**BEYOND THE BASICS**

**4.1 Multitasking**

* Iterative servers handle clients sequentially, finishing with one client before servicing the next. They work best for applications where each client requires a small => Need servers to handle many clients **simultaneously**.

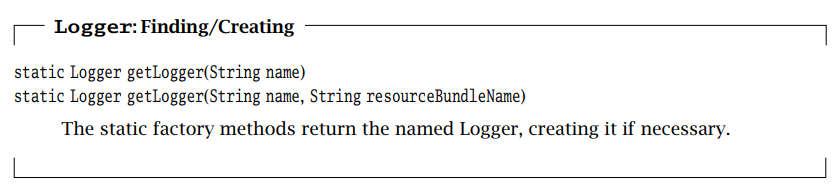
**4.1.1 Java Threads**

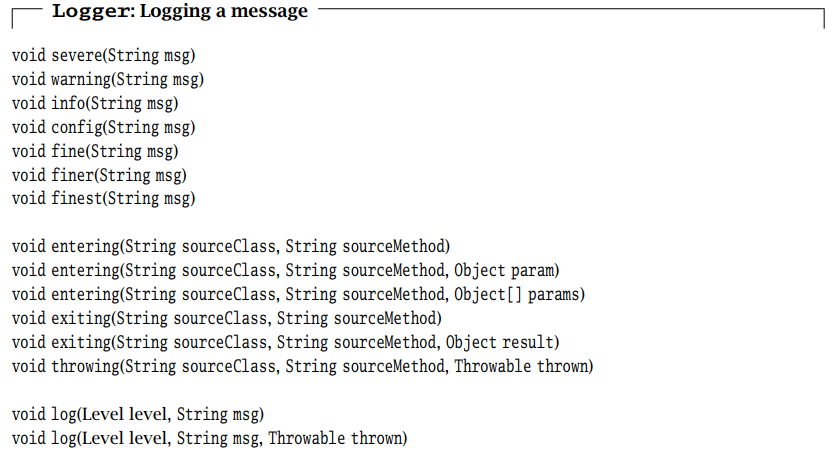
* Java provides **2 approaches** for **performing a task in a new thread**:

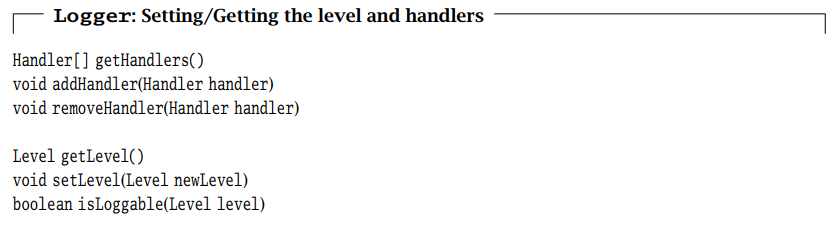
1. Define a subclass of the Thread class
2. Define a class that implements the Runnable interface and passing an instance of that class to the Thread constructor.

* Threads are perfect for implementing servers in which each client’s processing is independent.
* However, it is a different story when client processing involves updating information that is shared across threads on the server.

**4.1.2 Server Protocol**







**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**

* As the number of threads increases, more and more system resources are consumed by thread overhead => increase client service time.

=> limiting the total number of threads and reusing threads is necessary. 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.

* Since each thread in the pool loops forever, processing connections one by one.

=> A thread-pool server is 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.

**4.1.5 System-Managed Dispatching: The Executor Interface**

The **interface Executor** 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**

* Socket I/O calls may block for several reasons.
* Example:
  + Data input methods read() and receive() block if data is not available.
  + A write() on a TCP socket may block if there is not sufficient space to buffer the transmitted data.
  + The accept() method of ServerSocket() and the Socket constructor both block until a connection has been established

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

Set a bound on the maximum time (in milliseconds) to block, using the **setSoTimeout()** method.

