Prof. Heonyoung Yeom

System Programming (001)

Lab #5 / Proxy Lab

: Writing a Caching Web Proxy

Report

Dept. of Computer Science and Engineering
2021-13194
Jaejun Ko

2023.6.18

Lab #5 Report

Proxy Lab : Writing a Caching Web Proxy

Introduction

In this lab session, the assignment requires implementing a web proxy. It should be able to deal with multiple concurrent requests, and cache web objects to retrieve the web objects not accessing the server directly. The only method implemented in this implementation is HTTP/1.0 GET, which gets web objects from the server.

Specification

Part 1. Implementing a sequential web proxy

In part 1, it is required to implement a basic sequential proxy that handles HTTP/1.0 GET requests. When started, the proxy should listen for incoming connections on a port whose number will be specified on the command line. Once a connection is established, the proxy should read the entirety of the request from the client and parse the request. It should determine whether the client has sent a valid HTTP request; if so, it can then establish its connection to the appropriate web server then request the object the client specified. Finally, your proxy should read the server's response and forward it to the client.

I. HTTP/1.0 GET requests

When an end user enters a URL into the address bar of a web browser, the browser will send an HTTP request to the proxy having the following form:

GET http://*hostname*[:*port*][/*path*][?*query*][...] HTTP/1.1

In that case, the proxy should parse the request into at least the following fields: the hostname, path if it exists, and the path or query and everything following it. It should be an absolute path, not a relative path. That way, the proxy can determine that it should open a connection to the host and send an HTTP request with the relative path. Although Modern web browsers will generate HTTP/1.1 requests, this proxy handles them and forwards them as HTTP/1.0 requests.

II. Request headers

The important request headers for this lab are the Host, User-Agent, Connection, and Proxy-Connection headers. It should always send the following headers:

Host: *hostname*
Connection: close
Proxy-Connection: close

Web browsers may attach their Host headers to their HTTP requests. If that is the case, your proxy should use the same "Host" header as the browser. The "Connection" and "Proxy-Connection" headers are used to specify whether a connection will be kept alive after the first request/response exchange is completed. It is perfectly acceptable (and suggested) to have your proxy open a new connection for each request. Specifying close as the value of these headers alerts web servers that your proxy intends to close connections after the first request/response exchange.

Sending the following User-Agent header is optional, which has the following form:

```
  User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:10.0.3)
Gecko/20120305 Firefox/10.0.3
```

The User-Agent header identifies the client (in terms of parameters such as the operating system and browser), and web servers often use the identifying information to manipulate the content they serve. Sending this header may improve, in content and diversity, the material that you get back during simple telnet-style testing.

Finally, if a browser sends any additional request headers as part of an HTTP request, your proxy should forward them unchanged.

III. Port numbers

There are two significant classes of port numbers for this lab: HTTP request ports and the proxy's listening port.

The HTTP request port is an optional field in the URL of an HTTP request. If the port is not specified on the URL, the default HTTP port, which is port 80, should be used. Your proxy must properly function whether the port number is included in the URL. The listening port is the port on which the proxy should listen for incoming connections, which is given by a command line argument.

Part 2. Dealing with multiple concurrent requests

In part 2, a working sequential proxy should be altered to simultaneously handle multiple requests. The simplest way to implement a concurrent server is to spawn a new thread to handle each new connection request. Other designs are also possible, such as the pre-threaded server. The threads should run in detached mode to avoid memory leaks.

Part 3. Caching web objects

For the final part of the lab, a cache must be added to the proxy that stores recently used Web objects in memory. When the proxy receives a web object from a server, it should cache it in memory as it transmits the object to the client. If another client requests the same object from the same server, the proxy need not reconnect to the server; it can simply resend the cached object.

If the proxy were to cache every object that is ever requested, it would require an unlimited amount of memory. Moreover, because some web objects are larger than others, it might be the case that one giant object will consume the entire cache, preventing other objects from being cached at all. To avoid those problems, the proxy should have both a maximum cache size and a maximum cache object size. The maximum cache size is 1 MiB, and the maximum object size is 100 KiB. When calculating the size of its cache, your proxy must only count bytes used to store the actual web objects; any extraneous bytes, including metadata, should be ignored. Also, the proxy should only cache web objects that do not exceed the limit.

