CS 425: Computer Networking
IIT Kanpur
Project 2 Assignment

0. Due Date
This project is due on Aug 31, 2016 at 11:55 pm via moodle. The late submission policy and penalty discussed in the first class applies.

1. Introduction

The Hypertext Transfer Protocol (HTTP) is the protocol used for communication on this web: it defines how your web browser requests resources from a web server and how the server responds. For simplicity, in this project, we will be dealing only with version 1.0 of the HTTP protocol, defined in detail in RFC 1945. You may refer to that RFC while completing this assignment, but our instructions should be self-contained. In Project 1, you are already introduced to HTTP protocol, and implemented a server. This project will take you a bit further by construction of an HTTP proxy server. For the sake of completeness, we will discuss HTTP protocol first.

HTTP communications happen in the form of transactions; a transaction consists of a client sending a request to a server and then reading the response. Request and response messages share a common basic format:

- An initial line (a request or response line, as defined below)
- Zero or more header lines
- A blank line (CRLF)
- An optional message body.

The initial line and header lines are each followed by a "carriage-return line-feed" (\r\n) signifying the end-of-line.

For most common HTTP transactions, the protocol boils down to a relatively simple series of steps (important sections of RFC 1945 are in parenthesis):

1. A client creates a connection to the server.
2. The client issues a request by sending a line of text to the server. This **request line** consists of a HTTP *method* (most often GET, but POST, PUT, and others are possible), a *request URI* (like a URL), and the protocol version that the client wants to use (HTTP/1.0). The request line is followed by one or more header lines. The message body of the initial request is typically empty. (5.1-5.2, 8.1-8.3, 10, D.1)
3. The server sends a response message, with its initial line consisting of a **status line**, indicating if the request was successful. The status line consists of the HTTP version (HTTP/1.0), a *response status code* (a numerical value that indicates whether or not the request was completed successfully), and a *reason phrase*, an English-language message providing description of the status code. Just as with the the request message, there can be as many or as few header fields in the response as the server wants to return. Following the CRLF field separator, the message body contains the data requested by the client in the event of a successful request. (6.1-6.2, 9.1-9.5, 10)

4. Once the server has returned the response to the client, it closes the connection.

It's fairly easy to see this process in action without using a web browser. From a Unix prompt, type:

```
telnet www.yahoo.com 80
```

This opens a TCP connection to the server at www.yahoo.com listening on port 80 (the default HTTP port). You should see something like this:

```
Trying 69.147.125.65...
Connected to any-fp.wa1.b.yahoo.com.
Escape character is '^]'.
```

type the following:

```
GET http://www.yahoo.com/ HTTP/1.0
```

and hit enter twice. You should see something like the following:

```
HTTP/1.0 200 OK
Date: Tue, 16 Feb 2010 19:21:24 GMT
(More HTTP headers...)
Content-Type: text/html; charset=utf-8

<html><head>
<title>Yahoo!</title>
(More HTML follows)
```

There may be some additional pieces of header information as well- setting cookies, instructions to the browser or proxy on caching behavior, etc. What you are seeing is exactly what your web browser sees when it goes to the Yahoo home page: the HTTP status line, the header fields, and finally the HTTP message body- consisting of the HTML that your browser interprets to create a web page. You may notice here that the server responds with HTTP 1.1 even though you requested 1.0. Some web servers refuse to serve HTTP 1.0 content.

1. 1 HTTP Proxies

Ordinarily, HTTP is a client-server protocol. The client (usually your web browser) communicates directly with the server (the web server software). However, in some circumstances it may be useful to introduce an intermediate entity called a proxy. Conceptually, the proxy sits between the client and the server. In the simplest case, instead of sending requests directly to the server the client sends all its requests to the proxy. The proxy then opens a connection to the server, and passes on the client's request. The proxy receives the reply from the server, and then sends that reply back to the client. Notice that the proxy is essentially acting like both a HTTP client (to the remote server) and a HTTP server (to the initial client).

