the `TCP_NODELAY` socket option, which determines whether Nagle's algorithm is used.
- `void shutdownInput()` throws `java.io.IOException`—closes the input stream associated with this socket and discards any further information that is sent. Further reads to the input stream will encounter the end of the stream marker.
- `void shutdownOutput()` throws `java.io.IOException`—closes the output stream associated with this socket. Any data previously written, but not yet sent, will be flushed, followed by a TCP connection-termination sequence, which notifies the application that no more data will be available (and in the case of a Java application, that the end of the stream has been reached). Further writes to the socket will cause an `IOException` to be thrown.

6.4.3 Reading from and Writing to TCP Sockets

Creating client software that uses TCP for communication is extremely easy in Java, no matter what operating system is being used. The Java Networking API provides a consistent, platform-neutral interface that allows client applications to connect to remote services. Once a socket is created, it is connected

and ready to read/write by using the socket's input and output streams. These streams don't need to be created; they are provided by the `Socket.getInputStream()` and `Socket.getOutputStream()` methods. As was shown in Chapter 4 on I/O streams, filtered streams provide easy I/O access.

A filter can easily be connected to a socket stream, to make for simpler programming. The following code snippet demonstrates a simple TCP client that connects a `BufferedReader` to the socket input stream, and a `PrintStream` to the socket output stream.

```
try
{
       // Connect a socket to some host machine and port
       Socket socket = new Socket ( somehost, someport );

       // Connect a buffered reader
       BufferedReader reader = new BufferedReader (
            new InputStreamReader ( socket.getInputStream() ) );

       // Connect a print stream
       PrintStream pstream =
            new PrintStream( socket.getOutputStream() );
}
catch (Exception e)
{
       System.err.println ("Error - " + e);
}
```

6.4.4 Socket Options

Socket options are settings that modify how sockets work, and they can affect (both positively and negatively) the performance of applications. Support for socket options was introduced in Java 1.1, and some refinements have been made in later versions (such as support for the `SO_KEEPALIVE` option in Java 2 v 1.3). Generally, socket options should not be changed unless there is a good reason for doing so, as changes may negatively affect application and network performance (for example, enabling Nagle's algorithm may increase performance of telnet type applications but lower the available bandwidth). The one exception to this caveat is the `SO_TIMEOUT` option—virtually every TCP application should handle timeouts gracefully rather than stalling if the application the socket is connected to fails to transmit data when required.

6.4.4.1 SO_KEEPALIVE Socket Option

The keepalive socket option is controversial; its use is a topic that some developers feel very strongly about. By default, no data is sent between two connected sockets unless an application has data to send. This means that an

idle socket may not have data submitted for minutes, hours, or even days in the case of long-lived processes. Suppose, however, that a client crashes, and the end-of-connection sequence is not sent to a TCP server. Valuable resources (CPU time and memory) might be wasted on a client that will never respond. When the keepalive socket option is enabled, the other end of the socket is probed to verify it is still active. However, the application doesn't have any control over how often keepalive probes are sent. To enable keepalive, the `Socket.setSoKeepAlive(boolean)` method is called with a value of "true" (a value of "false" will disable it). For example, to enable keepalive on a socket, the following code would be used.

```
// Enable SO_KEEPALIVE
someSocket.setSoKeepAlive(true);
```

Although keepalive does have some advantages, many developers advocate controlling timeouts and dead sockets at a higher level, in application code. It should also be kept in mind that keepalive doesn't allow you to specify a value for probing socket endpoints. A better solution than keepalive, and one that developers are advised to use, is to instead modify the timeout socket option.

6.4.4.2 SO_RCVBUF Socket Option

The receive buffer socket option controls the buffer used for receiving data. Changes can be made to the size by calling the `Socket.setReceiveBufferSize(int)` method. For example, to increase the receive buffer size to 4,096 bytes, the following code would be used.

```
// Modify receive buffer size
someSocket.setReceiveBufferSize(4096);
```

Note that a request to modify the size of the receive buffer does not guarantee that it will change. For example, some operating systems may not allow this socket option to be modified, and will ignore any changes to the value. The current buffer size can be determined by invoking the `Socket.getReceiveBufferSize()` method. A better choice for buffering is to use a `BufferedInputStream/BufferedReader`.

