Network and Multithreaded Programming

Network Sockets, Streams and Threads in Java

Zbigniew Dendzik

March 2026

Lecture Plan

- Introduction to network programming
- Network sockets
- Data streams in network communication
- Internet chat application
- Multithreaded programming
- HTTP server
- HTTP protocol and request handling

Network Programming in Java

Java Platform API provides rich support for network applications. **Key packages:**

- `java.net` – network sockets, IP addresses, URLs
- `java.io` – data streams

Stream types:

- `InputStream, OutputStream` – binary streams
- `Reader, Writer` – text streams
- `BufferedReader, PrintWriter` – buffered streams

Network Sockets

Socket – abstraction of network connection endpoint. **Two socket types:**

- **Socket** – client socket
- **ServerSocket** – server socket

Connection scheme:

1. Server creates ServerSocket and listens
2. Client creates Socket and connects to server
3. Server accepts connection (accept())
4. Communication through streams
5. Connection closed

Port Numbers

Port – number identifying network service. **Port number range:** 0 – 65535

- **0–1023:** Well-known ports
  - Port 80: HTTP
  - Port 443: HTTPS
  - Port 22: SSH
  - Port 25: SMTP (mail)
- **1024–49151:** Registered ports
- **49152–65535:** Dynamic/private ports

Simple Server – Implementation

```
import java.io.*;
import java.net.*;

public class Server
{
    public static final int PORT = 50007;

    public static void main(String args[]) throws IOException
    {
        // Creating server socket
        ServerSocket serv = new ServerSocket(PORT);

        // Waiting for connection
        System.out.println("Listening: " + serv);
        Socket sock = serv.accept();
        System.out.println("Connection established: " + sock);

        // Creating input stream
```

Simple Server – cont.

```
        // Receiving data
        String str = inp.readLine();
        System.out.println("<Received:> " + str);

        // Closing connection
        inp.close();
        sock.close();
        serv.close();
    }
}
```

Simple Client – Implementation

```
import java.io.*;
import java.net.*;

public class Client
{
    public static final int PORT = 50007;
    public static final String HOST = "127.0.0.1";

    public static void main(String[] args) throws IOException
    {
        // Establishing connection with server
        Socket sock = new Socket(HOST, PORT);
        System.out.println("Connected: " + sock);

        // Creating streams
        BufferedReader keyboard = new BufferedReader(
            new InputStreamReader(System.in));
        PrintWriter outp = new PrintWriter(
```

Simple Client – cont.

```
        // Sending data
        System.out.print("<Sending:> ");
        String str = keyboard.readLine();
        outp.println(str);
        outp.flush();

        // Closing connection
        keyboard.close();
        outp.close();
        sock.close();
    }
}
```

Simplex Chat

Simplex mode – alternating communication (question-answer). **Implementation**

stages:

1. Connection establishment
2. Stream creation (keyboard, socket)
3. Communication loop
   - Receive message
   - Send response
4. Termination condition (“end”)
5. Connection closure

Simplex Communication Loop

```
while(true)
{
    // Receive message
    String received = inp.readLine();
    System.out.println("<Received:> " + received);

    if(received.equalsIgnoreCase("end"))
    {
        System.out.println("Connection closed");
        break;
    }

    // Send response
    System.out.print("<Sending:> ");
    String sent = keyboard.readLine();
    outp.println(sent);
    outp.flush();
```

Network Exception Handling

Common exceptions:

- `ConnectException` – no server at given address
- `SocketException` – TCP protocol error
- `IOException` – general I/O error
- `BindException` – port already in use

Best practices:

- Catch specific exceptions
- Inform user about problem
- Close resources in finally block

Exception Handling Example

```
try
{
    Socket sock = new Socket(HOST, PORT);
    // communication...
}
catch(ConnectException e)
{
    System.out.println("Connection interrupted");
    System.out.println("Server not responding");
}
catch(SocketException e)
{
    System.out.println("TCP protocol error");
    System.out.println("Check network connection");
}
catch(IOException e)
{
    System.out.println("I/O error: " + e.getMessage());
```

Multithreaded Programming

Thread – independent flow of program execution. **Applications:**

- Concurrent execution of multiple tasks
- Duplex communication (send and receive simultaneously)
- Servers handling multiple clients
- User interfaces (GUI)

