Michigan State University
Computer Science & Engineering Department
CSE422 Computer Networks, Spring 2017

Laboratory 2: Web Proxy Server

Due: 23:59 Thursday, March 23, 2017

1 Outline

Note that you are neither required to follow these steps nor required to use the skeleton code [link]. In the skeleton code, the missing parts that require your implementation is marked with a comment `/********TO BE IMPLEMENTED********/`. Please feel free to modify the helper classes if you prefer.

1. Run the client program (`client.h/cc`) and understand the usage of the helper classes. Refer to `.h` files for function definitions and their purposes. Refer to `.cc` files for implementations. More information can be found in Section 4.
2. Complete the missing part of `proxy.cc`. Create a TCP socket to accept connection. For `TCPSocket` class, please use `try/catch` to capture exceptions. You can find samples in client program. Refer to Section 12 for logs.
3. Complete the five functions that has missing part in `ProxyWorker.cc`, in the following order, `getRequest`, `checkRequest`, `forwardRequest`, `getResponse`, and `returnResponse`. More details can be found in Section 8.
4. In `getResponse`, complete identify/default transfer encoding before working on chunked transfer encoding (Section 4). Refer to client program for more details.
5. Work on keyword filtering. Note that one requirement is filtering the hostname and the other is filtering the path. (Section 8).
6. Work on subliminal messages (Section 5).
7. Try your proxy with real browsers (not required).

2 Goals

Apply your knowledge of socket programming in order to implement a real-world application and gain some basic understanding of HTTP.

3 Overview

In this lab, you will implement a simple *proxy server* for HTTP that forwards requests from *clients* to *end servers* and returns responses from *end servers* to the *clients*. You will also implement a special function that inserts *subliminal messages* between HTTP web pages.

This lab is worth for 12% of the final grade and is composed of 120 points. This lab is due no later than 23:59 (11:59 PM) on Thursday, March 23, 2017. **No late submission will be accepted.** You will submit your lab using the CSE *handin utility* (http://secure.cse.msu.edu/handin/).

3.1 The HyperText Transfer Protocol, HTTP

The **HyperText Transfer Protocol (HTTP)** is the World Wide Web's application-layer protocol. HTTP operates by having a *client* (usually a browser) initiate a connection to a *server*, send an HTTP request, and then read and display the server's response. HTTP defines the structure of these messages and how the clients and servers exchange messages.

A *web object* is simply a file, such as an HTML file, a JPEG image, or a video clip. A *web page* usually consists of one HTML file with several referenced objects. A page or an object is addressed by a single *Uniform Resource Locator (URL)*. When one wants to access an HTML page, the web browser initiates a request to the server and asks for the HTML file. If the request is successful, the server replies to the web browser with a response that contains the HTML file. The web browser examines the HTML file, identifies the referenced objects, and for each referenced object, initiates a request to retrieve the object.

An example of an HTTP request/response is shown in Figure 1. The request has only a message header and does not have a message body. The response consist of a message header followed by a message body, which is the HTML file requested. The header is composed of several lines, separated by a carriage return and line feed (CRLF, "`\r\n`"). For each message, the first line of the header indicates the type of the message. Zero or more header lines follow the first line; these lines specify additional information about this message. The end of the header is indicated by an empty line. The message body may contain text, binary data, or even nothing at all.

There are actually 8 different HTTP request methods, however, in this lab we consider only the `GET` method, which is used to request objects from the server. The `GET` request must include the `path` to the object the client wishes to download and the HTTP version. In the above example, the path is `/~liuchinj/cse422ss17/index.html` and the HTTP version is `HTTP/1.1`. Some request methods, such as `POST`, transmit data to the server in a message body. However, the `GET` method does not have a message body.

In its response, the server indicates the HTTP version, status code and status description.

