

CSE 7345  
Web Server

\* An example of a very simple, multi-threaded HTTP server.  
\* Implementation notes are in WebServer.html, and also  
\* as comments in the source code.

\*/

```
1 import java.io.*;
2 import java.net.*;
3 import java.util.*;
4 class WebServer implements HttpConstants {
5
6     /* static class data/methods */
7
8     /* print to stdout */
9     protected static void p(String s) {
10         System.out.println(s);
11     }
12
13     /* print to the log file */
14     protected static void log(String s) {
15         synchronized (log) {
16             log.println(s);
17             log.flush();
18         }
19     }
20
21     static PrintStream log = null;
22     /* our server's configuration information is stored
23      * in these properties
24      */
25     protected static Properties props = new Properties();
26
27     /* Where worker threads stand idle */
28     static Vector threads = new Vector();
29
30     /* the web server's virtual root */
31     static File root;
32
33     /* timeout on client connections */
34     static int timeout = 0;
35
36     /* max # worker threads */
37     static int workers = 5;
38
39
40     /* load www-server.properties from java.home */
41     static void loadProps() throws IOException {
42         File f = new File
```

```

43         (System.getProperty("java.home")+File.separator+
44             "lib"+File.separator+"www-server.properties");
45     if (f.exists()) {
46         InputStream is =new BufferedInputStream(new
47             FileInputStream(f));
48         props.load(is);
49         is.close();
50         String r = props.getProperty("root");
51         if (r != null) {
52             root = new File(r);
53             if (!root.exists()) {
54                 throw new Error(root + " doesn't exist as server
55 root");
56             }
57         }
58         r = props.getProperty("timeout");
59         if (r != null) {
60             timeout = Integer.parseInt(r);
61         }
62         r = props.getProperty("workers");
63         if (r != null) {
64             workers = Integer.parseInt(r);
65         }
66         r = props.getProperty("log");
67         if (r != null) {
68             p("opening log file: " + r);
69             log = new PrintStream(new BufferedOutputStream(
70                 new FileOutputStream(r)));
71         }
72     }
73
74     /* if no properties were specified, choose defaults */
75     if (root == null) {
76         root = new File(System.getProperty("user.dir"));
77     }
78     if (timeout <= 1000) {
79         timeout = 5000;
80     }
81     if (workers < 25) {
82         workers = 5;
83     }
84     if (log == null) {
85         p("logging to stdout");
86         log = System.out;
87     }
88 }
89
90 static void printProps() {
91     p("root="+root);

```

```

92         p("timeout="+timeout);
93         p("workers="+workers);
94     }
95
96     public static void main(String[] a) throws Exception {
97         int port = 8080;
98         if (a.length > 0) {
99             port = Integer.parseInt(a[0]);
100        }
101        loadProps();
102        printProps();
103        /* start worker threads */
104        for (int i = 0; i < workers; ++i) {
105            Worker w = new Worker();
106            (new Thread(w, "worker #" + i)).start();
107            threads.addElement(w);
108        }
109
110        ServerSocket ss = new ServerSocket(port);
111        while (true) {
112
113            Socket s = ss.accept();
114
115            Worker w = null;
116            synchronized (threads) {
117                if (threads.isEmpty()) {
118                    Worker ws = new Worker();
119                    ws.setSocket(s);
120                    (new Thread(ws, "additional worker")).start();
121                } else {
122                    w = (Worker) threads.elementAt(0);
123                    threads.removeElementAt(0);
124                    w.setSocket(s);
125                }
126            }
127        }
128    }
129 }
130
131
132 class Worker extends WebServer implements HttpConstants, Runnable {
133     final static int BUF_SIZE = 2048;
134
135     static final byte[] EOL = {(byte)'\r', (byte)'\n' };
136
137     /* buffer to use for requests */
138     byte[] buf;
139     /* Socket to client we're handling */
140     private Socket s;

```

```

141
142     Worker() {
143         buf = new byte[BUF_SIZE];
144         s = null;
145     }
146
147     synchronized void setSocket(Socket s) {
148         this.s = s;
149         notify();
150     }
151
152     public synchronized void run() {
153         while(true) {
154             if (s == null) {
155                 /* nothing to do */
156                 try {
157                     wait();
158                 } catch (InterruptedException e) {
159                     /* should not happen */
160                     continue;
161                 }
162             }
163             try {
164                 handleClient();
165             } catch (Exception e) {
166                 e.printStackTrace();
167             }
168             /* go back in wait queue if there's fewer
169              * than numHandler connections.
170              */
171             s = null;
172             Vector pool = WebServer.threads;
173             synchronized (pool) {
174                 if (pool.size() >= WebServer.workers) {
175                     /* too many threads, exit this one */
176                     return;
177                 } else {
178                     pool.addElement(this);
179                 }
180             }
181         }
182     }
183
184     void handleClient() throws IOException {
185         InputStream is = new BufferedInputStream(s.getInputStream());
186         PrintStream ps = new PrintStream(s.getOutputStream());
187         /* we will only block in read for this many milliseconds
188          * before we fail with java.io.InterruptedIOException,
189          * at which point we will abandon the connection.

