

A decorative graphic on the left side of the slide consisting of a network of blue and teal lines and circles, resembling a circuit board or a neural network, extending from the top and bottom edges.

# NETWORK PROGRAMMING


CHAPTER 7 : SOCKETS FOR SERVERS

CHANDAN GUPTA BHAGAT

<https://me.chandanbhagat.com.np>



# CONTENT

- Using ServerSockets: Serving Binary Data, Multithreaded Servers, Writing to Servers with Sockets and Closing Server Sockets
  - Logging: What to Log and How to Log
  - Constructing Server Sockets: Constructing Without Binding
  - Getting Information about Server Socket
  - Socket Options: SO\_TIMEOUT, SO\_RSUMEADDR, SO\_RCVBUF and Class of Service
  - HTTP Servers: A Single File Server, A Redirector and A Full-Fledged HTTP Server
- 

# Sockets for Server

- The previous chapter discussed sockets from the standpoint of *clients*: programs that open a socket to a server that's listening for connections. However, client sockets themselves aren't enough; clients aren't much use unless they can talk to a server, and the Socket class discussed in the previous chapter is not sufficient for writing servers. To create a Socket, you need to know the Internet host to which you want to connect. When you're writing a server, you don't know in advance who will contact you; and even if you did, you wouldn't know when that host wanted to contact you. In other words, servers are like receptionists who sit by the phone and wait for incoming calls. They don't know who will call or when, only that when the phone rings, they have to pick it up and talk to whoever is there. You can't program that behavior with the Socket class alone.
- For servers that accept connections, Java provides a ServerSocket class that represents server sockets. In essence, a server socket's job is to sit by the phone and wait for incoming calls. More technically, a server socket runs on the server and listens for incoming TCP connections. Each server socket listens on a particular port on the server machine. When a client on a remote host attempts to connect to that port, the server wakes up, negotiates the connection between the client and the server, and returns a regular Socket object representing the socket between the two hosts. In other words, server sockets wait for connections while client sockets initiate connections. Once a ServerSocket has set up the connection, the server uses a regular Socket object to send data to the client. Data always travels over the regular socket.

The background features a series of concentric, light gray circles centered on the page. In each of the four corners, there are stylized circuit board traces in a dark blue color. These traces consist of straight lines of varying lengths and angles, some terminating in small open circles, creating a technical, digital aesthetic.

# Using Server Sockets

# Using Server Sockets

- The `ServerSocket` class contains everything needed to write servers in Java. It has constructors that create new `ServerSocket` objects, methods that listen for connections on a specified port, methods that configure the various server socket options, and the usual miscellaneous methods such as `toString()`. In Java, the basic life cycle of a server program is this:
  1. A new `ServerSocket` is created on a particular port using a `ServerSocket()` constructor.
  2. The `ServerSocket` listens for incoming connection attempts on that port using its `accept()` method. `accept()` blocks until a client attempts to make a connection, at which point `accept()` returns a `Socket` object connecting the client and the server.
  3. Depending on the type of server, either the `Socket`'s `getInputStream()` method, `getOutputStream()` method, or both are called to get input and output streams that communicate with the client.
  4. The server and the client interact according to an agreed-upon protocol until it is time to close the connection.
  5. The server, the client, or both close the connection.
  6. The server returns to step 2 and waits for the next connection.

# Using Server Sockets

- Implementing your own daytime server is easy. First, create a server socket that listens on port 13:

```
ServerSocket server = new ServerSocket(13);
```

Next, accept a connection:

```
Socket connection = server.accept();
```

- The `accept()` call *blocks*. That is, the program stops here and waits, possibly for hours or days, until a client connects on port 13. When a client does connect, the `accept()` method returns a `Socket` object.
- The daytime protocol requires the server (and only the server) to talk, so get an `OutputStream` from the socket. Because the daytime protocol requires text, chain this to an `OutputStreamWriter`:

```
OutputStream out = connection.getOutputStream();
```

```
Writer writer = new OutputStreamWriter(out, "ASCII")
```

- The daytime protocol doesn't require any particular format other than that it be human readable, so let Java pick for you:

```
Date now = new Date();
```

```
out.write(now.toString() + "\r\n");
```

# Using Server Sockets

- Finally, flush the connection and close it:

```
out.flush();
```

```
connection.close();
```

- You won't always have to close the connection after just one write. Many protocols, dict and HTTP 1.1 for instance, allow clients to send multiple requests over a single socket and expect the server to send multiple responses.
- If the client closes the connection while the server is still operating, the input and/or output streams that connect the server to the client throw an `IOException` on the next read or write. In either case, the server should then get ready to process the next incoming connection.

