Overview

- Application Protocol: HTTP (HyperText Transmission Protocol)
- Implementation of Stream-based Clients and Servers on TCP/IP connection
Application Protocols

- An application protocol facilitates communication between applications through the services of the lower-level layers of the OSI (Open Systems Interconnection) model.
- TCP/IP Stack:

```
<table>
<thead>
<tr>
<th>HTTP</th>
<th>POP3</th>
<th>SMTP</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.....</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internet Protocol (IP)</td>
<td>Application Layer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transmission Control Protocol (TCP)</td>
<td>Transport Layer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>Network Layer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- An application protocol specification comprises the syntax and semantics of the protocol.
- An application protocol specification is published as a RFC (Request For Comment) document.
- Application Protocols: stateless and request-response-based
  - SMTP: Simple Mail Transfer Protocol (RFC 2821) for Internet electronic mail delivery.
  - POP3: Post Office Protocol (RFC 1939) for retrieval of mail from a mail server.

HTTP: HyperText Transmission Protocol

- Used for retrieving resources from and posting information to a web server.
- Two important versions: HTTP/1.0 (RFC 1945), HTTP/1.1 (RFC 2616)
- A HTTP server usually runs on port 80 (or 8080).
- A client is a user of services.
- A server is a provider of services.
- The client opens a connection to the server and sends a request to the server. The client receives a response from the server and closes the connection.
- Regarding the Internet:
  - A web browser is a HTTP client
  - A web server is a HTTP server
  - A resource is identified by a URI (Universal Resource Identifier)
- The web client should be able to construct an appropriate HTTP request, and interpret a HTTP response.
- The web server should be able to interpret the HTTP request, and construct an appropriate HTTP response.
Universal Resource Identifier: URI

- A URI is a superset of URL and URN. It is an identifier that identifies a resource. The resource may or may not exist. Neither does it imply how we can retrieve the resource.

ATIJ/lecture-notes-kam/atij-application-protocols

Universal Resource Locator: URL

- A URL specifies a unique address/location for a resource on the Web.
- Common form:
  
  `<protocol>://<hostname>[:<TCP port number>]/<pathname>[?<query>][#<reference>]

  http://www.ii.uib.no:80/~khalid/pgjc2e/
  mailto:khalid@ii.uib.no?Subject=Urgent%20Message
  http://www.w3.org/TR/REC-html32#intro  <!-- Tag to indicate particular part of a document.

Universal Resource Name: URN

- A URN is a unique identifier that identifies a resource, irrespective of its location and mode of retrieval.

  ISBN: 0–201–72828–1

HTTP Message Format

<table>
<thead>
<tr>
<th>HTTP Message Parts</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>The start line</td>
<td>Specifies the purpose of the message: a <em>method line</em> in case of a request and a <em>status line</em> in the case of a response.</td>
</tr>
<tr>
<td>The header section</td>
<td>Comprises a list of request/response header fields and/or entity header fields.</td>
</tr>
<tr>
<td></td>
<td>Request/response header fields allows additional information about the message to be passed.</td>
</tr>
<tr>
<td></td>
<td>Entity header fields specify the metainformation (size, type, encoding, etc.) about the content of the entity-body.</td>
</tr>
<tr>
<td>A blank line (CRLF)</td>
<td>Terminates the header section. Mandatory.</td>
</tr>
<tr>
<td>Optional entity-body</td>
<td>The content of the message.</td>
</tr>
</tbody>
</table>

- HTTP message is either a *request* from a client to a server, or a *response* from a server to a client.

- Each line of the HTTP message is terminated with a CRLF ("\r\n").
Cookies

- HTTP application protocol is stateless, i.e. it has no "memory".
- A cookie is a small piece of persistent data stored in a text file on the client side, which the server side can utilize in its interaction with the client.
- The server introduces a cookie to the client by including a Set-Cookie header field as part of an HTTP response.
- Any future HTTP requests made by the client will include a transmittal of the cookie in a Cookie header field from the client back to the server.
- A cookies comprises of the following information:
  - a list of name/value pairs
  - a URI
  - an optional expiry date
In subsequent requests involving this URI, the client will always include the list of name/value pairs from the cookie.
- Cookies facilitate tracking of client requests and customizing of server response for individual clients.

CGI: Common Gateway Interface

- CGI is an interface that allows servers to start server-side programs to handle client requests.
- CGI makes possible interaction or passing of information between a client and a server-side application.

MIME Type

- MIME (Multipurpose Internet Mail Extensions) is a message representation protocol, i.e. it specifies the structure of MIME messages.
- A MIME type specifies the media type and subtype of data in the body of a message (as the value of a Content-Type header field):
  \(<\text{type}>/\text{<subtype>}\>
- A MIME type allows an appropriate application to be used to interpret the data in a resource.
  application/msword
  application/zip
  text/html
  image/jpeg
  video/mpeg
HTTP Request Specification

- The general syntax of the method line of a HTTP request is:
  \(<\text{method name}> \ <\text{URI}> \ <\text{http version number}>\)

- A HTTP method specifies the action that the client wants the server to perform.

HTTP/1.0 methods that a web client can issue to a web server:

GET Retrieve the resource identified by the URI (specified as the local path):
  \(\text{GET} \ <\text{URI}> \ <\text{http version no.}>\)

Examples:

- GET /index.html HTTP/1.0
- GET /book/chapter1/figure1.jpg HTTP/1.0
- GET /servlet/register?first=khalid&last=mughal HTTP/1.0

The first two examples specify a passive resource, where as the last one specifies an active resource which does processing.

The query string specifies parameter-value pairs which are passed to the server-side application. The URI may need to be encoded, and then decoded at the server-side. See classes URLEncoder and URLDecoder.
**HEAD**

Retrieve only metainformation about the resource identified by the URI:

