Chapter 4: Beyond the Basics

4.1 Multitasking

4.1.1 Java Threads:

**ThreadExample.java**0 import java.util.concurrent.TimeUnit;  
1 2  
public class ThreadExample implements Runnable {  
3 4  
private String greeting; // Message to print to console  
4.1 Multitasking **75**

|  |  |
| --- | --- |
| 5 6 | public ThreadExample(String greeting) { |
| 7 | this.greeting = greeting; |

8 }  
9  
10 public void run() {  
11 while (true) {  
12 System.out.println(Thread.currentThread().getName() + ": " + greeting);  
13 try {  
14 // Sleep 0 to 100 milliseconds  
15 TimeUnit.MILLISECONDS.sleep(((long) Math.random() \* 100));  
16 } catch (InterruptedException e) {  
17 } // Should not happen  
18 }  
19 }  
20  
21 public static void main(String[] args) {  
22 new Thread(new ThreadExample("Hello")).start();  
23 new Thread(new ThreadExample("Aloha")).start();  
24 new Thread(new ThreadExample("Ciao")).start();  
25 }  
26 }

1. **Declaration of implementation of the Runnable interface:** line 2  
   Since ThreadExample implements the Runnable interface, it can be passed to the constructor  
   of Thread. If ThreadExample fails to provide a run() method, the compiler will complain.  
   2. **Member variables and constructor:** lines 4–8  
   Each instance of ThreadExample contains its own greeting string.  
   3. run()**:** lines 10–19  
   Loop forever performing:  
   **Print the thread name and instance greeting:** line 12  
   The static method Thread.currentThread() returns a reference to the thread from which  
   it is called, and getName() returns a string containing the name of that thread.  
   **Suspend thread:** lines 13–17  
   After printing its instance’s greeting message, each thread sleeps for a random amount  
   of time (between 0 and 100 milliseconds) by calling the static method Thread.sleep(),  
   which takes the number of milliseconds to sleep as a parameter. Math.random() returns  
   a random double between 0.0 and 1.0. Thread.sleep() can be interrupted by another  
   thread, in which case an InterruptedException is thrown. Our example does not include  
   an interrupt call, so the exception will not happen in this application.  
   4. main()**:** lines 21–25  
   Each of the three statements in main() does the following: 1) creates a new instance of  
   ThreadExample with a different greeting string, 2) passes this new instance to the constructor of Thread, and 3) calls the new Thread instance’s start() method. Each thread  
   independently executes the run() method of ThreadExample, while the main thread terminates. Note that the JVM does not terminate until all nondaemon (see Thread API) threads  
   terminate.

4.1.2 Server Protocol

**EchoProtocol.java**0 import java.io.IOException;  
1 import java.io.InputStream;  
2 import java.io.OutputStream;  
3 import java.net.Socket;  
4 import java.util.logging.Level;  
5 import java.util.logging.Logger;  
6 7  
public class EchoProtocol implements Runnable {  
8 private static final int BUFSIZE = 32; // Size (in bytes) of I/O buffer  
4.1 Multitasking **77**9 private Socket clntSock; // Socket connect to client  
10 private Logger logger; // Server logger  
11  
12 public EchoProtocol(Socket clntSock, Logger logger) {  
13 this.clntSock = clntSock;  
14 this.logger = logger;  
15 }  
16  
17 public static void handleEchoClient(Socket clntSock, Logger logger) {  
18 try {  
19 // Get the input and output I/O streams from socket  
20 InputStream in = clntSock.getInputStream();  
21 OutputStream out = clntSock.getOutputStream();  
22  
23 int recvMsgSize; // Size of received message  
24 int totalBytesEchoed = 0; // Bytes received from client  
25 byte[] echoBuffer = new byte[BUFSIZE]; // Receive Buffer  
26 // Receive until client closes connection, indicated by -1  
27 while ((recvMsgSize = in.read(echoBuffer)) != -1) {  
28 out.write(echoBuffer, 0, recvMsgSize);  
29 totalBytesEchoed += recvMsgSize;

