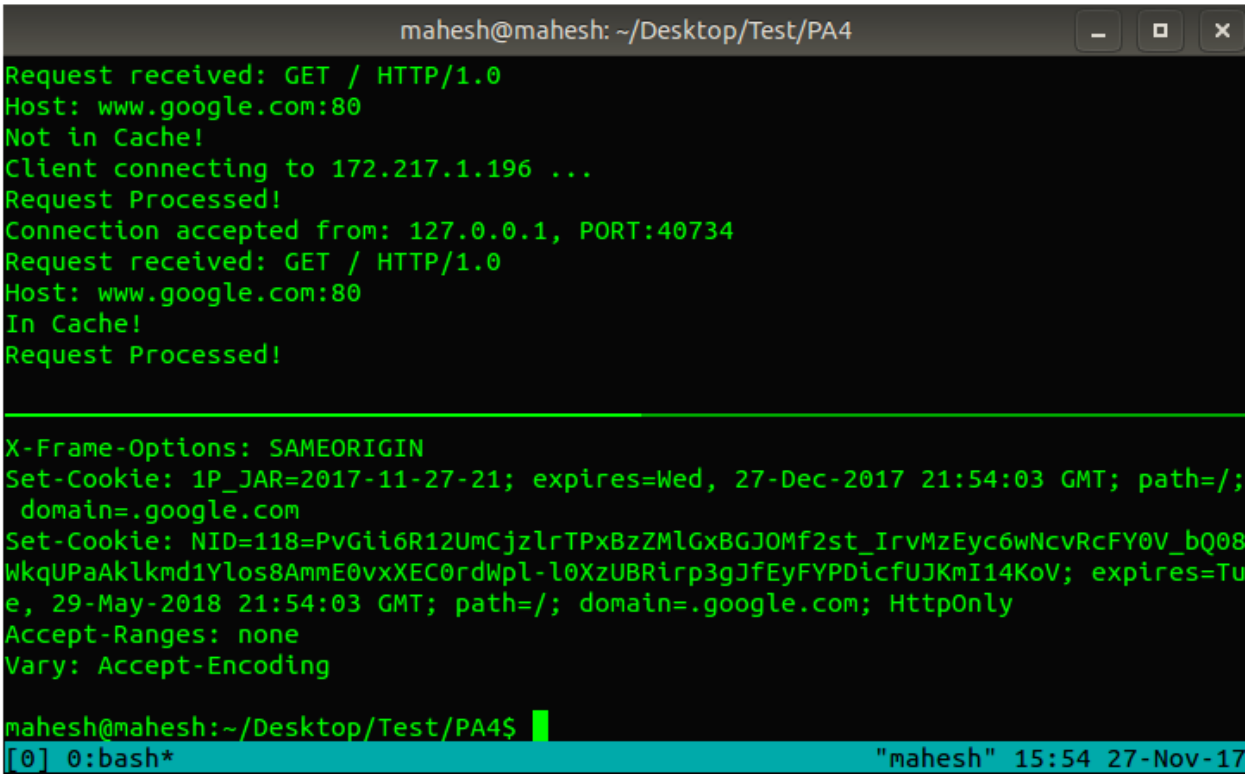
Read me:

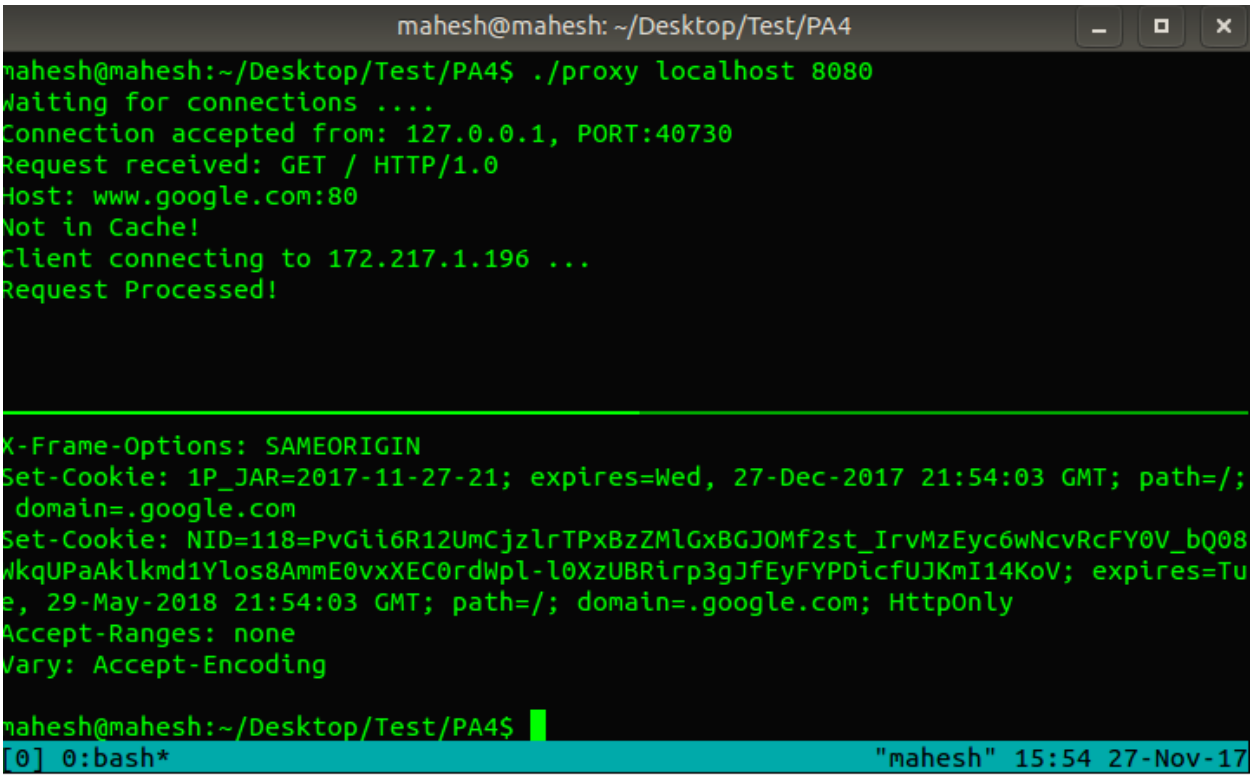
In this programming assignment we program an HTTP Proxy Server. HTTP proxy server is a type of server which sits in between the host and the server on the Internet. This assignment is based on the HTML 1.1. Every time a host wants to make a request it has to establish a new TCP connection, so we end up wasting 2 round trip time even before the required date is requested. This makes the connection very slow when the server is sitting far away from the client. Hence, some people created the concept of proxy server to cater frequently used pages faster. Every time a client/host makes a request for some webpage, the proxy server checks for that webpage to see if that request has been accessed earlier. If that request was made earlier proxy server would have cached that page in the memory, so it instead of sending the request to the server on internet it directly caters the request. Therefore, this makes it faster to access the webpages. If the webpage is not present in the Cache, or expired. The proxy server will forward the request to required host. After, the server receives the response from the external host. It caches the result to serve the similar requests in future. Every time it caches the page on cache it sets the expire time field to make sure that no client is served with the expired data. Our proxy server only stores the 10 least recently used entries into the cache.

Test Cases:

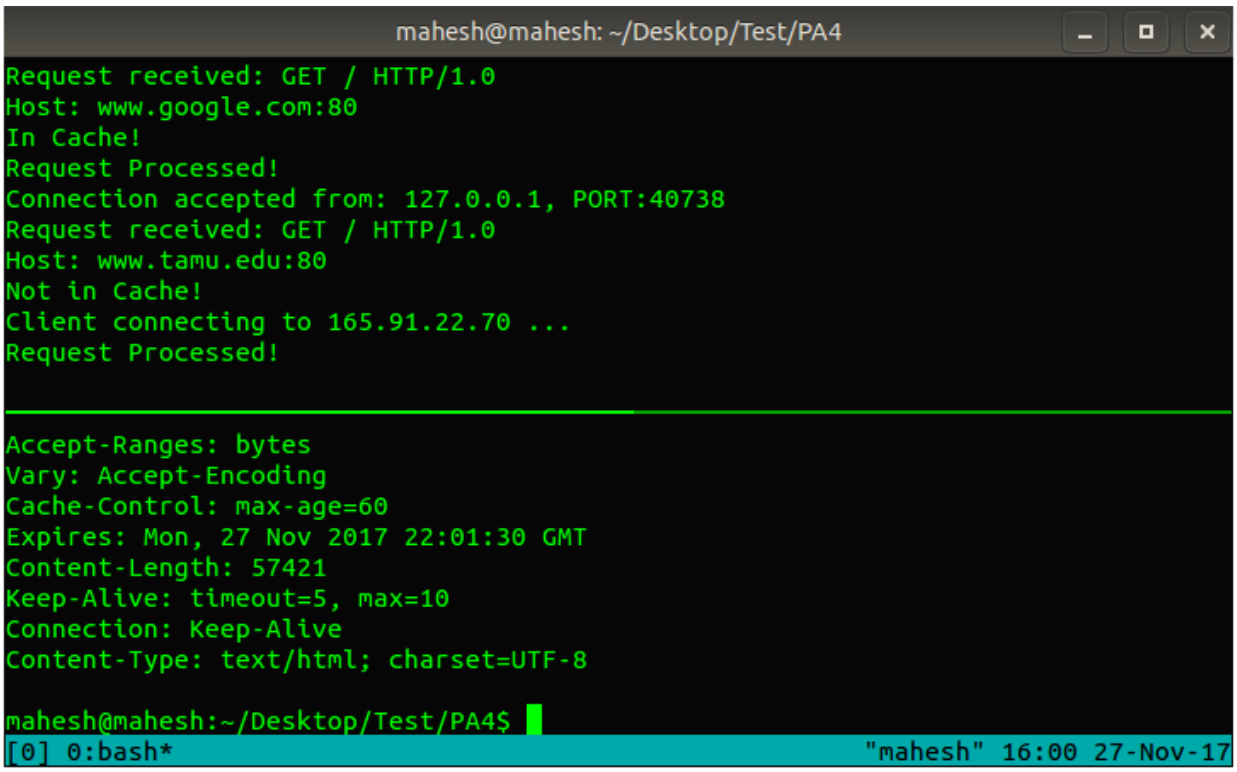
Case (1) a cache hit returns the saved data to the requester



