Chapter 4: Beyond the Basics

4.1 Multitasking

*4.1.1 Java Threads:*

Java provides two approaches for performing a task in a new thread: 1) defining a subclass  
of the Thread class with a run() method that performs the task, and instantiating it; or 2)  
defining a class that implements the Runnable interface with a run() method that performs  
the task, and passing an instance of that class to the Thread constructor. In either case, the  
new thread does not begin execution until its start() method is invoked. The first approach  
can only be used for classes that do not already extend some other class; therefore, we stick  
with the second approach, which is always applicable. The Runnable interface contains a single  
method prototype:  
interface Runnable {  
void run();  
}  
When the start() method of an instance of Thread is invoked, the JVM causes the  
instance’s run() method to be executed in a new thread, concurrently with all others.  
Meanwhile, the *original* thread returns from its call to start() and continues its execution  
independently. (Note that directly calling run() does not create a new thread; instead, the  
run() method is simply executed in the caller’s thread, just like any other method call.) The  
statements of each thread’s run() method are interleaved in a nondeterministic fashion, so in  
general it is not possible to predict precisely the order in which things will happen in different  
threads.

*4.1.2 Server Protocol*

Since the multitasking server approaches we are going to describe are independent of  
the particular client-server protocol, we want to be able to use the same protocol implementation for both. The code for the echo protocol is given in the class EchoProtocol.  
This class encapsulates the per-client processing in the static method handleEchoClient().  
This code is almost identical to the connection-handling portion of TCPEchoServer.java,  
except that a logging capability (described shortly) has been added; the method takes references to the client Socket and the Logger instance as arguments.  
The class implements Runnable (the run() method simply invokes handle EchoClient()  
with the instance’s Socket and Logger references), so we can create a thread that independently  
executes run(). Alternatively, the server-side protocol processing can be invoked by calling  
the static method directly (passing it the Socket and Logger references).

*4.1.3 Thread-per-Client*

In a *thread-per-client* server, a new thread is created to handle each connection. The server  
executes a loop that runs forever, listening for connections on a specified port and repeatedly  
accepting an incoming connection from a client and then spawning a new thread to handle  
that connection.

*4.1.4 Thread Pool*

Every new thread consumes system resources: spawning a thread takes CPU cycles and each  
thread has its own data structures (e.g., stacks) that consume system memory. In addition,  
when one thread *blocks*, the JVM saves its state, selects another thread to run, and restores  
the state of the chosen thread in what is called a *context switch*. As the number of threads  
increases, more and more system resources are consumed by thread overhead. Eventually, the  
system is spending more time dealing with context switching and thread management than  
with servicing connections. At that point, adding an additional thread may actually *increase*client service time.  
We can avoid this problem by limiting the total number of threads and reusing threads.  
Instead of spawning a new thread for each connection, the server creates a *thread pool* on  
start-up by spawning a fixed number of threads. When a new client connection arrives at the  
server, it is assigned to a thread from the pool. When the thread finishes with the client, it  
returns to the pool, ready to handle another request. Connection requests that arrive when all  
threads in the pool are busy are queued to be serviced by the next available thread.  
Like the thread-per-client server, a thread-pool server begins by creating a ServerSocket.  
Then it spawns *N* threads, each of which loops forever, accepting connections from the (shared)  
ServerSocket instance. When multiple threads simultaneously call accept() on the same ServerSocket instance, they all block until a connection is established. Then the system selects one  
thread, and the Socket instance for the new connection is returned *only in that thread*. The other  
threads remain blocked until the next connection is established and another lucky winner is  
chosen.  
Since each thread in the pool loops forever, processing connections one by one, a  
thread-pool server is really like a set of iterative servers. Unlike the thread-per-client server,  
a thread-pool thread does not terminate when it finishes with a client. Instead, it starts  
over again, blocking on accept().

*4.1.5 System-Managed Dispatching: The Executor Interface*

ava provides a number of built-in implementations of Executor that are convenient and  
simple to use, and others that are extensively configurable. Some of these offer handling for  
messy details like thread maintenance. For example, if a thread stops because of an uncaught  
exception or other failure, they automatically spawn a new thread to replace it.