Creating thread:

- Extend `Thread` class
- Implement `run()` method
- Call `start()` method

Creating Thread

```
class Receiver extends Thread
{
    Socket sock;
    BufferedReader sockReader;

    public Receiver(Socket sock) throws IOException
    {
        this.sock = sock;
        this.sockReader = new BufferedReader(
            new InputStreamReader(sock.getInputStream()));
    }

    public void run()
    {
        // Code executed in thread
        try
        {
            String str;
```

Starting Thread

```
public static void main(String[] args) throws IOException
{
    Socket sock = new Socket(HOST, PORT);

    // Create and start receiving thread
    new Receiver(sock).start();

    // Main thread - sending messages
    BufferedReader keyboard = new BufferedReader(
        new InputStreamReader(System.in));
    PrintWriter outp = new PrintWriter(
        sock.getOutputStream());

    String str;
    while((str = keyboard.readLine()) != null)
    {
        outp.println(str);
        outp.flush();
```

Duplex Chat

Duplex mode – parallel communication in both directions. **Architecture:**

- **Main thread** – sending messages
- **Receiving thread** – receiving messages

Advantages:

- No need to wait for turn
- More natural communication
- Better responsiveness

Multithreaded Server

Multithreaded server – handles multiple clients simultaneously. **Operation scheme:**

1. Server listens on port
2. For each connection:
   - Accept connection
   - Create new handler thread
   - Pass socket to thread
   - Thread handles client
3. Return to listening

Benefits:

- Multiple clients simultaneously
- Each client has dedicated thread
- Better resource utilization

Multithreaded Server – Implementation

```
class TaskHandler extends Thread
{
    Socket sock;

    TaskHandler(Socket clientSocket)
    {
        this.sock = clientSocket;
    }

    public void run()
    {
        // Client handling
        try
        {
            BufferedReader inp = new BufferedReader(
                new InputStreamReader(sock.getInputStream()));
            PrintWriter outp = new PrintWriter(
                sock.getOutputStream());
```

Multithreaded Server – Main Loop

```
public class Server
{
    public static void main(String[] args) throws IOException
    {
        ServerSocket serv = new ServerSocket(50007);

        while(true)
        {
            System.out.println("Waiting for connection...");
            Socket sock = serv.accept();

            // Create handler thread
            new TaskHandler(sock).start();
        }
    }
}
```

HTTP Protocol

HTTP (HyperText Transfer Protocol) – WWW communication protocol.
Operation scheme:

1. Client (browser) sends request
2. Server processes request
3. Server sends response
4. Connection closed

Request structure:

- Method (GET, POST, PUT, DELETE)
- Resource path
- Protocol version
- Headers

HTTP Request Example

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
Accept: text/html
Accept-Language: en
Connection: keep-alive
```

Response structure:

- Status code (200 OK, 404 Not Found, 500 Error)
- Headers (Content-Type, Content-Length)
- Response body (HTML, JSON, binary data)

Simple HTTP Server

```
import java.io.*;
import java.net.*;

public class HTTPServer
{
    public static void main(String[] args) throws IOException
    {
        ServerSocket serv = new ServerSocket(80);

        while(true)
        {
            Socket sock = serv.accept();

            BufferedReader inp = new BufferedReader(
                new InputStreamReader(sock.getInputStream()));
            DataOutputStream outp = new DataOutputStream(
                sock.getOutputStream());
```

Simple HTTP Server – cont.

```
            if(request.startsWith("GET"))
            {
                // Response header
                outp.writeBytes("HTTP/1.0 200 OK\r\n");
                outp.writeBytes("Content-Type: text/html\r\n");
                outp.writeBytes("\r\n");

                // Response body
                outp.writeBytes("<html>\r\n");
                outp.writeBytes("<H1>Test Page</H1>\r\n");
                outp.writeBytes("</html>\r\n");
            }
            else
            {
                outp.writeBytes("HTTP/1.1 501 Not supported.\r\
                    n");
            }
```

HTTP Status Codes

Important status codes:

- **200 OK** – request successful
- **404 Not Found** – resource does not exist
- **500 Internal Server Error** – server error
- **501 Not Implemented** – method not supported
- **403 Forbidden** – access denied

Code groups:

- **1xx** – informational
- **2xx** – success
- **3xx** – redirection
- **4xx** – client error
- **5xx** – server error

File Handling in HTTP Server