```
HEAD <URI> <http version no.>
```

Examples:

```
HEAD /index.html HTTP/1.0
HEAD /book/chapter1/figure1.jpg HTTP/1.0
```

The format of the HEAD request is identical to the GET request. The response is identical to the response to a GET request, but the response does not contain a entity-body, only the header fields. This method is often used for testing hypertext links for validity, accessibility, and recent modification.

---

**POST**

The POST method is used to pass information in the entity-body to the active resource identified by the URI:

```
POST <URI> <http version no.>
```

```
<header section>
<CRLF>
<entity-body>
```

The headers describe the information in the entity-body, for example the type of the content (Content-Type) and the number of bytes in the entity-body (Content-Length).

Example:

```
POST /servlet/magicSocket HTTP/1.0
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; ...)
Content-Type: application/x-www-form-urlencoded
Content-Length: ...
<CRLF>
question=Socket,%20socket%20in%20the%20wall.%20Who%20is
fairest%20client%20of%20them%20all?
```

The entity-body is encoded for unsafe characters when sent server-side. See classes URLEncoder and URLDecoder.
### HTTP Request Header Fields

<table>
<thead>
<tr>
<th>Request Header Field</th>
<th>Description</th>
</tr>
</thead>
</table>
| **Accept**           | Specify media types which are acceptable for the response.  
  Example:  
  - Accept: text/plain, text/html, audio/*  
  - Accept: text/*  
  
  "/*" indicates all subtypes in the specified media category. |
| **Accept-Charset**   | Specify character sets which are acceptable for the response.  
  Example:  
  - Accept-Charset: iso-8859-5 |
| **Accept-Encoding**  | Specify character encodings which are acceptable for the response.  
  Example:  
  - Accept-Encoding: compress, gzip |
| **Accept-Language**  | Specify natural languages that are preferred as a response.  
  Example:  
  - Accept-Language: no, en-us |
| **From**             | E-mail address of person controlling the requesting user-agent.  
  From: khalid.mughal@i1.uib.no |
| **Host**             | Specifies the Internet host and port number of the requested resource in the URI. No trailing port implies the default port for the service requested (for example, "80" for an HTTP URL).  
  Example:  
  - Host: localhost |
| **If-Modified-Since**| Conditional request to return an entity only if it has been modified since the specified date.  
  Example:  
  - If-Modified-Since: Sat, 1 Mar 2004 12:45:15 GMT |
| **If-Unmodified-Since**| Conditional request to return an entity only if it has *not* been modified since the specified date.  
  Example:  
  - If-Unmodified-Since: Sat, 1 Mar 2004 12:45:15 GMT |
### Request Header Field

<table>
<thead>
<tr>
<th>Request Header Field</th>
<th>Description</th>
</tr>
</thead>
</table>
| Referer              | Specify the URI of the resource that linked to the requested URI (*a back link*).  
Example: Referer: http://www.ii.uib.no/~khalid/index.html |
| User-Agent           | Specify information about the user-agent (software) originating the request.  
Example: User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; ...) |
| Cookie:              | The name/value pairs of all cookies matching the requested URI are specified with this header.  
Example: Cookie: passenger=High_Flyer |
| Connection:          | A general header which specifies options that are desired for that particular connection. Options close and Keep-Alive indicate a non-persistent and persistent connection, respectively.  
Example: Connection: Keep-Alive |

### HTTP Response Specification

- The general syntax of the status line of a HTTP response is:

  `<http version number> <response status code> <description>`

Examples of single-line responses:

```
HTTP/1.0 200 OK