**4.2 Blocking and Timeouts**

*4.2.1 accept(), read(), and receive()*

For these methods, we can set a bound on the maximum time (in milliseconds) to block, using  
the setSoTimeout() method of Socket, ServerSocket, and DatagramSocket. If the specified time  
elapses before the method returns, an InterruptedIOException is thrown. For Socket instances,  
we can also use the available() method of the socket’s InputStream to check for available data  
before calling read().  
*4.2.2 Connecting and Writing*

The Socket constructor attempts to establish a connection to the host and port supplied as  
arguments, blocking until either the connection is established or a system-imposed timeout  
occurs. Unfortunately, the system-imposed timeout is long, and Java does not provide any  
means of shortening it. To fix this, call the parameterless constructor for Socket, which returns  
an unconnected instance. To establish a connection, call the connect() method on the newly  
constructed socket and specify both a remote endpoint and timeout (milliseconds).  
A write() call blocks until the last byte written is copied into the TCP implementation’s  
local buffer; if the available buffer space is smaller than the size of the write, some data must be  
successfully transferred to the other end of the connection before the call to write() will return  
(see Section 6.1 for details). Thus, the amount of time that a write() may block is ultimately  
controlled by the receiving application. Unfortunately, Java currently does not provide any  
way to cause a write() to time out, nor can it be interrupted by another thread. Therefore,  
any protocol that sends a large enough amount of data over a Socket instance can block for  
an unbounded amount of time. (See Section 6.2 for a discussion of the potentially disastrous  
consequences of this.)

*4.2.3 Limiting Per-Client Time*

Suppose we want to implement the Echo protocol with a limit on the amount of time taken  
to service each client. That is, we define a target, timelimit, and implement the protocol in  
such a way that after timelimit milliseconds, the protocol instance is terminated. The protocol  
instance keeps track of the amount of time remaining, and uses setSoTimeout() to ensure that  
no read() call blocks for longer than that time. Since there is no way to bound the duration  
of a write() call, we cannot really guarantee that the time limit will hold. Nevertheless, TimelimitEchoProtocol.java implements this approach; to use it with TCPEchoServerExecutor.java,  
simply change the second line of the body of the while loop to:  
service.execute(new TimeLimitEchoProtocol(clntSock, logger));  
Again, Chapter 5 will cover more powerful mechanisms that can limit the time that threads  
can block—on all I/O calls, including writes—using the facilities of the NIO package.

**4.3 Multiple Recipients**

*4.3.1 Broadcast*

Broadcasting UDP datagrams is similar to unicasting datagrams, except that a *broadcast  
address* is used instead of a regular (unicast) IP address. Note that IPv6 does not explicitly  
provide broadcast addresses; however, there is a special all-nodes, link-local-scope multicast  
address, FFO2::1, that multicasts to all nodes on a link. The IPv4 *local broadcast* address  
(255.255.255.255) sends the message to every host on the same broadcast network. Local  
broadcast messages are never forwarded by routers. A host on an Ethernet network can send  
a message to all other hosts on that same Ethernet, but the message will not be forwarded by a  
router. IPv4 also specifies *directed broadcast* addresses, which allow broadcasts to all hosts on  
a specified network; however, since most Internet routers do not forward directed broadcasts,  
we do not deal with them here.  
There is no networkwide broadcast address that can be used to send a message to all  
hosts. To see why, consider the impact of a broadcast to every host on the Internet. Sending  
a single datagram would result in a very, very large number of packet duplications by the  
routers, and bandwidth would be consumed on each and every network. The consequences of  
misuse (malicious or accidental) are too great, so the designers of IP left such an Internetwide  
broadcast facility out on purpose.  
Even so, local broadcast can be very useful. Often, it is used in state exchange for network games where the players are all on the same local (broadcast) network. In Java, the code  
for unicasting and broadcasting is the same. To play with broadcasting applications, we can  
simply use our VoteClientUDP.java with a broadcast destination address. There is one problem. Can you find it? *Hint:* You cannot use connect with broadcast. Run VoteServerUDP.java as  
you did before (except that you can run several receivers at one time). *Caveat:* Some operating systems do not give regular users permission to broadcast, in which case this will  
not work.