# Using Server Sockets

```
ServerSocket server = new ServerSocket(port);  
  
while (true) {  
    try (Socket connection = server.accept()) {  
        Writer out = new OutputStreamWriter(connection.getOutputStream());  
        Date now = new Date();  
        out.write(now.toString() + "\r\n");  
        out.flush();  
    } catch (IOException ex) {  
        // problem with one client; don't shut down the server  
        System.err.println(ex.getMessage());  
    }  
}
```



# Using Server Sockets

- This is called an *iterative* server. There's one big loop, and in each pass through the loop a single connection is completely processed. This works well for a very simple protocol with very small requests and responses like daytime, though even with this simple a protocol it's possible for one slow client to delay other faster clients. Upcoming examples will address this with multiple threads or asynchronous I/O.
- When exception handling is added, the code becomes somewhat more convoluted. It's important to distinguish between exceptions that should probably shut down the server and log an error message, and exceptions that should just close that active connection.
- Exceptions within the scope of a particular connection should close that connection, but not affect other connections or shut down the server. Exceptions outside the scope of an individual request probably should shut down the server.

# Using Server Sockets

```
ServerSocket server = null;
try {
    server = new ServerSocket(port);
    while (true) {
        Socket connection = null;
        try {
            connection = server.accept();
            Writer out = new
OutputStreamWriter(connection.getOutputStream());

            Date now = new Date();
            out.write(now.toString() + "\r\n");
            out.flush();
            connection.close();
        } catch (IOException ex) {
            // this request only; ignore
```

```
} finally {
    try {
        if (connection != null)
            connection.close();
    } catch (IOException ex) {}
}
} catch (IOException ex) {
    ex.printStackTrace();
} finally {
    try {
        if (server != null) server.close();
    } catch (IOException ex) {}
}
}
```

***Always close a socket when you're finished with it.***

# Serving Binary Data

- Sending binary, nontext data is not significantly harder. You just use an Output Stream that writes a byte array rather than a Writer that writes a String.
- Code here demonstrates with an iterative time server that follows the time protocol outlined in RFC 868. When a client connects, the server sends a 4-byte, big-endian, unsigned integer specifying the number of seconds that have passed since 12:00 A.M., January 1, 1900, GMT (the epoch).
- Once again, the current time is found by creating a new Date object. However, because Java's Date class counts milliseconds since 12:00 A.M., January 1, 1970, GMT rather than seconds since 12:00 A.M., January 1, 1900, GMT, some conversion is necessary.

# Using Server Sockets

```
long differenceBetweenEpochs = 2208988800L;
try (ServerSocket server = new ServerSocket(PORT)) {
    while (true) {
        try (Socket connection = server.accept()) {
            OutputStream out =
connection.getOutputStream();

            Date now = new Date();
            long msSince1970 = now.getTime();
            long secondsSince1970 = msSince1970/1000;
            long secondsSince1900 = secondsSince1970
+ differenceBetweenEpochs;

            byte[] time = new byte[4];
            time[0] = (byte) ((secondsSince1900 &
0x00000000FF000000L) >> 24);
            time[1] = (byte) ((secondsSince1900 &
```

```
0x0000000000FF0000L) >> 16);
            time[2] = (byte) ((secondsSince1900 &
0x000000000000FF00L) >> 8);
            time[3] = (byte) (secondsSince1900 &
0x00000000000000FFL);

            out.write(time);
            out.flush();
        } catch (IOException ex) {
            System.err.println(ex.getMessage());
        }
    }
} catch (IOException ex) {
    System.err.println(ex);
}
```

# Multithreaded Servers

- Daytime and time are both very quick protocols. The server sends a few dozen bytes at most and then closes the connection. It's plausible here to process each connection fully before moving on to the next one. Even in that case, though, it is possible that a slow or crashed client might hang the server for a few seconds until it notices the socket is broken. If the sending of data can take a significant amount of time even when client and server are behaving, you really don't want each connection to wait for the next.
- Old-fashioned Unix servers such as `wu-ftpd` create a new process to handle each connection so that multiple clients can be serviced at the same time. Java programs should spawn a thread to interact with the client so that the server can be ready to process the next connection sooner. A thread places a far smaller load on the server than a complete child process. In fact, the overhead of forking too many processes is why the typical Unix FTP server can't handle more than roughly 400 connections without slowing to a crawl. On the other hand, if the protocol is simple and quick and allows the server to close the connection when it's through, it will be more efficient for the server to process the client request immediately without spawning a thread.

