CS 425: Computer Networking
IIT Kanpur

Project 1 Assignment

0. Due Date

This project is due on Aug 19, 2016 at 11:55 pm via moodle. The late submission policy and penalty discussed in the first class applies.

1. Introduction

The educational objective of this project is to reinforce concepts related to client-server protocols, server design, and multi-threaded design. The design and programming objectives of this project are to design, implement, and test a concurrent HyperText Transfer Protocol (HTTP) server that (almost) conforms to HTTP version 1.1 (HTTP/1.1). The specification of HTTP/1.1 is contained in RFC 2616. For this assignment you only need to be concerned with a minimal set of HTTP/1.1 features, as described briefly in Section 2.1 below and in the "Project 1 Notes" located at the end of this document. Note: currently HTTP version 2.0 is supported by most web servers and clients. But we will stick to version 1.1.

Note that Project 2 will require you to modify the HTTP server developed in Project 1. Therefore, it is particularly important that your design be modular and based on well-defined functions or objects.

2. Server Requirements

2.1. Summary of HTTP/1.1 Requirements for this Project

Your server should accept Request messages from a client (a web browser such as Microsoft Internet Explorer or Chrome or Firefox) and respond with an appropriate Response message. Requirements are as follows.

- Your server only needs to support the GET method. Clients use the GET method to retrieve files from a server. (The HEAD and POST methods are optional.)
- Your server only needs to support the absolute path (abs_path) type of requested uniform resource identifier (URI). Requested resources will be files at the server.
- Your server should support persistent connections.
- Your server should provide Content-Length and Content-Type fields in its response.
- Your server should attempt to provide appropriate Status-Code and Response-Phrase values in response to errors.

Further information is provided in the "Project 1 Notes" in the appendix of this project specification.

2.2. Additional Server Requirements

Your server should also adhere to the following requirements.

- The server program should be designed to run continuously until an unrecoverable error occurs. The server should be able to concurrently service requests from multiple clients.
- The server should define a base directory for all documents. For example, if the base directory is "c:\webfiles," a reference to "/gorp.html" should retrieve "c:\webfiles\gorp.html" and a reference to "/sub/hokie.gif" should retrieve "c:\webfiles\sub\hokie.gif." Note that a reference to a directory, including "/" should return the file "index.html" from the referenced directory. If "index.html" does not exist, the request can be treated as an error.

3. Implementation and Testing Constraints

The following constraints apply to the implementation and testing of the server program.

- The program may be developed in either C or C++.
- Object-oriented or procedural design and coding techniques can be used.

- Project must be developed on the given linux virtual machine.
- You may use socket class libraries with appropriate reference to the source in the comments sections of the code. You must always acknowledge code or libraries used. **You may not use high-level libraries or code that implement any part of the HTTP protocol.**
- All code must be clearly written, easily understandable by a knowledgeable reviewer, and neatly formatted. Code should be "self-documenting," i.e., all information needed to understand the structure and operation of your code and any data structures should be contained with the code itself.
- You may test your server on a single stand-alone host using "localhost" as the server address. If possible, test your server across a network with client and server executing on different hosts. You should be able to use any standard web browser as the client for testing.
- A set of test files is provided as a single zip file. Use the test files to verify the correct operation of your server.

4. Grading

4.1. Grade Distribution

Project grades will be based on the following factors as documented in your report and exhibited by your program.

15 points	Completeness and quality of project report
15 points	Completeness of test procedures
65 points	Correct operation of the server
5 points	Implementation of optional features (see below)

4.2. Optional Add-Ons for Full Credit

The maximum grade that you can earn by implementing just the mandatory features is 95 points out of a maximum of 100 points. Up to 5 additional points can be earned by completing optional features, as listed below. To receive credit optional features must be implemented, fully tested, and documented in the project report.

1 point	Allow the server port to be initialized at start up, for example via a command line argument or an initialization file.
1 point	Allow the document base directory to be initialized at start up, for example via a command line argument or an initialization file.
2 points	Include the Date and Server fields in the Response message header.
2 points	Log all client requests to a file. Each entry should be on one line and contain at least the following items: (i) date and time of request, (ii) client's host IP address, and (iii) the request (the URI).
2 points	Also implement the HEAD method.
3 points	Also implement the POST method.
3 points	Reply with a directory listing if a directory is the requested resource.
5 points	Reply with a hyperlinked directory listing if a directory is the requested resource. (This option supersedes the previous option.)

5. Submission Requirements

You must submit a project report and all source files as specified below.

5.1. Report Requirements

The report should include the following items in the specified order. Note that mandatory page limits are given for some items. You may be penalized for excess verbiage.

- Cover page with name of project, date, number and name of class, and student's name, student roll number, e-mail address and list of implemented options (1 page).