Request

```
GET /~liuchinj/cse422ss17/index.html HTTP/1.1
Connection: close
Host: www.cse.msu.edu
If-Modified-Since: 0
[blank line]
```

Response

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Connection: close
Content-Length: 7619
Content-Type: text/html
Date: Tue, 14 Feb 2017 04:20:52 GMT
ETag: "1dc3-5475aaccd8938"
Last-Modified: Tue, 31 Jan 2017 02:27:35 GMT
Server: Apache/2.4.10 (Debian)
Vary: Accept-Encoding
[blank line]
<!DOCTYPE html>
<html lang="en">
<head>
  <title>CSE422 Computer Networks at MSU</title>
  ....
```

Figure 1: Example HTTP request and response message. In the request, the client asks for `/~liuchinj/cse422ss17/index.html` from the Department's web server `www.cse.msu.edu` over HTTP/1.1. In the server's response, the server informs the client that the request was successful with the status code `200` and several additional header lines that carry information about this response. Note that each line is ended by a CRLF.

The status code and status description indicate whether the request was successful and, if not, why the request failed. Common status codes and status description include:

- `200 OK`: Request succeeded
- `403 Forbidden`: The request failed because access to the resource is not allowed.
- `404 Not Found`: The request is failed because the referenced object could not be found.
- `500 Internal Server Error`: There is something wrong at the server side.

For a more detailed information about HTTP, please see:

- Computer Networking: A Top-Down Approach, Sixth Edition, page 97 - 105.
- Wikipedia Entry
- RFC 2612: HTTP/1.1 and RFC 1945: HTTP/1.0

3.2 Proxy Server

As shown in Figure 2, a *proxy server* is a program that acts as a middleman between a *client* and an *end server*. Instead of requesting an object from the server directly, the client sends the request to the proxy, which forwards the request to the server. When the server replies to the proxy, the proxy returns the response to the requesting client.

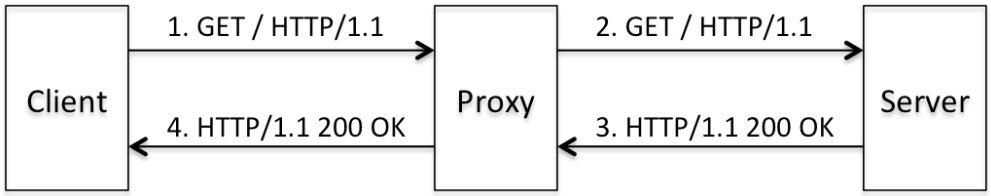

Figure 2: The client sends request to the proxy and the proxy forwards the request to the server. The proxy awaits the server's response and returns it to the client.

Proxies are used for many purposes. Sometimes proxies are used as firewalls, such that the proxy is the only way for a client behind a firewall to contact any server outside. Proxies are also used as anonymizers. By removing or modifying a request header, a proxy can make the client anonymous to the server. In addition, by examining the request header, a proxy can filter and block requests, for example, blocking any request where the URL contains the "umich.edu" keyword.

An important application of proxies is to cache web objects by storing a copy when the first request is made, and then serving that copy in response to future requests rather than going

to the server. For large businesses or ISPs, caching of frequently requested object can greatly reduce the communication costs.

More information can be found in lecture note 2-107 to note 2-112.

4 A simple HTTP client

In order to help you understand the HTTP message exchange and focus on the implementation of proxy, a simple command line HTTP client is provided to you along with the skeleton code [link]. Several classes are provided along with this simple HTTP client. (As demonstrated in class, you will be able to use your proxy with a commercial web browser, but this simple client likely be helpful in constructing and debugging your proxy. Moreover, the client program also serves as a sample code to demonstrate how to use the classes provided.)

The following description of the client, combined with your studying the code (`client.cc` and various utilities) should be helpful in constructing your proxy.