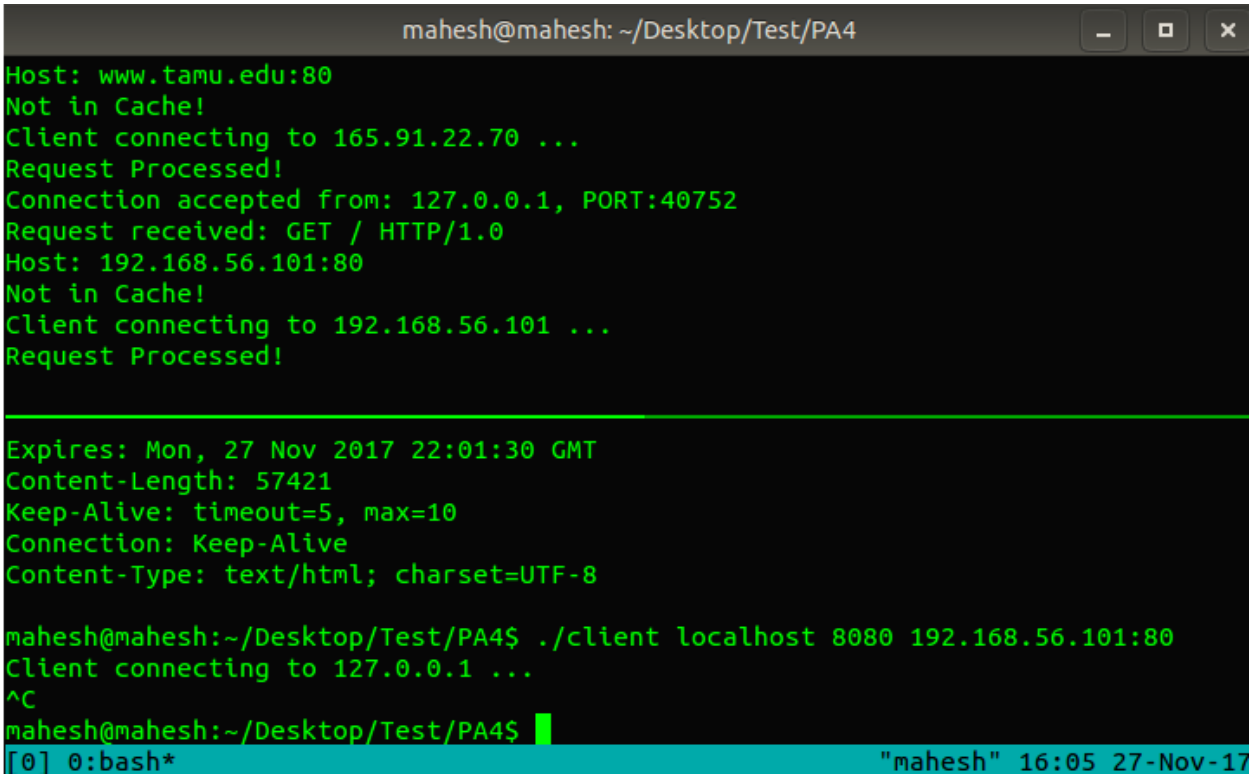
Case (2) a request that is not in the cache is proxied, saved in the cache, and returned to the requester:



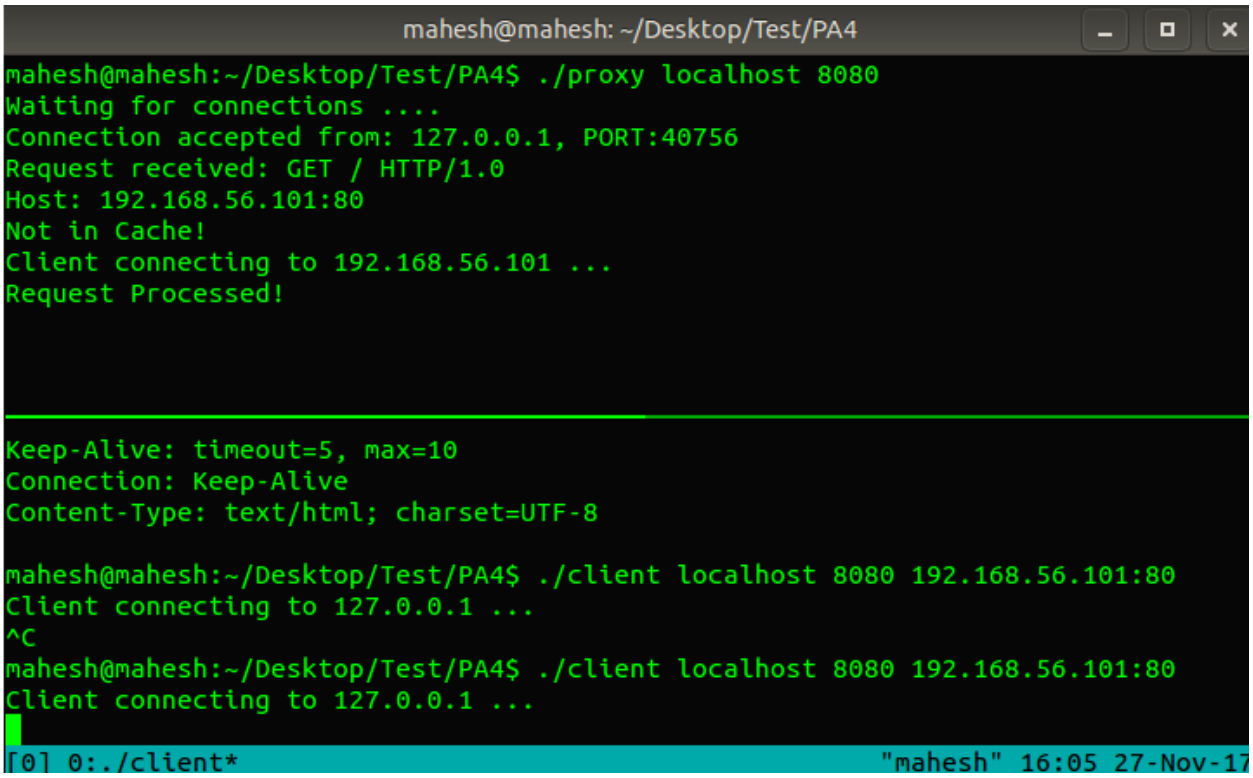
Case (3) a cache miss with 10 items already in the cache is proxied, saved in the LRU location in cache, and the data is returned to the requester