*4.3.2 Multicast*

As with broadcast, one of the main differences between multicast and unicast is the form of  
the address. A multicast address identifies a set of receivers. The designers of IP allocated arange of the address space dedicated to multicast, specifically 224.0.0.0 to 239.255.255.255  
for IPv4 and any address starting with FF for IPv6. With the exception of a few reserved multicast addresses, a sender can send datagrams addressed to any address in this range. In Java,  
multicast applications generally communicate using an instance of MulticastSocket, a subclass  
of DatagramSocket. It is important to understand that a MulticastSocket is actually a UDP socket  
(DatagramSocket), with some extra multicast-specific attributes that can be controlled. Our next  
examples implement a multicast sender and receiver of vote messages.

**4.4 Controlling Default Behaviors**

*4.4.1 Keep-Alive*

If no data has been exchanged for a while, each endpoint may be wondering if the other is  
still around. TCP provides a keep-alive mechanism where, after a certain time of inactivity, a  
probe message is sent to the other endpoint. If the endpoint is alive and well, it sends an  
acknowledgment. After a few retries without acknowledgment, the probe sender gives up and  
closes the socket, eliciting an exception on the next attempted I/O operation. Note that the  
application only sees keep-alive working if the probes *fail*.  
**Socket: KeepAlive**boolean getKeepAlive()  
void setKeepAlive(boolean on)  
By default, keep-alive is disabled. Call the setKeepAlive() method with true to enable  
keep-alive.  
4.4.2 Send and Receive Buffer Size  
When a Socket or DatagramSocket is created, the operating system must allocate buffers to hold  
incoming and outgoing data. (We talk about this in much greater detail in Section 6.1.)  
**Socket, DatagramSocket: Setting/Getting Send/Receive Buffer Size**int getReceiveBufferSize()  
void setReceiveBufferSize(int size)  
int getSendBufferSize()  
void setSendBufferSize(int size)  
The getReceiveBufferSize(), setReceiveBufferSize(), getSendBufferSize(), and setSendBufferSize() methods get and set the size (bytes) of the receive and send buffers. Note that  
these sizes are taken as suggestions so the actual size may not be what you specified.  
You can also specify the receive buffer size on a ServerSocket; however, this actually sets  
the receive buffer size for new Socket instances created by accept(). Why can you only set thereceive buffer size and not the send buffer? When you accept a new Socket, it can immediately  
begin receiving data so you need the receive buffer size set before accept() completes the  
connection. On the other hand, you control when you send on a newly accepted socket, which  
gives you time to set the send buffer size before sending.  
**ServerSocket: Setting/Getting Accepted Socket Receive Buffer Size**int getReceiveBufferSize()  
void setReceiveBufferSize(int size)  
The getReceiveBufferSize() and setReceiveBufferSize() methods get and set the size  
(bytes) of the receive buffer for Socket instances created by the accept().

*4.4.3 Timeout*

As we’ve already seen, many I/O operations will block if they cannot complete immediately:  
reads block until at least 1 byte is available and accept blocks until a connection is initiated.  
Unfortunately, the blocking time is not bounded. We can specify a maximum blocking time for  
the various operations.  
**Socket, ServerSocket, DatagramSocket: Setting/Getting I/O Timeouts**int getSoTimeout()  
void setSoTimeout(int timeout)  
The getSoTimeout() and setSoTimeout() methods get and set the maximum time (milliseconds) to allow read/receive and accept operations to block. A timeout of 0 means the  
operation *never* times out. If the timeout expires, an exception is thrown.