```
Usage: ./client [options]
The following options are available:
    -s host URL
    -p proxy URL
    -h display help message
```

The URL to the desired web object must be specified by the argument `-s`. `-p` argument is optional.

Example invocation without proxy:
`./client -s http://www.cse.msu.edu/~liuchinj/cse422ss17/index.html`

This invocation does exactly the same thing as in Figure 1 and stores a copy of `index.html` in a `Downloads` folder under current directory.

Example invocation with proxy running on `smithers.cse.msu.edu` at port `31187`:
`./client -s http://www.cse.msu.edu/~liuchinj/cse422ss17/index.html -p smithers.cse.msu.edu:31187`

If the proxy port is not specified in the command line, the client assumes it to be 8080. **NOTE:** However, in this lab, the port used by the proxy is assigned by the operating system, so you will need to enter that port number in the command line when starting the client.

If there is no proxy waiting at the specified address (IP address and port number), then the connection to the proxy fails and the client program is terminated. If there is a proxy running at the address specified, then the download should be successful and store a local copy in the

subdirectory `Downloads`. Each invocation of the client program initiates a separate HTTP request and handles the response for that request.

4.1 Initiating a Request

To initiate a request, the HTTP client has to connect to the server, construct a request message, and send the message to the server (or to the proxy, depending on how the client is invoked). In the remainder of this section, *server* means either the end server or the proxy server.) The `TCPSocket` class provides the functionality for the communication and handles details of setting up the socket.

Processing HTTP messages requires a lot of string parsing and formatting. A `URL` class is provided to help you parse the given URL and store it as an object. The method `URL::parse` takes a string as the argument and returns the pointer to the parsed URL object if the string is a valid URL, or NULL otherwise.

An `HTTPRequest` class is provided to handle the construction of new HTTP requests, for sending/receiving of requests, and for parsing an incoming HTTP request (which is not needed by the client, but is needed by the proxy.)

We summarize the initiation and sending of requests as follows:

- Parse the server URL string by invoking `URL::parse`.
- Create a `TCPSocket` object: `TCPSocket clientSock`. The method `clientSock.Connect` connects to the corresponding server.
- Create an `HTTPRequest` object `request` by invoking `HTTPRequest::createGetRequest`.
- Configure this HTTP Request.
- The method `HTTPRequest::send(clientSock)` sends the request to the server.

4.2 Handling the Response

Next, the client expects the HTTP response from the server. We provide an `HTTPResponse` class for sending/receiving of responses, for parsing of incoming HTTP response, and to handle the creation of new HTTP requests.

Handling a response is a two-step procedure, first handling the response header and then the response body. Two steps are needed because the length of the message body varies, and the client does not know in advance *when to stop receiving incoming data* from the socket. When a process invokes the `read/recv` system call, the system call either immediately returns the number of bytes received, or the process blocks and waits for incoming bytes. Without

knowing the length of the message body, the client does not know when to stop calling `read/recv`. Therefore, a client has to receive the header first and examine the header fields to determine the number of bytes to expect in the body.

The message header comprises several lines, each ending with a CRLF, and the end of the header is marked by a blank line. The client keeps reading one line of data until two consecutive CRLFs are found in the buffer; the rest of incoming data belong to response body. The `readHeader` method is provided in both `HTTPRequest` and `HTTPResponse`. If you wish to handle the data yourself, the method `readLine` is provided in `TCPSocket`.

The *transfer encoding* of an HTTP response defines how the data is encoded. Several transfer encoding mechanisms are supported in HTTP (e.g., identity, chunked, compress, deflate, gzip). Most of these are designated by a `Transfer-Encoding` line in the header. In this lab, we are concerned only with two of these: identity encoding and chunked transfer encoding.

4.2.1 Identity Encoding

Identity encoding is the default transfer encoding mechanism defined in HTTP. The `Transfer-Encoding` line is not present in the header. The `Content-Length` line specifies the length of the response body. The client simply receives this specified amount of data and stores it as the response body.

4.2.2 Chunked Transfer Encoding

Chunked transfer encoding, defined in HTTP version 1.1, enables a web object to be sent from the server as a series of "chunks." The advantage of chunked transfer encoding is that the server does not need to know the length of the response body before starting to send parts of it to the client. This capability is particularly important for returning dynamically generated content (such as a shopping cart).

Each chunk is separated by a CRLF and begins with a hexadecimal chunk size followed by an extra CRLF. After reading the header, if this response uses chunked transfer encoding, the client reads one more line, which indicates the length of the first chunk. The client receives data until the chunk is completely downloaded. The client reads two more lines, the first line is the blank line between chunks and the second line is the size of next chunk. The client continues this process until it receives a zero chunk size, which indicates the end of the transfer.

4.3 Response Summary

The handling of an HTTP response is summarized as follows:

- Create an `HTTPResponse` object.
- Receive the response header by invoking `HTTPResponse::receiveHeader`, and parse the header.
- Receive the response body. You can check if this response is chunked by invoking `HTTPResponse::isChunked`.
- Store the received data as a file.

5 Subliminal Messages

We can insert a simple type of "subliminal messages" (messages below the threshold of consciousness) into HTTP responses by taking advantage of the proxy and HTML meta refresh. Here, we insert images to remind the user of the success of the MSU football team.

5.1 HTML Meta Refresh

The HTML Meta refresh method instructs a browser to load a given URL after a given amount of time, in seconds. This automatic refresh can be done by setting an HTML meta parameter `http-equiv` to `refresh`. The time interval and target URL are specified by setting the parameter `content` to `<time>; url=<target-url>`. A webpage that performs HTML Meta refresh is shown below.