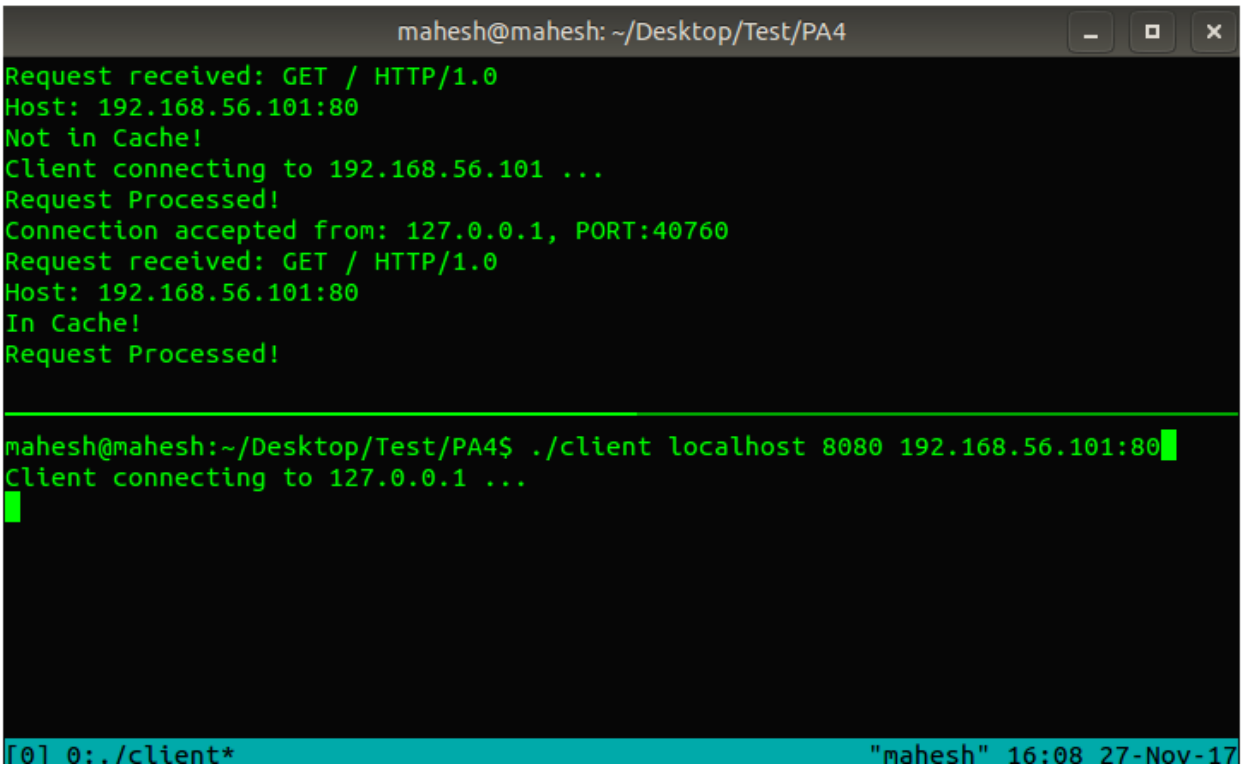
Case (4) A stale Expires header in the cache is accessed, the cache entry is replaced with a fresh copy, and the fresh data is delivered to the requester



