Advanced Programming
CSE 201
Instructor: Sambuddho

(Semester: Monsoon 2025)
Week 12 - Networking and Network Programming - II

Thread-Per-Client Server: Concept

- Architecture

[ServerSocket] --accept()--> Socket for client 1 -- handled by Thread T1

Socket for client 2 -- handled by Thread T2

Socket for client 3 -- handled by Thread T3

- Flow

- Main thread:
  - Loops on accept().
  - For each accepted Socket, starts a new thread to handle that client.
- Each client:
  - Has its own handler thread executing the same handleClient logic.

Advantage

- Multiple clients can be served in parallel.
- A slow or chatty client does not block other clients from being served.

Using Thread for Parallel Connection Handling

```
while (true) {
    Socket incoming = server.accept();
    Thread t = new Thread(() → {
        try {
            handleClient(incoming); // same handleClient as before
        } catch (IOException e) {
            e.printStackTrace();
        }
    });
    t.start();
}
```

Drawbacks

- Platform threads are heavier in resource usage.
- Handling thousands of simultaneous clients becomes harder.
- Still fine for moderate loads and educational examples.

Interruptible vs Non-Interruptible Blocking I/O

Problem with classic blocking I/O

- A thread blocked in read() or accept() on a classic Socket may not be easily interrupted with Thread.interrupt() on some platforms.
- This can make it hard to cancel long-running or stuck network operations using platform threads.

Solutions

- Use virtual threads (recommended in modern Java):
  - Blocking I/O is cooperatively cancellable.
- Or use SocketChannel (NIO) which responds to interrupts and closure.

Guidance

- For this course: prefer Socket + virtual threads for simplicity.
- Introduce SocketChannel as a more advanced tool for high-performance or event-driven servers.

Using Virtual Threads for Client Handling

Modern Java (19+/21+) supports virtual threads which are:

- Very lightweight.
- Perfect for thread-per-client servers.

```
ExecutorService service =
    Executors.newVirtualThreadPerTaskExecutor();


try (ServerSocket server = new ServerSocket(8189)) {
    System.out.println("Multi-client echo server on port 8189");


    while (true) {
        Socket incoming = server.accept(); // wait for client
        service.submit(() -> {new Runnable() {

            public void run() {
            try {
            handleClient(incoming);
              }
             catch (IOException e) {
            e.printStackTrace();
             }
          }
       });
    }
}
```

Benefits

- Simplified logic (no fancy NIO needed).
- Handles many clients without exhausting OS threads.

Interruptible I/O with SocketChannel

Motivation

- Threads blocked on classic Socket I/O cannot always be cleanly interrupted using Thread.interrupt() when using platform threads.
- SocketChannel (NIO) allows blocking operations that can respond to interrupts or channel close.

Basic usage:

```
SocketChannel channel =
    SocketChannel.open(new InetSocketAddress(host, port));
Scanner in = new Scanner(channel);
while (!Thread.currentThread().isInterrupted() && in.hasNextLine()) {
    System.out.println(in.nextLine());
}
```

Key ideas

- Closing the channel or interrupting the thread can cause blocked reads to return or throw.
- Useful when you must support cancellation with platform threads or integrate with NIO selectors.

When to Use Socket vs SocketChannel

Use Socket **when:**

- Protocol is simple and blocking I/O is acceptable.
- You are using virtual threads, which make blocking I/O scalable and cancellable.
- You want the simplest code and do not need non-blocking features.

Use SocketChannel **when:**

- You need non-blocking I/O and event-driven architectures using selectors.
- You must interrupt or cancel blocking I/O on platform threads reliably.
- You are building high-performance or multiplexed network servers manually.

Why Security Matters: TLS/SSL Overview

Network threats

- Eavesdropping on plain-text connections.
- Tampering with data in transit.
- Impersonating servers (man-in-the-middle attacks).

TLS/SSL provides

- Encryption: attackers cannot read the cleartext data.
- Integrity: tampering is detected via MACs or signatures.
- Authentication: certificates bind public keys to identities (e.g., domain names).

Java SSL stack

- `SSLSocketFactory` / `SSLServerSocketFactory` for secure sockets.
- Under the hood, Java uses JSSE (Java Secure Socket Extension).

Key message

- HTTPS = HTTP over TLS (SSL) on top of sockets; your own protocols can use TLS the same way.