```
<html>
  <head>
    <meta http-equiv="refresh" content="5;url=http://www.cse.msu.edu/">
  </head>
  <body>
    <center>
      <H1>
        Demonstration of HTML Meta Refresh <br>
        Redirecting to the MSU CSE homepage in 5 seconds...
      </H1>
    </center>
  </body>
</html>
```

This webpage automatically redirects to MSU CSE homepage in 5 seconds. You can create an HTML file containing the code above and load it in a browser to see how it works.

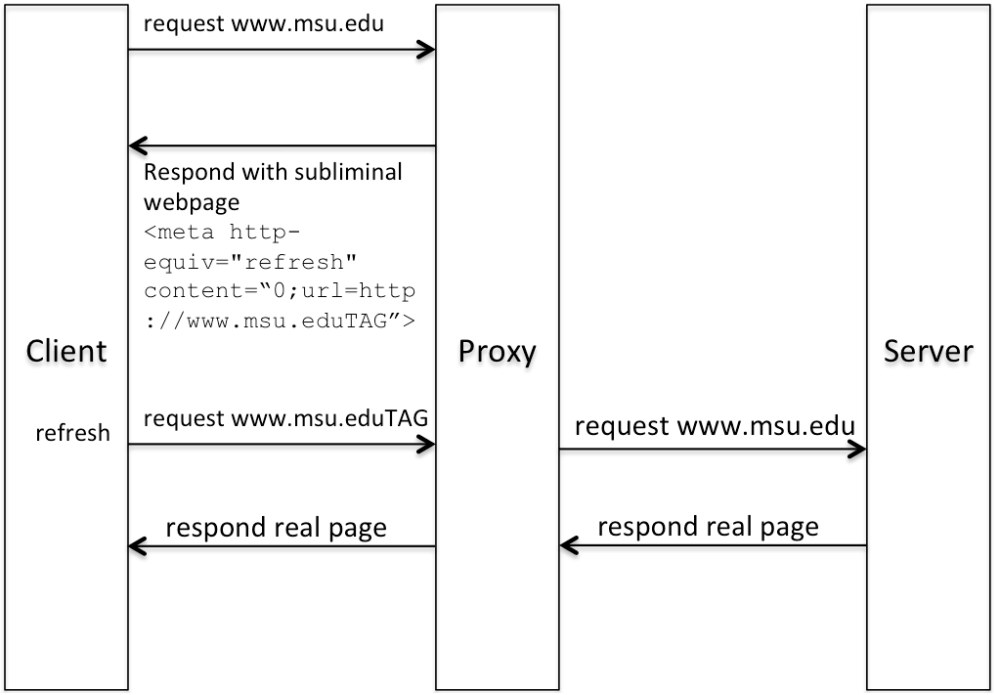

Figure 3: The client sends request to the proxy and the proxy forwards the request to the server. The proxy awaits the server's response and returns it to the client.

5.2 Inserting a Subliminal Message Webpage

Similarly, we can create a webpage that contains an image and redirects the browser after a very short amount of time, say 0 seconds, to another webpage. But how do we force the the browser to load the subliminal message webpage? We can implement the web proxy functionality depicted in Figure 3.

When the proxy receives a request, it checks if the URL contains a special tag. If the URL does not contain the special tag, instead of forwarding this request to the server, the proxy responds with a subliminal message webpage that redirects to the originally requested URL (but with a a special tag appended to the URL). When the browser refreshes, it requests the URL containing the special tag. The proxy uses this tag to identify that the redirection has already occurred. It then simply forwards the original request to the server and delivers the response to the client.

Take Figure 3 as an example. The browser requests the URL `http://www.msu.edu/`. The proxy receives the request and checks if the URL contains the special tag. Because this URL does not have the tag, the proxy responds with a subliminal message webpage that redirects to `http://www.msu.eduTAG/`. When the browser refreshes, the browser requests `http://www.msu.eduTAG`. The proxy strips off the tag and handles this request as a normal proxy. The skeleton code includes a function, `subliminalResponse`, that responds with a

subliminal message webpage. You can check if a subliminal message has been inserted to a webpage or not by invoking the function `ProxyWorker::hasSubliminalTag`.

6 Threads

Our proxy is a multithreaded program. The multithreading part is provided in the skeleton. You are not required to implement this part in this lab. However, in case you are unfamiliar with threads, here are some information about threads.

A multitasking operating system schedules and interrupts tasks periodically so that the tasks appear to run concurrently. In most modern operating systems, processes and (kernel-level) threads are the task units to be scheduled. Threads are finer-grained than processes and sometimes referred to as lightweight processes. One or more threads may reside in the same process.

When a parent process forks a child process, the child process has its own address space and its own copy of all data of the parent process. The child has its own copies of global variables and resources such as descriptors to open files; for example, the child process may close an open file without affecting that of its parent process. In contrast, the threads in a process share the same address space; as a result, communication and context switching among threads are faster than among processes. Creating a new thread also requires less overhead than spawning a new process, in part because the system does not need to create an entirely new address space. Therefore, multitasking based on threads is more efficient than multitasking based on processes.

6.1 POSIX Thread Library (`pthread.h`)

The POSIX thread, or *pthread* library, is a standard thread API for C/C++. It defines an API for manipulating threads on Unix/Linux operating systems. The functions and data types are declared in the header file `pthread.h`. The library includes operations for thread management, mutexes, conditional variables and synchronization.

- Each thread in a process is identified by a thread ID of type `pthread_t`.
- Thread creation [man page]:

  ```
  int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                     void *(*start_routine) (void *), void *arg);
  ```

  The `pthread_create()` function creates a new thread in the invoking process.

  - `thread` is the thread ID of type `pthread_t`.
  - `attr` describes the attributes of the thread. In this lab assignment, the default thread attribute is sufficient. We can simply set this parameter to `NULL`.
  - `start_routine` is the function where thread execution will begin.
  - `arg` is a pointer to the (only) argument of the above function. To pass multiple arguments, point to a structure.

  To the invoking thread, the call to `pthread_create()` returns immediately and execution continues. At the same time, the new thread begins its execution of the specified function.

- Thread termination: The thread terminates by calling the function `pthread_exit()`.

The following are some online resources for `pthreads`.

```
Linux tutorial: http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
Another pthreads page: https://computing.llnl.gov/tutorials/pthreads/
```

7 Guidelines

Working alone or in pairs. You may work on this assignment individually or in pairs (not in groups of 3 or more, however). If you prefer to work in a pair, **both** students must submit a copy of the solution and identify their MSU NetID in a README file. If you prefer to work individually, please clearly state that you are working individually and include your MSU NetID in the README file.

Programming Language. You must implement this project using `C++`. The skeleton code is written in `C++`. Please clearly state the command to compile your project submission in your README file.

Testing your code. Each Linux distribution might be slightly different. It is the students' responsibility to make sure that the lab submissions compile on *at least one* of the following machines in 3353 EB: `carl`, `ned`, `marge`, `mcclure`, `apu`, `krusty`, `rod` or `skinner`. A statement must be provided in the README file's header. You will **not** be awarded any credit if your lab submission does not compile on any of those machines.

8 Specification

In this lab, you are required to implement a proxy that forwards `GET` requests from a client to the server and returns the responses from the server back to the client. The port for

listening to incoming request is assigned by the operating system. This lab addresses only non-persistent connections. The proxy is expected to be able to handle multiple requests by spawning a separate thread to handle each request. Both default (identity) encoding and chunked transfer encoding must be handled by this proxy. The requirement of the proxy is listed as follows.

Subliminal messages: The proxy needs to insert subliminal message webpages for HTTP requests where the target webpages has a filename extension of either `.html` or `.htm`. A supporting function `HTTPRequest::isHtml` is also provided for this purpose. A supporting function `ProxyWorker::subliminalResponse` is provided to generate the subliminal response. Set the message duration to 0 will makes the response to flash and act like a subliminal message.

Modify `Server` field in the response header: To help with debugging, you are required to add/modify a field in the response header, saying that this response is returned by your proxy. Specifically, you are required to add (or modify) the field `Server` with a string, such as your MSU NetID, showing the header has been modified. The method `HTTPResponse::setHeaderField` is able to do this.

Host not found: The proxy is expected to respond with error messages to bad requests. For a request that tries to download an object from a host that does not exist, the proxy returns a `404 Not Found` response. As long as the end server exists, it is the responsibility of the server, not the proxy, to determine whether or not the requested web object exists. The proxy simply forwards the request and returns the response. A supporting function `ProxyWorker::proxyResponse` is provided for you to create response when error occurs.

Filtering: The proxy is expected to perform simple filtering.