6.4.4.3 SO_SNDBUF Socket Option

The send buffer socket option controls the size of the buffer used for sending data. By calling the `Socket.setSendBufferSize(int)` method, you can attempt to change the buffer size, but requests to change the size may be rejected by the operating system.

```
// Set the send buffer size to 4096 bytes
someSocket.setSendBufferSize(4096);
```

To determine the size of the current send buffer, you can call the `Socket.getSendBufferSize()` method, which returns an int value.

```
// Get the default size
int size = someSocket.getSendBufferSize();
```

Changing buffer size will be more effective with the `DatagramSocket` class. When buffering writes, the preferable choice is to use a `BufferedOutputStream` or a `BufferedWriter`.

6.4.4.4 SO_LINGER Socket Option

When a TCP socket connection is closed, it is possible that data may be queued for delivery and not yet sent (particularly if an IP datagram becomes lost in transit and must be resent). The linger socket option controls the amount of time during which unsent data may be sent, after which it is discarded completely. It is possible to enable/disable the linger option entirely, or to modify the duration of a linger, by using the `Socket.setSoLinger(boolean onFlag, int duration)` method:

```
// Enable linger, for fifty seconds
someSocket.setSoLinger( true, 50 );
```

6.4.4.5 TCP_NODELAY Socket Option

This socket option is a flag, the state of which controls whether Nagle's algorithm (RFC 896) is enabled or not. Because TCP data is sent over the network using IP datagrams, a fair bit of overhead exists for each packet, such as IP and TCP header information. If only a few bytes at a time are sent in each packet, the size of the header information will far exceed that of the data. On a local area network, the extra amount of data sent probably won't amount to much, but on the Internet, where hundreds, thousands, or even millions of clients may be sending such packets through individual routers, this adds up to a significant amount of bandwidth consumption.

The solution is Nagle's algorithm, which states that TCP may send only one datagram at a time. When an acknowledgment comes back for each IP datagram, a new packet is sent containing any data that has been queued up. This limits the amount of bandwidth being consumed by packet header information, but at a not insignificant cost—network latency. Since data is being queued, it isn't dispatched immediately, so systems that require quick response times such as X-Windows or telnet are slowed. Disabling Nagle's algorithm may improve performance, but if used by too many clients, network performance is reduced.

Nagle's algorithm is enabled or disabled by invoking the `Socket.setTcpNoDelay(boolean state)` method. For example, to deactivate the algorithm, the following code would be used:

```
// Disable Nagle's algorithm for faster response times
someSocket.setTcpNoDelay(false);
```

To determine the state of Nagle's algorithm and the `TCP_NODELAY` flag, the `Socket.getTcpNoDelay()` method is used:

```
// Get the state of the TCP_NODELAY flag
boolean state = someSocket.getTcpNoDelay();
```

6.4.4.6 SO_TIMEOUT Socket Option

This timeout option is the most useful socket option. By default, I/O operations (be they file- or network-based) are blocking. An attempt to read data from an `InputStream` will wait indefinitely until input arrives. If the input never arrives, the application stalls and in most cases becomes unusable (unless multithreading is used). Users are not fond of unresponsive applications, and find such application behavior annoying, to say the least. A more robust application will anticipate such problems and take corrective action.

NOTE: In a local intranet environment during testing, network problems are rare, but on the Internet stalled applications are probable. Server applications are not immune—a server connection to a client uses the Socket class as well, and can just as easily stall. For this reason, all applications (be they client or server) should handle network timeouts gracefully.

When the `SO_TIMEOUT` option is enabled, any read request to the `InputStream` of a socket starts a timer. When no data arrives in time and the timer expires, a `java.io.InterruptedIOException` is thrown, which can be caught to check for a timeout. What happens then is up to the application developer—a retry attempt might be made, the user might be notified, or the connection aborted. The duration of the timer is controlled by calling the `Socket.setSoTimeout(int)` method, which accepts as a parameter the number of milliseconds to wait for data. For example, to set a five-second timeout, the following code would be used:

```
// Set a five second timeout
someSocket.setSoTimeout ( 5 * 1000 );
```

Once enabled, any attempt to read could potentially throw an `InterruptedIOException`, which is extended from the `java.io.IOException` class.

