# 15-441 Networks - Project 1
# Mini Web Server

Project 1 Lead TA: Justin Weisz (jweisz@andrew.cmu.edu)

Assigned: Wednesday, September 4th, 2002
Due Date: Monday, September 23rd, 2002

## 1 Abstract

In this assignment, you are going to write a web server. This web server will be single threaded and non-blocking, and will handle static content and secure connections using SSL. This assignment is to be done individually, and the work you submit must be your own. See the collaboration policy on the course web site for more information. Please note that this is a long handout, and should be read over carefully.

### 1.1 Suggested Checkpoints and Deadlines

Below is a table of *suggested* checkpoints. While it is not required that you meet all of the requirements by the specified dates, you should keep these deadlines in mind to gauge your progress. This assignment is due on **Monday, September 23rd, by 11:59 PM**. The late policy is explained on the course web site.

| Date | Description |
|---|---|
| September 4 | Assignment handed out. **PLEASE START EARLY!** |
| September 7 | Your server should be able to accept connections with a client and exchange data. |
| September 17 | You should have a working HTTP server at this point. This is the bulk of the assignment. |
| September 21 | Your server should be able to support SSL connections. |
| September 23 | Assignment due date. You will be turning in all of your source code, as well as a Makefile, and a tests.txt file explaining how you tested your program. The handin directory is: `/afs/cs.cmu.edu/academic/class/15441-f02-users/group-XX` Remember that the assignment is due before 11:59PM! |

## 2 Where to get help

If you have a question, please do not hesitate to ask us for help, that's why we're here! General questions should be posted to the class bulletin board, **academic.cs.15-441**. If you have more specific questions (especially ones that require us looking at your code), please drop by our office hours. Office hours are listed below:

| Professors | Office | Hours |
|---|---|---|
| Mor Harchol-Balter | Wean 8119 | Tuesday 2-3p |
| Srini Seshan | Wean 8212 | Thursday 3-4p |

| TAs | | |
|---|---|---|
| Justin Weisz | Wean 3604 | Tuesday 5-6p |
| Xavier Appe | Wean 3108 | Friday 2-3p |
| Umair Shah | Wean 3710 | Wednesday 2-3p |
| Julio Lopez | Wean 8205 | Monday 2:30-3:30p |

# 3 Introduction

During the course of this project, you will do the following:

- Learn and implement a small subset of the Hypertext Transfer Protocol (HTTP)
- Become familiar with socket programming
- Write a web server
- Get very good at debugging and testing network programs
- Learn how to use OpenSSL to write secure network applications

# 4 Project specification

## 4.1 Background

An HTTP web server is a program that listens for incoming TCP connections (typically, but not always, on port 80). Client requests and server replies flow on each connection in a mixture of human-readable text and binary data until either the client or the server decides to close the connection. The HTTP protocol has many optional features that allow an implementation to be bewilderingly complicated and bug-ridden.

In this project, you will be implementing an HTTP web server, named `mws-server`. The following resources may prove to be useful in the design and implementation of your project:

1. **HTTP Made Really Easy**
   `http://www.jmarshall.com/easy/http/`
   *Very good introduction to the HTTP protocol. Provides a lot more detail than what we require in this assignment.*

2. **BSD Sockets Programming**
   `http://world.std.com/~jimf/papers/sockets/sockets.html`
   *Gives some code examples for creating sockets and writing to them.*

3. **Sockets FAQ**
   `http://www.developerweb.net/sock-faq/`
   *Huge FAQ providing many questions and answers to socket programming. Search here if you have specific socket programming problems.*

4. **RFC 2068 — Hypertext Transfer Protocol – HTTP/1.1**
   `http://www.cs.cmu.edu/afs/cs/archive/internet-rfc/rfc2068.txt`
   *The definitive HTTP/1.1 specification. This is a very long document, and we do not expect you to read all of it. RFCs are written in a style that you may find unfamiliar, so it is wise for you to become comfortable reading them. If you are unsure of how your web server is supposed to behave in a certain situation, this is the place to find an answer.*

5. **RFC 1630 — Universal Resource Identifiers in WWW**
   `http://www.cs.cmu.edu/afs/cs/archive/internet-rfc/rfc1630.txt`
   *This RFC gives a very detailed breakdown of URLs and URIs. ASCII character encoding is also discussed in this RFC. As with RFC 2068, we do not expect you to read this entire document.*

