Streams, Network Sockets and Threads II – HTTP Server

Zbigniew Dendzik
Institute of Physics, University of Silesia

2025

Introduction

Another example of programming for the Java platform in the area of network sockets, data streams and multithreading will be the implementation of a simple HTTP server. The HTTP protocol is described in the appropriate document RFC 2616 (http 1.1). Our implementation will use character streams to retrieve data sent by the browser to the server, as well as byte streams – to transmit requested documents (remember that in addition to html text, your server will also send binary data – graphics, sound, or applet bytecodes).

Example I

Below you will find a simple implementation of an HTTP server in Java. Analyse the example carefully, prepare an appropriate compilation unit (package, source files, etc.), and then compile and test the example. Of course, the server's client will be a web browser. In the above example, it would actually be sufficient to use `InputStream` and `OutputStream` streams. The specialised streams `BufferedReader` and `DataOutputStream` were used for convenience. `BufferedReader` has functionality (methods) that allows convenient operation on text data streams (request). `DataOutputStream` has functionality that allows convenient operation on binary streams (response). Sending text to the browser directly via an `OutputStream` object would require using the `getBytes()` method and writing the retrieved byte array using the `write(byte[] bytes)` method.

```
1  import java.io.*;
2  import java.net.*;
3
4  public class HTTPServer
5  {
6     public static void main(String[] args) throws IOException
7     {
8        ServerSocket serv=new ServerSocket(80);
9
10       while(true)
11       {
12          //accepting connection
```

```
13            System.out.println("Waiting for connection...");
14            Socket sock=serv.accept();
15
16            //data streams
17            InputStream is=sock.getInputStream();
18            OutputStream os=sock.getOutputStream();
19            BufferedReader inp=new BufferedReader(new
                 InputStreamReader(is));
20            DataOutputStream outp=new DataOutputStream(os);
21
22            //receiving request
23            String request=inp.readLine();
24
25            //sending response
26            if(request.startsWith("GET"))
27            {
28               //response header
29               outp.writeBytes("HTTP/1.0 200 OK\r\n");
30               outp.writeBytes("Content-Type: text/html\r\n");
31               outp.writeBytes("Content-Length: \r\n");
32               outp.writeBytes("\r\n");
33
34               //response body
35               outp.writeBytes("<html>\r\n");
36               outp.writeBytes("<H1>Test page</H1>\r\n");
37               outp.writeBytes("</html>\r\n");
38            }
39            else
40            {
41               outp.writeBytes("HTTP/1.1 501 Not supported.\r\n");
42            }
43
44            //closing streams
45            inp.close();
46            outp.close();
47            sock.close();
48         }
49      }
50   }
```

Listing 1: HTTPServer.java

Ex. 6.1

Test the above example. Recall the basic concepts related to the HTTP protocol, i.e. the way a web browser communicates with a www server. Review and familiarise yourself with the structure of the relevant RFC documents describing the HTTP and HTTPS protocols.

Ex. 6.2

Write a code fragment that will print to standard output the request that the browser sends to the server. Analyse its content. In the above example, only the first line of the request is written to the `request` variable (in general there may be more). Write a code fragment that will print to standard output (`System.out.println()`) the complete information sent by the browser.

Ex. 6.3

Write a code fragment that will retrieve from the (text) variable `request` the name of the file that the browser is requesting and print that name to standard output. You can use methods from the `java.lang.String` class for this purpose.

Ex. 6.4

Write an appropriate code fragment that will create a `java.io.FileInputStream` object for a file with the given name. If such a file does not exist, your server should send appropriate information to the browser preceded by the header: "HTTP/1.0 404 Not Found".