|  |  |
| --- | --- |
| 30 | } |
| 31 |  |

32 logger.info("Client " + clntSock.getRemoteSocketAddress() + ", echoed "  
33 + totalBytesEchoed + " bytes.");  
34  
35 } catch (IOException ex) {  
36 logger.log(Level.WARNING, "Exception in echo protocol", ex);  
37 } finally {  
38 try {  
39 clntSock.close();  
40 } catch (IOException e) {  
41 }  
42 }  
43 }  
44  
45 public void run() {  
46 handleEchoClient(clntSock, logger);  
47 }  
48 }  
**EchoProtocol.java**1. **Declaration of implementation of the Runnable interface:** line 7  
2. **Member variables and constructor:** lines 8–15  
Each instance of EchoProtocol contains a socket for the connection and a reference to the  
logger instance.  
3. handleEchoClient()**:** lines 17–43  
Implement the echo protocol:  
**Get the input/output streams from the socket:** lines 20–21  
**Receive and echo:** lines 25–30  
Loop until the connection is closed (as indicated by read() returning *-*1), writing  
whatever is received back immediately.  
**Record the connection details in the log:** lines 32–33  
Record the SocketAddress of the remote end along with the number of bytes echoed.  
**Handle exceptions:** line 36  
Log any exceptions.

4.1.3 Thread-per-Client

**TCPEchoServerThread.java**0 import java.io.IOException;  
1 import java.net.ServerSocket;  
2 import java.net.Socket;  
3 import java.util.logging.Logger;  
4 5  
public class TCPEchoServerThread {  
6 7  
public static void main(String[] args) throws IOException {  
8 9  
if (args.length != 1) { // Test for correct # of args  
10 throw new IllegalArgumentException("Parameter(s): <Port>");  
11 }  
12  
13 int echoServPort = Integer.parseInt(args[0]); // Server port  
14  
15 // Create a server socket to accept client connection requests  
16 ServerSocket servSock = new ServerSocket(echoServPort);  
17  
18 Logger logger = Logger.getLogger("practical");  
19  
20 // Run forever, accepting and spawning a thread for each connection  
21 while (true) {  
22 Socket clntSock = servSock.accept(); // Block waiting for connection  
23 // Spawn thread to handle new connection  
24 Thread thread = new Thread(new EchoProtocol(clntSock, logger));  
25 thread.start();  
26 logger.info("Created and started Thread " + thread.getName());  
27 }  
28 /\* NOT REACHED \*/  
29 }  
30 }  
**TCPEchoServerThread.java**1. **Parameter parsing and server socket/logger creation:** lines 9–18  
2. **Loop forever, handling incoming connections:** lines 21–27  
**Accept an incoming connection:** line 22  
**Create a new instance of** Thread **to handle the new connection:** line 24  
Since EchoProtocol implements the Runnable interface, we can give our new instance  
to the Thread constructor, and the new thread will execute the run() method of  
EchoProtocol (which calls handleEchoClient()) when start() is invoked.  
**Start the new thread for the connection and log it:** lines 25–26  
The getName() method of Thread returns a String containing a name for the new thread.

4.1.4 Thread Pool

**TCPEchoServerPool.java**0 import java.io.IOException;  
1 import java.net.ServerSocket;  
2 import java.net.Socket;  
3 import java.util.logging.Level;  
4 import java.util.logging.Logger;  
5  
4.1 Multitasking **83**6 public class TCPEchoServerPool {

|  |  |
| --- | --- |
| 7 8 | public static void main(String[] args) throws IOException { |
| 9 |  |
| 10 | if (args.length != 2) { // Test for correct # of args |