- List and brief discussion of any optional features that you implemented (1 page maximum). Note that you do not need to describe implementation of mandatory features.
- Testing results, including the following (2 pages maximum).
  - Test procedure used. Be sure to test all implemented functionality. Be sure to specify browser(s) used for testing.
  - Screen shots and/or other evidence showing test results, including results when using the supplied test files. Log files and other long print-outs can be included as an appendix, but must be referenced and discussed in the main body of the report.
  - Summary of test results indicating which features work or do not work properly.
- Neatly formatted source file for your server program as an appendix to your report.

5.2. Submission Logistics

You must submit the report and all source files and a Makefile with a target "all" to build the server as a single tar file using the moodle class web site. The tar file must be given a name of the following form: *LAST_FIRST*_p1.tar where *LAST* is your last or family name and *FIRST* is your first or given name. For example, student John Doe would submit the file doe_john_p1.tar. Make only one submission. If you make more than one submission, your first submission will be graded.

Files contained within the tar file should adhere to the following requirements.

- The report must be an Adobe Acrobat PDF file. The full report, including formatted source code listing, should be in a single file.
- Submit all source files for your program. These are the *.c, *.cpp, and *.h files. Do not submit executable files, object files, library files or other intermediate or supporting files. You must submit your Makefile. Makefile must have a target "all" to build the server. The server executable should be named "http-server", and the Makefile should also have a "clean" target to clean all intermediate files and the executable.

6. Honor Code

You must work alone on this project. Teams are not allowed. You should not share your code with other students or borrow code from other students. You may not discuss your design or code with anyone except the instructor or teaching assistants for this class. You may not help other students in debugging their code or have others help you. Simply stated, you may not discuss or in any way share any aspect of your original work with anyone except the instructor or teaching assistants for this class. If you use libraries or any code developed by others, its use must be properly acknowledged.

You may discuss the details of system calls with other students. You may also discuss the protocol specification and the requirements of this assignment with others. Contact the instructor if you have any questions about the honor code requirements.

7. Questions

Use the slack forum (https://cs425a.slack.com/messages) ask questions about this assignment. Do not post questions that contain specific information about the solution.

Appendix: Project 1 Notes

1. Introduction

These notes are intended to aid your understanding of the HTTP/1.1 server. The notes are not a replacement for the assignment.

Section numbers below refer to sections in RFC 2616[1]. Further information about HTTP/1.1 and related materials are available from the World Wide Web Consortium (W3C) at http://www.w3.org/.

2. Requests from the Client

The Request message sent by the client has the following specification (Section 5).

```
Request = Request-Line
          *(( general-header
           | request-header
           | entity-header ) CRLF)
          CRLF
          [ message-body ]
```

You only need to support the GET method in your server. For the GET method, there is no entity-header or message-body. There may be a general-header and a request-header. Note that the Request message is terminated by adjacent occurrences of CRLF CRLF (i.e., by the carriage return character followed by the line feed character followed by the carriage return character followed by the line feed character).

2.1. Request-Line

For the GET method, the important part of the request is the Request-Line. The Request-Line has the following specification.

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

2.1.1. Method

The server for this assignment is required to support only the GET method (Section 9.3). Implementing the HEAD method (Section 9.4) and POST method (Section 9.5) are options. Note that a fully functional HTTP/1.1 server is required to support both the GET and HEAD methods, but all other methods are optional.

An error status should be sent in the response if an unsupported method appears in the request. See the description of the Response message in Section 3 below for further information.

2.1.2. Request-URI

A URI is a Universal Resource Identifier. For the GET method, the Request-URI indicates the resource to be accessed. For this assignment, the Request-URI is a file at the server. In general, the URI could be a reference to all files, a reference to a resource on another host, or some other resource.

```
Request-URI = "*" | absoluteURI | abs_path | authority
```

Files are indicated by an absolute path (abs_path), such as “/index.html” (a specific file relative to the server’s base directory) or such as “/images/" or “/” (directories relative to the server's base directory).

2.2. General-Header and Connection Field

The general-header may contain various fields. For this assignment, only the Connection field (Section 14.10) is relevant. Its grammar is defined as follows.

```
Connection = "Connection" ":" 1#(connection-token)
connection-token = token
```

[1] R. Fielding, et al., “Hypertext Transfer Protocol – HTTP/1.1,” RFC 2616, June 1999.

The Connection field, if present, will specify one of two options (which are not case sensitive): Keep-Alive and Close.

2.2.1. Keep-Alive Option

The Keep-Alive option specifies that the server may keep the connection open after satisfying this request for possible later requests. This is the default option for HTTP/1.1, so if no Connection field is present, the server may keep the connection open. For this assignment, the server should keep the connection open if the Keep-Alive option is specified or if no Connection field is present in the general-header, thus implementing persistent connection.