Why use a proxy? There are a few possible reasons:

- **Performance:** By saving a copy of the pages that it fetches, a proxy can reduce the need to create connections to remote servers. This can reduce the overall delay involved in retrieving a page, particularly if a server is remote or under heavy load.
- **Content Filtering and Transformation:** While in the simplest case the proxy merely fetches a resource without inspecting it, there is nothing that says that a proxy is limited to blindly fetching and serving files. The proxy can inspect the requested URL and selectively block access to certain domains, reformat web pages (for instances, by stripping out images to make a page easier to display on a handheld or other limited-resource client), or perform other transformations and filtering.
- **Privacy:** Normally, web servers log all incoming requests for resources. This information typically includes at least the IP address of the client, the browser or other client program that they are using (called the User-Agent), the date and time, and the requested file. If a client does not wish to have this personally identifiable information recorded, routing HTTP requests through a proxy is one solution. All requests coming from clients using the same proxy appear to come from the IP address and User-Agent of the proxy itself, rather than the individual clients. If a number of clients use the same proxy (say, an entire business or university), it becomes much harder to link a particular HTTP transaction to a single computer or individual.

References:

- RFC 1945 The Hypertext Transfer Protocol, version 1.0

2. Project Details

2.1 The Basics

Your task is to build a web proxy capable of accepting HTTP requests, forwarding requests to remote (origin) servers, and returning response data to a client. The proxy MUST handle **concurrent** requests by forking a process for each new client request using the fork() system call. You will only be responsible for implementing the GET method. All other request methods received by the proxy should elicit a "Not Implemented" (501) error (see RFC 1945 section 9.5 - Server Error).

This assignment can be completed in either C or C++. It should compile and run (using g++) without errors or warnings, producing a binary called proxy that takes as its first argument a port to listen from. Don't use a hard-coded port number.

You shouldn't assume that your server will be running on a particular IP address, or that clients will be coming from a pre-determined IP.

2.2 Listening

When your proxy starts, the first thing that it will need to do is establish a socket connection that it can use to listen for incoming connections. Your proxy should listen on the port specified from the command line and wait for incoming client connections. Each new client request is accepted, and a new process is spawned using fork() to handle the request. To avoid overwhelming your server, you should not create more than a reasonable number of child

processes (for this experiment, use at most 20), in which case your server should wait until one of its ongoing child processes exits before forking a new one to handle the new request.

Once a client has connected, the proxy should read data from the client and then check for a properly-formatted HTTP request. Your project 1 server solution code could be reused for this or you can use the parsing library provided with the skeleton code. Specifically, you will use our libraries to ensure that the proxy receives a request that contains a valid request line:

<METHOD> <URL> <HTTP VERSION>
All other headers just need to be properly formatted:
<HEADER NAME>: <HEADER VALUE>
In this assignment, client requests to the proxy must be in their absolute URI form (see RFC 1945, Section 5.1.2), e.g.,
GET http://www.cs.princeton.edu/index.html HTTP/1.0
Your browser will send absolute URI if properly configured to explicitly use a proxy (as opposed to a transparent on-path proxies that some ISPs deploy, unbeknownst to their users). On the other form, your proxy should issue requests to the webserver properly specifying *relative* URLs, e.g.,
GET /index.html HTTP/1.0
Host: www.cs.princeton.edu
An invalid request from the client should be answered with an appropriate error code, i.e. "Bad Request" (400) or "Not Implemented" (501) for valid HTTP methods other than GET. Similarly, if headers are not properly formatted for parsing, your proxy should also generate a type-400 message.

2.2 Parsing Library

The skeleton code provided has a parsing library to do string parsing on the header of the request. This library is in proxy_parse.[c|h] in the skeleton code. The library can parse the request into a structure called ParsedRequest which has fields for things like the host name (domain name) and the port. It also parses the custom headers into a set of ParsedHeader structs which each contain a key for the header field name and value corresponding to the value to which the header is set. You can search for headers by the key or header field name and modify them. The library can also recompile the headers into a string given the information in the structs.

More details as well as an example of how to use the library is included in the library file, proxy_parse.h. This library can also be used to verify that the headers are in the correct format since the parsing functions return error codes if this is not the case.

2. 4 Parsing the URL

Once the proxy receives a valid HTTP request, it will need to parse the requested URL. The proxy needs at least three pieces of information: the requested host, port, and path. See the URL (7) manual page for more info. You will need to parse the absolute URL specified in the given request line. **You can use the parsing library to help you**. If the hostname indicated in the absolute URL does not have a port specified, you should use the default HTTP port 80.