11 throw new IllegalArgumentException("Parameter(s): <Port> <Threads>");  
12 }  
13  
14 int echoServPort = Integer.parseInt(args[0]); // Server port  
15 int threadPoolSize = Integer.parseInt(args[1]);  
16  
17 // Create a server socket to accept client connection requests  
18 final ServerSocket servSock = new ServerSocket(echoServPort);  
19  
20 final Logger logger = Logger.getLogger("practical");  
21  
22 // Spawn a fixed number of threads to service clients  
23 for (int i = 0; i < threadPoolSize; i++) {  
24 Thread thread = new Thread() {  
25 public void run() {  
26 while (true) {  
27 try {  
28 Socket clntSock = servSock.accept(); // Wait for a connection  
29 EchoProtocol.handleEchoClient(clntSock, logger); // Handle it  
30 } catch (IOException ex) {  
31 logger.log(Level.WARNING, "Client accept failed", ex);  
32 }  
33 }  
34 }  
35 };  
36 thread.start();  
37 logger.info("Created and started Thread = " + thread.getName());  
38 }  
39 }  
40 }  
**TCPEchoServerPool.java**1. **Setup:** lines 10–20  
The port number to listen on and the number of threads are both passed as arguments  
to main(). After parsing them we create the ServerSocket and Logger instances. Note that  
both have to be declared final, because they are referenced inside the anonymous class  
instance created below.  
2. **Create and start *threadPoolSize* new threads:** lines 23–38  
For each loop iteration, an instance of an anonymous class that extends Thread is created.  
When the start() method of this instance is called, the thread executes the run() method  
of this anonymous class. The run() method loops forever, accepting a connection and  
then giving it to EchoProtocol for service.  
**Accept an incoming connection:** line 28  
Since there are *N* different threads executing the same loop, up to *N* threads can  
be blocked on *servSock*’s accept(), waiting for an incoming connection. The system  
ensures that only one thread gets a Socket for any particular connection. If no threads  
are blocked on accept() when a client connection is established (that is, if they are all  
busy servicing other connections), the new connection is queued by the system until  
the next call to accept() (see Section 6.4.1).  
**Pass the client socket to** EchoProtocol.handleEchoClient**:** line 29  
The handleEchoClient() method encapsulates knowledge of the protocol details. It logs  
the connection when it finishes, as well as any exceptions encountered along the way.  
**Handle exception from** accept()**:** line 31

4.1.5 System-Managed Dispatching: The Executor Interface

**TCPEchoServerExecutor.java**0 import java.io.IOException;  
1 import java.net.ServerSocket;  
2 import java.net.Socket;  
3 import java.util.concurrent.Executor;  
4 import java.util.concurrent.Executors;  
5 import java.util.logging.Logger;  
6 7  
public class TCPEchoServerExecutor {

|  |  |
| --- | --- |
| 8 9 | public static void main(String[] args) throws IOException { |
| 10 |  |
| 11 | if (args.length != 1) { // Test for correct # of args |

12 throw new IllegalArgumentException("Parameter(s): <Port>");  
13 }  
14  
15 int echoServPort = Integer.parseInt(args[0]); // Server port  
16  
17 // Create a server socket to accept client connection requests  
18 ServerSocket servSock = new ServerSocket(echoServPort);  
19  
20 Logger logger = Logger.getLogger("practical");  
21  
22 Executor service = Executors.newCachedThreadPool(); // Dispatch svc  
23  
24 // Run forever, accepting and spawning a thread for each connection  
25 while (true) {  
26 Socket clntSock = servSock.accept(); // Block waiting for connection  
27 service.execute(new EchoProtocol(clntSock, logger));  
28 }  
29 /\* NOT REACHED \*/  
30 }  
31 }  
**TCPEchoServerExecutor.java**  
1. **Setup:** lines 11–20  
The port is the only argument. We create the ServerSocket and Logger instances as before;  
they need not be declared final here, because we do not need an anonymous Thread  
subclass.  
2. **Get an** Executor**:** line 22  
The static factory method newCachedThreadPool() of class Executors creates an instance  
of ExecutorService. When its execute() method is invoked with a Runnable instance, the  
executor service creates a new thread to handle the task if necessary. However, it first  
tries to reuse an existing thread. When a thread has been idle for at least 60 seconds, it  
is removed from the pool. This is almost always going to be more efficient than either of  
the last two TCPEchoServer\* examples.  
3. **Loop forever, accepting connections and executing them:** lines 25–28  
When a new connection arrives, a new EchoProtocol instance is created and passed to the  
execute() method of *service*, which either hands it off to an already-existing thread or  
creates a new thread to handle it. Note that in the steady state, the cached thread pool  
Executor service ends up having about the right number of threads, so that each thread  
stays busy and creation/destruction of threads is rare.