```
// Extract file name from request
String[] parts = request.split(" ");
String fileName = parts[1].substring(1);

try
{
    FileInputStream fis = new FileInputStream(fileName);

    // Headers
    outp.writeBytes("HTTP/1.0 200 OK\r\n");
    outp.writeBytes("Content-Type: text/html\r\n");
    outp.writeBytes("Content-Length: " + fis.available() + "\r\
       n");
    outp.writeBytes("\r\n");

    // Transfer file
    byte[] buffer = new byte[1024];
    int n;
```

Handling 404 Error

```
catch(FileNotFoundException e)
{
    outp.writeBytes("HTTP/1.0 404 Not Found\r\n");
    outp.writeBytes("Content-Type: text/html\r\n");
    outp.writeBytes("\r\n");
    outp.writeBytes("<html>\r\n");
    outp.writeBytes("<H1>404 - Not Found</H1>\r\n");
    outp.writeBytes("<p>File " + fileName +
                    " does not exist.</p>\r\n");
    outp.writeBytes("</html>\r\n");
}
```

Content Type

Content-Type header specifies data format. **Popular MIME types:**

- `text/html` – HTML documents
- `text/plain` – plain text
- `image/jpeg` – JPEG images
- `image/png` – PNG images
- `application/json` – JSON data
- `application/pdf` – PDF documents

Determining type:

- Based on file extension
- .html, .htm → text/html
- .jpg, .jpeg → image/jpeg
- .png → image/png

Multithreaded HTTP Server

Why multithreaded?

- Typical page requires multiple requests
- HTML, CSS, JavaScript, images, fonts
- Browser sends multiple parallel requests
- Single-threaded server handles sequentially
- Multithreaded server handles in parallel

Example:

- Page with 10 images
- 11 requests: 1 HTML + 10 images
- Single-threaded server: 11 times sequentially
- Multithreaded server: 11 threads simultaneously

Server Activity Logging

Log file – server activity registry. **Typical log information:**

- Request date and time
- Client IP address
- Request (method, path)
- Response code
- Data size sent

Used classes:

- `java.util.Date` – current date and time
- `socket.getInetAddress()` – client IP address
- `PrintWriter` – file writing

Logging Example

```
PrintWriter log = new PrintWriter(
    new FileWriter("server.log", true));

Date now = new Date();
String clientIP = sock.getInetAddress().toString();

log.println("[" + now + "] " + clientIP + " " + request);
log.flush();
log.close();
```

Sample log entry:

```
[Fri Mar 13 01:45:23 CET 2026] /192.168.1.100 GET /index.html
   HTTP/1.1
```

HTTPS and SSL/TLS

HTTPS – HTTP with SSL/TLS encryption. **SSL (Secure Socket Layer) / TLS (Transport Layer Security):**

- Communication encryption
- Server authentication
- Protection against eavesdropping

Java Platform API:

- Package `javax.net.ssl`
- `SSLServerSocketFactory`
- `SSLSocket`
- Automatic encryption/decryption

HTTPS Server Skeleton

```
import java.io.*;
import java.net.*;
import javax.net.ssl.*;

public class HTTPSServer
{
    public static void main(String[] args) throws IOException
    {
        SSLServerSocketFactory fact =
            (SSLServerSocketFactory)SSLServerSocketFactory.
                getDefault();
        ServerSocket servsock = fact.createServerSocket(8080);

        while(true)
        {
            Socket sock = servsock.accept();

            // Communication as in HTTP
```

Summary

- **Network sockets** – Socket, ServerSocket
- **Streams** – BufferedReader, PrintWriter, DataOutputStream
- **Communication** – simplex, duplex
- **Exceptions** – ConnectException, SocketException
- **Threads** – Thread, run(), start()
- **Multithreaded servers** – handling multiple clients
- **HTTP** – requests, responses, status codes
- **HTTPS** – SSL/TLS, secure communication