Ex. 6.5

Create a buffer (byte array of size e.g. 1024) that will be used to store bytes retrieved from the file before sending them to the browser. Write an appropriate code fragment that will retrieve bytes from the file to the buffer, and then send the contents of the buffer to the browser that sent the request. You can implement this, for example, according to the following scheme:

```
1  FileInputStream fis = new FileInputStream(fileName);
2
3  byte[] buffer;
4  buffer=new byte[1024];
5  int n=0;
6
7  while ((n = fis.read(buffer)) != -1 )
8  {
9     outp.write(buffer, 0, n);
10 }
```

Listing 2: Buffer scheme

Ex. 6.6

Using the appropriate `int available()` method of the `java.io.FileInputStream` class, retrieve information about the number of bytes in the file and use it to define the `Content-Length` header. Define the appropriate `Content-Type` header based on the file extension – for html and htm files: "Content-Type: text/html\r\n", for other files: "Content-Type: \r\n".

Example II

Below you will find the skeleton implementation of a multithreaded server that creates a separate thread for each incoming request that handles that request. The `run()` method contains the "life cycle" of the thread. Production servers typically create a certain thread pool. From this pool, each request is assigned a thread that will handle it. After handling the request, the thread returns to the pool.

```
1  import java.io.*;
2  import java.net.*;
3
4  class TaskHandler extends Thread
5  {
6     Socket sock;
7
8     TaskHandler(Socket clientSocket)
9     {
10       this.sock=clientSocket;
11    }
12
13    public void run()
14    {
15
16    }
17 }
18
19 public class HTTPServer
20 {
21    public static void main(String[] args) throws IOException
22    {
23       ServerSocket serv=new ServerSocket(80);
24
25       while(true)
26       {
27          //accepting connection
28          System.out.println("Waiting for connection...");
29          Socket sock=serv.accept();
30
31          //creating thread to handle this connection
32          new TaskHandler(sock).start();
33       }
34    }
35 }
```

Listing 3: HTTPServer.java – multithreaded server

Ex. 6.7

Based on the above example, write an implementation of a multithreaded server that will create a new thread for each request whose task will be to handle that request. Write a web page (html) containing at least 5 images and several Java applets and test the entire

server implementation thoroughly. Answer the question of how many requests the browser must send to receive such a page. Observe whether there is a difference in performance between an iterative (single-threaded) server and a multithreaded server.

Ex. 6.8

Extend your server in such a way that during startup you can specify any port number as a parameter passed from the operating system command line. To convert text to an `int` variable, you can use the static method `parseInt()` from the `java.lang.Integer` class. In case this port number is already being used by another network application, a `java.net.BindException` exception will occur. Handle this exception.

Ex. 6.9

Implement logging of server "activity" to a log file. Each entry should contain the request and be marked with a date (`java.util.Date` or `java.util.Calendar`) and the IP address (appropriate method of the `java.net.Socket` class) of the browser that submitted the request.

Example III

Below you will find the skeleton of a simplified implementation of an HTTPS server, using the capabilities of the Java Platform API in the area of secure network sockets using the SSL (Secure Socket Layer) protocol.

```
1  import java.io.*;
2  import java.net.*;
3  import javax.net.ssl.*;
4
5  public class HTTPSServer
6  {
7
8     public static void main(String[] args) throws IOException
9     {
10       SSLServerSocketFactory fact;
11       fact = (SSLServerSocketFactory)SSLServerSocketFactory.
            getDefault();
12       ServerSocket servsock = fact.createServerSocket(8080);
13
14       while (true)
15       {
16          try
17          {
18             Socket sock = servsock.accept();
19
20             OutputStream out;
21             out=sock.getOutputStream();
22
23             BufferedReader in;
```

```
24              in=new BufferedReader(new InputStreamReader(sock.
                   getInputStream()));
25
26
27
28              in.close();
29              out.close();
30              sock.close();
31          }
32          catch (Exception e)
33          {
34              e.printStackTrace();
35          }
36      }
37  }
38 }
```

Listing 4: HTTPSServer.java

Ex. 6.10*

Based on the above example, implement HTTPS protocol support.