HTTP/1.0 404 Not Found
```

Example of a multiple-line response:

```
HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: ...

<html>
<head>Answer</head>
<body>
  <h1>You are the lucky winner of $1000000!</h1>
</body>
</html>
```

- The metainformation about the content of the entity-body is given by the entity header fields.
Response Status Codes in HTTP Response Status Line

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>OK</td>
</tr>
<tr>
<td>201</td>
<td>Created</td>
</tr>
<tr>
<td>202</td>
<td>Accepted</td>
</tr>
<tr>
<td>204</td>
<td>No Content</td>
</tr>
<tr>
<td>300</td>
<td>Multiple Choices</td>
</tr>
<tr>
<td>301</td>
<td>Moved Permanently</td>
</tr>
<tr>
<td>302</td>
<td>Moved Temporarily</td>
</tr>
<tr>
<td>304</td>
<td>Not Modified</td>
</tr>
<tr>
<td>400</td>
<td>Bad Request</td>
</tr>
<tr>
<td>401</td>
<td>Unauthorized</td>
</tr>
<tr>
<td>403</td>
<td>Forbidden</td>
</tr>
<tr>
<td>404</td>
<td>Not Found</td>
</tr>
<tr>
<td>303</td>
<td>See Other</td>
</tr>
<tr>
<td>307</td>
<td>Temporary Redirect</td>
</tr>
<tr>
<td>500</td>
<td>Internal Server Error</td>
</tr>
<tr>
<td>501</td>
<td>Not Implemented</td>
</tr>
<tr>
<td>502</td>
<td>Bad Gateway</td>
</tr>
<tr>
<td>503</td>
<td>Service Unavailable</td>
</tr>
</tbody>
</table>

HTTP Response Header Fields

<table>
<thead>
<tr>
<th>Response Header Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location:</td>
<td>Specifies the absolute path of the resource in the case of redirection. \Example: Location: <a href="http://www.ii.uib.no/~khalid/pjgc2e/">http://www.ii.uib.no/~khalid/pjgc2e/</a></td>
</tr>
<tr>
<td>Server:</td>
<td>Specifies the software used by the server to handle the request. \Example: Server: MegaServer/1.0</td>
</tr>
<tr>
<td>Pragma:</td>
<td>A general header which specifies implementation-specific directives. \For example, to prevent caching of a resource, a server can specify the following directive: Pragma: no-cache</td>
</tr>
<tr>
<td>Set-Cookie:</td>
<td>Specifies a cookie which the client will include in all subsequent requests to the server. \Example: Set-cookie: passenger=High_Flyer; path=/</td>
</tr>
</tbody>
</table>
## HTTP Entity Header Fields in a HTTP Message

<table>
<thead>
<tr>
<th>Entity Header Field</th>
<th>Description</th>
</tr>
</thead>
</table>
| **Content-Encoding:** | Specifies what additional content coding has been applied to the resource, and thus what decoding mechanism must be applied in order to obtain the media-type referenced by the Content-Type header field.  
Example:  
Content-Encoding: gzip |
| **Content-Length:** | Specifies the size of the entity body, i.e. the number of bytes comprising the entity-body. Client can take this information into consideration when dealing with the entity-body.  
Example:  
Content-Length: 1949 |
| **Content-Type:** | Specifies the media type of the entity body.  
Example:  
Content-Type: text/html |
| **Expires:** | Specifies the date/time when the resource would become stale. This does not imply that the original resource will be affected.  
Example:  
Expires: WED, 24 Dec 2004 12:00:00 GMT |
| **Last-Modified:** | Specifies the date/time when the resource was last modified.  
Example:  
Last-modified: WED, 24 Dec 2004 12:00:00 GMT |
Some Useful Classes

- Lexical analysis of messages: the java.util.StringTokenizer class, the java.util.Scanner class (See notes on new features in Java 5.0.)
- Dealing with files and directories: the java.io.File class
- Encoding and decoding URLs: java.net.URLEncoder, java.net.URLDecoder classes
- Constructing messages: the String, StringBuilder, and StringBuffer classes
- Check out the java.util.regex class.

Lexical Analysis with the StringTokenizer Class

- A StringTokenizer breaks a string into words, also called tokens.
- Delimiter characters are used to recognize tokens in the string.