2.2.2. Close Option

This option specifies that the server should close the connection after satisfying this connection. The server should also close the connection, after satisfying any pending or new requests, if the client closes its connection after sending its request.

2.3. Request Examples

2.3.1. Request from Microsoft Internet Explorer

Here's an example Request message generated by Microsoft Internet Explorerto retrieve the file "/index.html" from an HTTP server at host “bodhran.irean.vt.edu.” A carriage return, line feed pair is indicated by “[CRLF].”

```
GET /index.html HTTP/1.1[CRLF]
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-powerpoint, application/vnd.ms-excel,
application/msword, */*[CRLF]
Accept-Language: en-us[CRLF]
Accept-Encoding: gzip, deflate[CRLF]
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)[CRLF]
Host: bodhran.irean.vt.edu[CRLF]
Connection: Keep-Alive[CRLF]
[CRLF]
```

3. Response From the Server

The server replies to the client’s request with a Response message specified as follows (Section 6).

```
Response = Status-Line
           *(( general-header
            | response-header
            | entity-header ) CRLF)
           CRLF
           [ message-body ]
```

To support the GET method, the server must provide a Status-Line, some header fields, and a message-body. The message body is the returned file itself, assuming that the URI requested is valid. See Section 4 below for a discussion of how the server manages the connection after the response is sent.

3.1. Status Line

The Status-Line has the following form (Section 6.1).

```
Status-Line = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

Your server should provide at least the following Status-Code and Reason-Phrase pairs (Section 6.1.1) when appropriate. You may want to use additional codes. Note that the particular status codes that are used imply design decisions in how errors are detected and handled.

Status-Code	*Reason-Phrase*	*Comments*
200	OK	Success (Section 10.2.1)
400	Bad Request	Malformed request (Section 10.4.1)
404	Not Found	File not found (Section 10.4.5)
500	Internal Server Error	Unexpected error (Section 10.5.1)
501	Not Implemented	Unsupported method (Section 10.5.2)

3.2. General Header

The general-header portion of the Response message has the following specification (Section 4.5).

```
general-header = Cache-Control ;
               | Connection
               | Date
               | Pragma
               | Trailer
               | Transfer-Encoding
               | Upgrade
               | Via
               | Warning
```

To maintain compatibility with HTTP/1.0 clients, the Connection field should always be returned. The "Connection: Close" option should be sent if the server will close the connection after providing this response. The "Connection: Keep-Alive" option should be sent if the server will keep the connection open after providing this response. This is the default mode for HTTP/1.1 but not for HTTP/1.0, so providing the "Connection: Keep-Alive" field in the general-header is needed to let HTTP/1.0 clients know that your server will keep the connection open. The optional feature to return the date also requires use of the general-header's Date field (Section 14.18).

3.3. Response Header

The response-header portion of the Response message has the following specification (Section 6.2).

```
response-header = Accept-Ranges
                | Age
                | ETag
                | Location
                | Proxy-Authenticate
                | Retry-After
                | Server
                | Vary
                | WWW-Authenticate
```

Your server does not need to provide a response-header unless you choose to provide the Server field (Section 14.38) as an optional feature.

3.4. Entity Header

The entity-header portion of the Response message provides information about the resource or entity and has the following specification (Section 7.1).

```
entity-header = Allow ; Section 14.7
              | Content-Encoding ; Section 14.11
              | Content-Language ; Section 14.12
              | Content-Length ; Section 14.13
              | Content-Location ; Section 14.14
              | Content-MD5 ; Section 14.15
              | Content-Range ; Section 14.16
              | Content-Type ; Section 14.17
              | Expires
              | Last-Modified
              | extension-header
```

Your server needs to provide Content-Length and Content-Type fields to support the GET method.

3.4.1. Content-Length

The Content-Length field (Section 14.13) indicates the length of the file being returned as a decimal number of bytes (octets). The length value is only that of the entity-body; it does not include header bytes. The client can use this value to determine the end of the response since the connection may not be closed with persistent connections. The Content-Length field has the following form.

```
Content-Length = "Content-Length" ":" 1*DIGIT
```

Be sure to read Section 14.3 of RFC 2616 if you plan to implement the HEAD method.

3.4.2. Content-Type

The Content-Type field (Section 14.17) indicates the type of content. The Content-Type field has the following form.

```
Content-Type = "Content-Type" ":" media-type
```

Your server should support at least the following values for media-type that can be determined from the file type or file extension of the requested resource.

File Extensions	*Values for media-type*
*.html, *.htm	text/html
*.txt	text/plain
*.jpeg, *.jpg	image/jpeg
*.gif	image/gif
*.pdf	Application/pdf

For unknown types, the server is allowed to omit the Content-Type field or use the media-type application/octet-stream.

3.5. Response Examples

3.5.1. Successful Response

The following response was received from a production HTTP server for a request for “/index.html.” The notation “[CRLF]” has been added to indicate the locations of carriage return-line feed pairs in the response. In the interest of conserving space in this document, the message-body is omitted here. The message body is 10,230 bytes in length.