**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.  
**TimeLimitEchoProtocol.java**0 import java.io.IOException;  
1 import java.io.InputStream;  
2 import java.io.OutputStream;  
3 import java.net.Socket;  
4 import java.util.logging.Level;  
5 import java.util.logging.Logger;  
6 7  
class TimelimitEchoProtocol implements Runnable {  
8 private static final int BUFSIZE = 32; // Size (bytes) of buffer  
9 private static final String TIMELIMIT = "10000"; // Default limit (ms)  
10 private static final String TIMELIMITPROP = "Timelimit"; // Property  
11  
12 private static int timelimit;  
13 private Socket clntSock;  
14 private Logger logger;  
15  
16 public TimelimitEchoProtocol(Socket clntSock, Logger logger) {  
17 this.clntSock = clntSock;  
18 this.logger = logger;  
19 // Get the time limit from the System properties or take the default  
20 timelimit = Integer.parseInt(System.getProperty(TIMELIMITPROP,TIMELIMIT));  
21 }  
22  
23 public static void handleEchoClient(Socket clntSock, Logger logger) {  
24  
25 try {  
26 // Get the input and output I/O streams from socket  
27 InputStream in = clntSock.getInputStream();  
28 OutputStream out = clntSock.getOutputStream();  
29 int recvMsgSize; // Size of received message  
30 int totalBytesEchoed = 0; // Bytes received from client  
31 byte[] echoBuffer = new byte[BUFSIZE]; // Receive buffer  
32 long endTime = System.currentTimeMillis() + timelimit;  
33 int timeBoundMillis = timelimit;34  
35 clntSock.setSoTimeout(timeBoundMillis);  
36 // Receive until client closes connection, indicated by -1  
37 while ((timeBoundMillis > 0) && // catch zero values  
38 ((recvMsgSize = in.read(echoBuffer)) != -1)) {  
39 out.write(echoBuffer, 0, recvMsgSize);  
40 totalBytesEchoed += recvMsgSize;  
41 timeBoundMillis = (int) (endTime - System.currentTimeMillis()) ;  
42 clntSock.setSoTimeout(timeBoundMillis);  
43 }  
44 logger.info("Client " + clntSock.getRemoteSocketAddress() +  
45 ", echoed " + totalBytesEchoed + " bytes.");  
46 } catch (IOException ex) {  
47 logger.log(Level.WARNING, "Exception in echo protocol", ex);  
48 }  
49 }  
50  
51 public void run() {  
52 handleEchoClient(this.clntSock, this.logger);  
53 }  
54 }