# Multithreaded Servers

- The operating system stores incoming connection requests addressed to a particular port in a first-in, first-out queue. By default, Java sets the length of this queue to 50, although it can vary from operating system to operating system. Some operating systems (not Solaris) have a maximum queue length. For instance, on FreeBSD, the default maximum queue length is 128. On these systems, the queue length for a Java server socket will be the largest operating-system allowed value less than or equal to 50. After the queue fills to capacity with unprocessed connections, the host refuses additional connections on that port until slots in the queue open up. Many (though not all) clients will try to make a connection multiple times if their initial attempt is refused. Several
- ServerSocket constructors allow you to change the length of the queue if its default length isn't large enough. However, you won't be able to increase the queue beyond the maximum size that the operating system supports. Whatever the queue size, though, you want to be able to empty it faster than new connections are coming in, even if it takes a while to process each connection. The solution here is to give each connection its own thread, separate from the thread that accepts incoming connections into the queue.

# Using Server Sockets

```
public final static int PORT = 13;
public static void main(String[] args) {
    try (ServerSocket server = new ServerSocket(PORT)) {
        while (true) {
            try {
                Socket connection = server.accept();
                Thread task = new DaytimeThread(connection);
                task.start();
            } catch (IOException ex) {}
        }
    } catch (IOException ex) {
        System.err.println("Couldn't start server");
    }
}
```

# Using Server Sockets

```
private static class DaytimeThread extends Thread {  
    private Socket connection;  
    DaytimeThread(Socket connection) {  
        this.connection = connection;  
    }  
}
```

```
@Override  
public void run() {  
    try {  
        Writer out = new  
            OutputStreamWriter  
(connection.getOutputStream());  
        Date now = new Date();
```

```
        out.write(now.toString() + "\r\n");  
        out.flush();  
    } catch (IOException ex) {  
        System.err.println(ex);  
    } finally {  
        try {  
            connection.close();  
        } catch (IOException e) {  
            // ignore;  
        }  
    }  
}
```



# Writing to Servers with Sockets

- In the examples so far, the server has only written to client sockets. It hasn't read from them. Most protocols, however, require the server to do both. This isn't hard. You'll accept a connection as before, but this time ask for both an `InputStream` and an `OutputStream`. Read from the client using the `InputStream` and write to it using the `OutputStream`. The main trick is understanding the protocol: when to write and when to read.
- The echo protocol, defined in RFC 862, is one of the simplest interactive TCP services. The client opens a socket to port 7 on the echo server and sends data. The server sends the data back. This continues until the client closes the connection. The echo protocol is useful for testing the network to make sure that data is not mangled by a misbehaving router or firewall.

```
$ telnet rama.poly.edu 7
Trying 128.238.10.212...
Connected to rama.poly.edu.
Escape character is '^]'.
This is a test
This is a test
```

```
This is another test
This is another test
9876543210
9876543210
^]
telnet> close
Connection closed.
```

# Closing Server Sockets

- When finished, the server socket should be closed. This frees up the port for other programs that may wish to use it. Closing a `ServerSocket` should not be confused with closing a `Socket`. Closing a `ServerSocket` frees a port on the local host, allowing another server to bind to the port; it also breaks all currently open sockets that the `ServerSocket` has accepted.
- Server sockets are closed automatically when a program dies, so it's not absolutely necessary to close them in programs that terminate shortly after the `ServerSocket` is no longer needed. Nonetheless, it doesn't hurt. Programmers often follow the same close if-not-null pattern in a try-finally block that you're already familiar with from streams and client-side sockets.
- In Java 7, `ServerSocket` implements `AutoCloseable` so you can take advantage of try-with-resources instead:

```
try (ServerSocket server = new ServerSocket(port)) {  
    // ... work with the server socket  
}
```