6. **Lecture 3 Notes**
   `http://www.cs.cmu.edu/~srini/15-441/F02/lectures/lec3.ps`
   *These are the notes from lecture 3. They provide a very easy introduction to socket programming. If you are new to socket programming, read these notes over before you write any actual code.*

7. **Introduction to OpenSSL, Parts 1 and 2**
   `http://www.cs.cmu.edu/~srini/15-441/F02/Projects/lab01/reference/part1.pdf` & `part2.pdf`
   *Part one provides a good, quick tutorial on how to write a simple secure client and server. An overview on the entire SSL connection process is also provided. Part two is listed solely as a reference, and contains information on more advanced SSL topics, such as session resumption, client authentication, and re-handshaking. Read part one before you start the SSL component of this project, and save part two for when you have free time.*

8. **SSL and TLS, Chapter 9 — HTTP over SSL**
   `http://www.rtfm.com/sslbook/chap9-sample.pdf`
   *Provides a great introduction to HTTP from a security perspective. Read this before you start the assignment.*

9. **Google Groups**
   `http://groups.google.com/`
   *This is Google's archive of USENET postings. If you have a programming question (i.e. how do I do _____ in C?), or if you can't figure out what a certain function does, this is a great place to find answers.*

10. **The Class Bulletin Board**
    `academic.cs.15-441`
    *We fully encourage you to read and post questions to the class bboard. The TAs will be reading the bboard daily, and this is the best place to get quick answers to short questions. If your question involves a lot of code, or if it will take more one minute to answer, please come to our office hours instead.*

## 4.2 Programming Guidelines

Your web server must be written in the C programming language. You are not allowed to use any custom socket classes or libraries, except for libssl. C++ submissions will not be accepted. You can

use any development environment you like, but please note that we will be compiling and testing your programs on Andrew Linux machines (i.e. linux.andrew.cmu.edu), so you are responsible for making sure your program compiles and runs correctly on these machines. We recommend using `gcc` to compile your program and `gdb` to debug it. You should also use the `-Wall` flag when compiling to generate full warnings and to help debug.

For this project, you will also be responsible for turning in a GNU Make (`gmake`) compatible Makefile. Everything you need to know (and much much more) about `gmake` can be found at `http://www.gnu.org/manual/make/html_mono/make.html`. The Makefile for this assignment should be very simple.

### 4.3 The web server: mws-server

This is the main component of the project. Your server must be HTTP/1.1 compliant (see RFC 2068 in section 4.1), in that it must support GET and HEAD requests, as well as persistent connections. Note that persistent connections are not listed the HTTP/1.0 specification.

#### 4.3.1 Invocation

Your web server program must accept the following command line options, in any arbitrary order. If an argument is left out when running the server, then use the specified default.

- `-p` *port*: Overrides the default port the server should run on. The default is port 8000 (note that ports under 1024 require root access to use, so we cannot use port 80 as default).
- `-root` *path*: Sets the root directory of the web server. This is where the files are served from. For example, if the server is run with `-root ./networks/website`, then a request for `http://www.myserver.com/index.html` will result in serving the file `./networks/website/index.html`. If this option is not present, than use `./www` as the default root directory.
- `-sslport`: Listens for an SSL connection on the specified port. Your server should be able to listen on a normal port and on an SSL port at the same time, and handle both types of connections. The default SSL port is 8001.
- `-debug`: When this option is present, you may print out debugging messages to the console (stdout). When this option is not present, your program may not print **any** output to the console.

#### 4.3.2 HTTP Operations

HTTP is a fairly simple protocol. Clients send the server a request, and the server processes that request and sends back a response. The format of an HTTP request is `<method><request-uri><http-version>`, followed by an arbitrary number of additional header lines, and terminated with two CRLFs (\r\n\r\n); one on the last header line, and one by itself on the next line. HTTP/1.1 defines seven methods: GET, HEAD, POST, PUT, DELETE, TRACE, and OPTIONS. You will only need to support the first two, GET and HEAD. The request URI field is the path to the object the client wants (RFC 1630 has more information about the format of a URI). Finally, the http version field tells the server what version of HTTP the client supports, and will be `HTTP/1.1` in your project. Note that some older web browsers may send `HTTP/1.0`, but you shouldn't have to worry about this in your server.