Creating an SSL Client

Pattern for a secure client:

```
SocketFactory factory = SSLSocketFactory.getDefault();
try (Socket s = factory.createSocket(host, port);
     Scanner in = new Scanner(s.getInputStream());
     PrintWriter out = new PrintWriter(s.getOutputStream(),
true)) {


    out.println("Hello securely!");
    while (in.hasNextLine()) {
        System.out.println(in.nextLine());
    }
}
```

Noteworthy:

- Same basic pattern as with a plain Socket.
- The SSL/TLS handshake, encryption, and integrity checks are handled under the hood.
- For self-signed certificates or custom CAs, you must configure keystores and truststores explicitly.

Creating an SSL Server

Secure server pattern

```
ServerSocketFactory factory =
    SSLServerSocketFactory.getDefault();


try (ServerSocket server = factory.createServerSocket(8443)) {
    while (true) {
        Socket incoming = server.accept();
        handleClient(incoming); // same echo or protocol
logic, now encrypted
    }
}
```

Key points

- The server presents a certificate to the client.
- The client must trust that certificate (directly or via a CA).
- Your existing handleClient logic usually does not change; all encryption is transparent to your I/O code.

Keystores and Certificates (High-Level)

Concepts

- A keystore file (e.g., JKS) holds keys and certificates.
- A certificate binds a public key to an identity (like a domain name).

Typical steps with keytool

- Generate a key pair for the server and self-signed certificate.
- Store them in a server keystore.
- Export the certificate and import it into a client truststore so the client will trust the server.

System properties

- Server might use: -Djavax.net.ssl.keyStore=server.jks
- Client might use: -Djavax.net.ssl.trustStore=client.jks

Takeaway

- HTTPS and other TLS-secured protocols all fundamentally rely on this certificate and trust mechanism.

Shift to Web Data: URI vs URL

Definitions

URI (Uniform Resource Identifier)

- Pure syntax identifying a resource.
- Java class: java.net.URI.
- Useful for parsing, normalizing, and resolving relative references.

URL (Uniform Resource Locator)

- A URI with an attached “open connection” behavior.
- Java class: java.net.URL.
- Legacy constructors are discouraged in modern Java.

Preferred pattern

```
URI uri = new URI("https://example.com/resource?x=1");
URL url = uri.toURL();
```

Reason

- URI handles encoding and parsing more robustly.
- URL focuses on I/O operations (openConnection, openStream).

Simple URL Fetch with openStream()

Minimal HTTP fetch

```
URI uri = new URI("https://example.com/");
URL url = uri.toURL();
try (InputStream in = url.openStream();
     Scanner scanner = new Scanner(in)) {
    while (scanner.hasNextLine()) {
        System.out.println(scanner.nextLine());
    }
}
```

Pros

- Extremely simple to write.
- Good for quick tests or small utilities.

Cons

- No control over HTTP headers.
- Hard to send POST requests.
- Hard to handle errors, redirects, or authentication cleanly.

Motivation

- Use URLConnection or HttpClient for non-trivial applications.

URLConnection: The 5-Step Life Cycle

Typical usage pattern

1. Create a URL

   ```
   URL url = new URI(urlString).toURL();
   ```

2. Open a connection (but not yet connected)

   ```
   URLConnection connection = url.openConnection();
   ```

3. Configure the request

   - Set headers, timeouts, and doOutput for POST, etc.

4. Connect

   ```
   connection.connect();
   // often implicit when reading/writing first time
   ```

5. Read the response

   - Headers via `getHeaderFields()` and convenience methods.
   - Body via `getInputStream()` (or `getErrorStream()` on error).

Core idea

- URLConnection exposes lower-level HTTP details, but its API is older compared to HttpClient.

Setting Request Properties (Headers)

Examples of configuring a request

```
connection.setConnectTimeout(5000);
connection.setReadTimeout(10000);

connection.setRequestProperty("User-Agent", "MyJavaClient/1.0");
connection.setRequestProperty("Accept", "text/html; q=0.9, */*; q=0.8");

Setting Content-Type for POST form data
connection.setRequestProperty(
    "Content-Type",
    "application/x-www-form-urlencoded; charset=utf-8"
);

Use cases
- Mimic a browser's headers.
- Specify accepted formats (JSON, XML, HTML).
- Send authentication tokens or API keys in Authorization headers.
```

Reading Response Headers

Inspecting headers

```
Map<String, List<String>> headers =
connection.getHeaderFields();
for (Map.Entry<String, List<String>> e : headers.entrySet()) {
    String name = e.getKey();
    for (String value : e.getValue()) {
        System.out.println((name != null ? name + ": " : "") +
value);
    }
}
```

Convenience methods

- String type = connection.getContentType();
- int length = connection.getContentLength();
- long lastMod = connection.getLastModified();

GET vs POST: Conceptual Difference

GET

- Parameters are encoded in the URL:
  https://example.com/search?q=java&lang=en
- Often cached by browsers.
- Visible in address bar and logs.
- Typically used for idempotent, read-only operations.