The proxy's cache should employ an eviction policy that approximates a least-recently-used (LRU) eviction policy. It doesn't have to be strictly LRU, but it should be something reasonably close.

Furthermore, accesses to the cache must be thread-safe, and ensuring that cache access is free of race conditions will likely be the more interesting aspect of this part of the lab.

Implementation

Part 1. Implementing a sequential web proxy

There are some constant strings used in various parts of the proxy server code to construct the appropriate headers for the requests and to define default values.

`user_agent_hdr`: Represents the User-Agent header that will be sent in the requests forwarded by the proxy.

`connection_hdr`: Represents the Connection header that will be sent in the requests forwarded by the proxy. It specifies that the connection should be closed after the response is received.

`proxy_connection_hdr`: Represents the Proxy-Connection header that will be sent in the requests forwarded by the proxy. It also specifies that the connection should be closed after the response is received.

`server_version`: Represents the version of the HTTP protocol that the proxy server will use when communicating with the destination server. In this case, it is set to "HTTP/1.0".

`default_port`: Represents the default port number to be used if the requested URI does not explicitly specify a port. In this case, the default port is set to "80", which is the standard port for HTTP communication.

```
static const char *user_agent_hdr =
    "User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:10.0.3) Gecko/20120305 "
    "Firefox/10.0.3\r\n";
static const char *connection_hdr = "Connection: close\r\n";
static const char *proxy_connection_hdr = "Proxy-Connection: close\r\n";
static const char *server_version = "HTTP/1.0";
static const char *default_port = "80";
```

There are three functions to implement a sequential web proxy, `void doit(int fd)`, `int parse_uri(char *uri, char *host, char *port, char *path)`, `void convert_reqhdrs(char *reqhdrs, char *host, char *port).`

The `doit` function is responsible for handling a single HTTP request/response transaction. It handles the entire request/response cycle, including caching and forwarding requests to the server. Here is a summary of its main steps:

1. Read the request line and headers from the client.
2. Parse the request line to extract the method, URI, and HTTP version.
3. Check if the request method is "GET". If not, return an error message.
4. Parse the URI to extract the host, port, and path components using `parse_uri`.
5. Establish a connection with the server.
6. Convert the given request header using `convert_reqhdrs` and send the request line and headers to the server.
7. Read the response headers from the server and extract the content length.
8. Send the response headers to the client.
9. Read the content from the server and send it to the client.
10. Close the connection to the server.

```
void doit(int fd) {
  char req_buf[MAXLINE], reqhdrs[MAXLINE] = "";
  char method[MAXLINE], uri[MAXLINE], version[MAXLINE];
  char host[MAXLINE], port[MAXLINE], path[MAXLINE];
  char rsp_buf[MAXLINE] = "", rsp[MAXLINE] = "";
  int serverfd;
  rio_t proxy_req, server_rsp;
  char *content_buf = NULL;
  int content_length;
  Rio_readinitb(&proxy_req, fd);
  if (!Rio_readlineb(&proxy_req, req_buf, MAXLINE)) {
    return;
  }
  sscanf(req_buf, "%s %s %s", method, uri, version);
  while (strcmp(req_buf, "\r\n")) {
    Rio_readlineb(&proxy_req, req_buf, MAXLINE);
    strcat(reqhdrs, req_buf);
  }
  if (strcasecmp(method, "GET")) {
    printf("Proxy does not implement the method\n");
    return;
  }
  if (parse_uri(uri, host, port, path) == -1) {
    printf("Not http protocol\n");
    return;
  }
  while ((serverfd = Open_clientfd(host, port)) < 0)
    ;
  sprintf(req_buf, "%s %s %s\r\n", method, path, server_version);
  Rio_writen(serverfd, req_buf, strlen(req_buf));

  convert_reqhdrs(reqhdrs, host, port);
  Rio_writen(serverfd, reqhdrs, strlen(reqhdrs));
  Rio_readinitb(&server_rsp, serverfd);

  do {
    Rio_readlineb(&server_rsp, rsp_buf, MAXLINE);
    if (!strncasecmp(rsp_buf, "Content-Length:", 15)) {
      sscanf(rsp_buf + 15, "%d\r\n", &content_length);
    }
    strcat(rsp, rsp_buf);
  } while (strcmp(rsp_buf, "\r\n"));
  Rio_writen(fd, rsp, strlen(rsp));
  content_buf = (char *)Malloc(content_length);
  Rio_readnb(&server_rsp, content_buf, content_length);
  Rio_writen(fd, content_buf, content_length);
  Close(serverfd);
}
```