```
HTTP/1.0 200 OK[CRLF]
Content-Length: 10230[CRLF]
Content-Type: text/html[CRLF]
[CRLF]
Message body (the file) here
```

Given the assignment and notes above, your server should provide a similar, but slightly different response. A suitable response to the request for "/index.html" is shown below. Note that the HTTP-Version field is different and that the Connection field has been added to ensure compatibility with HTTP/1.0 clients.

```
HTTP/1.1 200 OK[CRLF]
Content-Length: 10230[CRLF]
Content-Type: text/html[CRLF]
Connection: Keep-Alive[CRLF]
[CRLF]
Message body (the file) here
```

3.5.2. Unsuccessful Response

The following response was received from a production HTTP server for a request for file "/x.html" which did not exist at the server. Note that this server provides a "File not found" error message as an HTML document. This is not required for this assignment.

```
HTTP/1.0 404 Not Found[CRLF]
Date: Sat, 25 Sep 1999 19:36:12 GMT[CRLF]
Server: NCSA/1.5[CRLF]
Content-type: text/html[CRLF]
[CRLF]
<HEAD><TITLE>404 Not Found</TITLE></HEAD>[CRLF]
<BODY><H1>404 Not Found</H1>[CRLF]
The requested URL /x.html was not found on this server.[CRLF]
</BODY>
```

4. Implementing Persistent Connections

Persistent connections provide a way for a client and server to exchange multiple requests and responses while incurring the overhead of just one TCP connection. With HTTP versions earlier than 1.1, the default mode of operation was that the server would close the connection after it sent its response. With HTTP/1.1, the default mode of operation is for the server to keep the connection open.

4.1. Closing the Connection

The server should close the connection after the occurrence of one or more of the following three events:

i) the client closes the connection, e.g., detected by recv() returning 0 or by the client resetting the connection,

ii) the client sends a general-header with a Connection field specifying the Close option, or

iii) an error is encountered.

A server is also allowed, but not required, to close a connection that has been idle for some period. This "connection time-out" feature is not required for this project. (See Section 4.1 below if you want to implement a time-out feature.)

The server must check for a Connection field with the Close option in the request. If this is the case, then the server must close the connection after it responds to the request.

The server must also check for the case where the client closes the connection. If this occurs before a full request is received, then the server should close its side of the connection. The client may close the connection immediately after sending a request to the server. In this case, the server should send the response to the client and then close the connection.

The server should also check for the case where the client resets or aborts the connection. It is observed that MSIE 6.0 resets the connection after about 90 seconds of inactivity. If this occurs, then a call to recv() will return SOCKET_ERROR and WSAGetLastError() will then return WSAECONNRESET if the connection has been reset by the client or WSAECONNABORTED if the client has aborted the

connection. The server should treat WSAECONNRESET or WSAECONNABORTED as valid conditions, not true errors. The server should close the client socket after detecting this condition.

4.2. Using select() to Time-Out a Connection

As stated above, the HTTP specification allows the server and client to terminate the connection when they decide to do so. When to terminate the connection at the server is a trade-off between performance for the specific client (needing to create a new connection for a subsequent request) and resources and performance at the server in general (since the connection ties up server resources). Time-out values must also accommodate remote clients that may experience substantial delay in sending requests to the server and receiving responses from the server.

A server can use asynchronous socket calls to “time-out” or terminate a connection after a certain period of inactivity. The following is an example of how the select() call can be used to "time-out" waiting for data from the client. Of course, the code for the HTTP server of Project 1 will need to be structured differently.

```
fd_set  fds;
int     rc;
TIMEVAL timeout;

timeout.tv_sec = 30;  // set timeout to 30 secs
timeout.tv_usec = 0;  // and 0 microsecs

FD_ZERO(&fds);        // clear descriptor list
FD_SET(sock, &fds);   // include our client socket

rc = select(FD_SETSIZE,&fds, (fd_set *)NULL,(fd_set *)NULL,&timeout);
if(rc == SOCKET_ERROR)
   HANDLE A SOCKET ERROR
else if(rc == 0)
   HANDLE A TIMEOUT
else if(rc == 1)
   CALL recv() TO GET DATA FROM CLIENT
```

5. Other Comments

Socket testing tools such as https://sourceforge.net/projects/sockettest/ can be useful in testing your server. It can be used as a client to send a request to a standard HTTP server to monitor the response or it can be used as a server to examine the format of a request. This is how the request and response examples above were collected. It can also be used as a client to test your server.