```

```

190         */
191         s.setSoTimeout(WebServer.timeout);
192         s.setTcpNoDelay(true);
193         /* zero out the buffer from last time */
194         for (int i = 0; i < BUF_SIZE; i++) {
195             buf[i] = 0;
196         }
197         try {
198             /* We only support HTTP GET/HEAD, and don't
199              * support any fancy HTTP options,
200              * so we're only interested really in
201              * the first line.
202              */
203             int nread = 0, r = 0;
204
205             outerloop:
206             while (nread < BUF_SIZE) {
207                 r = is.read(buf, nread, BUF_SIZE - nread);
208                 if (r == -1) {
209                     /* EOF */
210                     return;
211                 }
212                 int i = nread;
213                 nread += r;
214                 for (; i < nread; i++) {
215                     if (buf[i] == (byte)'\n' || buf[i] == (byte)'\r') {
216                         /* read one line */
217                         break outerloop;
218                     }
219                 }
220             }
221
222             /* are we doing a GET or just a HEAD */
223             boolean doingGet;
224             /* beginning of file name */
225             int index;
226             if (buf[0] == (byte)'G' &&
227                 buf[1] == (byte)'E' &&
228                 buf[2] == (byte)'T' &&
229                 buf[3] == (byte)' ') {
230                 doingGet = true;
231                 index = 4;
232             } else if (buf[0] == (byte)'H' &&
233                 buf[1] == (byte)'E' &&
234                 buf[2] == (byte)'A' &&
235                 buf[3] == (byte)'D' &&
236                 buf[4] == (byte)' ') {
237                 doingGet = false;
238                 index = 5;

```

```

239     } else {
240         /* we don't support this method */
241         ps.print("HTTP/1.0 " + HTTP_BAD_METHOD +
242             " unsupported method type: ");
243         ps.write(buf, 0, 5);
244         ps.write(EOL);
245         ps.flush();
246         s.close();
247         return;
248     }
249
250     int i = 0;
251     /* find the file name, from:
252     * GET /foo/bar.html HTTP/1.0
253     * extract "/foo/bar.html"
254     */
255     for (i = index; i < nread; i++) {
256         if (buf[i] == (byte)' ') {
257             break;
258         }
259     }
260     String fname = (new String(buf, 0, index,
261         i-index)).replace('/', File.separatorChar);
262     if (fname.startsWith(File.separator)) {
263         fname = fname.substring(1);
264     }
265     File targ = new File(WebServer.root, fname);
266     if (targ.isDirectory()) {
267         File ind = new File(targ, "index.html");
268         if (ind.exists()) {
269             targ = ind;
270         }
271     }
272     boolean OK = printHeaders(targ, ps);
273     if (doingGet) {
274         if (OK) {
275             sendFile(targ, ps);
276         } else {
277             send404(targ, ps);
278         }
279     }
280 } finally {
281     s.close();
282 }
283 }
284
285 boolean printHeaders(File targ, PrintStream ps) throws IOException {
286     boolean ret = false;
287     int rCode = 0;

```

```

288     if (!targ.exists()) {
289         rCode = HTTP_NOT_FOUND;
290         ps.print("HTTP/1.0 " + HTTP_NOT_FOUND + " not found");
291         ps.write(EOL);
292         ret = false;
293     } else {
294         rCode = HTTP_OK;
295         ps.print("HTTP/1.0 " + HTTP_OK+" OK");
296         ps.write(EOL);
297         ret = true;
298     }
299     log("From " +s.getInetAddress().getHostAddress()+" : GET " +
300         targ.getAbsolutePath()+"-->" +rCode);
301     ps.print("Server: Simple java");
302     ps.write(EOL);
303     ps.print("Date: " + (new Date()));
304     ps.write(EOL);
305     if (ret) {
306         if (!targ.isDirectory()) {
307             ps.print("Content-length: "+targ.length());
308             ps.write(EOL);
309             ps.print("Last Modified: " + (new
310                 Date(targ.lastModified())));
311             ps.write(EOL);
312             String name = targ.getName();
313             int ind = name.lastIndexOf('.');
314             String ct = null;
315             if (ind > 0) {
316                 ct = (String) map.get(name.substring(ind));
317             }
318             if (ct == null) {
319                 ct = "unknown/unknown";
320             }
321             ps.print("Content-type: " + ct);
322             ps.write(EOL);
323         } else {
324             ps.print("Content-type: text/html");
325             ps.write(EOL);
326         }
327     }
328     return ret;
329 }
330
331 void send404(File targ, PrintStream ps) throws IOException {
332     ps.write(EOL);
333     ps.write(EOL);
334     ps.println("Not Found\n\n"+
335         "The requested resource was not found.\n");
336 }