*4.4.4 Address Reuse*

Under some circumstances, you may want to allow multiple sockets to bind to the same socket  
address. In the case of UDP multicast, you may have multiple applications on the same host  
participating in the same multicast group. For TCP, when a connection is closed, one (or both)  
endpoints must hang around for a while in “Time-Wait” state to vacuum up stray packets  
(see Section 6.4.2). Unfortunately, you may not be able to wait for the Time-Wait to expire. In  
both cases, you need the ability to bind to an address that’s in use. To enable this, you must  
allow address reuse.  
**Socket, ServerSocket, DatagramSocket: Setting/Getting Address Reuse**boolean getReuseAddress()  
void setReuseAddress(boolean on)  
The getReuseAddress() and setReuseAddress() methods get and set reuse address permissions. A value of true means that address reuse is enabled.

*4.4.5 Eliminating Buffering Delay*

TCP attempts to help you avoid sending small packets, which waste network resources. It  
does this by buffering data until it has more to send. While this is good for the network, your  
application may not be so tolerant of this buffering delay. Fortunately, you can disable this  
behavior.  
**Socket: Setting/Getting TCP Buffering Delay**boolean getTcpNoDelay()  
void setTcpNoDelay(boolean on)  
The getTcpNoDelay() and setTcpNoDelay() methods get and set the elimination of buffering  
delay. A value of true means that buffering delay is disabled.

*4.4.6 Urgent Data*

Suppose you’ve sent a bunch of data to a slow receiver and suddenly you have some data that  
the receiver needs right now. If you send the data in the output stream, it gets queued up  
behind all of the regular data, and who knows when the receiver will see it? To deal with this  
TCP includes the concept of *urgent* data that can (theoretically) skip ahead. Such data is called  
out-of-band because it bypasses the normal stream.  
**Socket: Urgent Data**void sendUrgentData(int data)  
boolean getOOBInline()  
void setOOBInline(boolean on)  
To send urgent data, call the sendUrgentData() method, which sends the least significant  
byte of the int parameter. To receive this byte, the receiver must enable out-of-band data  
by passing true to setOOBInline(). The byte is received in the input stream of the receiver.Data sent before the urgent byte will precede the urgent byte in the receiver’s input stream. If  
reception of out-of-band data is not enabled, the urgent byte is silently discarded.  
Note that Java can get little use from urgent data because urgent bytes are mixed in with  
regular bytes *in the order of transmission*. In fact, a Java receiver cannot even tell that it’s  
receiving urgent data.

*4.4.7 Lingering after close*

When you call close() on a socket, it immediately returns even if the socket is buffering unsent  
data. The problem is that your host could then fail at a later time without sending all of the  
data. You may optionally ask close() to “linger,” or block, by blocking until all of the data is  
sent and acked or a timeout expires. See Section 6.4.2 for more details.  
**Socket: Linger on** close()  
int getSoLinger()  
void setSoLinger(boolean on, int linger)  
If you call setSoLinger() with on set to true, then a subsequent close() will block until  
all data is acknowledged by the remote endpoint or the specified timeout (seconds) expires. If  
the timeout expires, the TCP connection is forceably closed. The getSoLinger() method returns  
the timeout if linger is enabled and *-*1 otherwise.

*4.4.8 Broadcast Permission*

Some operating systems require that you explicitly request permission to broadcast. You  
can control broadcast permissions. As you already know, DatagramSockets provide broadcast  
service.  
**DatagramSocket: Setting/Getting Broadcast Permissions**boolean getBroadcast()  
void setBroadcast(boolean on)  
The getBroadcast() and setBroadcast() methods get and set broadcast permissions.  
A value of true means that broadcast is permitted. By default, Java permits broadcast.

