

UNIT 7

Socket for Servers

(5 Hours) — Network Programming (CAC355)

Disclaimer: These notes were generated automatically from the CACS355 syllabus and may contain errors or omissions. Please verify important details (syntax, method names, default values) against your textbook (Java Network Programming, E. R. Harold) and class notes before the exam.

7.1 Using ServerSockets

`java.net.ServerSocket` lets a server **listen** on a port and accept incoming TCP connections. The basic life cycle is:

- **Create** a `ServerSocket` bound to a port: `new ServerSocket(port)`.
- **accept()** — blocks until a client connects and returns a **Socket** for that client.
- Use the returned socket's **input/output streams** to talk to the client.
- **Close** the client socket (and repeat `accept()` for the next client).
- Close the `ServerSocket` when the server shuts down.

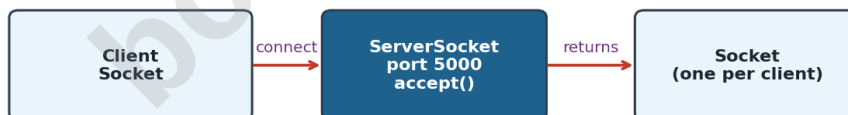


Figure 7.1 — `accept()` returns a new `Socket` for each connected client

```
try (ServerSocket server = new ServerSocket(13)) { // daytime port
    while (true) {
        try (Socket c = server.accept()) {
            Writer out = new OutputStreamWriter(c.getOutputStream(), "US-ASCII");
            out.write(new Date().toString() + "\r\n");
            out.flush();
        } catch (IOException e) { /* this client failed; keep serving */ }
    }
}
```

Listing 7.1 — Iterative daytime server

*Note: Ports below 1024 usually require administrator/root rights on Unix-like systems. If the port is in use, the constructor throws **BindException**.*

Serving binary data

Servers are not limited to text. For binary data (images, files) use the raw *OutputStream* (or *DataOutputStream*) and write bytes; e.g., a server that sends the current time as a 4-byte integer (the **time** protocol, port 37). Write the whole message, then *flush()*. Text protocols wrap the stream in a *Writer*, but binary ones must not, because a *Writer* changes the bytes through character encoding.

Multithreaded servers

An iterative server serves one client at a time; a slow client blocks everyone else. A **multithreaded server** hands each accepted socket to a separate thread, so *accept()* is called again immediately.

```
ExecutorService pool = Executors.newFixedThreadPool(50);
try (ServerSocket server = new ServerSocket(5000)) {
    while (true) {
        Socket c = server.accept();
        pool.submit(() -> handle(c)); // each client in its own task
    }
}
// ...
static void handle(Socket c) {
    try (c; BufferedReader in = new BufferedReader(new InputStreamReader(c.getInputStream()));
        PrintWriter out = new PrintWriter(c.getOutputStream(), true)) {
        String line;
        while ((line = in.readLine()) != null) out.println("Echo: " + line);
    } catch (IOException e) { e.printStackTrace(); }
}
```

Listing 7.2 — Multithreaded echo server using a thread pool

Approach	Pros	Cons
One thread per connection (new Thread)	Simple	Thread-creation cost; can exhaust resources under heavy load
Thread pool (ExecutorService)	Limits threads, reuses them	Clients wait when the pool is full
Nonblocking I/O with Selector (Unit 9)	Many clients with few threads	More complex code

Writing to servers with sockets

Servers write to a client in the same way as clients do: get the socket's `OutputStream`, wrap it if needed, write and flush. Both sides must agree on the message format (line-based text, length-prefixed binary, etc.) and on who speaks first.

Closing server sockets

Call `close()` on the `ServerSocket` when the server is finished. This frees the port and makes any thread blocked in `accept()` throw a `SocketException` — a convenient way to stop the server from another thread. `isClosed()` tells whether it has been closed. Closing a `ServerSocket` does **not** close the client sockets already accepted; close those separately.

7.2 Logging

What to log

- **Requests:** time, client IP address and port, requested resource or command.
- **Responses:** status and size sent.
- **Errors and exceptions** with stack traces, plus security events (failed logins).
- Server start/stop and configuration; avoid logging passwords or private data.