The format of an HTTP response is `<http-version><status-code><reason-phrase>`, followed by an arbitrary number of additional header lines, and terminated with two CRLFs; one on the last header line, and one by itself on the next line. The HTTP version is the same as above. The status code is an integer representing the action the server took, or an error code if something went wrong. For example, you should be familiar with the ubiquitous 404 status code, meaning a file was not found. Section 6.1.1 of RFC 2068 gives a complete listing of status codes. You will only need to implement a small subset of these, namely, 200, 400, 404, and 501. You may also want to implement 500, since it may help with debugging. Finally, the reason phrase is a piece of English text describing the status code. For a status code of 200, the reason phrase will be "OK". RFC 2068 gives the reason phrases for all of the status codes.

Below is more information on the two HTTP methods you need to support.

1. GET

   The simplest HTTP operation of all is GET. This is a simple two-message exchange. The client initiates it by sending a GET message to the server. The message identifies the object the client is requesting with a Uniform Resource Identifier (URI, see RFC 1630). If the server can return the requested object, it indicates success with an appropriate response (in this case a 200 OK status code along with the requested object). If the server cannot return the requested object (or chooses not to), then it can return any number of other status codes.

2. HEAD

   The HEAD operation is just like a GET operation, except that the server does not return the actual object requested, just the HTTP headers of the response. HEAD is just a short form of "header". Clients usually use a HEAD message when they want to verify that an object exists, but don't need to actually retrieve the object. Programs that verify links in web pages and cache servers also use the HEAD operation.

A GET request from the web client will look like this:

```
GET /index.html HTTP/1.1
Host: unix47.andrew.cmu.edu:1234
User-Agent: mini-webclient/1.0 (Unix)
Connection: close
```

Assuming this request is successful, the headers for the response from your web server will look like this:

```
HTTP/1.1 200 OK
Date: Mon, 02 Sep 2002 15:31:31 GMT
Server: mini-webserver/1.0 (Unix)
Content-Length: 94
Connection: close
Content-Type: text/html
```

Note that these are just sample headers. A real web browser may emit more headers, such as "Accept-Encoding" or "Accept-Charset", which you will not have to process. Also note that the time is in GMT, and your server should send all timestamps in GMT. You can do this using the `gmtime()` and `strftime()` functions.

### 4.3.3 Persistent connections

Obtaining an HTML document typically involves several HTTP requests to the web server (to fetch embedded images, etc.). In HTTP/1.0 and earlier, TCP connections were closed after each request and response, so each resource to be retrieved required its own connection. Opening and closing TCP connections, however, takes a substantial amount of CPU time, bandwidth and memory. In practice, most web pages consist of several files on the same server, so much can be saved by allowing several requests and responses to be sent through a single persistent connection.

HTTP/1.1 enables browsers to send several HTTP requests to the server on a single TCP connection. In anticipation of receiving further requests, the server keeps the connection open for a configurable interval after receiving a request. This method amortizes the overhead of establishing a TCP connection over multiple HTTP requests, and it allows the client to send several requests in series (called pipelining). Moreover, sending multiple server responses on a single TCP connection in short succession avoids multiple TCP slow-starts, thus increasing network utilization and effective bandwidth perceived by the client. For more information on persistent connections in HTTP/1.1, read section 8.1 of RFC 2068.

If an HTTP/1.1 client sends multiple requests through a single connection, the server MUST keep the connection open and send responses back in the same order as the requests. If a request includes the "Connection: close" header, then that request is the final one for the connection and the server should close the connection after sending the response. Also, the server should close an idle connection after some timeout period (can be anything, but yours should be 15 seconds).

Your server must support these persistent connections. *Please remember that a single client may issue additional requests while your server is still reading and the first request. In this case, your server must read in and process all requests before closing the connection.*

### 4.3.4 Multiple connections

A web server that accepts only one connection at a time is probably impractical and definitely not very useful. As such, your server should also be written to accept multiple connections (usually from multiple hosts). It should be able to simultaneously listen for incoming connections, as well as keep reading from the connections which are already open. Note that today's web browsers may open two connections to your server (as per RFC 2068), so your server should be able to handle these multiple connections.

Many of the functions you will use for multiple connections (e.g. `accept()` and `read()`) will block when called; they go to sleep and stall program execution until some data arrives. So, if your server is blocking on `accept()`, it cannot `read()` data at the same time. This could lead to starvation of certain clients, which means that they never get served. The reason for this behavior is because the sockets created with `socket()` are initially set to be *blocking* sockets.