Since read attempts can already throw an `IOException`, no further code is required to handle the exception—however, some applications may want to specifically trap timeout-related exceptions, in which case an additional exception handler may be added.

```
try
{
       Socket s = new Socket (...);
       s.setSoTimeout ( 2000 );

       // do some read operation ....
}
catch (InterruptedIOException iioe)
{
       timeoutFlag = true; // do something special like set a flag
}
catch (IOException ioe)
{
       System.err.println ("IO error " + ioe);
       System.exit(0);
}
```

To determine the length of the TCP timer, the `Socket.getSoTimeout()` method, which returns an `int`, can be used. A value of zero indicates that timeouts are disabled, and read operations will block indefinitely.

```
// Check to see if timeout is not zero
if ( someSocket.getSoTimeout() == 0)
someSocket.setSoTimeout (500);
```

6.5 Creating a TCP Client

Having discussed the functionality of the Socket class, we will now examine a complete TCP client. The client we'll look at here is a daytime client, which, as its name suggests, connects to a daytime server to read the current day and time. Establishing a socket connection and reading from it is a fairly simple process, requiring very little code. By default, the daytime service runs on port 13. Not every machine has a daytime server running, but a Unix server would be a good system to run the client against. If you do not have access to a Unix server, code for a TCP daytime server is given in Section 6.7—the client can be run against it.

Code for DaytimeClient

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

```
// Chapter 6, Listing 1
public class DaytimeClient
{
      public static final int SERVICE_PORT = 13;

      public static void main(String args[])
      {
            // Check for hostname parameter
            if (args.length != 1)
            {
                  System.out.println ("Syntax - DaytimeClient host");
                  return;
            }

            // Get the hostname of server
            String hostname = args[0];

            try
            {
                  // Get a socket to the daytime service
                  Socket daytime = new Socket (hostname,
                  SERVICE_PORT);

                  System.out.println ("Connection established");

                  // Set the socket option just in case server stalls
                  daytime.setSoTimeout ( 2000 );

                  // Read from the server
                  BufferedReader reader = new BufferedReader (
                    new InputStreamReader
                    (daytime.getInputStream()
                  ));

                  System.out.println ("Results : " +
                  reader.readLine());

                  // Close the connection
                  daytime.close();
            }
            catch (IOException ioe)
            {
                  System.err.println ("Error " + ioe);
            }
      }
}
```

How DaytimeClient Works

The daytime application is straightforward, and uses concepts discussed earlier in the chapter. A socket is created, an input stream is obtained, and timeouts

are enabled in the rare event that a server as simple as daytime fails during a connection. Rather than connecting a filtered stream, a buffered reader is connected to the socket input stream, and the results are displayed to the user. Finally, the client terminates after closing the socket connection. This is about as simple a socket client as you can get—complexity comes from implementing network protocols, not from network-specific coding.

Running DaytimeClient

Running the application is easy. Simply specify the hostname of a machine running the daytime service as a command-line parameter and run it. If you use a nonstandard port for the daytime server (discussed later), remember to change the port number in the client and recompile.

For example, to run the client against a server running on the local machine, the following command would be used:

```
java DaytimeClient localhost
```


NOTE: The daytime server must be running, or the client will be unable to establish a connection. If you're using, for example, a Wintel system, instead of Unix, then you'll need to run the `DaytimeServer` from later in this chapter.

6.6 ServerSocket Class

A special type of socket, the server socket, is used to provide TCP services. Client sockets bind to any free port on the local machine, and connect to a specific server port and host. The difference with server sockets is that they bind to a specific port on the local machine, so that remote clients may locate a service. Client socket connections will connect to only one machine, whereas server sockets are capable of fulfilling the requests of multiple clients.