```java
String input = "To be. Or not to be?";
StringTokenizer tokenlist1 = new StringTokenizer(input); // default " \t\n\r\f"
StringTokenizer tokenlist2 = new StringTokenizer(input, ".?" ); // Delimiters ".?"

- Delimiters are not part of a token.
- The tokens in a StringTokenizer can be read as a String one at a time by calling the nextToken() method successively.
- The method hasMoreTokens() can be used to check if all tokens have been read.
- These two methods should be used in sync.

    while (tokenlist1.hasMoreTokens()) {
        System.out.println(tokenlist1.nextToken());
    }

- Delimiters can be changed on the fly when using a StringTokenizer.
  - use the overloaded method nextToken(String delimiters).
```
Example: StringTokenizer

```java
import java.util.*;
public class UsingStringTokenizer {
    public static void main(String[] args) {
        // Input string.
        String input = "To be. Or not to be?";
        // Create a tokenizer. Default delimiters: " 	
\r\f"
        StringTokenizer tokenlist1 = new StringTokenizer(input);
        // Create a tokenizer. Delimiters "."?
        StringTokenizer tokenlist2 = new StringTokenizer(input, ".?";
        System.out.println("--------------");
        printTokens(tokenlist1);
        System.out.println("--------------");
        printTokens(tokenlist2);
    }
    public static void printTokens(StringTokenizer tokenlist) {
        while (tokenlist.hasMoreTokens()) {
            String token = tokenlist.nextToken();
            System.out.println(token);
        }
    }
}
```

Running the program:
> java UsingStringTokenizer
--------------
To
be.
Or
not
to
be?
--------------
To be
Or not to be
The File Class

- A File object is an immutable abstract representation of a pathname (URI) for a file or a directory.
- A File object can represent an absolute or a relative pathname.
- The File class provides methods to manipulate pathnames in a platform-independent way.
- The File class has many methods which return a String representation of the pathname components.