1. The proxy rejects any request to a host containing the keyword "umich.edu" but allows requests where the file path contains the "umich.edu" keyword. The proxy returns a `403 Forbidden` response for the former request. For example, the proxy shoud reject a request to `http://umich.edu`, but should forward to the server a request for `http://www.cse.msu.edu/~liuchinj/cse422ss17/lab2_files/umich.html`

2. The proxy redirects any request to a path containing the keyword "harbaugh" to `http://www.cse.msu.edu/~liuchinj/cse422ss17/lab2_files/whoa.html`

You can operate on the host and path information in the request using the supporting functions `HTTPRequest::getHost`, `HTTPRequest::getPath`, `HTTPRequest::setHost` and `HTTPRequest::setPath` before forwarding the request to the server. Alternatively, your proxy can also create a response that gives the client the Webpage you want using `HTTPResponse` class and replies the new response to the client without contacting the actual server.

Transfer Encoding: The proxy is required to handle both default transfer encoding and

chunked transfer encoding. For default transfer encoding, the proxy is required to display (print to the console) the content length. For chunked transfer encoding, the proxy is required to display (print to the console) the length of each chunk.

The Work Flow of the Proxy

We outline the work flow of this proxy in this section. The proxy starts running and waits for incoming connections. For each connection, the proxy has to do the following:

- Get the request string from the client, check if the request is valid by parsing it. (the method `HTTPRequest::receive` receives the request and also parses it.)
- From the parsed `HTTPrequest` object, obtain the server address by invoking `HTTPRequest::getHost`. Also check the validity of this server address by invoking `URL::parse`. If this server is invalid (NULL is returned), respond to this request with `404 Not Found`. If the proxy is supposed to block requests to this server (i.e., contains facebook keyword), respond to this request with `403 forbidden`. If the server is valid and not filtered, continue.
- If this request's path does not contain the special tag (by invoking the function `ProxyWorker::hasSubliminalTag`), the proxy responds this request with `subliminalResponse` and the proxy is done serving this request. If this request's path contains the special tag, the proxy serves this request as a normal proxy.
- Forward this request to the server by invoking `HTTPRequest::send`.
- Receive the response header and modify the `Server` header field. (`HTTPResponse::setHeaderField`)
- Receive the response body. Handle both default and chunked transfer encoding.
- Return the modified response to the client.

The proxy returns a `404 Not Found` response to the client whenever it cannot reach the server specified in the request, including:

- The parsed server URL is NULL, which means the URL is not valid.
- Failure to connect to the server.
- Unable to resolve the server URL.
- Server does not exist.
- etc.

When the server responds with a `403` or `404` message, the content/body of the response is usually a webpage showing related information. However, the provided `HTTPResponse` class constructor generates responses for these cases that contain only a header. In this lab, your proxy server does not need to provide response content/body for error cases. It is fine as long as the header is correct.

A skeleton file is provided to you along with the simple client. [link].

9 Deliverables

You will submit your lab using the *handin* utility (http://secure.cse.msu.edu/handin/). Please submit all files in your project directory. If you start your lab with the skeleton code, submit all files, even for files that are not modified.

This lab is due no later than 23:59 (11:59 PM) on Thursday, March 23, 2017. No late submission will be accepted.

The compilation must be done using a makefile. The code should compile and link on CSE machines in 3353 EB. (See Section 11. bullet 1) You will not be awarded any points if your submission does not compile using makefile. Please test your proxy thoroughly before submitting the code.

A README file is required. You will run your proxy with the client program provided and record the log in your README file. A sample README file is also included in the skeleton code. You are also encouraged to include any relevant comments in the README file.

10 Grading

You will not be awarded any points if your submission does not compile.