# Closing Server Sockets

- After a server socket has been closed, it cannot be reconnected, even to the same port. The `isClosed()` method returns true if the `ServerSocket` has been closed, false if it hasn't:

```
public boolean isClosed()
```

- `ServerSocket` objects that were created with the noargs `ServerSocket()` constructor and not yet bound to a port are not considered to be closed. Invoking `isClosed()` on these objects returns false. The `isBound()` method tells you whether the `ServerSocket` has been bound to a port:

```
public boolean isBound()
```

- As with the `isBound()` method of the `Socket` class discussed in the [Chapter 8](#), the name is a little misleading. `isBound()` returns true if the `ServerSocket` has ever been bound to a port, even if it's currently closed. If you need to test whether a `ServerSocket` is open, you must check both that `isBound()` returns true and that `isClosed()` returns false. For example:

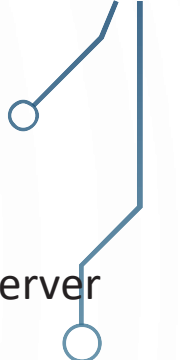
```
public static boolean isOpen(ServerSocket ss) {  
    return ss.isBound() && !ss.isClosed();  
}
```



# Logging



# Logging

- Servers run unattended for long periods of time. It's often important to debug what happened when in a server long after the fact.
  - It's advisable to store server logs for at least some period of time
- 

# What to Log

- There are two primary things you want to store in your logs:
  - Requests
  - Server errors
- The audit log usually contains one entry for each connection made to the server. Servers that perform multiple operations per connection may have one entry per operation instead. For instance, a dict server might log one entry for each word a client looks up.
- The error log contains mostly unexpected exceptions that occurred while the server was running. For instance, any `NullPointerException` that happens should be logged here because it indicates a bug in the server you'll need to fix.
- The error log does not contain client errors, such as a client that unexpectedly disconnects or sends a malformed request. These go into the request log. The error log is exclusively for unexpected exceptions.

# What to Log

- The general rule of thumb for error logs is that every line in the error log should be looked at and resolved. The ideal number of entries in an error log is zero. Every entry in this log represents a bug to be investigated and resolved.
- If investigation of an error log entry ends with the decision that that exception is not really a problem, and the code is working as intended, remove the log statement.
- Error logs that fill up with too many false alarms rapidly become ignored and useless. For the same reason, do not keep debug logs in production. Do not log every time you enter a method, every time a condition is met, and so on. No one ever looks at these logs. They just waste space and hide real problems.
- If you need method-level logging for debugging, put it in a separate file, and turn it off in the global properties file when running in production
- More advanced logging systems provide log analysis tools that enable you to do things like show only messages with priority INFO or higher, or only show messages that originated from a certain part of the code.

# What to Log

- These tools make it more feasible to keep a single logfile or database, perhaps even share one log among many different binaries or programs. Nonetheless, the principle still applies that a log record no one will ever look at is worthless at best and more often than not distracting or confusing.
- Do not follow the common antipattern of logging everything you can think of just in case someone might need it someday. In practice, programmers are terrible at guessing in advance which log messages they might need for debugging production problems.
- Once a problem occurs, it is sometimes obvious what messages you need; but it is rare to be able to anticipate this in advance. Adding “just in case” messages to logfiles usually means that when a problem does occur, you’re frantically hunting for the relevant messages in an even bigger sea of irrelevant data.



# How to Log

- Many legacy programs dating back to Java 1.3 and earlier still use third-party logging libraries such as log4j or Apache Commons Logging, but the `java.util.logging` package available since Java 1.4 suffices for most needs. Choosing it avoids a lot of complex third-party dependencies. Although you can load a logger on demand, it's usually easiest to just create one per class like so:

```
private final static Logger auditLogger = Logger.getLogger("requests");
```

- Loggers are thread safe, so there's no problem storing them in a shared static field. Indeed, they almost have to be because even if the `Logger` object were not shared between threads, the logfile or database would be. This is important in highly multithreaded servers. This example outputs to a log named "requests." Multiple `Logger` objects can output to the same log, but each logger always logs to exactly one log. What and where the log is depends on external configuration.
- Most commonly it's a file, which may or may not be named "requests"; but it can be a database, a SOAP service running on a different server, another Java program on the same host, or something else.