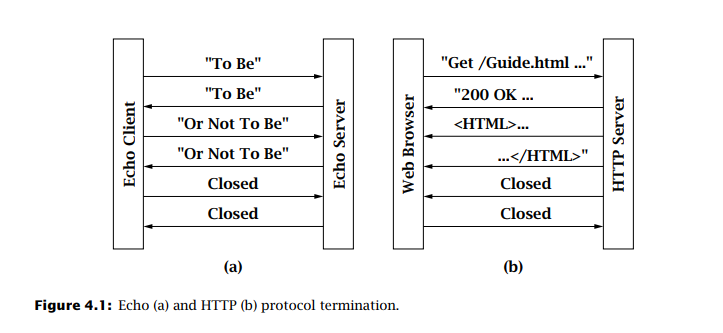
**4.3 Multiple Recipients**

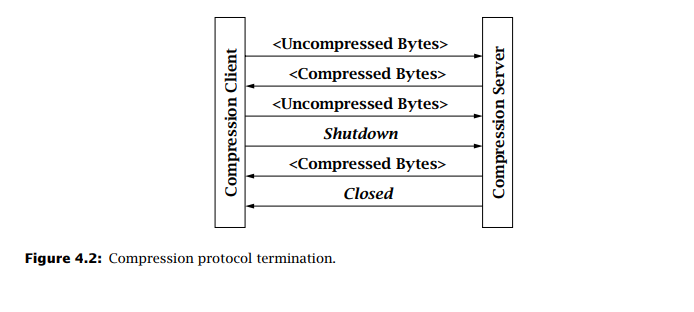
4.3.1 Broadcast  
Broadcasting UDP datagrams is similar to unicasting datagrams, except that a *broadcast  
address* is used instead of a regular (unicast) IP address. Note that IPv6 does not explicitly  
provide broadcast addresses; however, there is a special all-nodes, link-local-scope multicast  
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**





**CompressClient.java**0 import java.net.Socket;  
1 import java.io.IOException;  
2 import java.io.InputStream;  
3 import java.io.OutputStream;  
4 import java.io.FileInputStream;  
5 import java.io.FileOutputStream;  
6 7  
/\* WARNING: this code can deadlock if a large file (more than a few  
8 \* 10's of thousands of bytes) is sent.  
9 \*/  
10  
11 public class CompressClient {  
12  
13 public static final int BUFSIZE = 256; // Size of read buffer  
14  
15 public static void main(String[] args) throws IOException {  
16  
17 if (args.length != 3) { // Test for correct # of args  
18 throw new IllegalArgumentException("Parameter(s): <Server> <Port> <File>");  
19 }  
20  
21 String server = args[0]; // Server name or IP address  
22 int port = Integer.parseInt(args[1]); // Server port  
23 String filename = args[2]; // File to read data from  
24  
25 // Open input and output file (named input.gz)  
26 FileInputStream fileIn = new FileInputStream(filename);  
27 FileOutputStream fileOut = new FileOutputStream(filename + ".gz");  
28  
29 // Create socket connected to server on specified port  
30 Socket sock = new Socket(server, port);  
31  
32 // Send uncompressed byte stream to server  
33 sendBytes(sock, fileIn);  
34  
35 // Receive compressed byte stream from server  
36 InputStream sockIn = sock.getInputStream();  
37 int bytesRead; // Number of bytes read  
38 byte[] buffer = new byte[BUFSIZE]; // Byte buffer  
39 while ((bytesRead = sockIn.read(buffer)) != -1) {  
40 fileOut.write(buffer, 0, bytesRead);  
41 System.out.print("R"); // Reading progress indicator  
42 }  
43 System.out.println(); // End progress indicator line  
44  
45 sock.close(); // Close the socket and its streams  
46 fileIn.close(); // Close file streams  
47 fileOut.close();  
48 }  
49  
50 private static void sendBytes(Socket sock, InputStream fileIn)  
51 throws IOException {  
52 OutputStream sockOut = sock.getOutputStream();  
53 int bytesRead; // Number of bytes read  
54 byte[] buffer = new byte[BUFSIZE]; // Byte buffer  
55 while ((bytesRead = fileIn.read(buffer)) != -1) {  
56 sockOut.write(buffer, 0, bytesRead);  
57 System.out.print("W"); // Writing progress indicator  
58 }  
59 sock.shutdownOutput(); // Finished sending  
60 }  
61 }  
**CompressClient.java**1. **Application setup and parameter parsing:** lines 17–23  
2. **Create socket and open files:** lines 25–30  
3. **Invoke** sendBytes() **to transmit bytes:** line 33  
4. **Receive the compressed data stream:** lines 35–42  
The while loop receives the compressed data stream and writes the bytes to the output  
file until an end-of-stream is signaled by a *-*1 from read().  
5. **Close socket and file streams:** lines 45–47  
6. sendBytes()**:** lines 50–60  
Given a socket connected to a compression server and the file input stream, read all of  
the uncompressed bytes from the file and write them to the socket output stream.  
**Get socket output stream:** line 52 **Send uncompressed bytes to compression server:** lines 55–58  
The while loop reads from the input stream (in this case from a file) and repeats the  
bytes to the socket output stream until end-of-file, indicated by *-*1 from read(). Each  
write is indicated by a “W” printed to the console.  
**Shut down the socket output stream:** line 59  
After reading and sending all of the bytes from the input file, shut down the output  
stream, notifying the server that the client is finished sending. The close will cause a  
*-*1 return from read() on the server.