```
General requirements: 5 points
_____  2 points Coding standard, comments ... etc
_____  1 points README file
_____  2 points Descriptive messages/Reasonable output.
             Display the headers and (content length or chunk sizes)

Proxy basic functions: 65 points
_____  5 points get the request
_____  5 points forward the request
_____  5 points checking the request
_____ 10 points Return the default transfer encoding responses.
```

```
_____ 35 points Return the chunked transfer encoding responses.
_____  5 points Add/Modify the Server header field

Proxy handling special cases: 30 points
_____  5 points Respond with 404 not found to request for non-exist URL
_____  8 points Filter out requests to any "host" contains "umich.edu"
                and return a 403 forbidden.
_____  5 points Allow requests to a "path" contains "umich.edu"
_____ 12 points Redirect requests to any "path" contains "harbaugh" or
                "Harbaugh" to the video we love very much

Proxy inserts subliminal messages: 20 points
_____ 15 points Inserting subliminal messages
```

11 Notes

- Please develop your program on machines in CSE labs, instead of the servers. If possible, make sure your program compiles on `carl, ned, marge or skinner` before submitting.
- This lab only uses non-persistent connections. The constructor of `HTTPRequest` class sets the `Connection` header to `close` for you already.
- Please spend some time tracing the code in the provided classes. One should be able to build the entire proxy using those classes. Tracing the client code would be a good start.
- We plan to modify this proxy in future lab exercises.
- Obviously, the default transfer encoding (identity) is easier to implement than chunked transfer encoding. For your convenience, the requests to the following URLs are guaranteed to reply with default encoding responses. In fact, responses to most web objects that are not HTMLs should use the default transfer encoding.
  - `http://www.cse.msu.edu/~liuchinj/cse422ss17/lab2_files/whoa.html`
  - `http://www.cse.msu.edu/~liuchinj/cse422ss17/images/0.jpg`
- This lab does not require the proxy to work with real browsers. However, if the functions required in this lab are implemented correctly, this proxy should be able to work with real browsers and should be able to display most webpages.

Please feel free to mail the instructor Chin-Jung Liu, liuchinj AT cse.msu.edu for questions or clarifications. Additional notes and FAQ will be posted on Piazza as well.

12 Examples

The following examples show output from the client or proxy for various scenarios. Note that these examples do not include subliminal messages. You need to test subliminal messages on a real browser.

12.1 Client without using proxy.

```
./client -s http://www.cse.msu.edu
Request sent...
============================================================
GET / HTTP/1.1
Connection: close
Host: www.cse.msu.edu
If-Modified-Since: 0
============================================================

Response header received
============================================================
HTTP/1.1 200 OK
Connection: close
Content-Type: text/html; charset=UTF-8
Date: Sat, 18 Feb 2017 04:42:28 GMT
Server: Apache/2.4.10 (Debian)
Transfer-Encoding: chunked
Vary: Accept-Encoding
============================================================

Downloading rest of the file ...
Chunked transfer encoding
       chunk length: 8744
responseBody length: 1244
       chunk length: 8744
responseBody length: 8752
       chunk length: 7431
responseBody length: 120
       chunk length: 7431
```

```
responseBody length: 7436
       chunk length: 0
responseBody length: 2
Download complete (16175 bytes written)
```

12.2 Client with proxy

```
>./proxy
Proxy running at 58854...
New connection established.
New proxy child thread started.
Getting request from client...

Received request:
============================================================
GET /CSE422 HTTP/1.1
Connection: close
Host: www.cse.msu.edu
If-Modified-Since: 0
============================================================
Checking request...
Done. The request is valid.