# How to Log

- Once you have a logger, you can write to it using any of several methods. The most basic is `log()`. For example, this catch block logs an unexpected runtime exception at the highest level:


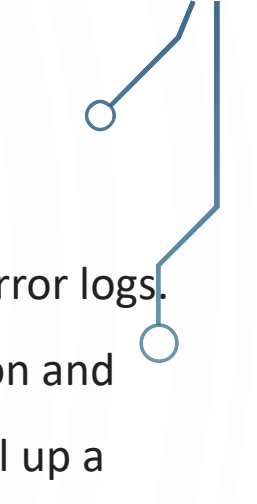
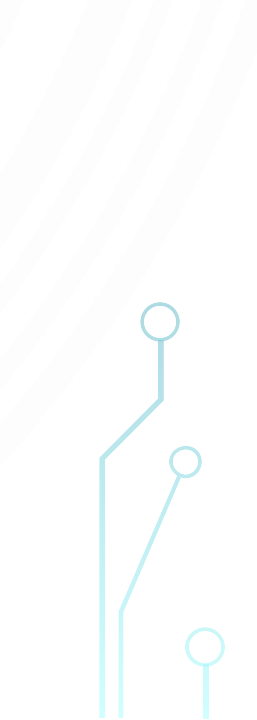
```
catch (RuntimeException ex) {  
    logger.log(Level.SEVERE, "unexpected error " + ex.getMessage(), ex);  
}
```

- Including the exception instead of just a message is optional but customary when logging from a catch block. There are seven levels defined as named constants in `java.util.logging.Level` in descending order of seriousness:

- `Level.SEVERE` (highest value)
- `Level.WARNING`
- `Level.INFO`
- `Level.CONFIG`
- `Level.FINE`
- `Level.FINER`
- `Level.FINEST` (lowest value)



# How to Log

- Finally, once you've configured your servers with logging, don't forget to look in them, especially the error logs. There's no point to a logfile no one ever looks at. You'll also want to plan for and implement log rotation and retention policies. Hard drives get bigger every year, but it's still possible for a high-volume server to fill up a filesystem with log data if you aren't paying attention.
  - Murphy's law says this is most likely to happen at 4:00 A.M. on New Year's Day when you're on vacation halfway around the world.
- 
- 
- 



# Constructing Server Sockets

# Constructing Server Sockets

- There are four public ServerSocket constructors:

```
public ServerSocket(int port) throws BindException, IOException
```

```
public ServerSocket(int port, int queueLength) throws BindException, IOException
```

```
public ServerSocket(int port, int queueLength, InetAddress bindAddress) throws IOException
```

```
public ServerSocket() throws IOException
```

- These constructors specify the port, the length of the queue used to hold incoming connection requests, and the local network interface to bind to. They pretty much all do the same thing, though some use default values for the queue length and the address to bind to. For example, to create a server socket that would be used by an HTTP server on port 80, you would write:

```
ServerSocket httpd = new ServerSocket(80)
```

- To create a server socket that would be used by an HTTP server on port 80 and queues up to 50 unaccepted connections at a time:

```
ServerSocket httpd = new ServerSocket(80, 50);
```

# Constructing Server Sockets

- If you try to expand the queue past the operating system's maximum queue length, the maximum queue length is used instead.
- By default, if a host has multiple network interfaces or IP addresses, the server socket listens on the specified port on all the interfaces and IP addresses. However, you can add a third argument to bind only to one particular local IP address. That is, the server socket only listens for incoming connections on the specified address; it won't listen for connections that come in through the host's other addresses.
- For example, *login.ibiblio.org* is a particular Linux box in North Carolina. It's connected to the Internet with the IP address 152.2.210.122. The same box has a second Ethernet card with the local IP address 192.168.210.122 that is not visible from the public Internet, only from the local network. If, for some reason, you wanted to run a server on this host that only responded to local connections from within the same network, you could create a server socket that listens on port 5776 of 192.168.210.122 but not on port 5776 of 152.2.210.122, like so:

```
InetAddress local = InetAddress.getByName("192.168.210.122");
```

```
ServerSocket httpd = new ServerSocket(5776, 10, local);
```

# Constructing Server Sockets

- In all three constructors, you can pass 0 for the port number so the system will select an available port for you. A port chosen by the system like this is sometimes called an *anonymous port* because you don't know its number in advance (though you can find out after the port has been chosen). This is often useful in multiset protocols such as FTP.
- In passive FTP the client first connects to a server on the well-known port 21, so the server has to specify that port. However, when a file needs to be transferred, the server starts listening on any available port. The server then tells the client what other port it should connect to for data using the command connection already open on port 21. Thus, the data port can change from one session to the next and does not need to be known in advance.
- All these constructors throw an `IOException`, specifically, a `BindException`, if the socket cannot be created and bound to the requested port.