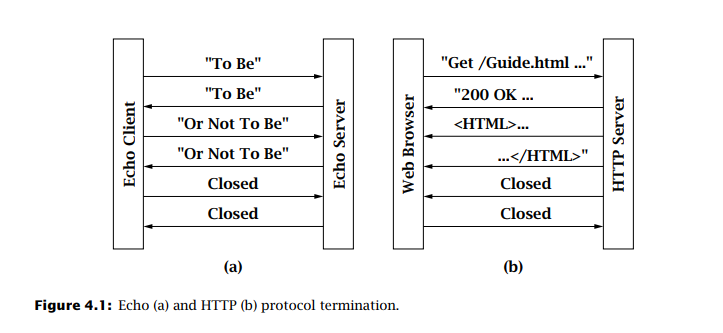
*4.4.9 Traffic Class*

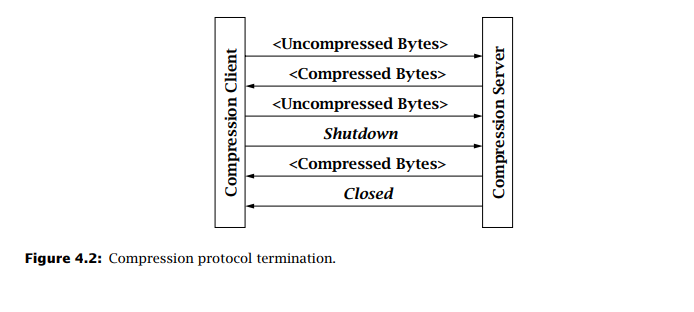
Some networks offer enhanced or “premium” services to packets classified as being eligible  
for the service. The *traffic class* of a packet is indicated by a value carried in the packet as it  
is transmitted through the network. For example, some networks might give packets in the  
“gold service” class higher priority, to provide reduced delay and/or reduced loss probability.  
Others might use the indicated traffic class to choose a route for the packet. Beware, however,  
that network providers charge extra for such services, so there is no guarantee these options  
will actually have any effect.  
**Socket, DatagramSocket: Setting/Getting Traffic Class**int getTrafficClass()  
void setTrafficClass(int tc)  
The traffic class is specified as an integer or a set of bit flags. The number and meaning  
of the bits depend on the version of IP you are using.

*4.4.10 Performance-Based Protocol Selection*

TCP may not be the only protocol available to a socket. Which protocol to use depends on  
what’s important to your application. Java allows you to give “advice” to the implementation  
regarding the importance of different performance characteristics to your application. The  
underlying network system may use the advice to choose among different protocols that can  
provide equivalent stream services with different performance characteristics.  
**Socket, ServerSocket: Specifying Protocol Preferences**void setPerformancePreferences(int connectionTime, int latency, int bandwidth)  
The performance preference for the socket is expressed by three integers representing  
connection time, delay, and bandwith. The specific values are not important; instead, Java  
compares the relative values for each criterion and returns the closest-matching, available  
protocol. For example, if connectionTime and latency both equal 0 and bandwidth equals 1, the  
protocol maximizing bandwidth will be selected. Note that this method must be called *be*

**4.5 Closing Connections**





**4.6 Applets**Applets can perform network communication using TCP/IP sockets, but there are severe  
restrictions on how and with whom they can converse. Without such restrictions, unsuspecting Web browsers might execute malicious applets that could, for example, send fake email,  
attempt to hack other systems while the browser user gets the blame, and so on. These security restrictions are enforced by the Java security manager, and violations by the applet result  
in a SecurityException. Typically, browsers only allow applets to communicate with the host  
that served the applet. This means that applets are usually restricted to communicating with  
applications executing on that host, usually a Web server originating the applet. The list of  
security restrictions and general applet programming is beyond the scope of this book. It is  
worth noting, however, that the default security restrictions can be altered, if allowed by the  
browser user.  
Suppose that you wanted to implement an applet that allowed users to type and save  
notes to themselves on their browser. Browser security restrictions prevent applets from  
saving data directly on the local file system, so you would need some other means besides  
local disk I/O to save the notes. FileClientApplet.java (available from the book’s Web site)  
is an applet that allows the user to type text into an editor window and, by clicking the  
“Save” button, copy the text over the network to a server (running on port 5000). The server,  
TCPFileServer.java (also on the book’s Web site), saves the data to a file. It takes a port (use  
5000 to work with the applet) and the name of the file. The server must execute on the Web  
server that serves the applet to the browser. Note that there is nothing applet-specific about  
the server. FileClientApplet.html on the Web site demonstrates how to integrate the applet  
into a Web page.