Fortunately, there is a function call which lets you turn a blocking socket into a *non-blocking* socket. A `read()` call on a non-blocking socket will always return immediately with the data in the socket buffer, even if there was no data available. However, *in most cases*, putting your program on a busy-wait loop looking for data on a non-blocking socket consumes enormous amounts of CPU time and is a bad idea. Instead, the `select()` function gives you the power to monitor several sockets at the same time, blocking until there is data to be dealt with. It will tell you which sockets are ready for reading, writing and accepting, and which have raised exceptions (if you really want to know). For non-SSL connections, you will want to use the `select()` function with blocking sockets.

However, in order to use `select()` with SSL connections (sockets on which you have called

`SSL_accept()`), you must set those sockets to be non-blocking. This can be done by calling `fcntl(sockfd,F_SETFL,O_NONBLOCK)`. The reason for this is because of the way SSL chunks data into records. An SSL socket may have some data in the buffer, but not enough for an entire record. Since the socket buffer has some data in it, `select()` will tell you that there is data waiting on the socket. However, when you actually try to read this data, the read call will block, because SSL has to wait for the rest of the data to fill the entire record. But, the purpose of `select()` is to prevent your read calls from blocking! So, in order to prevent your server from blocking, **your SSL sockets must be non-blocking, and your non-SSL sockets must be blocking**. See section 4.3.5 for more information.

Your server must deal with multiple connections. In this project, you must use the `select()` call to implement multiple connections in your server. *Please refer to the lecture 3 notes to see how to use the select() call, as well as the man pages.*

#### 4.3.5 SSL connections

If you've ever made a purchase online, you know the importance of having a layer of security between your application and the network. Since Ethernet is a broadcast-based protocol, and most networks today are based on Ethernet, the packets your computer send out have a good chance of being sniffed by malicious users on the network. So, the last thing you want to do is have someone else steal your credit card number because your web browser sent it over the network in a human readable form.

This is where SSL comes in. SSL provides a layer of security between your application and the network, by chunking data into records, and then encrypting those records. While you do not need to know the specifics of the encryption methods used, you should be familiar with how an SSL connection is created. First and foremost, the server needs to have a public key certificate and corresponding private key. When a client initiates an SSL connection to the server, the server sends its certificate to the client. The client then generates a random encryption key and encrypts it using the server's public key, and sends it back to the server. Then, the server decodes the client's message with it's own private key, and now the client and server both possess the secret encryption key, which they use for all future communications.

For this project, we will be using OpenSSL 0.9.6d, which is installed on the Andrew Linux servers. Included with this project is a sample SSL echo client and server, which are documented in section 5. The SSL code from that example can be easily adopted for use in your server. Another good SSL reference is the Introduction to SSL, which is listed in section 4.1.

Your server must handle SSL connections. Since you will need to use a certificate with your server, we have created a self-signed server certificate specifically for this assignment. You can get it, along with the SSL echo client and server source, from `http://www.cs.cmu.edu/~srini/15-441/F02/Projects/lab01/ssl_echo/`. The certificate file you will need for your server is named "server.pem". Keep this file in the same directory as your web server.

One final note with SSL connections: since you will be using `select()` to figure out which sockets have data, you **MUST make your SSL sockets non-blocking.** Otherwise, your reads from an SSL connection may block, because there was some data waiting in the socket buffer, but not enough for a full SSL record. To make a socket non-blocking, issue the following function call: `fcntl(sockfd,F_SETFL,O_NONBLOCK)`. Make sure you do this **after** you establish the SSL connection (i.e. after `SSL_accept()`), otherwise you will get an `SSL_WANT_READ` error.

#### 4.3.6 Extra hints!

Since there are a lot of problems which need to be solved in order to create a successful, working web server, here is a list of things you should consider to make your life a little easier.

- Spend some time thinking about how your server will work, and the order in which it accepts connections and processes them. Draw a flowchart of what the server's lifetime looks like.
- Remember to make your SSL sockets non-blocking! Make sure you do this **after** you accept the SSL connection, or you will get an `SSL_WANT_READ` error.
- You will want to make a data structure which stores information about a connected client. This structure should be used for both regular connections and SSL connections.
- You will want to write a wrapper for the `read()` function which takes in your client structure, and calls the appropriate read function depending on if the client is connected normally or via SSL. Remember to call `read()` on normal clients, and `BIO_read()` on SSL clients. See section 5.2 for more information on BIO objects.
- You will want to give each client a read buffer, such that data is read into the buffer, and then requests are parsed from the buffer. This will prevent your server from starving other clients when one client is sending multiple requests on one connection. The same holds true with a write buffer, where if your server is sending a large file to one client, other clients will starve. However, a write buffer is not necessary for this assignment.
- After calling `BIO_write()`, you will want to call `BIO_flush()` to make sure the data gets sent back to the client immediately.
- When you read from a normal socket, just call `read()` with a large buffer size of your choosing (1024 bytes, for example). `Read()` will return when it is finished reading data, even if it couldn't fill the entire buffer. The value returned by `read()` is how much data it was able to read. If it returns a value of -1, that means that an error occurred, and you should check the value of errno to find out what happened. See `man read` for more details.
- You can do this same trick with `BIO_read()`. Call `BIO_read()` with a large buffer size, and it will return once all available data has been read, even if it could not fill the entire buffer.