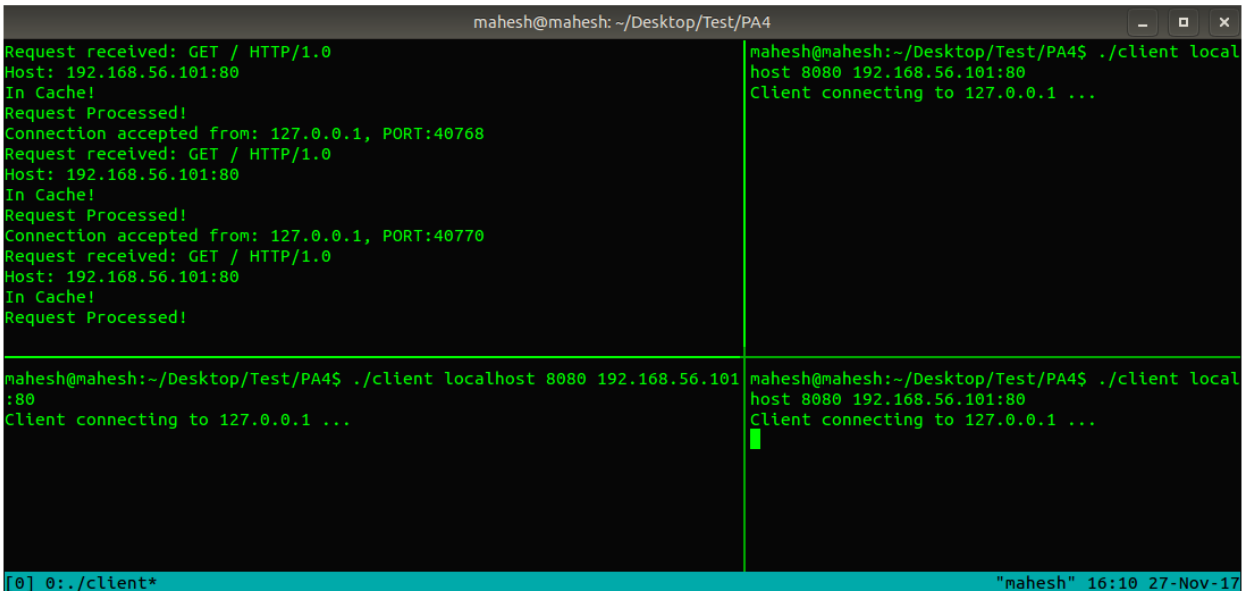
Case (5) A stale entry in the cache without an Expires header is determined based on the last Web server access time and last modification time, the stale cache entry is replaced with fresh data, and the fresh data is delivered to the requester



Case (6) A cache entry without an Expires header that has been previously accessed from the Web server in the last 24 hours and was last modified more than one month ago is returned to the requester



Case (7) Three clients can simultaneously access the proxy server and get the correct data



Code:

MakeFile:

CC=g++

```
all: proxy client

proxy: server.cpp
	$(CC) -o proxy server.cpp -lpthread
client: client.cpp
	$(CC) -o client client.cpp

.PHONY: clean

clean:
	rm -f proxy client
```

Server:

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <netdb.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string>
#include <arpa/inet.h>
#include <pthread.h>
#include <map>

using namespace std;

#define MAX_CLIENT_MSG_LEN 100
#define MAX_CLIENTS 10

struct html {
	string header;
	string content;
};

map<string, html> cache;

// Signal Handler for Child Process
void sigchld_handler(int signo);
// Thread that handles every client
void *client_handler(void *);
// Gets a webpage from the internet
html get_url(string, string);
```

```
int main(int argc, char const *argv[])
{
	int sd, new_sd;
	int err;
	int yes=1;
	int client_port;
	int bytes_recv;
	struct addrinfo addr_info, *server_info, *it;
	struct sigaction sig;
	struct sockaddr_storage client_addr;
	socklen_t sockaddr_size;
	char ip_addr[INET6_ADDRSTRLEN];
	char *msg, client_msg[MAX_CLIENT_MSG_LEN];
	void *client_ip;
	pthread_t handler;

	// Populate the address structure
	memset(&addr_info, 0, sizeof(addr_info));
	// IPv4 IPv6 independent
	addr_info.ai_family = AF_UNSPEC;
	// Use TCP
	addr_info.ai_socktype = SOCK_STREAM;
	addr_info.ai_flags = AI_PASSIVE;

	// Check command line arguments
	if (argc < 3) {
		printf("Invalid usage. Try: \n");
		printf("$ ./proxy <ip to bind> <port to bind>\n");
		return 1;
	}

	if ((err = getaddrinfo(argv[1], argv[2], &addr_info, &server_info)) != 0) {
		fprintf(stderr, "getaddrinfo() failed: %s\n", gai_strerror(err));
		return 1;
	}

	// Iterate through output of getaddrinfo() and find a port to bind to
	for (it = server_info; it != NULL; it = it->ai_next) {
		sd = socket(it->ai_family, it->ai_socktype, it->ai_protocol);
		if (sd == -1) {
			printf("%s:%d ", __FILE__, __LINE__);
			fflush(stdout);
			perror("socket()");
			continue;
		}
		if (setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1) {
```

```
                printf("%s:%d ", __FILE__, __LINE__);
                fflush(stdout);
                perror("setsockopt()");
                return 1;
        }
        if (bind(sd, it->ai_addr, it->ai_addrlen) == -1) {
                close(sd);
                printf("%s:%d ", __FILE__, __LINE__);
                fflush(stdout);
                perror("bind()");
                // If bind was unsuccesssful, try the next port
                // Note: In this case, there happens to be only one port and
                // so the list server_info only has one element
                continue;
        }
        break;
}

freeaddrinfo(server_info);