How to log

Use the `java.util.logging` package (or Log4j/SLF4J). Typical use:

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

audit.info("Client " + c.getRemoteSocketAddress());           // normal event
errors.log(Level.SEVERE, "Accept failed", exception);         // error with exception
```

- **Levels:** SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST.
- **Handlers** decide where records go (`ConsoleHandler`, `FileHandler`, `SocketHandler`); **Formatters** (`SimpleFormatter`, `XMLFormatter`) decide how they look.
- Use separate loggers for requests and errors, and rotate log files to control size.
- Loggers are thread-safe, which matters in multithreaded servers.

7.3 Constructing Server Sockets

Constructor	Description
<code>ServerSocket(int port)</code>	Listen on the port; port 0 picks any free port (find it with <code>getLocalPort()</code>)
<code>ServerSocket(int port, int backlog)</code>	backlog = maximum length of the queue of pending connections (default 50)
<code>ServerSocket(int port, int backlog, InetAddress bindAddr)</code>	Listen only on one local interface/address
<code>ServerSocket()</code>	Create unbound (see below)

If the queue is full, further connection attempts are refused. Trying to use a port already in use throws **BindException**.

Constructing without binding

new ServerSocket() creates an unbound socket, so options such as `SO_REUSEADDR` or receive buffer size can be set *before* binding. Then call **bind(SocketAddress)** or **bind(SocketAddress, backlog)**.

```
ServerSocket ss = new ServerSocket();
ss.setReuseAddress(true);
ss.bind(new InetSocketAddress(5000));
```

7.4 Getting Information about a Server Socket

Method	Returns
<code>getInetAddress()</code>	The local address the server socket is bound to (0.0.0.0 = all interfaces)
<code>getLocalPort()</code>	The port being listened on (useful when 0 was given)
<code>getLocalSocketAddress()</code>	Address + port object
<code>isBound()</code> / <code>isClosed()</code>	Whether bound to a port / closed
<code>toString()</code>	E.g. <code>ServerSocket[addr=0.0.0.0/0.0.0.0,localport=5000]</code>

7.5 Socket Options

Option	Methods	Effect on a ServerSocket
<code>SO_TIMEOUT</code>	<code>setSoTimeout(int ms)</code>	<code>accept()</code> throws <code>SocketTimeoutException</code> after this time; the <code>ServerSocket</code> is still usable. 0 = wait forever

Option	Methods	Effect on a ServerSocket
SO_REUSEADDR	setReuseAddress(boolean)	Lets the server rebind quickly to a port whose old connections are in TIME_WAIT; set before bind()
SO_RCVBUF	setReceiveBufferSize(int)	Default receive buffer size for accepted sockets; set before bind()
Class of service	setPerformancePreferences(connectionTime, latency, bandwidth)	Relative importance of the three; the JVM may ignore it (traffic class itself is set on the accepted Socket via setTrafficClass)

7.6 HTTP Servers

An HTTP server reads a request (line + headers), then writes a status line, headers, a blank line and the body. Three progressively more capable examples are usual:

A single file server

Always serves the same file, whatever the request. It reads the file once, then for each client reads (and ignores) the request headers up to the blank line, and sends *HTTP/1.0 200 OK*, Content-Type, Content-Length and the data.

```
out.write("HTTP/1.0 200 OK\r\n");
out.write("Content-Type: text/html; charset=UTF-8\r\n");
out.write("Content-Length: " + data.length + "\r\n\r\n");
out.flush();
raw.write(data);           // raw = the socket's OutputStream
raw.flush();
```

A redirector

Sends every client to another URL. It reads the first request line and replies with a 3xx status and a **Location** header.

```
HTTP/1.0 302 Found
Location: http://www.newsite.com/
Content-Type: text/html

<html><body>Document moved <a href="http://www.newsite.com/">here</a></body></html>
```

A full-fledged HTTP server

Parses the request line (*method, path, version*), maps the path to a file under a **document root**, chooses the MIME type from the extension (`guessContentTypeFromName`), and sends the file. Additional requirements:

- Return **404 Not Found** for missing files and **501 Not Implemented** for unsupported methods.
- Prevent **directory traversal**: reject paths containing “..” that would leave the document root; default to `index.html` for “/”.
- Serve requests in a thread pool, and log each request.
- Send correct **Content-Type** and **Content-Length** headers.

Example — Typical responses

200 OK — file found and sent

301/302 — resource moved (Location header)

404 Not Found — file missing

501 Not Implemented — method not supported

Key Takeaways

- ✓ `ServerSocket` listens; `accept()` returns a `Socket` per client.
- ✓ Iterative servers handle one client at a time; multithreaded servers use a thread (pool) per client.
- ✓ Log requests and errors with `java.util.logging` (levels, handlers, formatters).
- ✓ Constructors: `(port)`, `(port, backlog)`, `(port, backlog, bindAddr)`, or `unbound + bind()`.
- ✓ Options: `SO_TIMEOUT` (accept timeout), `SO_REUSEADDR`, `SO_RCVBUF`, performance preferences.
- ✓ HTTP server examples: single-file server, redirector, and a full server with MIME types, error codes and directory-traversal protection.

— End of Unit 7 —