**CompressProtocol.java**0 import java.net.Socket;  
1 import java.io.IOException;  
2 import java.io.InputStream;  
3 import java.io.OutputStream;  
4 import java.util.zip.GZIPOutputStream;  
5 import java.util.logging.Logger;  
6 import java.util.logging.Level;  
7 8  
public class CompressProtocol implements Runnable {  
9  
10 public static final int BUFSIZE = 1024; // Size of receive buffer  
11 private Socket clntSock;  
12 private Logger logger;  
13  
14 public CompressProtocol(Socket clntSock, Logger logger) {  
15 this.clntSock = clntSock;  
16 this.logger = logger;  
17 }  
18  
19 public static void handleCompressClient(Socket clntSock, Logger logger) {  
20 try {  
21 // Get the input and output streams from socket  
22 InputStream in = clntSock.getInputStream();  
23 GZIPOutputStream out = new GZIPOutputStream(clntSock.getOutputStream());  
24  
25 byte[] buffer = new byte[BUFSIZE]; // Allocate read/write buffer  
26 int bytesRead; // Number of bytes read  
27 // Receive until client closes connection, indicated by -1 return  
28 while ((bytesRead = in.read(buffer)) != -1)  
29 out.write(buffer, 0, bytesRead);  
30 out.finish(); // Flush bytes from GZIPOutputStream  
31  
32 logger.info("Client " + clntSock.getRemoteSocketAddress() + " finished");  
33 } catch (IOException ex) {  
34 logger.log(Level.WARNING, "Exception in echo protocol", ex);  
35 }  
36  
37 try { // Close socket  
38 clntSock.close();  
39 } catch (IOException e) {  
40 logger.info("Exception = " + e.getMessage());  
41 }  
42 }  
43  
44 public void run() {  
45 handleCompressClient(this.clntSock, this.logger);  
46 }  
47 }  
**CompressProtocol.java**1. **Variables and constructors:** lines 10–17  
2. handleCompressClient()**:** lines 19–42  
Given a socket connected to the compression client, read the uncompressed bytes from  
the client and write the compressed bytes back.  
**Get socket I/O streams:** lines 22–23  
The socket’s output stream is wrapped in a GZIPOutputStream. The sequence of bytes  
written to this stream is compressed, using the GZIP algorithm, before being written  
to the underlying output stream.  
**Read uncompressed and write compressed bytes:** lines 28–29  
The while loop reads from the socket input stream and writes to the GZIPOutputStream,  
which in turn writes to the socket output stream, until the end-of-stream indication is  
received.  
**Flush and close:** lines 30–42  
Calling finish on the GZIPOutputStream is necessary to flush any bytes that may be  
buffered by the compression algorithm.  
run() **method:** lines 44–46  
The run() method simply calls the handleCompressClient() method.To use this protocol we simply modify TCPEchoServerExecutor.java to create an instance  
of CompressProtocol instead of EchoProtocol:  
service.execute(new CompressProtocol(clntSock, logger));

**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.  
**4.7 Wrapping Up**We have discussed some of the ways Java provides access to advanced features of the sockets  
API, and how built-in features such as threads and executors can be used with socket programs.  
In addition to these facilities, Java provides several mechanisms (not discussed here) that  
operate on top of TCP or UDP and attempt to hide the complexity of protocol development. For  
example, Java Remote Method Invocation (RMI) allows Java objects on different hosts to invoke  
one another’s methods as if the objects all reside locally. The URL class and associated classes  
provide a framework for developing Web-related programs. Many other standard Java library  
mechanisms exist, providing an amazing range of services. These mechanisms are beyond the  
scope of this book; however, we encourage you to look at the book’s Web site for descriptions  
and code examples for some of these libraries.