# Constructing Without Binding

- The noargs constructor creates a ServerSocket object but does not actually bind it to a port, so it cannot initially accept any connections. It can be bound later using the bind() methods:

```
public void bind(SocketAddress endpoint) throws IOException
```

```
public void bind(SocketAddress endpoint, int queueLength) throws IOException
```

- The primary use for this feature is to allow programs to set server socket options before binding to a port. Some options are fixed after the server socket has been bound. The general pattern looks like thi:

```
ServerSocket ss = new ServerSocket();
```

```
// set socket options...
```

```
SocketAddress http = new InetSocketAddress(80);
```

```
ss.bind(http);
```

- You can also pass null for the SocketAddress to select an arbitrary port. This is like passing 0 for the port number in the other constructors.



The image features a light gray background with a subtle pattern of concentric circles. In the four corners, there are decorative circuit-like lines in a dark blue color. These lines consist of straight segments and small circles, resembling a stylized electronic circuit or network diagram. The central text is in a bold, black, sans-serif font.

# Getting Information about Server Socket

# Getting Information about Server Socket

- The ServerSocket class provides two getter methods that tell you the local address and port occupied by the server socket. These are useful if you've opened a server socket on an anonymous port and/or an unspecified network interface. This would be the case, for one example, in the data connection of an FTP session:

```
public InetAddress getInetAddress()
```

- This method returns the address being used by the server (the local host). If the local host has a single IP address (as most do), this is the address returned by InetAddress.getLocalHost(). If the local host has more than one IP address, the specific address returned is one of the host's IP addresses. You can't predict which address you will get. For example:

```
ServerSocket httpd = new ServerSocket(80);
```

```
InetAddress ia = httpd.getInetAddress();
```

- If the ServerSocket has not yet bound to a network interface, this method returns null:

```
public int getLocalPort()
```

# Getting Information about Server Socket

- The ServerSocket constructors allow you to listen on an unspecified port by passing 0 for the port number. This method lets you find out what port you're listening on.
- You might use this in a peer-to-peer multiset program where you already have a means to inform other peers of your location. Or a server might spawn several smaller servers to perform particular operations. The well-known server could inform clients on what ports they can find the smaller servers.
- If the ServerSocket has not yet bound to a port, getLocalPort() returns -1. As with most Java objects, you can also just print out a ServerSocket using its toString() method. A String returned by a ServerSocket's toString() method looks like this:

```
ServerSocket[addr=0.0.0.0,port=0,localport=5776]
```

- addr is the address of the local network interface to which the server socket is bound. This will be 0.0.0.0 if it's bound to all interfaces, as is commonly the case. port is always 0. The localport is the local port on which the server is listening for connections. This method is sometimes useful for debugging, but not much more.



# Socket Options

# Socket Options

- Socket options specify how the native sockets on which the ServerSocket class relies send and receive data. For server sockets, Java supports three options:
  - `SO_TIMEOUT`
  - `SO_REUSEADDR`
  - `SO_RCVBUF`
- It also allows you to set performance preferences for the socket's packets.

# SO\_TIMEOUT

- SO\_TIMEOUT is the amount of time, in milliseconds, that accept() waits for an incoming connection before throwing a java.io.InterruptedIOException. If SO\_TIMEOUT is 0, accept() will never time out. The default is to never time out.
- We might need to set it if you were implementing a complicated and secure protocol that required multiple connections between the client and the server where responses needed to occur within a fixed amount of time.
- However, most servers are designed to run for indefinite periods of time and therefore just use the default timeout value, 0 (never time out). If you want to change this, the setTimeout() method sets the SO\_TIMEOUT field for this server socket object:

```
public void setTimeout(int timeout) throws SocketException
```

```
public int getTimeout() throws IOException
```

- The countdown starts when accept() is invoked. When the timeout expires, accept() throws a SocketTimeoutException, a subclass of IOException