The way it works is simple—clients are aware of a service running on a particular port (usually the port number is well known, and used for particular protocols, but servers may run on nonstandard port numbers as well). They establish a connection, and within the server, the connection is accepted. Multiple connections can be accepted at the same time, or a server may choose to accept only one connection at any given moment. Once accepted, the connection is represented as a normal socket, in the form of a Socket object—once you have mastered the Socket class, it becomes almost as simple to write servers as it does clients. The only difference between a server and a client is that the server binds to a specific port, using a `ServerSocket` object. This `ServerSocket` object acts as a factory for client connections—you don't need to create

instances of the Socket class yourself. These connections are modeled as a normal socket, so you can connect input and output filter streams (or even a reader and writer) to the connection.

6.6.1 Creating a ServerSocket

Once a server socket is created, it will be bound to a local port and ready to accept incoming connections. When clients attempt to connect, they are placed into a queue. Once all free space in the queue is exhausted, further clients will be refused.

Constructors

The simplest way to create a server socket is to bind to a local address, which is specified as the only parameter, using a constructor. For example, to provide a service on port 80 (usually used for Web servers), the following snippet of code would be used:

```
try
{
       // Bind to port 80, to provide a TCP service (like HTTP)
       ServerSocket myServer = new ServerSocket ( 80 );

       // ......
}
catch (IOException ioe)
{
       System.err.println ("I/O error - " + ioe);
}
```

This is the simplest form of the ServerSocket constructor, but there are several others that allow additional customization. All of these constructors are marked as public.

- ServerSocket(int port) throws java.io.IOException, java.lang.SecurityException—binds the server socket to the specified port number, so that remote clients may locate the TCP service. If a value of zero is passed, any free port will be used—however, clients will be unable to access the service unless notified somehow of the port number. By default, the queue size is set to 50, but an alternate constructor is provided that allows modification of this setting. If the port is already bound, or security restrictions (such as security polices or operating system restrictions on well-known ports) prevent access, an exception is thrown.

- `ServerSocket(int port, int numberOfClients)` throws `java.io.IOException, java.lang.SecurityException`—binds the server socket to the specified port number and allocates sufficient space to the queue to support the specified number of client sockets. This is an overloaded version of the `ServerSocket(int port)` constructor, and if the port is already bound or security restrictions prevent access, an exception is thrown.
- `ServerSocket(int port, int numberOfClients, InetAddress address)` throws `java.io.IOException, java.lang.SecurityException`—binds the server socket to the specified port number, and allocates sufficient space to the queue to support the specified number of client sockets. This is an overloaded version of the `ServerSocket(int port, int numberOfClients)` constructor that allows a server socket to bind to a specific IP address, in the case of a multihomed machine. For example, a machine may have two network cards, or may be configured to represent itself as several machines by using virtual IP addresses. Specifying a null value for the address will cause the server socket to accept requests on all local addresses. If the port is already bound or security restrictions prevent access, an exception is thrown.

6.6.2 Using a ServerSocket

While the `Socket` class is fairly versatile, and has many methods, the `ServerSocket` class doesn't really do that much, other than accept connections and act as a factory for `Socket` objects that model the connection between client and server. The most important method is the `accept()` method, which accepts client connection requests, but there are several others that developers may find useful.

Methods

All methods are public unless otherwise noted.

- `Socket accept()` throws `java.io.IOException, java.lang.SecurityException`—waits for a client to request a connection to the server socket, and accepts it. This is a blocking I/O operation, and will not return until a connection is made (unless the timeout socket option is set). When a connection is established, it will be returned as a Socket object. When accepting connections, each client request will be verified by the default security manager, which makes it possible to accept certain IP addresses and block others, causing an exception to be thrown. However, servers do not need to rely on the security manager to block

or terminate connections—the identity of a client can be determined by calling the `getInetAddress()` method of the client socket.