```java
File myPathname = new File("..\assignments\week1"); // "./assignments/week1"
// Given that the current directory is d:\course\.

getName() Returns the last component of the pathname ("week1").
getParent() Returns the pathname minus the last component ("..\assignments").
getPath() Returns whatever pathname is in the File object. ("..\assignments\week1")
```

getAbsolutePath() Returns the absolute pathname which is a concatenation of the current directory, the file separator character, and the pathname of the File object.

A file or directory can have many absolute paths. ("d:\course\..\assignments\week1")

getCanonicalPath() Returns an absolute pathname in which all relative path references (such as ".", ".") have been completely resolved.

A file or a directory has a unique canonical path. ("d:\assignments\week1")
Encoding and Decoding URLs

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static String encode(String urlStr, String encoding) throws UnsupportedEncodingException()</td>
<td>This method is defined in class URLEncoder. It translates a string into MIME type application/x-www-form-urlencoded format using a specific encoding scheme.</td>
</tr>
<tr>
<td>static String decode(String urlStr, String encoding)</td>
<td>This method is defined in class URLDecoder. It decodes a string in MIME type application/x-www-form-urlencoded format using a specific encoding scheme.</td>
</tr>
</tbody>
</table>

- Encoding procedure:
  - The alphanumeric characters and the special characters '.', '-', '*', and '_' are unchanged.
  - The space character ' ' is converted into a plus sign '+'. (Might also see "%20").
  - All other characters are unsafe and are first converted into one or more bytes using some encoding scheme (recommended: "UTF8"). Then each byte is represented by the 3-character string "%xy", where xy is the two-digit hexadecimal representation of the byte.

- For example, "/servlet/register?first=khalid azim&last=mughal" is encoded as "/%2Fregister%3Ffirst%3Dkhalid+azim&last%3Dmughal".

Example: A Directory Navigator Server

Directory: atij-application-protocols

File: HTTPServer.java

```java
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

public class HTTPServer {
    private Socket serverSocket; // local port
    private String directory; // Content Directory

    public HTTPServer(String directory, int port) throws IOException {
        this.directory = directory;
        this.port = port;
        // Create the socket server.
        serverSocket = new ServerSocket(this.port);
    }

    public void handleClients() {
        for (;;) {
            try {
                // Accept a new socket connection for the next client
                Socket clientSocket = serverSocket.accept();

                // Create a new server object to process the request
                ServerRequestHandler serverRequestHandler = new ServerRequestHandler(clientSocket, directory);
                serverRequestHandler.start();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Example: A Directory Navigator Server