### 4.4 The web client: mws-client

We have provided you with a web client which you can use during the testing of your web server. It can be found at:

`http://www.cs.cmu.edu/~srini/15-441/F02/Projects/lab01/mws-client`

This is the same client we will be using to grade your programs with. It supports GET and HEAD requests, multiple connections, persistent connections, and SSL connections.

### 4.4.1 Invocation

The following is a summary of the command line options the web client accepts:

- `-g` *url*: Issues a GET request for the specified url, and prints the returned results to console (stdout). Prefixing your URL with "http://" or "https://" is optional. "https://" means secure HTTP, and will turn on SSL.
- `-h` *url*: Issues a HEAD request for the specified url, and prints the returned results to console.
- `-d`: Outputs the HTTP headers received from the server before outputting the content received.
- `-o` *filename*: Saves the requested file to filename. This is needed for working with binary files. If this option is not present, then the client will print the results to console, which may not be pretty in the case of a binary file.
- `-p` *port*: Overrides the default port the client should connect to. The default is port 8000.
- `-s` *filename*: Read a script from the specified filename. The file format is described in section 4.4.2. If this option is present, then all of {`-g`, `-h`, `-o`, `-p`, `-ssl`} are ignored.
- `-ssl`: Uses SSL to initiate a connection to the server. If there is no port specified when using SSL, the default port will be changed to 8001.

### 4.4.2 Persistent connections

Since persistent connections count on being able to issue multiple requests to a server before the connection is closed, we provide a mechanism for doing this in the client. The following is a script which will issue a series of GET and HEAD requests to a web server in succession:

```
http://www.server.com:80
G /index.html
G /dir/file.html
H /index.html
```

The first line of the file specifies the server address and port. If the server address is prefixed with "https://", then an SSL connection will be established with the server. The remaining lines specify an action (`G` for GET, `H` for HEAD), along with the path to the file. All of the requests in this file will have `Connection: Keep-Alive` in the header, except for the last one, which will have `Connection: close`. The client processes this file line by line, issuing HTTP requests in succession until all lines have been read in. All lines in this file are first processed by the client, and then all HTTP requests are sent to the server before any responses are read in and processed. Responses from the web server are concatenated into a single file, named "out.txt".

# 5 SSL Programming Tutorial

This section explains the SSL echo server and client programs. You can get the full source code from `http://www.cs.cmu.edu/~srini/15-441/F02/Projects/lab01/ssl_echo/`.

## 5.1 Files

Here are descriptions of the files included in the example source.

| | |
|---|---|
| *echo_server.c* | The source code for the echo server. Uses the SSL functions defined in ssl_common.c. |
| *echo_client.c* | The source code for the echo client. Uses the SSL functions defined in ssl_common.c. |
| *ssl_common.c / .h* | Contains SSL related functions. |
| *server.pem* | This file has both the server's private key and public key certificate. It is loaded by the echo server on startup, and **must** be named "server.pem". You will be using this file for your web server as well. |
| *Makefile* | A GNU compatible Makefile for this project. There are two targets defined, "echo_server" and "echo_client". Feel free to use this as a base for your own Makefile. |

## 5.2 SSL Connection Walkthrough

This is a walkthrough of the specific SSL function calls performed in both the server and the client.