```

```

337
338 void sendFile(File targ, PrintStream ps) throws IOException {
339     InputStream is = null;
340     ps.write(EOL);
341     if (targ.isDirectory()) {
342         listDirectory(targ, ps);
343         return;
344     } else {
345         is = new FileInputStream(targ.getAbsolutePath());
346     }
347
348     try {
349         int n;
350         while ((n = is.read(buf)) > 0) {
351             ps.write(buf, 0, n);
352         }
353     } finally {
354         is.close();
355     }
356 }
357
358 /* mapping of file extensions to content-types */
359 static java.util.Hashtable map = new java.util.Hashtable();
360
361 static {
362     fillMap();
363 }
364 static void setSuffix(String k, String v) {
365     map.put(k, v);
366 }
367
368 static void fillMap() {
369     setSuffix("", "content/unknown");
370     setSuffix(".uu", "application/octet-stream");
371     setSuffix(".exe", "application/octet-stream");
372     setSuffix(".ps", "application/postscript");
373     setSuffix(".zip", "application/zip");
374     setSuffix(".sh", "application/x-shar");
375     setSuffix(".tar", "application/x-tar");
376     setSuffix(".snd", "audio/basic");
377     setSuffix(".au", "audio/basic");
378     setSuffix(".wav", "audio/x-wav");
379     setSuffix(".gif", "image/gif");
380     setSuffix(".jpg", "image/jpeg");
381     setSuffix(".jpeg", "image/jpeg");
382     setSuffix(".htm", "text/html");
383     setSuffix(".html", "text/html");
384     setSuffix(".text", "text/plain");
385     setSuffix(".c", "text/plain");

```

```

386         setSuffix(".cc", "text/plain");
387         setSuffix(".c++", "text/plain");
388         setSuffix(".h", "text/plain");
389         setSuffix(".pl", "text/plain");
390         setSuffix(".txt", "text/plain");
391         setSuffix(".java", "text/plain");
392     }
393
394     void listDirectory(File dir, PrintStream ps) throws IOException {
395         ps.println("<TITLE>Directory listing</TITLE><P>\n");
396         ps.println("<A HREF=\"..\>Parent Directory</A><BR>\n");
397         String[] list = dir.list();
398         for (int i = 0; list != null && i < list.length; i++) {
399             File f = new File(dir, list[i]);
400             if (f.isDirectory()) {
401                 ps.println("<A
402 HREF=\""+list[i]+"/>"+list[i]+"/</A><BR>");
403             } else {
404                 ps.println("<A
405 HREF=\""+list[i]+\">"+list[i]+"/</A><BR>");
406             }
407         }
408         ps.println("<P><HR><BR><I>" + (new Date()) + "</I>");
409     }
410 }
411 }
412
413 interface HttpConstants {
414     /** 2XX: generally "OK" */
415     public static final int HTTP_OK = 200;
416     public static final int HTTP_CREATED = 201;
417     public static final int HTTP_ACCEPTED = 202;
418     public static final int HTTP_NOT_AUTHORITATIVE = 203;
419     public static final int HTTP_NO_CONTENT = 204;
420     public static final int HTTP_RESET = 205;
421     public static final int HTTP_PARTIAL = 206;
422
423     /** 3XX: relocation/redirect */
424     public static final int HTTP_MULT_CHOICE = 300;
425     public static final int HTTP_MOVED_PERM = 301;
426     public static final int HTTP_MOVED_TEMP = 302;
427     public static final int HTTP_SEE_OTHER = 303;
428     public static final int HTTP_NOT_MODIFIED = 304;
429     public static final int HTTP_USE_PROXY = 305;
430
431     /** 4XX: client error */
432     public static final int HTTP_BAD_REQUEST = 400;
433     public static final int HTTP_UNAUTHORIZED = 401;
434     public static final int HTTP_PAYMENT_REQUIRED = 402;

```

```
435     public static final int HTTP_FORBIDDEN = 403;
436     public static final int HTTP_NOT_FOUND = 404;
437     public static final int HTTP_BAD_METHOD = 405;
438     public static final int HTTP_NOT_ACCEPTABLE = 406;
439     public static final int HTTP_PROXY_AUTH = 407;
440     public static final int HTTP_CLIENT_TIMEOUT = 408;
441     public static final int HTTP_CONFLICT = 409;
442     public static final int HTTP_GONE = 410;
443     public static final int HTTP_LENGTH_REQUIRED = 411;
444     public static final int HTTP_PRECON_FAILED = 412;
445     public static final int HTTP_ENTITY_TOO_LARGE = 413;
446     public static final int HTTP_REQ_TOO_LONG = 414;
447     public static final int HTTP_UNSUPPORTED_TYPE = 415;
448
449     /** 5XX: server error */
450     public static final int HTTP_SERVER_ERROR = 500;
451     public static final int HTTP_INTERNAL_ERROR = 501;
452     public static final int HTTP_BAD_GATEWAY = 502;
453     public static final int HTTP_UNAVAILABLE = 503;
454     public static final int HTTP_GATEWAY_TIMEOUT = 504;
455     public static final int HTTP_VERSION = 505;
456 }
457
458
459
460
461
```