Directory: atij-application-protocols

File: HTTPServer.java

```java
import java.io.*;
import java.net.*;
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

public class HTTPServer {
    private Socket serverSocket; // local port
    private String directory; // Content Directory

    public HTTPServer(String directory, int port) throws IOException {
        this.directory = directory;
        this.port = port;
        // Create the socket server.
        serverSocket = new ServerSocket(this.port);
    }

    public void handleClients() {
        for (;;) {
            try {
                // Accept a new socket connection for the next client
                Socket clientSocket = serverSocket.accept();

                // Create a new server object to process the request
                ServerRequestHandler serverRequestHandler = new ServerRequestHandler(clientSocket, directory);
                serverRequestHandler.start();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```
Running the Directory Navigator Server

- Source: HTTPServerDemo.java, HTTPServerV3.java

1. Run the following command at the command line:
   ```
   >java HTTPServerDemo
   ```

2. Point the web browser to the following URI:
   ```
   http://localhost/
   ```

3. Navigate by clicking on the links in the web page.

4. The output at the command line shows the HTML content that the server is sending to the web client.

Content Type in HTTP Responses: HTML

- For a file:
  ```
  <html><head><title>File: HTTPServer.java</title></head>
  <body><h1>File: HTTPServer.java</h1>
  <pre>
  ... contents of the file ...
  </pre>
  </body></html>
  ```

- For a directory:
  ```
  <html><head><title>Directory: client-sever-examples</title></head>
  <body><h1>Directory: client-sever-examples</h1>
  <table border="0">
  <tr><td><a href="ThreadedEchoServer.java">ThreadedEchoServer.java</a></td></tr>
  <tr><td><a href="EchoServer.java">EchoServer.java</a></td></tr>
  </table>
  </body></html>
  ```
Implementation of the Directory Navigator Server

- Source: HTTPServerV3.java, HTTPServerDemo.java

1. Create a server socket on a local port (HTTPServerV2()):

   serverSocket = new ServerSocket(this.port);

2. Accept a connection on this local port, and spin off a new thread (handleClients()).

   for (;;) {
     try {
       // Accept a new socket connection for the next client
       Socket clientSocket = serverSocket.accept();

       // Start a new handler instance to process the request
       new ConnectionHandler(clientSocket, directory).start();
     } catch (IOException ioe) {
       ioe.printStackTrace();
     }
   }

3. Implementation of connection handler.

   class ConnectionHandler extends Thread {
     // Nested class.
     private Socket socket;
     private PrintWriter socketWriter;
     private BufferedReader socketReader;
     private File directory;
     private String methodLine;

     public ConnectionHandler(Socket socket, String directory)
       throws IOException {
       this.socket = socket;
       // Get the canonical form of the pathname
       this.directory = new File(directory).getCanonicalFile();
     }

     private String getMethodLine() throws IOException {
       String line = socketReader.readLine();
       System.out.println("<" + line);
       return line;
     }
   }
private String getURLFromMethodLine() throws IOException {
    StringTokenizer tokens = new StringTokenizer(methodLine, " ");
    tokens.nextToken();        // Skip method name
    return tokens.nextToken(); // Requested resource
}

private void readHeaderSection() throws IOException {
    for (; ; ) {
        String line = socketReader.readLine();
        if (line == null)
            throw new IOException("No more input on socket.");
        System.out.println("<" + line);
        if (line.equals("")) break;
    }
}

private String getEntityBody() throws IOException {
    String content = "";
    for (; ; ) {
        String line = socketReader.readLine();
        if (line == null) break;
        System.out.println("<" + line);
        content += line;
    }
    return content;
}

private void sendResponseLine(String line) {
    /*
     * final String CRLF = "\r\n";
     * socketWriter.print(line + CRLF);
     * socketWriter.flush(); // ONLY println() flushes!!!
     */
    socketWriter.println(line); // Flushes.
    System.out.println(">" + line);
}

private void sendBlankLine() {
    sendResponseLine("");
}

private File constructCanonicalPathname(String baseURL,
    String relURL)
    throws IOException {
    ...
}

public void run() { ... }
private void doNotImplemented() { ... }
private void doGetMethod() throws IOException { ... }
private void doPostMethod() throws IOException { ... }
private void determineResponse(File file) throws IOException
{ ... }
private void sendFileResponse(PrintWriter socketWriter,
File file)
throws IOException { ... }
private void sendDirResponse(PrintWriter socketWriter, File dir)
{ ... }

4. Create input and output streams for response/request messages (run()):

   // Connect socketReader and socketWriter to input and output
   // streams of the socket.
   socketReader = new BufferedReader(new InputStreamReader(
       socket.getInputStream()));
   socketWriter = new PrintWriter(socket.getOutputStream(),
       true);// Flushes with println()

5. Determine the Http method in the request (run()):
   // Read the method line of the HTTP request from client.
   methodLine = getMethodLine();

   // Any request at all?
   if(methodLine != null) {
      // Handle GET and POST requests.
      if(methodLine.toUpperCase().startsWith("GET"))
          doGetMethod();
      else if(methodLine.toUpperCase().startsWith("POST"))
          doPostMethod();
      else
          doNotImplemented();
   }
6. Handle a GET method request (doGetMethod()):