**Step 1 - server and client** First, we initialize the SSL library, and load the error strings. Error strings need to be loaded so that if OpenSSL encounters an error, we can print a more detailed explanation of that error.

```
// initialize the SSL library
SSL_load_error_strings();   /* readable error messages */
SSL_library_init();         /* initialize library */
```

**Step 2 - server and client** Now we initialize the SSL context. The SSL context keeps track of the public and private keys, as well as a list of trusted authorities. `ssl_initialize_context()` takes in two strings. The first is the path to the pem file where the public and private keys which should be loaded from, and the second is the path to the pem file of a trusted certificate authority.

```
ssl_context = ssl_initialize_context( ... );
```

**Step 3 - server** Next, the server waits for a connection, and accepts it. After the server accepts the connection, the server creates a new Buffered I/O object (BIO) which we attach to the client's socket. This BIO object will be used for reading and writing SSL data. The reason we want to use a buffered I/O object is because it lets us work with more natural units (lines and characters) rather than with SSL records. Next, the server makes a new SSL object, which will use the context we set up in the previous step to negotiate the SSL connection. Finally, after we attach the BIO object to the SSL connection, we call `SSL_accept()`, which will handle all of the gory details of negotiating the SSL connection. Note that it is highly important that you do not read from or

write to the client socket before `SSL_accept()`. If you do, the SSL negotiation process will fail, and you will be left with cryptic error messages, such as "wrong SSL version number".

```
clientSocket = accept( listenSocket,
                       (struct sockaddr *)&clientAddress,
                       &clientLength );
...
ssl_client_bio = BIO_new_socket( clientSocket, BIO_NOCLOSE );
ssl_connection = SSL_new( ssl_context );
SSL_set_bio( ssl_connection, ssl_client_bio, ssl_client_bio );
...
if ( (ssl_error_code = SSL_accept(ssl_connection)) <= 0)
```

**Step 3.5 - client** The client does the same SSL context initialization and creation of SSL and BIO objects as the server. Then, while the server is sitting in `accept()`, the client will attempt to connect to the server. When this is successful, the client will then call `SSL_connect()` to initiate the SSL handshake. Again, please note that you cannot read from or write to the connection socket until after `SSL_connect()` is called, otherwise the SSL connection will not be able to negotiate.

```
ssl_connection = SSL_new(ssl_context);
ssl_server_bio = BIO_new_socket(connectSocket, BIO_NOCLOSE);
SSL_set_bio( ssl_connection, ssl_server_bio, ssl_server_bio );
...
if (connect(connectSocket,
        (struct sockaddr*)&serverAddress,
        sizeof(serverAddress)) < 0) {
...
if ( SSL_connect(ssl_connection) <= 0 )
```

**Step 4 - client** Next, the client verifies that the server has sent a valid X.509 certificate. This function will also print out the common name in the certificate, which will be "15-441 Networks Server Certificate". The `ssl_check_cert()` function is fully implemented in `ssl_common.c`, and you will not have to perform any additional certificate checks other than the one already implemented.

```
ssl_check_cert(ssl_connection);
```

**Step 5 - server (and client)** Now that we have established an SSL connection, we can send data over the network by writing to the BIO object we created in steps 3 and 3.5. Writing and reading is the same for the server and the client, except in reverse order (the client sends data to the server, the server reads this data and writes it back to the client, and the client then reads that data). We use the `BIO_write()` and `BIO_read()` functions respectively. After we perform a read or a write, we want to flush the BIO object to make sure that the data is actually sent over the network. Since SSL chunks data into records, a situation may occur where we write some data to the BIO object, but it is not enough to fill a record, so OpenSSL will wait until we write more data. However, we don't necessarily want to wait to write more data before we flush the BIO object, because we may have run out of data to write (say, the final bytes of an HTML file). So, we flush the BIO object to make sure that everything does in fact get sent over the network.

```
dataTransmitted = BIO_write(ssl_server_bio, buf, strlen(buf));
BIO_flush(ssl_server_bio);
```

**Step 6 - server and client** Once the user types in a control-d character, the client initiates a shutdown of the SSL connection. When the server sees the control-d, the server will then call `SSL_shutdown()`, closing the SSL connection. We do some extra error checking in the actual source code, which is also documented there.

```
ssl_error_code = SSL_shutdown(ssl_connection);
```

**Step 7 - server and client** Finally, we free the SSL object and close the socket. The client will free the SSL context and terminate, and the server will loop back and wait for another connection. The server does NOT free the SSL context until it is done with accepting SSL connections (which is never, since the server loops indefinitely until it is forcibly terminated). The code from the client is shown below.

```
SSL_free(ssl_connection);
ssl_destroy_context(ssl_context);
...
close(connectSocket);
```

# 6 Testing

## 6.1 How you should test your programs

The goal of this section is to give you a better idea of how your project will be evaluated, and what you should be on the lookout for while writing your code. The following list is not guaranteed to be exhaustive:

1. Make sure your server can handle GET and HEAD requests from the client. Test requests for different types of files, because sometimes a mistake will be more apparent in one type than another. For example, it's easier to see a corrupted GIF file rather than a prematurely ending text file. Try sending HTML files, GIFs, JPEGs, QuickTime movies, MPEG movies, Shockwave files, ZIP files, etc. Your server should be able to handle all of these types of files once you write the code to handle a GET request.