# SO\_TIMEOUT

- You need to set this option before calling `accept()`; you cannot change the timeout value while `accept()` is waiting for a connection. The timeout argument must be greater than or equal to zero; if it isn't, the method throws an `IllegalArgumentException`.

```
try (ServerSocket server = new ServerSocket(port)) {  
    server.setSoTimeout(30000); // block for no more than 30 seconds  
    try {  
        Socket s = server.accept();  
        // handle the connection  
    } catch (SocketTimeoutException ex) {  
        System.err.println("No connection within 30 seconds");  
    }  
} catch (IOException ex) {  
    System.err.println("Unexpected IOException: " + e);  
}
```

# SO\_REUSEADDR

- The SO\_REUSEADDR option for server sockets is very similar to the same option for client sockets, discussed in the previous chapter. It determines whether a new socket will be allowed to bind to a previously used port while there might still be data traversing the network addressed to the old socket. As you probably expect, there are two methods to get and set this option:

```
public boolean getReuseAddress() throws SocketException
```

```
public void setReuseAddress(boolean on) throws SocketException
```

- The default value is platform dependent. This code fragment determines the default value by creating a new ServerSocket and then calling getReuseAddress():

```
ServerSocket ss = new ServerSocket(10240);  
System.out.println("Reusable: " + ss.getReuseAddress());
```



# SO\_RCVBUF

- The SO\_RCVBUF option sets the default receive buffer size for client sockets accepted by the server socket. It's read and written by these two methods:

```
public int getReceiveBufferSize() throws SocketException
```

```
public void setReceiveBufferSize(int size) throws SocketException
```

- Setting SO\_RCVBUF on a server socket is like calling setReceiveBufferSize() on each individual socket returned by accept() (except that you can't change the receive buffer size after the socket has been accepted). Recall from the previous chapter that this option suggests a value for the size of the individual IP packets in the stream. Faster connections will want to use larger buffers, although most of the time the default value is fine.
- You can set this option before or after the server socket is bound, unless you want to set a receive buffer size larger than 64K. In that case, you must set the option on an unbound ServerSocket before binding it.

# SO\_RCVBUF

```
ServerSocket ss = new ServerSocket();  
  
int receiveBufferSize = ss.getReceiveBufferSize();  
  
if (receiveBufferSize < 131072) {  
    ss.setReceiveBufferSize(131072);  
}  
  
ss.bind(new InetSocketAddress(8000));  
//...
```

# Class of Service

- As you learned in the previous chapter, different types of Internet services have different performance needs. For instance, live streaming video of sports needs relatively high bandwidth. On the other hand, a movie might still need high bandwidth but be able to tolerate more delay and latency. Email can be passed over low-bandwidth connections and even held up for several hours without major harm.
- Four general traffic classes are defined for TCP:
  - Low cost
  - High reliability
  - Maximum throughput
  - Minimum delay
- These traffic classes can be requested for a given Socket. For instance, you can request the minimum delay available at low cost. These measures are all fuzzy and relative, not guarantees of service. Not all routers and native TCP stacks support these classes.

# Class of Service

- The `setPerformancePreferences()` method expresses the relative preferences given to connection time, latency, and bandwidth for sockets accepted on this server:

```
public void setPerformancePreferences(int connectionTime, int latency, int bandwidth)
```

- For instance, by setting `connectionTime` to 2, `latency` to 1, and `bandwidth` to 3, you indicate that maximum bandwidth is the most important characteristic, minimum latency is the least important, and connection time is in the middle:

```
ss.setPerformancePreferences(2, 1, 3);
```

- Exactly how any given VM implements this is implementation dependent. The underlying socket implementation is not required to respect any of these requests. They only provide a hint to the TCP stack about the desired policy. Many implementations including Android ignore these values completely.



# HTTP Servers

# HTTP Servers

- A full-featured HTTP server must respond to requests for files, convert URLs into filenames on the local system, respond to POST and GET requests, handle requests for files that don't exist, interpret MIME types, and much, much more. However, many HTTP servers don't need all of these features.
- For example, many sites simply display an “under construction” message. Clearly, Apache is overkill for a site like this. Such a site is a candidate for a custom server that does only one thing. Java's network class library makes writing simple servers like this almost trivial.
- Custom servers aren't useful only for small sites. High-traffic sites like Yahoo! are also candidates for custom servers because a server that does only one thing can often be much faster than a general-purpose server such as Apache or Microsoft IIS. It is easy to optimize a special-purpose server for a particular task; the result is often much more efficient than a general-purpose server that needs to respond to
- For instance, icons and images that are used repeatedly across many pages or on high-traffic pages might be better handled by a server that read all the image files into memory on startup and then served them straight out of RAM, rather than having to read them off disk for each request. many different kinds of requests.