// Check if server successfully binded to the given port
if (it == NULL) {
        fprintf(stderr, "Server failed to bind to given port.\n");
        return 1;
}

// Listen on the port
if (listen(sd, MAX_CLIENTS) == -1) {
        printf("%s:%d ", __FILE__, __LINE__);
        fflush(stdout);
        perror("listen()");
        return 1;
}

// Register the signal handler for SIGCHLD event
// so that the resulting process is not a zombie while exiting
sig.sa_handler = sigchld_handler;
sigemptyset(&sig.sa_mask);
sig.sa_flags = SA_RESTART;
if (sigaction(SIGCHLD, &sig, NULL) == -1) {
        printf("%s:%d ", __FILE__, __LINE__);
        fflush(stdout);
        perror("sigaction()");
        return 1;
}
```

```
	printf("Waiting for connections ....\n");

	while (1) {
		sockaddr_size = sizeof(client_addr);
		// Accept incoming connection
		new_sd = accept(sd, (struct sockaddr *)&client_addr, &sockaddr_size);
		if (new_sd == -1) {
			printf("%s:%d ", __FILE__, __LINE__);
			fflush(stdout);
			perror("accept()");
			continue;
		}

		// Print the client IP Address and Port Number
		if ((*(struct sockaddr *)&client_addr).sa_family == AF_INET) {
			client_ip = &(((struct sockaddr_in *)&client_addr)->sin_addr);
		} else {
			client_ip = &(((struct sockaddr_in6 *)&client_addr)->sin6_addr);
		}
		client_port = ntohs((*(struct sockaddr_in *)&client_addr).sin_port);
		inet_ntop(client_addr.ss_family, client_ip, ip_addr, INET6_ADDRSTRLEN);
		printf("Connection accepted from: %s, PORT:%d\n", ip_addr, client_port);

		// Create a child thread to handle the client
		if (pthread_create(&handler, NULL, client_handler, (void *)&new_sd) < 0) {
			printf("%s:%d ", __FILE__, __LINE__);
			fflush(stdout);
			perror("pthread_create()");
			return 1;
		}
	}
}

// SIGCHLD handler
void sigchld_handler(int signo)
{
	int saved_errno = errno;
	while(waitpid(-1, NULL, WNOHANG) > 0);
	errno = saved_errno;
}

// This thread handles each client connected
void *client_handler(void *socket)
{
	int sd = *(int *)socket;
	int bytes_recv;
```

```
int index;
char *msg;
char client_msg[MAX_CLIENT_MSG_LEN];
struct html page;

// Wait for client to send data
while ((bytes_recv = recv(sd, client_msg, MAX_CLIENT_MSG_LEN, 0)) > 0) {
	client_msg[bytes_recv] = '\0';

	printf("Request received: %s\n", client_msg);
	index = 0;
	for (int i = 1; i < bytes_recv; i++) {
		if ((client_msg[i-1] == 'H') && (client_msg[i] == 'o')) {
			index = i;
			break;
		}
	}
	string host = "";
	string ip = "";
	string port = "";
	for (int i = index+5; i < bytes_recv; i++) {
		host += client_msg[i];
	}

	for (int i = 0; i < host.size(); i++) {
		if (host[i] == ':') {
			index = i;
			break;
		}
		ip += host[i];
	}
	for (int i = index+1; i < host.size(); i++) {
		port += host[i];
	}

	if (cache.find(host) != cache.end()) {
		page = cache[host];
		printf("In Cache!\n");
	} else {
		printf("Not in Cache!\n");
		page = get_url(ip, port);
		cache[host] = page;
	}

	string data = page.header + page.content;
	if (send(sd, data.c_str(), 1000000, 0) == -1) {
```

```
                        perror("send()");
                }

                printf("Request Processed!\n");
                while(1);

        }
        // If client closes connection, print it on standard output
        if (bytes_recv == 0) {
                printf("Client Closed Connection ...\n");
//              printf("Client closed connection: %s, PORT:%d\n", ip_addr, client_port);
                fflush(stdout);
        }
        // If recv() fails, output with error code
        else {
                perror("recv()");
        }

        // Free object
        close(sd);
        pthread_exit(NULL);
}


html get_url(string ip, string port)
{
        int sd;
        int err;
        int bytes_recv;
        struct addrinfo addr_info, *server_info, *it;
        char ip_addr[INET6_ADDRSTRLEN];
        char msg[MAX_CLIENT_MSG_LEN];
        void *client_ip;
        bool end;
        int index;
        struct html page;
        string request;
        string data;

        // TCP, IPv4/6
        memset(&addr_info, 0, sizeof(addr_info));
        addr_info.ai_family = AF_UNSPEC;
        addr_info.ai_socktype = SOCK_STREAM;

        page.header = "";
        page.content = "";
```

```
// Get a Port Number for the IP
if ((err = getaddrinfo(ip.c_str(), port.c_str(), &addr_info, &server_info)) != 0) {
	fprintf(stderr, "getaddrinfo() failed: %s\n", gai_strerror(err));
	return page;
}