2.5 Getting Data from the Remote Server

Once the proxy has parsed the URL, it can make a connection to the requested host (using the appropriate remote port, or the default of 80 if none is specified) and send the HTTP request for the appropriate resource. The proxy should always send the request in the relative URL + Host header format regardless of how the request was received from the client:

Accept from client:

GET http://www.princeton.edu/ HTTP/1.0

Send to remote server:

GET / HTTP/1.0
Host: www.princeton.edu
Connection: close
(Additional client specified headers, if any...)

Note that we always send HTTP/1.0 flags and a Connection: close header to the server, so that it will close the connection after its response is fully transmitted, as opposed to keeping open a persistent connection. So while you should pass the client headers you receive on to the server, you should make sure you replace any Connection header received from the client with one specifying close. To add new headers or modify existing ones, use the HTTP Request Parsing Library provided.

2.6 Returning Data to the Client

After the response from the remote server is received, the proxy should send the response message as-is to the client via the appropriate socket. To be strict, the proxy would be required to ensure a Connection: close is present in the server's response to let the client decide if it should close its end of the connection after receiving the response. However, checking this is not required in this project for the following reasons. First, a well-behaving server would respond with a Connection: close anyway given that we ensure that we sent the server a close token. Second, we configure Firefox to always send a Connection: close by setting keepalive to false. Finally, it simplifies the assignment so you wouldn't have to parse the server response.

The following summarizes how status replies should be sent from the proxy to the client:

1. For any error your proxy should return the status 500 'Internal Error'. This means for any request method other than GET, your proxy should return the status 500 'Internal Error' rather than 501 'Not Implemented'. Likewise, for any invalid, incorrectly formed headers or requests, your proxy should return the status 500 'Internal Error' rather than 400 'Bad Request' to the client. For any error that your proxy has in processing a request such as failed memory allocation or missing files, your proxy should also return the status 500 'Internal Error'. (This is what is done by default in this case.)
2. Your proxy should simply forward status replies from the remote server to the client. This means most 1xx, 2xx, 3xx, 4xx, and 5xx status replies should go directly from the remote server to the client through your proxy. Most often this should be the status 200 'OK'. However, it may also be the status 404 'Not Found' from the remote server. (While you are debugging, make sure you are getting valid 404 status replies from the remote server and not the result of poorly forwarded requests from your proxy.)

2.7 Testing Your Proxy

Run your client with the following command:

./proxy <port>, where port is the port number that the proxy should listen on. As a basic test of functionality, try requesting a page using telnet:

```
telnet localhost <port>
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
GET http://www.google.com/ HTTP/1.0
```

If your proxy is working correctly, the headers and HTML of the Google homepage should be displayed on your terminal screen. Notice here that we request the absolute URL (http://www.google.com/) instead of just the relative URL (/). A good sanity check of proxy behavior would be to compare the HTTP response (headers and body) obtained via your proxy with the response from a direct telnet connection to the remote server. Additionally, try requesting a page using telnet concurrently from two different shells.

For a slightly more complex test, you can configure Firefox to use your proxy server as its web proxy as follows:

1. Go to the 'Edit' menu.
2. Select 'Preferences'. Select 'Advanced' and then select 'Network'.
3. Under 'Connection', select 'Settings...'.
4. Select 'Manual Proxy Configuration'. If you are using localhost, remove the default 'No Proxy for: localhost 127.0.0.1". Enter the hostname and port where your proxy program is running.
5. Save your changes by selecting 'OK' in the connection tab and then select 'Close' in the preferences tab.

2.8 Socket and Multi-Process Programming

By now through assignment 1, and Project 1 you should be familiar with socket programming.

In addition to the sockets library, there are some functions you will need to use for creating and managing multiple processes: fork and waitpid.