# HTTP Servers

- Furthermore, this server could avoid wasting time on logging if you didn't want to track the image requests separately from the requests for the pages in which they were included. Finally, Java isn't a bad language for full-featured web servers meant to compete with the likes of Apache or IIS. Even if you believe CPU-intensive Java programs are slower than CPU-intensive C and C++ programs (something I very much doubt is true in modern VMs), most HTTP servers are limited by network bandwidth and latency, not by CPU speed.

# A Single-File Server

- Our investigation of HTTP servers begins with a server that always sends out the same file, no matter what the request. The filename, local port, and content encoding are read from the command line. If the port is omitted, port 80 is assumed. If the encoding is omitted, ASCII is assumed.
- The `SingleFileHTTPServer` class holds the content to send, the header to send, and the port to bind to. The `start()` method creates a `ServerSocket` on the specified port, then enters an infinite loop that continually accepts connections and processes them.
- Each incoming socket is processed by a runnable `Handler` object that is submitted to a thread pool. Thus, one slow client can't starve other clients. Each `Handler` gets an `InputStream` from it which it reads the client request. It looks at the first line to see whether it contains the string `HTTP`. If it sees this string, the server assumes that the client understands `HTTP/1.0` or later and therefore sends a MIME header for the file; then it sends the data. If the client request doesn't contain the string `HTTP`, the server omits the header, sending the data by itself. Finally, the handler closes the connection.



# A Redirector

- *Redirection* is another simple but useful application for a special-purpose HTTP server. In this section, you develop a server that redirects users from one website to another for example, from *cnet.com* to [www.cnet.com](http://www.cnet.com). We can read a URL and a port number from the command line, opens a server socket on the port, and redirects all requests that it receives to the site indicated by the new URL using a 302 FOUND code. In this example, I chose to use a new thread rather than a thread pool for each connection. This is perhaps a little simpler to code and understand but somewhat less efficient.
- It is possible that the first word will be POST or PUT instead or that there will be no HTTP version. The second “word” is the file the client wants to retrieve. This *must* begin with a slash (/).
- Browsers are responsible for converting relative URLs to absolute URLs that begin with a slash; the server does not do this. The third word is the version of the HTTP protocol the browser understands. Possible values are nothing at all (pre-HTTP/1.0 browsers), HTTP/1.0, or HTTP/1.1.

# A Redirector

- To handle a request like this, Redirector ignores the first word. The second word is attached to the URL of the target server (stored in the field `newSite`) to give a full redirected URL. The third word is used to determine whether to send a MIME header; MIME headers are not used for old browsers that do not understand HTTP/1.0. If there is a version, a MIME header is sent; otherwise, it is omitted.
- Sending the data is almost trivial. The `Writer` out is used. Because all the data you send is pure ASCII, the exact encoding isn't too important. The only trick here is that the end-of-line character for HTTP requests is `\r\n`—a carriage return followed by a line feed.

# A Full-Fledged HTTP Server

- Enough special-purpose HTTP servers. This next section develops a full-blown HTTP server, called JHTTP, that can serve an entire document tree, including images, applets, HTML files, text files, and more. It will be very similar to the SingleFileHTTPServer, except that it pays attention to the GET requests. This server is still fairly lightweight; after looking at the code, we'll discuss other features you might want to add.
- Because this server may have to read and serve large files from the filesystem over potentially slow network connections, you'll change its approach. Rather than processing each request as it arrives in the main thread of execution, you'll place incoming connections in a pool. Separate instances of a RequestProcessor class will remove the connections from the pool and process them.

The slide features a light blue background with a subtle pattern of concentric circles. In the four corners, there are decorative circuit-like line art elements. The top-left and bottom-left corners have dark blue lines, while the top-right and bottom-right corners have light blue lines. These lines form various geometric shapes, some ending in small circles, resembling a network or circuit diagram.

THANK YOU

YOU CAN FIND THIS SLIDES/NOTES IN

<https://chandanbhagat.com.np/docs/network-programming/>

AND CODES IN

<https://github.com/chandan-g-bhagat/network-programming>