// Check if we got a port number (In this case, we only have one choice)
for (it = server_info; it != NULL; it = it->ai_next) {
	sd = socket(it->ai_family, it->ai_socktype, it->ai_protocol);
	if (sd == -1) {
		printf("%s:%d ", __FILE__, __LINE__);
		fflush(stdout);
		perror("socket()");
		continue;
	}
	if (connect(sd, it->ai_addr, it->ai_addrlen) == -1) {
		close(sd);
		printf("%s:%d ", __FILE__, __LINE__);
		fflush(stdout);
		perror("connect()");
		continue;
	}
	break;
}

if (it == NULL) {
	fprintf(stderr, "Client failed to connect!\n");
	return page;
}

if ((*(struct sockaddr *)it->ai_addr).sa_family == AF_INET) {
	client_ip = &(((struct sockaddr_in *)it->ai_addr)->sin_addr);
} else {
	client_ip = &(((struct sockaddr_in6 *)it->ai_addr)->sin6_addr);
}

inet_ntop(it->ai_family, client_ip, ip_addr, INET6_ADDRSTRLEN);
printf("Client connecting to %s ...\n", ip_addr);

freeaddrinfo(server_info);

request = "GET / HTTP/1.0\r\nHost: ";
request += ip;
request += ":";
```

```
request += port;
request += "\r\n\r\n";

if (send(sd, request.c_str(), request.size(), 0) == -1) {
	perror("send()");
}


data = "";
while (1) {
	memset(msg, 0, strlen(msg));
	bytes_recv = recv(sd, msg, MAX_CLIENT_MSG_LEN, 0);
	data += msg;
	fflush(stdout);

	end = false;
	for (int i = 6; i < bytes_recv; i++) {
		if ((msg[i-6] == '<') && (msg[i-5] == '/'))
			if ((msg[i-4] == 'h') && (msg[i-3] == 't'))
				if ((msg[i-2] == 'm') && (msg[i-1] == 'l'))
					if (msg[i] == '>') {
						end = true;
						break;
					}
	}

	if (end) {
		break;
	}
}

for (int i = 3; i < data.size(); i++) {
	if ((data[i-3] == '\r') && (data[i-2] == '\n'))
		if ((data[i-1] == '\r') && (data[i] == '\n')) {
			index = i;
			end = true;
			break;
		}
}

page.header = "";
for (int i = 0; i < index; i++) {
	page.header += data[i];
}
page.header += "\n";
```

```
	page.content = "";
	for (int i = index+1; i < data.size(); i++) {
		page.content += data[i];
	}

	close(sd);
	return page;
}
```

Client:

```
#include <stdio.h>
#include <string.h>
#include <string>
#include <netdb.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <stdlib.h>

using namespace std;

#define MAX_CLIENT_MSG_LEN 1000

struct html {
	string header;
	string content;
};

html get_url(string, string);

int main(int argc, char const *argv[])
{
	int sd;
	int err;
	int bytes_recv;
	struct addrinfo addr_info, *server_info, *it;
	char ip_addr[INET6_ADDRSTRLEN];
	char msg[MAX_CLIENT_MSG_LEN];
	void *client_ip;
	bool end;
	int index;

	// Check Input Parameters
	if (argc < 4) {
		printf("Invalid usage. Try: \n");
		printf("$ ./client <proxy address> <proxy port> <URL to retrieve>\n");
```

```
        return 1;
}

// TCP, IPv4/6
memset(&addr_info, 0, sizeof(addr_info));
addr_info.ai_family = AF_UNSPEC;
addr_info.ai_socktype = SOCK_STREAM;

// Get a Port Number for the IP
if ((err = getaddrinfo(argv[1], argv[2], &addr_info, &server_info)) != 0) {
        fprintf(stderr, "getaddrinfo() failed: %s\n", gai_strerror(err));
        return 1;
}

// Check if we got a port number (In this case, we only have one choice)
for (it = server_info; it != NULL; it = it->ai_next) {
        sd = socket(it->ai_family, it->ai_socktype, it->ai_protocol);
        if (sd == -1) {
                printf("%s:%d ", __FILE__, __LINE__);
                fflush(stdout);
                perror("socket()");
                continue;
        }
        if (connect(sd, it->ai_addr, it->ai_addrlen) == -1) {
                close(sd);
                printf("%s:%d ", __FILE__, __LINE__);
                fflush(stdout);
                perror("connect()");
                continue;
        }
        break;
}

if (it == NULL) {
        fprintf(stderr, "Client failed to connect!\n");
        return 2;
}

if ((*(struct sockaddr *)it->ai_addr).sa_family == AF_INET) {
        client_ip = &(((struct sockaddr_in *)it->ai_addr)->sin_addr);
} else {
        client_ip = &(((struct sockaddr_in6 *)it->ai_addr)->sin6_addr);
}

inet_ntop(it->ai_family, client_ip, ip_addr, INET6_ADDRSTRLEN);
printf("Client connecting to %s ...\n", ip_addr);
```