2. Make sure your server can handle GET requests from other web browsers. If a particular browser has problems with your web server, don't worry too much, because not all browsers are written to the exact HTTP/1.1 specification. We will only be grading with the client we supplied you with.

3. Try requesting files that don't exist, or files in subdirectories.

4. Try requesting a file outside of the server's root directory.

5. Make sure SSL works between your server and the client.

6. Try using Internet Explorer, Netscape, Mozilla, Opera, Amaya, wget, curl, etc. with your web server. You should be able to correctly load a web page using these browsers, but if there are some problems, do not worry too much. Not all browsers are written to the exact HTTP/1.1 specification. If you experience a long timeout delay when using a browser, this may be due to how persistent connections are implemented. Do not worry about any odd problems with web browsers, as long as your server works with the client we supplied.

### 6.2 The tests.txt file

As part of this assignment, you are responsible for testing your submission and making sure everything works. You should document all of your testing strategies in a file named `tests.txt` and submit it along with the rest of your assignment. This file is worth 5% of your grade!

Also, if you have any outstanding issues you know about, but are not able to fix, you should let us know about them. Or, if you have any comments about this assignment, felt that it was too hard or too easy, please let us know.

## 7 Grading Criteria

When we grade your submission, we will run `gmake` to compile your web server, and then run a combination of tests and grading scripts to evaluate your submission. If your project generates compiler warnings during compilation, you will lose credit; if your server dumps core during our testing, you will lose substantial credit. Remember to compile with the `-Wall` flag, otherwise you will lose points if this is not in your `Makefile`.

Poor design, documentation, or code structure will probably reduce your grade by making it hard for you to produce a working program, and making it hard for us to understand it. Remember the 15-212 slogan: *Think before you hack!*. Egregious failures in these areas will cause your grade to be lowered even if your implementation performs adequately. Putting all of your code in one module counts as an egregious design failure.

It is better to have partial functionality working solidly than lots of code that doesn't actually do anything correctly.

The following is a checklist of all of the features your web server must possess. As you write your programs, you should cross off each feature from the list once you have implemented and tested it. All of these features will be tested during grading.

| mws-server | | 100% |
|---|---|---|
| | accepts the correct command line arguments | 5% |
| | socket infrastructure (socket/bind/listen/accept) | 10% |
| | handles GET and HEAD requests and returns correct HTTP headers | 30% |
| | returns appropriate status codes (200, 400, 404, 501) | 5% |
| | multiple connections | 15% |
| | persistent connections | 15% |
| | SSL connections | 15% |
| | tests.txt file | 5% |

### 7.1 Extra credit

There is one improvement you can make to your server for extra credit, and it is worth 10 points (your maximum score on this lab is 110). For the extra credit, figure out some method for constraining the serving of files to the root directory. Based on this assignment specification, it is possible for an attacker to gain unauthorized access to files outside of the root directory. For example, a request for the URL `http://www.myserver.com/../file.html` will result in sending the client a file which is outside of your server's root directory, assuming that there really is a `file.html`. More realistically, what if a client tried to access `http://www.myserver.com/../../etc/passwd`?

Try this with your webserver and see for yourself. Note that the following URL is within the root directory: `http://www.myserver.com/subdirectory/../index.html`.

### 7.2 What to turn in

Below is a listing of all of the files you should submit. The handin directory is `/afs/cs.cmu.edu/academic/class/15441-f02-users/group-XX`, where XX is your assigned number on the course web site (see the pictures page). If you do not have an assigned number, please see the professors immediately!

Please note that we will be checking timestamps in order to determine late submissions. Check the late policy on the course web site and notify the TAs and professors if you will be submitting this assignment late.

- GNU compatible `Makefile`
- All of your source code (files ending with `.c` or `.h` only, no `.o` files)
- Documentation of your test cases and any known issues you have, named `tests.txt`

## 8 How to succeed in this assignment

Above all, please **start early**. We will be able to help you more if you **start early**. You will feel better if you **start early**. You will feel even **better** if you finish the assignment *before* it is due. Remember, lots of bad things can happen at 11:50pm the night the assignment is due (if you want war stories, see Justin). Keep in mind that this is the first of four labs, and doing well on this assignment will ease your mind later on in the course.