POST

- Parameters are sent in the request body.
- Can send much more data (forms, JSON, files).
- Used for operations that modify server state (login, purchase, upload).

Summary

- GET: read/query operations with parameters in the URL.
- POST: writes/updates, form submissions, large or sensitive payloads in the body.

POST with URLConnection

Steps to send form data via POST

1. Allow output on the connection

```
connection.setDoOutput(true);
connection.setRequestProperty(
    "Content-Type",
    "application/x-www-form-urlencoded; charset=utf-8"
);
```

2. Write URL-encoded form data

```
try (PrintWriter out = new PrintWriter(connection.getOutputStream())) {
    String name = URLEncoder.encode("Alice", StandardCharsets.UTF_8);
    String msg  = URLEncoder.encode("Hello World", StandardCharsets.UTF_8);
    out.print("name=" + name + "&message=" + msg);
    // closing the writer will flush the data
}
```

3. Read the response from connection.getInputStream() as usual.

Encoding rules

- Letters, digits, and safe punctuation remain unchanged.
- Spaces become +.
- Other characters are encoded as %XY (hex).

Handling Errors and Redirects

HTTP errors

- If the server returns an error code (4xx or 5xx),

  connection.getInputStream() may throw FileNotFoundException.

- You can access the error body using:

  InputStream err = ((HttpURLConnection) connection).getErrorStream();

Redirects

- Status codes 301, 302, 303 indicate redirects.
- For manual handling:
  - Read the Location header.
  - Resolve it against the original URL.
  - Open a new connection to the redirected URL.

Auto-redirect

- HttpURLConnection can follow redirects automatically, but HttpClient gives more explicit and modern control over redirect policies.

Why HttpClient Was Introduced

Limitations of HttpURLConnection

- Verbose and somewhat awkward API.
- Complicated error handling, redirects, and configuration.
- Limited direct support for asynchronous operations.
- HTTP/2 support bolted on later.

HttpClient advantages (java.net.http)

- Modern builder API (immutable, fluent).
- First-class support for HTTP/1.1 and HTTP/2.
- Easy synchronous and asynchronous requests.
- Better control over redirects, proxies, and SSL settings.
- Configurable body handling (strings, files, byte arrays, reactive streams).

Message

- For new HTTP code in Java, prefer HttpClient over HttpURLConnection.

Basic HttpClient GET Example

Creating a client and sending a GET request

```
HttpClient client = HttpClient.newBuilder()
    .followRedirects(HttpClient.Redirect.NORMAL)
    .build();

HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://httpbin.org/get?x=1"))
    .GET()
    .build();

HttpResponse<String> response =
    client.send(request, HttpResponse.BodyHandlers.ofString());

System.out.println("Status: " + response.statusCode());
System.out.println("Body:\n" + response.body());
```

Discussion

- `client.send(...)` is synchronous; it blocks until the response is available.
- `BodyHandlers.ofString()` collects the entire response body as a String.

HttpClient POST Example (JSON)

Sending JSON with POST

```
String json = """
    { "name": "Alice", "age": 23 }
    """;

HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("https://httpbin.org/post"))
    .header("Content-Type", "application/json")
    .POST(HttpRequest.BodyPublishers.ofString(json))
    .build();

HttpResponse<String> response =
    client.send(request, HttpResponse.BodyHandlers.ofString());

System.out.println("Status: " + response.statusCode());
System.out.println("Response JSON:\n" + response.body());
```

Why this matters

- This is how Java applications talk to RESTful JSON-based APIs.
- Very easy to integrate with your own backends or third-party web services.

Accessing Response Headers with HttpClient

Working with headers

```
HttpHeaders headers = response.headers();

// Print all headers
headers.map().forEach((name, values) -> {
    System.out.println(name + ": " + String.join(", ", values));
});



// Get a single header value
headers.firstValue("Content-Type").ifPresent(
    ct -> System.out.println("Content-Type = " + ct)
);
```

Connection to earlier material

- This is HttpClient’s equivalent of URLConnection.getHeaderFields().
- The map() view and Optional-based access are more convenient and modern.