- `void close()` throws `java.io.IOException`—closes the server socket, which unbinds the TCP port and allows other services to use it.
- `InetAddress getInetAddress()`—returns the address of the server socket, which may be different from the local address in the case of a multihomed machine (i.e., a machine whose localhost is known by two or more IP addresses).
- `int getLocalPort()`—returns the port number to which the server socket is bound.
- `int getSoTimeout()` throws `java.io.IOException`—returns the value of the timeout socket option, which determines how many milliseconds an `accept()` operation can block for. If a value of zero is returned, the accept operation blocks indefinitely.
- `void implAccept(Socket socket)` throws `java.io.IOException`—this method allows `ServerSocket` subclasses to pass an unconnected socket subclass, and to have that socket object accept an incoming request. Using the `implAccept` method to accept the connection, an overridden `ServerSocket.accept()` method can return a connected socket. Few developers will want to subclass the `ServerSocket`, and using this should be avoided unless required.
- `static void setSocketFactory ( SocketImplFactory factory )` throws `java.io.IOException, java.net.SocketException, java.lang.SecurityException` —assigns a server socket factory for the JVM. This is a static method, and should be called only once during the lifetime of a JVM. If assigning a new socket factory is prohibited, or one has already been assigned, an exception is thrown.
- `void setSoTimeout(int timeout)` throws `java.net.SocketException`—assigns a timeout value (specified in milliseconds) for the blocking `accept()` operation. If a value of zero is specified, timeouts are disabled and the operation will block indefinitely. Providing timeouts are enabled, however, whenever the `accept()` method is called a timer starts. When the timer expires, a `java.io.InterruptedIOException` is thrown, which allows a server to then take further actions.

6.6.3 Accepting and Processing Requests from TCP Clients

The most important function of a server socket is to accept client sockets. Once a client socket is obtained, the server can perform all the "real work" of server programming, which involves reading from and writing to the socket to imple-

ment a network protocol. The exact data that is sent or received is dependent on the details of the protocol. For example, a mail server that provides access to stored messages would listen to commands and send back message contents. A telnet server would listen for keystrokes and pass these to a log-in shell, and send back output to the network client. Protocol-specific actions are less network- and more programming-oriented.

The following snippet shows how client sockets are accepted, and how I/O streams may be connected to the client:

```
// Perform a blocking read operation, to read the next socket
// connection
Socket nextSocket = someServerSocket.accept();

// Connect a filter reader and writer to the stream
BufferedReader reader = new BufferedReader (new
                        InputStreamReader
                        (nextSocket.getInputStream() ) );
PrintWriter writer = new PrintWriter( new
                        OutputStreamWriter
                        (nextSocket.getOutputStream() ) );
```

From then on, the server may conduct the tasks needed to process and respond to client requests, or may choose to leave this task for code executing in another thread. Remember that just like any other form of I/O operation in Java, code will block indefinitely while reading a response from a client—so to service multiple clients concurrently, threads must be used. In simple cases, however, multiple threads of execution may not be necessary, particularly if requests are responded to quickly and take little time to process.

Creating fully-fledged client/server applications that implement popular Internet protocols involves a fair amount of effort, especially for those new to network programming. It also draws on other skills, such as multi-threaded programming, discussed in the next chapter. For now, we'll focus on a simple, bare-bones TCP server that executes as a single-threaded application.

6.7 Creating a TCP Server

One of the most enjoyable parts of networking is writing a network server. Clients send requests and respond to data sent back, but the server performs most of the real work. This next example is of a daytime server (which you can test using the client described in Section 6.5).

Code for DaytimeServer

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

```
// Chapter 6, Listing 2
public class DaytimeServer
{
      public static final int SERVICE_PORT = 13;

      public static void main(String args[])
      {
            try
            {
                  // Bind to the service port, to grant clients
                  // access to the TCP daytime service
                  ServerSocket server = new ServerSocket
                  (SERVICE_PORT);

                  System.out.println ("Daytime service started");

                  // Loop indefinitely, accepting clients
                  for (;;)
                  {
                        // Get the next TCP client
                        Socket nextClient = server.accept();

                        // Display connection details
                        System.out.println ("Received request from " +
                              nextClient.getInetAddress() + ":" +
                              nextClient.getPort() );

                        // Don't read, just write the message
                        OutputStream out =
                        nextClient.getOutputStream();
                        PrintStream pout = new PrintStream (out);

                        // Write the current date out to the user
                        pout.print( new java.util.Date() );

                        // Flush unsent bytes
                        out.flush();

                        // Close stream
                        out.close();

                        // Close the connection
                        nextClient.close();
                  }
            }
            catch (BindException be)
            {
                  System.err.println ("Service already running on port " + SERVICE_PORT );
            }
            catch (IOException ioe)
            {
```

```
                System.err.println ("I/O error - " + ioe);
            }
        }
}
```

How DaytimeServer Works

For a server, this is about as simple as it gets. The first step in this server is to create a `ServerSocket`. If this port is already bound, a `BindException` will be thrown, as no two servers can share the same port. Otherwise, the server socket is created; the next step is to wait for connections.