**4.2.2 Connecting and Writing**

* Call the **connect()** method on the newly constructed socket and specify both a remote endpoint and timeout.
* The amount of time that a **write()** may block is ultimately controlled by the receiving application.
* Notice: any protocol that sends a large enough amount of data over a Socket instance can block for an unbounded amount of time.

**4.2.3 Limiting Per-Client Time**

* 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.

**4.3 Multiple Recipients**

* Instead of making the sender responsible for duplicating packets, we can give this job to the network.
* There are 2 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

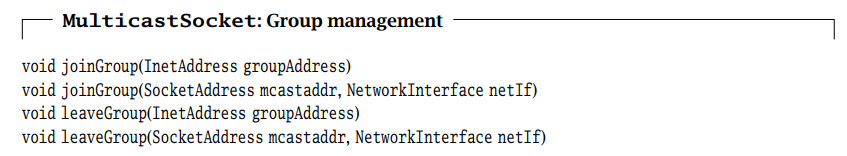
**4.3.1 Broadcast**

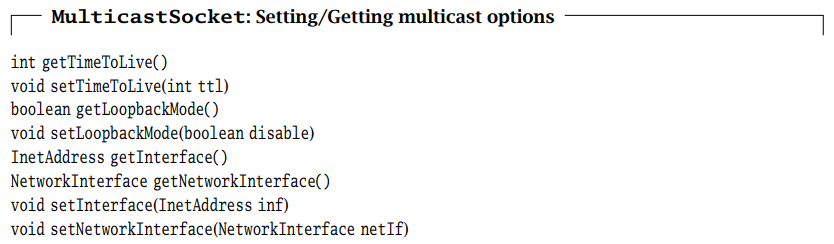
There is no networkwide **broadcast** address that can be used to send a message to all hosts.

**4.3.2 Multicast**

* With the exception of a few reserved multicast addresses, a sender can send datagrams addressed to any address in this range.
* Unlike broadcast, **network multicast** duplicates the message only to a specific set of receivers.







**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.

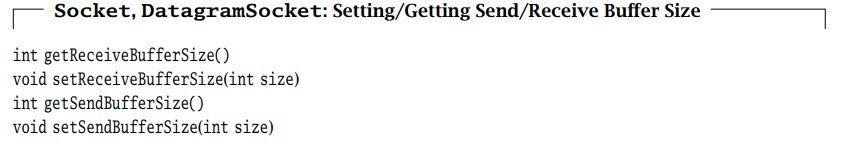
**4.4.1 Keep-Alive**

* If 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.



**4.4.2 Send and Receive Buffer Size**

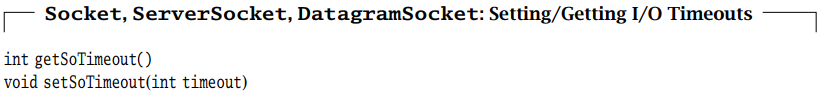
The OS must allocate **buffers** to hold incoming and outgoing data.





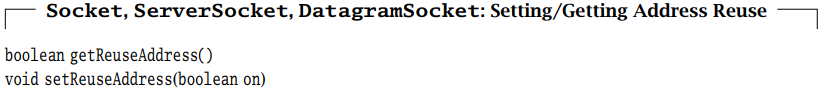
**4.4.3 Timeout**

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.



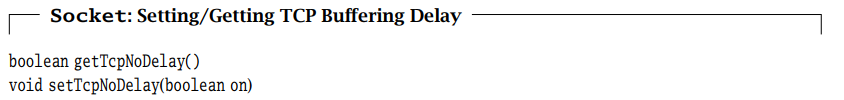
**4.4.4 Address Reuse**

Under some circumstances, you may want to allow multiple sockets to bind to the same socket address



**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.



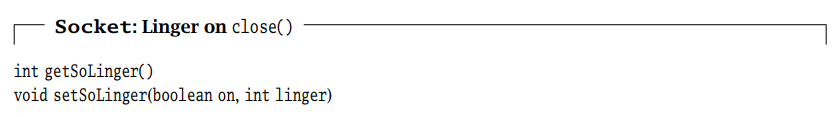
**4.4.6 Urgent Data**

If you send the data in the output stream, it gets queued up behind all of the regular data.



