# WSN Lab (CN)
# Assignment 1 - HTTP Proxy

The due date for this assignment is January 31, 2021, i.e., by 23:59 hrs.

The submissions will be accepted through moodle only. Additional files available for the assignment on moodle are as follows:

- a2_skeleton.zip: This file contains the skeleton code which you can use. It also helps you to parse client http request for a webpage.
- proxy_tester.py; The python script that will be used to test your proxy run- ning on single thread.
- proxy_tester_conc.py; The python script that will be used to test your proxy accepting concurrent connections.

## Contents

# 1 Overview

In this assignment, you will implement a web proxy that passes requests and data between multiple web clients and web servers, concurrently. This will give you a chance to get to know one of the most popular application protocols on the Internet – the Hypertext Transfer Protocol (HTTP) – and give you more exposure to the Berkeley sockets API. When you're done with the assignment, you should be able to configure your web browser to use your personal proxy server as a web proxy.

# 2 Introduction: The Hypertext Transfer Protocol

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 assignment, 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 the instructions in the assignment should be self-contained.

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 Linux prompt, type:

```
telnet www.iiita.ac.in 80
```

This opens a TCP connection to the server at www.iiita.ac.in listening on port 80 (the default HTTP port). In case you get host unreachable message, specially in our lab, then append "nameserver 172.31.1.21" (without quotations) in your /etc/resolv.conf file. You should see something like this:

```
Trying 172.31.1.34...
Connected to
www.iiita.ac.in. 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/1.1 200 OK
Date: Tue, 3 Sep 2020
07:29:24 GMT Server:
Apache/2.2.15 (CentOS)
X-Powered-By: PHP/5.3.3
Connection: close
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html5>
<html lang='en'>
<head>

(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.

## 2.1 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).

## 2.2 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

# 3 Assignment Details

## 3.1 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 must be completed in C or C++. It should compile and run (using g++) without errors, producing a binary called proxy that takes as its first argument a port to listen from. Do not 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.

## 3.2 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. There should be a reasonable limit on the number of processes that your proxy can create (e.g., 100). Once a client has connected, the proxy should read data from the client and then check for a properly-formatted HTTP request – but don't worry, we have provided you with libraries that parse the HTTP request lines and headers. 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) – as your browser will send if properly configured to explicitly use a proxy (as opposed to a transparent on-path proxies that some ISPs deploy, unbeknownst to their users). An invalid request from the client should be answered with an appropriate error code, i.e., "Bad Request" (400) or "Not Imple- mented" (501) for valid HTTP methods other than GET. Similarly, if headers are not properly formatted for parsing, your client should also generate a type-400 message.

## 3.3 Parsing Library

We have provided 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 and value corresponding to the header. You can lookup headers by the key and modify them. The library can also recompile the headers into a string given the information in the structs.

More details as well as sample usage is available in proxy_parse.h, as well as example code on how to use the library. 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.

## 3.4 Parsing the URL

Once the proxy sees a valid HTTP request, it will need to parse the requested URL. The proxy needs at least three pieces of information: the requested host and port, and the requested path. See the URL (7) manual page for more info. You will need to parse the absolute URL specified in the request line using the given 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.

## 3.5 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.iiita.ac.in/
HTTP/1.0
```

Send to remote server:

```
GET         /
HTTP/1.0
Host:
www.iiita.ac.in
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, as shown. To add new headers or modify existing ones, use the HTTP Request Parsing Library we provide.

## 3.6 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 it's end of the connection after receiving the response. However, checking this is not required in this assignment 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, we wanted to simplify 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:

- 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.)
- 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.

### 3.7 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:

- Go to the 'Edit' menu.
- Select 'Preferences'. Select 'Advanced' and then select 'Network'.
- Under 'Connection', select 'Settings...'.
- 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.
- Save your changes by selecting 'OK' in the connection tab and then select 'Close' in the preferences tab.

## 4 Socket and Multi-Process Programming

For socket programming refer to Guide to Network Programming Using Sockets (al- ready sent to you and will be available on moodle). In addition to the Berkeley sockets library, there are some functions you will need to use for creating and manag- ing multiple processes: fork and waitpid. Refer to the chat application link and Fork Wikipedia page.

## 5 Grading

You should submit your completed proxy on your Section moodle page. 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 README file describing your code and the design decisions that you made.

Your tarball should be named ass2.tgz. The sample Makefile in the skeleton zip file we provide will make this tarball for you with the make tar command. Your proxy will be graded out of twenty points, with the following criteria:

- When running make on your assignment, it should compile without errors and produce a binary named proxy. The first command line argument should be the port that the proxy will listen from.
- Your proxy should run silently: any status messages or diagnostic output should be off by default.
- Your proxy should work with Firefox.
- We'll first check that your proxy works correctly with a small number of major web pages, using the same scripts that we've given you to test your proxy. If your proxy passes all of these 'public' tests, you will get 10 of the possible points.
- 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 5 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.
- You'll get 2 additional points for writing a reasonable README.
- Well written code will get 3 additional points – for readability, error checking, and comments – for a total of 20 points.