```
freeaddrinfo(server_info);


string request = "GET / HTTP/1.0\r\nHost: ";
request += argv[3];
struct html page;

if (send(sd, request.c_str(), request.size(), 0) == -1) {
        perror("send()");
}

while(1) {
        memset(msg, 0, strlen(msg));
        bytes_recv = recv(sd, msg, MAX_CLIENT_MSG_LEN, 0);

        end = false;
        index = 0;
        for (int i = 3; i < bytes_recv; i++) {
                if ((msg[i-3] == '\r') && (msg[i-2] == '\n'))
                        if ((msg[i-1] == '\r') && (msg[i] == '\n')) {
                                index = i;
                                end = true;
                                break;
                        }
        }

        page.header = "";
        for (int i = 0; i < index; i++) {
                page.header += msg[i];
        }

        if (end) {
                end = false;
                page.content = "";
                for (int i = index+1; i < bytes_recv; i++) {
                        page.content += msg[i];
                }
                break;
        }
}

while (1) {
        memset(msg, 0, strlen(msg));
        bytes_recv = recv(sd, msg, MAX_CLIENT_MSG_LEN, 0);
        page.content += msg;
```

```
                end = false;
                for (int i = 6; i < bytes_recv; i++) {
                        if ((msg[i-6] == '<') && (msg[i-5] == '/'))
                                if ((msg[i-4] == 'h') && (msg[i-3] == 't'))
                                        if ((msg[i-2] == 'm') && (msg[i-1] == 'l'))
                                                if (msg[i] == '>') {
                                                        end = true;
                                                        break;
                                                }
                }

                if (end) {
                        break;
                }
        }

        printf("%s\n", page.header.c_str());
        fflush(stdout);
        FILE *fp;
        fp = fopen("file.html", "w");
        fputs(page.content.c_str(), fp);
        fclose(fp);

        close(sd);
        return 0;
}

html get_url(string ip, string port)
{
        int sd;
        int err;
        int bytes_recv;
        struct addrinfo addr_info, *server_info, *it;
        char ip_addr[INET6_ADDRSTRLEN];
        char msg[MAX_CLIENT_MSG_LEN];
        void *client_ip;
        bool end;
        int index;
        struct html page;
        string request;

        // TCP, IPv4/6
        memset(&addr_info, 0, sizeof(addr_info));
        addr_info.ai_family = AF_UNSPEC;
        addr_info.ai_socktype = SOCK_STREAM;
```

```
page.header = "";
page.content = "";

// Get a Port Number for the IP
if ((err = getaddrinfo(ip.c_str(), port.c_str(), &addr_info, &server_info)) != 0) {
	fprintf(stderr, "getaddrinfo() failed: %s\n", gai_strerror(err));
	return page;
}


// Check if we got a port number (In this case, we only have one choice)
for (it = server_info; it != NULL; it = it->ai_next) {
	sd = socket(it->ai_family, it->ai_socktype, it->ai_protocol);
	if (sd == -1) {
		printf("%s:%d ", __FILE__, __LINE__);
		fflush(stdout);
		perror("socket()");
		continue;
	}
	if (connect(sd, it->ai_addr, it->ai_addrlen) == -1) {
		close(sd);
		printf("%s:%d ", __FILE__, __LINE__);
		fflush(stdout);
		perror("connect()");
		continue;
	}
	break;
}

if (it == NULL) {
	fprintf(stderr, "Client failed to connect!\n");
	return page;
}

if ((*(struct sockaddr *)it->ai_addr).sa_family == AF_INET) {
	client_ip = &(((struct sockaddr_in *)it->ai_addr)->sin_addr);
} else {
	client_ip = &(((struct sockaddr_in6 *)it->ai_addr)->sin6_addr);
}

inet_ntop(it->ai_family, client_ip, ip_addr, INET6_ADDRSTRLEN);
printf("Client connecting to %s ...\n", ip_addr);

freeaddrinfo(server_info);
```

```
request = "GET / HTTP/1.1\r\nHost: ";
request += ip;
request += ":";
request += port;
request += "\r\n\r\n";

if (send(sd, request.c_str(), request.size(), 0) == -1) {
        perror("send()");
}

while(1) {
        memset(msg, 0, strlen(msg));
        bytes_recv = recv(sd, msg, MAX_CLIENT_MSG_LEN, 0);

        end = false;
        index = 0;
        for (int i = 3; i < bytes_recv; i++) {
                if ((msg[i-3] == '\r') && (msg[i-2] == '\n'))
                        if ((msg[i-1] == '\r') && (msg[i] == '\n')) {
                                index = i;
                                end = true;
                                break;
                        }
        }

        page.header = "";
        for (int i = 0; i < index; i++) {
                page.header += msg[i];
        }

        if (end) {
                end = false;
                page.content = "";
                for (int i = index+1; i < bytes_recv; i++) {
                        page.content += msg[i];
                }
                break;
        }
}

while (1) {
        memset(msg, 0, strlen(msg));
        bytes_recv = recv(sd, msg, MAX_CLIENT_MSG_LEN, 0);
        page.content += msg;

        end = false;
```

```
                for (int i = 6; i < bytes_recv; i++) {
                        if ((msg[i-6] == '<') && (msg[i-5] == '/'))
                                if ((msg[i-4] == 'h') && (msg[i-3] == 't'))
                                        if ((msg[i-2] == 'm') && (msg[i-1] == 'l'))
                                                if (msg[i] == '>') {
                                                        end = true;
                                                        break;
                                                }
                }

                if (end) {
                        break;
                }
        }

        close(sd);
        return page;
}
```