```java
// Get requested resource
String reqPathname = getURLFromMethodLine();

// Construct the full pathname.
File file =
    constructCanonicalPathname(directory.getPath(),
                               reqPathname);
determineResponse(file);
```

7. Handle a POST method request (doPostMethod()):

```java
// Get requested/base resource
String relURL1 = getURLFromMethodLine();
// Read the header section.
readHeaderSection();

// Entity-body is a relative URL.
String relURL2 = getEntityBody();

// First construct the requested pathname from
// relURL1 and URL2.
String reqPathname;
if(relURL2.startsWith("/")) ||
    relURL2.startsWith("\""))
    reqPathname = relURL1 + relURL2;
else
    reqPathname = relURL1 + File.separator + relURL2;
// Construct a full pathname using
// the directory pathname.
File file =
    constructCanonicalPathname(directory.getPath(),
                               reqPathname);
determineResponse(file);
```
8. Construct the full pathname of the URI (constructCanonicalPathname()).

String pathname;
if(relURL.startsWith("/")) ||
    relURL.startsWith("\"))
    pathname = baseURL + relURL;
else
    pathname = baseURL + File.separator + relURL;

// Decode pathname.
pathname = URLDecoder.decode(pathname, "UTF8");
// Return the canonical pathname of the resource.
return new File(pathname).getCanonicalFile();

9. Determine if the URI identifies a file or a directory (determineResponse()).

// Check to see if requested resource doesn't start
// with specified directory
if(!file.getCanonicalPath().
    startsWith(this.directory.getCanonicalPath())) {
    sendResponseLine("HTTP/1.0 403 Forbidden");
    sendBlankLine();
} else if(!file.exists()) {
    sendResponseLine("HTTP/1.0 404 File Not Found");
    sendBlankLine();
} else if(!file.canRead()) {
    sendResponseLine("HTTP/1.0 403 Forbidden");
    sendBlankLine();
} else if(file.isDirectory()) // A directory
    sendDirResponse(socketWriter, file);
else // A file
    sendFileResponse(socketWriter, file);
10. Response for a file (sendFileResponse()):

```java
BufferedReader source = new BufferedReader(
    new InputStreamReader(new FileInputStream(file)));
// Send the status line
sendResponseLine("HTTP/1.0 200 OK");
// Send the entity header fields
sendResponseLine("Content-Type: text/html");
// Mark end of the response headers section
sendBlankLine();

// Send entity body
sendResponseLine("<html><head><title>File: " +
    file.getName() + "</title></head>");
sendResponseLine("<body><h1>File: " +
    file.getName() + "</h1>");
sendResponseLine("<pre>");
for(;;) {
    String txtLine = source.readLine();
    if (txtLine == null) break;
    sendResponseLine(txtLine);
}
sendResponseLine("</pre>");
sendResponseLine("</body></html>");
```

11. Response for a directory (sendDirResponse()):

```java
// Send the status line
sendResponseLine("HTTP/1.0 200 Okay");

// Send the entity header fields
sendResponseLine("Content-Type: text/html");

// Mark end of the response headers
sendBlankLine();

// Send entity body
sendResponseLine("<html><head><title>Directory: " +
    dir.getName() + "</title></head>");
sendResponseLine("<body><h1>Directory: " + dir.getName() + "</h1>");
File[] contents = dir.listFiles();
sendResponseLine("<table border="0">");
for(int i=0; i < contents.length; i++) {
    String row = "</a href="/">" + contents[i].getName();
    if(contents[i].isDirectory())
        row += "/";
    row += ">";
```
if(contents[i].isDirectory())
    row += "<i>";
row += contents[i].getName();
if(contents[i].isDirectory())
    row += "</i>";
row += "</a></td></tr>";
sendResponseLine(row);
}
sendResponseLine("</table></body></html>");

12. Close the socket (run()).
    socket.close();