Chapter 4: Beyond the Basics

4.1 Multitasking

In this section we describe two approaches to coding concurrent servers, namely, thread per-client, where a new thread is spawned to handle each client connection, and thread pool, where connections are assigned to a prespawned set of threads. We shall also describe the built-in Java facilities that simplify the use of these strategies for multithreaded servers.

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.

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.

4.1.5 System-Managed Dispatching: The Executor Interface

In the previous subsections, we have seen that encapsulating the details of the client-server protocol (as in EchoProtocol.java) lets us use different “dispatching” methods with the same protocol implementation (e.g., TCPEchoServerThread.java and TCPEchoServerThreadPool.java). In fact the same thing is true for the dispatching methods themselves. The interface Executor (part of the java.util.concurrent package) represents an object that executes Runnable instances according to some strategy, which may include details about queueing and scheduling, or how jobs are selected for execution.

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).

4.2.3 Limiting Per-Client Time

The TimelimitEchoProtocol class is similar to the EchoProtocol class, except that it attempts to bound the total time an echo connection can exist to 10 seconds. At the time the handleEchoClient() method is invoked, a deadline is computed using the current time and the time bound. After each read(), the time between the current time and the deadline is computed, and the socket timeout is set to the remaining time.

4.3 Multiple Recipients

So far all of our sockets have dealt with communication between exactly two entities, usually a server and a client. Such one-to-one communication is sometimes called unicast. Some information is of interest to multiple recipients. In such cases, we could unicast a copy of the data to each recipient, but this may be very inefficient.

There are two types of one-to-many service: broadcast and multicast. With broadcast, all hosts on the (local) network receive a copy of the message. With multicast, the message is sent to a multicast address, and the network delivers it only to those hosts that have indicated that they want to receive messages sent to that address. In general, only UDP sockets are allowed to broadcast or multicast.

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

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 a range 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.

4.4 Controlling Default Behaviors

The TCP/IP protocol developers spent a good deal of time thinking about the default behaviors that would satisfy most applications.

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.

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.

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. 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.

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

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.

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.

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.

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.

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.

4.5 Closing Connections

Calling close() on a Socket terminates both directions (input and output) of data flow.

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.

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.