Since daytime is a very simple protocol and our first example of a TCP server should be a simple one, we use here a single-threaded server. A `for` loop that loops indefinitely is commonly used in simple TCP servers, or a while loop whose expression always evaluates to true. Inside this loop, the first line you will find is the `server.accept()` method, which blocks until a client attempts to connect. This method returns a socket that represents the connection to the client. For logging, the IP address and port of the connection is sent to `System.out`. You'll see this every time someone logs in and gets the time of day.

Daytime is a response-only protocol, so we don't need to worry about reading any input. We obtain an `OutputStream` and then wrap it in a `PrintStream` to make it easier to work with. Determining the date and time using the `java.util.Date` class, we send it over the TCP stream to the client. Finally, we flush all data in the print stream and close the connection by calling `close()` on the socket.

Running DaytimeServer

Running the server is very simple. The server has no command-line parameters. For this server example to run on UNIX, you will need to modify the `SERVICE_PORT` variable to a number above 1,024, unless you turn off the default daytime process and run this example as root. On Windows or other operating systems, this will not be a problem. To run the server on the local machine, the following command would be used:

```
java DaytimeServer
```

6.8 Exception Handling: Socket-Specific Exceptions

As a medium for communication, networks are fraught with problems. With so many machines connected to the global Internet, the prospect of encountering a host whose hostname cannot be resolved, one that is disconnected from the network, or one that locks up during a connection, is very likely in the lifetime of a software application. It is important, therefore, to be aware of the condi-

tions that might cause such problems to arise in an application and to deal with them gracefully. Of course, not every application will require precise control, and in simple applications you'll probably want to handle everything with a generic handler. For those more advanced applications, however, it is important to be aware of the socket-specific exceptions that can be thrown at runtime.

NOTE: All socket-specific exceptions extend from `SocketException`, so by simply catching that exception, you catch all of the socket-specific ones and write a single generic handler. In addition, `SocketException` extends from `java.io.IOException` if you want to provide a catchall for any I/O exception.

6.8.1 SocketException

The `java.net.SocketException` represents a generic socket error, which can represent a range of specific error conditions. For finer-grained control, applications should catch the subclasses discussed below.

6.8.2 BindException

The `java.net.BindException` represents an inability to bind a socket to a local port. The most common reason for this will be that the local port is already in use.

6.8.3 ConnectException

The `java.net.ConnectException` occurs when a socket can't connect to a specific remote host and port. There can be several reasons for this, such as that the remote server does not have a service bound to that port, or that it is so swamped by queued connections, it cannot accept any further ones.

6.8.4 NoRouteToHostException

The `java.net.NoRouteToHostException` is thrown when, due to a network error, it is impossible to find a route to the remote host. The cause of this may be local (i.e., the network on which the software application is running), may be a temporary gateway or router problem, or may be the fault of the remote network to which the socket is trying to connect. Another common cause of this is that firewalls and routers are blocking the client software, which is usually a permanent condition.

6.8.5 InterruptedIOException

The `java.net.InterruptedIOException` occurs when a read operation is blocked for sufficient time to cause a network timeout, as discussed earlier in the chapter. Handling timeouts is a good way to make your code more robust and reliable.

6.9 Summary

Communication over TCP with sockets is an important technique to master, as many of the most interesting application protocols in use today occur over TCP. The Java socket API provides a clear and easy mechanism by which developers are able to accept communications as a server or initiate communications as a client. By using the concepts discussed earlier involving input and output streams under Java, the transition to socket-based communication is straightforward. With the level of exception handling built into the `java.net` package, it's also very easy to deal with network errors that occur at runtime.

Chapter Highlights

In this chapter, you have learned:

- About the Transmission Control Protocol
- About clients and servers
- About how to write and run a simple TCP socket client, using `java.net.Socket`
- About how to write and run a simple TCP socket server, using `java.net.ServerSocket`
- About exception handling with sockets