Forwarding request to server http://www.cse.msu.edu/...
Getting the response from the server...
Response header received.
Receiving response body...
Chunked transfer encoding
       chunk length: 8744
responseBody length: 1244
       chunk length: 8744
responseBody length: 8752
       chunk length: 7431
responseBody length: 0
       chunk length: 7431
responseBody length: 7436
       chunk length: 0
responseBody length: 0
Returning the response to the client...

Returning response to client ...
```

```
============================================================
HTTP/1.1 200 OK
Connection: close
Content-Length: 16196
Content-Type: text/html; charset=UTF-8
Date: Sat, 18 Feb 2017 04:43:58 GMT
Server: MSU/CSE422/SS17
Transfer-Encoding: chunked
Vary: Accept-Encoding
============================================================
Connection served. Proxy child thread terminating.
```

Note that the subliminal tag is CSE422.

```
./client -s http://www.cse.msu.edu/CSE422 -p ned.cse.msu.edu:58854
Request sent...
============================================================
GET /CSE422 HTTP/1.1
Connection: close
Host: www.cse.msu.edu
If-Modified-Since: 0
============================================================

Response header received
============================================================
HTTP/1.1 200 OK
Connection: close
Content-Length: 16196
Content-Type: text/html; charset=UTF-8
Date: Sat, 18 Feb 2017 04:43:58 GMT
Server: MSU/CSE422/SS17
Transfer-Encoding: chunked
Vary: Accept-Encoding
============================================================

Downloading rest of the file ...
Chunked transfer encoding
       chunk length: 8744
responseBody length: 16190
       chunk length: 7431
responseBody length: 7438
       chunk length: 0
```

```
responseBody length: 2
Download complete (16175 bytes written)
```

12.3 Request URLs that are blocked

```
>./proxy
Proxy running at 57271...
Received request:
============================================================
GET / HTTP/1.1
Connection: close
Host: www.umich.edu
If-Modified-Since: 0
============================================================
Checking request...
Request to URL contains umich.edu.
Reject this request by a forbidden.

Returning 403 to client ...
============================================================
HTTP/1.1 403 Forbidden
Connection: close
Content-Length: 48
Content-Type: text/html
Date: Sat, 18 Feb 2017 04:47:13 GMT
Server: MSU/CSE422/SS17-Section001
============================================================
Connection served. Proxy child thread terminating.

>./client -s http://www.umich.edu -p localhost:58854
Request sent...
============================================================
GET / HTTP/1.1
Connection: close
Host: www.umich.edu
If-Modified-Since: 0
============================================================

Response header received
============================================================
HTTP/1.1 403 Forbidden
```

```
Connection: close
Content-Length: 48
Content-Type: text/html
Date: Sat, 18 Feb 2017 04:47:13 GMT
Server: MSU/CSE422/SS17-Section001
============================================================

Downloading rest of the file ...
Default transfer encoding
Content-length: 48
bytes written:48
data gotten:48
403 Forbidden
```

12.4 Requesting an URL that does not exist

```
./proxy
Proxy running at 38408...
New connection established.
New proxy child thread started.
Getting request from client...

Received request:
============================================================
GET /CSE422 HTTP/1.1
Connection: close
Host: www.cseeee.msu.edu
If-Modified-Since: 0
============================================================
Checking request...
Done. The request is valid.

Forwarding request to server http://www.cseeee.msu.edu/...
Unable to connect to server.

Returning 404 to client ...
============================================================
HTTP/1.1 404 Not Found
Connection: close
Content-Length: 48
Content-Type: text/html
```

```
Date: Sat, 18 Feb 2017 04:50:23 GMT
Server: MSU/CSE422/SS17-Section001
============================================================
Connection served. Proxy child thread terminating.

./client -s http://www.cseeee.msu.edu/CSE422 -p ned.cse.msu.edu:38408
Request sent...
============================================================
GET /CSE422 HTTP/1.1
Connection: close
Host: www.cseeee.msu.edu
If-Modified-Since: 0
============================================================

Response header received
============================================================
HTTP/1.1 404 Not Found
Connection: close
Content-Length: 48
Content-Type: text/html
Date: Sat, 18 Feb 2017 04:50:23 GMT
Server: MSU/CSE422/SS17-Section001
============================================================

Downloading rest of the file ...
Default transfer encoding
Content-length: 48
bytes written:48
data gotten:48
404 Not Found
```