References:

- Guide to Network Programming Using Sockets
- HTTP Made Really Easy- A Practical Guide to Writing Clients and Servers
- Wikipedia page on fork()

3. Submission and Grading

3.1. 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
70 points	Correct operation of the Proxy

Your proxy will be graded out of 70 points, with the following criteria:

1. When running make on your assignment, it should compile without errors or warnings on the course VM and produce a binary named proxy. The first command line argument should be the port that the proxy will listen from.
2. Your proxy should run silently: any status messages or diagnostic output should be off by default.
3. You can complete the assignment in either C or C++.
4. Your proxy should work with Firefox.
5. We'll first check that your proxy works correctly with a small number of major web pages, using the same scripts given to you to test your proxy. If your proxy passes all of these 'public' tests, you will get 35 of the possible points.
6. We'll then check a number of additional URLs and transactions that you will not know in advance. If your proxy passes **all** of these tests, you get 25 additional points. These tests will check the overall robustness of your proxy, and how you handle certain edge cases. This may include sending your proxy incorrectly formed HTTP requests, large transfers, etc.
7. Well written code will get 10 additional points -- for readability, error checking, and comments

4. Submission Requirements

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

4.1. 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 design choices you might have made (1 page maximum). Note that you do not need to describe implementation but only any design choices you made.
- 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.

4.2. Submission Logistics

You should submit your completed proxy by the due date via moodle page for this project. You will need to submit a tarball file containing the following:

- All of the source code for your proxy
- A Makefile that builds your proxy
- A Project Report describing your code and the design decisions that you made.

Your tarball should be named project2.tar. The sample Makefile in the skeleton zip file provided with this project specification will make this tarball for you with the make tar command.

5. 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.

6. 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.

7. FAQ

7.1 Address in use Errors

Q) Why do I keep getting the error "Address already in use" when I try to run my proxy server?

> **A)** This error typically means that there is already a socket bound to the same port that you are attempting to bind to. If you're testing your proxy on one of the cluster machines, this may just mean that another student is running a proxy on that port already. Otherwise, it could mean that a proxy shutdown without properly closing the port that it was running on; if you bind to, say, port 8000 and then kill your proxy without closing the port, it may take a few minutes for the operating system to notice that the port is no longer in use and make it available again. Try running your proxy again, binding to a different port (this is one reason why running with hard-coded port numbers is a bad idea).

If your proxy is looping forever to serve requests and being stopped with Ctrl-C, you might want to try registering a handler for the SIGINT signal to clean up after your proxy- it can greatly reduce instances of your favorite port number becoming 'stuck'. Take a look at the signal (2) man page for details. Another option is to set the SO_REUSEADDR option to allow the proxy to re-use an address that is already in use; look at the setsockopt man page for more information

7.2 Broken Images

Q) My proxy seems to work for web pages, but all of the images on the page are coming up broken. What's going on?

A) Remember that your proxy has to handle both text data- like web pages and HTTP requests- and binary data- like image files and downloads. You can easily run into problems if you try to use functions likefputs to send data to the client- it will work correctly for text data, but will likely fail to work correctly for binary data. Use the read/write fread/fwrite functions instead. Another possible cause for this problem could be that your proxy is running in a single-threaded mode, and it is missing subsequent HTTP requests while it is serving the request for the main page.

7.3 HTTP/1.1 Requests

Q) How should I handle requests from a client that uses HTTP/1.1 instead of HTTP/1.0?

A) Unless they are trying to use a method that isn't supported by HTTP/1.0, you should go ahead and process their request to the best of your ability. You're not required to support keeping the connection alive for HTTP/1.1 clients- they are required by the RFC to be able to handle their connection being closed by either end at any time. So parse their request, send out a HTTP/1.0 request to the remote server, and return their data using HTTP/1.0 as normal. You must return error 500 (Internal Server Error) for valid HTTP/1.1 methods that are not covered by RFC 1945.

7.4 Bad Requests

Q) What error should I return if I get a request from the client that I can't parse?

A) Again, you should send status code 500 for any malformed request that you are unable to process.

7.5 Internal Error (500)?

Q) When should I send status code 500 (Internal Error)?

A) We have greatly simplified the assignment. For any error that your proxy has in processing a client's request, including an incorrectly formed request, reply with the 500 status code.

7.6 Request/Response Line length?

Q) What is the maximum length for a request or response line that I have to support?

> **A)** The RFC does not specify a maximum value for these elements; you should not simply set a fixed value and then fail for larger requests. At the same time, you don't want to grow the buffer to an unbounded size; doing so sets up your proxy for an easy denial-of-service attack. Instead, you should start with your buffer at a small size, and then grow the buffer to some reasonable limit- say 8-16 KB. Document your decision in your assignment's README file. Take a look at the realloc man page for information on dynamically growing a buffer.

