**Multitasking**

What we need is some way for each connection to proceed independently, without interfering with other connections. Java threads provide exactly that: a convenient mechanism allowing servers to handle many clients simultaneously.

In this section we describe two approaches to coding concurrent servers, namely, threadper-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.

**Java Threads**

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.

Threads are perfect for implementing servers, in which each client’s processing is independent of that provided to every other client. However, it is a different story when client processing involves updating information that is shared across threads on the server. In that case, great care must be taken to ensure that different threads are properly synchronized with respect to the shared data; otherwise, the shared information can get into an inconsistent state, and moreover, the problem can be very difficult to trace.

**Server Protocol**

The code for the echo protocol is given in the class EchoProtocol. This class encapsulates the per-client processing in the static method handleEchoClient().

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().

Now a user reports a problem. How do you determine what happened? Is the problem at your server? Perhaps the client is violating the protocol. To deal with this scenario, most servers log their activities.

The logger sends messages to one or more Handlers, which “handle” publishing the messages. By default, a logger has a single ConsoleHandler that prints messages to System.err.

An important characteristic of Logger for our purposes is that it is thread-safe—that is, its methods can be called from different threads running concurrently without requiring additional synchronization among the callers. Without this feature, different messages logged by different threads might end up being interleaved in the log

**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.

**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.

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.

**System-Managed Dispatching: The executor Interface**

Java 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.

The ExecutorService interface extends Executor to provide a more sophisticated facility that allows a service to be shut down, either gracefully or abruptly. ExecutorService also allows for tasks to return a result, through the Callable interface, which is like Runnable, only with a return value.

Instances of ExecutorService can be obtained by calling various static factory methods of the convenience class Executors. The program TCPEchoServerExecutor.java illustrates the use of the basic Executor facilities.

**Blocking and Timeouts**

Socket I/O calls may block for several reasons. In all cases, the method returns only after the request has been satisfied. Of course, a blocked method call halts progress of the application (and makes the thread that is running it useless).

What about a program that has other tasks to perform while waiting for call completion (e.g., updating the “busy” cursor or responding to user requests)? These programs may have no time to wait on a blocked method call. What about lost UDP datagrams? If we block waiting to receive a datagram and it is lost, we could block indefinitely. Here we explore the various blocking methods and approaches for limiting blocking behavior.

**Connecting and Writing**

To establish a connection, call the connect() method on the newly constructed socket and specify both a remote endpoint and timeout (milliseconds).

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.)

**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.

**Multiple Recipients**

Fortunately, networks provide a way to use bandwidth more efficiently. Instead of making the sender responsible for duplicating packets, we can give this job to the network.

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.

**Broadcast**

Broadcasting UDP datagrams is similar to unicasting datagrams, except that a broadcast address is used instead of a regular (unicast) IP address.

There is no networkwide broadcast address that can be used to send a message to all hosts.

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.

**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.

The only significant differences between our unicast and multicast senders are that 1) we verify that the given address is multicast, and 2) we set the initial Time To Live (TTL) value for the multicast datagram.

Unlike broadcast, network multicast duplicates the message only to a specific set of receivers.

The decision to use broadcast or multicast depends on several factors, including the network location of receivers and the knowledge of the communicating parties.

UDP unicast, multicast, and broadcast are all implemented using an underlying UDP socket.

**Controlling Default Behaviors**

The TCP/IP protocol developers spent a good deal of time thinking about the default behaviors that would satisfy most applications. (If you doubt this, read RFCs 1122 and 1123, which describe in excruciating detail the recommended behaviors—based on years of experience— for implementations of the TCP/IP protocols.) For most applications, the designers did a good job; however, it is seldom the case that “one size fits all” really fits all

**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.

**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.)

**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.

**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.

**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.

**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.

**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.

**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.

**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.

**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.

**Closing Connections**

Network protocols, on the other hand, are typically very specific about who “closes” first. The close is a critical part of the protocol because without it the server doesn’t know when the client is finished sending characters to echo.

Calling close() on a Socket terminates both directions (input and output) of data flow. Once an endpoint (client or server) closes the socket, it can no longer send or receive data.

Let’s consider a different protocol. Suppose you want a compression server that takes a stream of bytes, compresses them, and sends the compressed stream back to the client. Which endpoint should close the connection?

Fortunately, sockets provide a way to do this. The shutdownInput() and shutdownOutput() methods of Socket allow the I/O streams to be closed independently.

**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.