Now that you are resolved to starting this assignment early, read over this handout several times. If anything is unclear, please send email to the bboard (**academic.cs.15-441**), or come to office hours. We are here to help!

Finally, now that you understand the assignment, here is a **suggested** plan of attack. Note that we will not hold you to this outline, this is here solely for guiding you through this assignment.

**Step 1** Use English, not HTTP. Only use one client.

Write two programs, called `mws-client.c` and `mws-server.c`. You will run `mws-client.c` on one machine, say unix46. You will run `mws-server.c` on another machine, say, unix47. You will need to pick a port number for your server. Your client program will need to know this port number (you will need port numbers in the future as well, see sections 4.3.1 and 4.4.1).

Your `mws-server.c` program should do the following: create a socket and bind, listen on the socket, accept and obtain a new socket descriptor, print the IP address of the client and the port number of the client (see `man inet_ntoa`), read a request from the client, print the client's request, and write a message to the client saying "I got your request".

Your `mws-client.c` program should do the following: create a socket, connect, write a request to the socket, where the request says "Hello server, here is my request", read the response from the server and print that response.

Please read the Lecture 3 notes for more information on socket programming. Man pages are another good source of information for the various arguments to socket calls.

**Step 1a - Quick Enhancement** Make your client send multiple requests on this same connection and have your server read the multiple requests and handle them all.

**Step 1b - Quick Enhancement** Start up multiple clients on the same machine or different machines and have them all send requests to the server . Do not use the `select()` call. Instead, try putting a loop around the `accept()` call. Make sure you understand when this approach works and when it doesn't.

**Step 2** Use English, not HTTP. Use `select()`. Use multiple clients.

Run `mws-client.c` on multiple machines. Each instance of `mws-client.c` should send a different message to the server. Your `mws-server.c` code should use the `select()` call to figure out which client has sent a request. The server should again reply to the request in simple English. Rather than having the server simply say to the client "I got your request", the server should greet the client with the client's IP and port number. For instance, "Thank you client with IP 128.2.19.5 and port 8000, I got your request and here is my response." This will help you check that your server is doing the right thing.

**Step 3** Use English, not HTTP. Use `select()`. Use multiple clients. Persistent connections.

Make up a delimiter that ends each client request, e.g. `\r\n`. Create a scenario where a client might send 2 requests before the server has a chance to do a read. Thus when the server reads the request, the server will really be reading 2 requests. Make sure that your `mws-server.c` program checks for end-delimiters and can tell when the client has sent multiple requests. This is also a good time to make sure that your server can handle the case where the client request is so long that it can't obtain the whole request with a single read.

**Step 4** Use English, not HTTP. Use `select()`. Retrieves files. Use multiple clients. Persistent connections.

Now edit the requests in `mws-client.c` to be of the following format: `GET filename\r\n`, where filename is a fully specified path, like `./file.html`. Make your server try to open the file in the specified directory. If the file does not exist, have your server reply with "Sorry, not there". If the file does exist, your server should send back the whole file.

**Step 5** Use HTTP. Use `select()`. Retrieves files. Use multiple clients. Persistent connections. GET.

Repeat Step 4, except that this time, each client should send a GET file HTTP message to the server, and the server should reply with an HTTP message containing the file, or an HTTP message with the appropriate error code. Be careful to include lots of error checking in your program; for example, the server should complain if the request being sent to the server is not written in proper HTTP.

Again, allow each client to request multiple files from the server. Experiment with every odd scenario you can imagine.

Read the handout given to you on application programming by Kurose/Ross. It contains general information about HTTP messages. Then read RFC 2068 on HTTP.

**Step 6** Use HTTP. Use `select()`. Retrieves files. Use multiple clients. Persistent connections. GET. HEAD.

Expand your server to handle HEAD requests.

**Step 7** Use HTTP. Use `select()`. Retrieves files. Use multiple clients. Persistent connections. GET. HEAD.

Test your web server with our client, as well as a real web browser. See section 6 for ideas on how to test your server.

**Step 8** Use HTTP. Use `select()`. Retrieves files. Use multiple clients. Persistent connections. GET. HEAD. SSL.

Add SSL support to your server. Take a look at the SSL echo client and server source for how to do this.

**Step 9** Use HTTP. Use `select()`. Retrieves files. Use multiple clients. Persistent connections. GET. HEAD. SSL.

Test your program thoroughly. Write the `tests.txt` file as described in section 7.2.

**Step 10** Take a deep breath, and turn in your submission. Congratulations, you have written a web server!