7.7 Checking Headers

Q) What sort of checking of HTTP headers in the request message needs to be performed?

> **A)** You need to check that the headers are correctly formatted (e.g.- that the delimiter is correct), but you do not need to do any checking of the header field values. The one exception is the Host header when the request line contains a relative URL that does not include the hostname. In this case, you should parse the Host header to extract the hostname value. Otherwise, say you are given a Date header by the client, you should check that the header line is properly delimited and terminated, but you do not need to check to make sure that the value of the field is a legal date.

7.8 Checking Server Replies

Q) What sort of checking should I do for the reply from the web server?

> **A)** You are not **required** to do any checking of the messages returned from the server. While you're free to implement whatever checking you feel is necessary, you should be very lenient about handing data back from the server. In particular, while you're free to do any fix up that you think would be a good idea for data returned from the server (such as eliminating extra whitespace in the response line, correcting partial line terminators, etc.), you shouldn't discard data intended for the client just because you think that the server is not replying in the correct way. It is preferable to let the client decide how to handle odd responses rather than denying it data from misbehaving servers. The client application can then decide when to make due with the response as-is, and when it is necessary to re-issue the request or discard the response data.

7.9 External Libraries

Q) Can we use any external libraries, or other third-party code?

> **A)** For this assignment, the **only** additional library that you can use (outside of ANSI C) is the ustr string library (details here). However, we'd prefer if you just use the built-in C++ string libraries instead. You can't use any embedded scripting languages, free socket or HTTP libraries, etc., in your solution. Using ustr is optional and discouraged.

7.10 Test Script Errors

Q) On some sites (such as Google), the test script says that I failed the test and displays the difference between the proxy data and the direct server data as consisting of a single line containing some sort of header information. What's wrong?

> **A)** In all likelihood, nothing. If the results differ by a single line that contains something like a Date or Etag header, then the difference is almost certainly due to the proxy request and the direct transfer request being directed to two different servers in a load-balanced cluster. In other words, your proxy is working correctly, but the data it retrieved is being compared to data from another machine that is also tasked with handling requests for that URL. The result is that there may be minor differences in the data- the specific machine that handles the request may insert its IP or hostname into the document header, there may be clock drift between the two servers, etc. Look at the returned data; if there is only a single line or two difference, and they consist of machine-dependent information (like cashing instructions or date/time fields), then you are likely passing the test, and your TAs will treat it as such.

7.11 An aside on the details of load balancing

Almost all large commercial websites implement some sort of load-balancing or distribution scheme in order to cope with large volumes of traffic and ensure availability in the face of hardware failure. When you send a request to a website like Google, rather than being directed to a single specific server your request is sent to one of several (possibly several thousand!) machines selected by some combination of software and hardware that is looking at inbound requests. One strategy is to keep track of the load on the cluster of servers, and then direct the next incoming request to the machine with the lowest load. Simpler schemes (like round robin balancing) just direct traffic to each server in the cluster in sequence without paying attention to existing load information.

The result is that two requests to the same URL, sent very close together, may be ultimately serviced by two different machines with different hardware and software configurations! In principle, this sort of load distribution is meant to be entirely transparent to the user; all the servers should return the exact same results and the same data. In practice, this is often not exactly the case. One cause of differences can be errors in the cluster setup. If a particular machine if misconfigured, or not properly updated, it may reply to requests with out-of-date data. A machine with a clock that drifts out of sync quickly may return timestamps that are very different from its siblings in the cluster.

Other differences are intentional; in order to debug problems with individual machines in the cluster, the machines may be configured to put an identifying marker in the headers or data of messages that they return, to ease debugging. Finally, there can sometimes be transient errors that occur as the cluster is updated; in order to improve redundancy, clustered machines may cache local copies of data rather than reading it from a single shared storage medium. Updates to the cached copy can either be pushed out to the cluster machines, or pulled off of a central server according to some schedule. It is possible to hit two different machines in a cluster while they are in the process of updating, in which case one machine will return the 'new' data, and another machine will still be serving the older, cached copy. Such windows are hopefully quite small, but can occasionally become visible to users who are engaging in unusual activities (such

as making repeated, rapid web queries to a single URL in order to test the web proxy that they wrote!).

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.