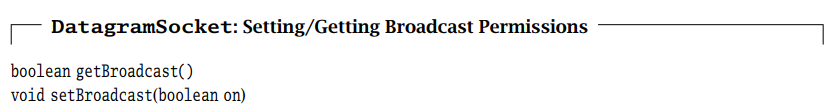
**4.4.7 Lingering after close**

When you call close() on a socket, it immediately returns even if the socket is buffering unsent data.



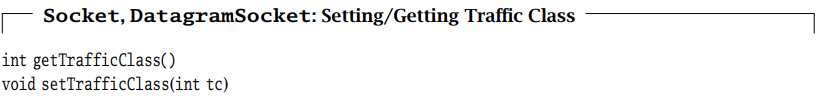
**4.4.8 Broadcast Permission**

Some operating systems require that you explicitly request permission to broadcast.



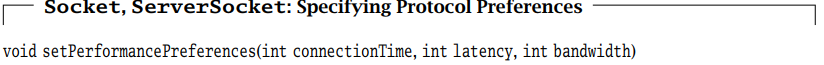
**4.4.9 Traffic Class**

Some networks offer enhanced or “premium” services to packets classified as being eligible for the service.

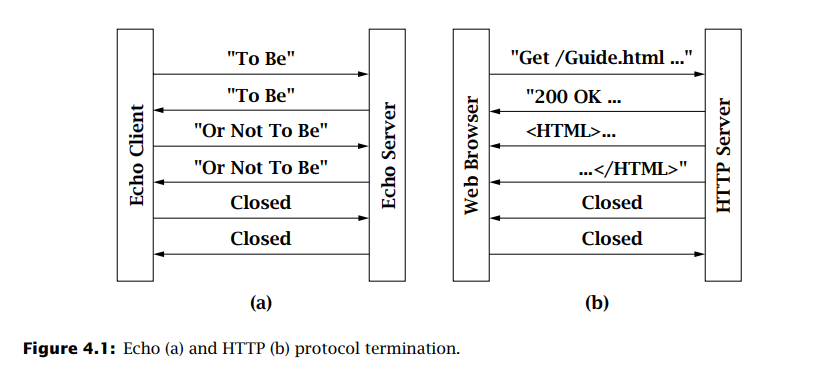


**4.4.10 Performance-Based Protocol Selection**

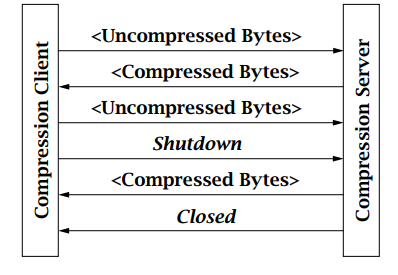
TCP may not be the only protocol available to a socket. Which protocol to use depends on what is important to your application.



**4.5 Closing Connections**

Network protocols are typically very specific about who “closes” first.

a) When the client is finished, it calls **close().** After the server has received and echoed all of the data sent before the client’s call to close(), its read operation returns a −1, indicating that the client is finished. The server then calls close() on its socket.

b) When the server that initiates the connection close. Then, the client sends a request (“get”) to the server, and the server responds by sending a header. Since the client does not know the size of the file, the server must indicate the end-of-file by closing the socket.

In the compression protocol:

1. The client writes the bytes to be compressed, closing the output stream and reads the compressed byte stream from the server.
2. The server repeatedly reads the uncompressed data and writes the compressed data until the client performs a shutdown.
3. The server then closes the connection and exits.