`parse_uri` takes a URI string as input and extracts the host, port, and path components. It returns 0 on success and -1 on error. The steps involved are as follows:

1. Check if the URI starts with "*http://*". If not, return an error.
2. Find the host by searching for the substring after "*http://*".
3. Find the end of the host, which can be a colon (indicating the presence of a port) or a forward slash (indicating the end of the host).
4. Extract the host by copying the characters between the start and end positions.
5. If a port is specified (after the colon), extract it similarly.
6. If no port is specified, use a default port 80.
7. Finally, extract the path by copying the characters from the end position until the end of the URI.

`convert_reqhdrs` converts the request headers by replacing specific headers with new ones. The input is the original request headers, host, and port. The function modifies the `reqhdrs` string in place. The steps involved are as follows:

1. Create a new string `new_hdrs` to hold the modified headers.
2. Construct the new headers by concatenating the host, port, user agent header, connection header, and proxy connection header.
3. Copy the other headers from the original headers to the new headers, excluding specific headers such as "Host", "User-Agent", "Connection", and "Proxy-Connection".
4. Append the end of the header marker ("`\r\n`") to the new headers.
5. Copy the new headers back to `reqhdrs` using `strncpy`.

```
int parse_uri(char *uri, char *host, char *port, char *path) {
  char *start = strstr(uri, "http://");
  if (start == NULL) {
    return -1;
  }
  start += 7;
  char *end = start;
  while (*end != ':' && *end != '/' && *end != '\0') {
    end++;
  }
  strncpy(host, start, end - start);
  if (*end == ':') {
    start = ++end;
    while (*end != '/' && *end != '\0') {
      end++;
    }
    strncpy(port, start, end - start);
  } else {
    strncpy(port, default_port, strlen(default_port));
  }
  strcpy(path, end);
  return 0;
}

void convert_reqhdrs(char *reqhdrs, char *host, char *port) {
  char new_hdrs[MAXLINE];
  char *line;
  sprintf(new_hdrs, "Host: %s:%s\r\n%s%s%s", host, port, user_agent_hdr,
          connection_hdr, proxy_connection_hdr);
  line = strtok(reqhdrs, "\r\n");
  while (line != NULL) {
    if (strncmp(line, "Host:", 5) != 0 &&
        strncmp(line, "User-Agent:", 11) != 0 &&
        strncmp(line, "Connection:", 11) != 0 &&
        strncmp(line, "Proxy-Connection:", 17) != 0) {
      strcat(new_hdrs, line);
      strcat(new_hdrs, "\r\n");
    }
    line = strtok(NULL, "\r\n");
  }
  strcat(new_hdrs, "\r\n");
  strncpy(reqhdrs, new_hdrs, MAXLINE);
}
```

The following is the `main` function for a server program. It sets up a server that listens on a specified port, accepts client connections, and delegates the handling of each connection to the `doit` function. Here is a summary of its functionality:

1. Check the command line arguments. If the number of arguments is not 2, it prints an error message with the correct usage and exits.
2. Listen on a specified port, whose number is provided as the first command line argument (`argv[1]`).
3. Initialize variables related to client connections: `connfd` to store the connection socket descriptor and `clientaddr` to store the client's address information.

   Enter a loop to continuously accept client connections and handle them. The loop continues indefinitely, accepting new connections as they arrive. Within the loop:
   a. Accept a client connection. Block until a client connects and returns a new socket descriptor (`connfd`) for the connection.
   b. Call the `doit` function to handle the client request with `connfd` as an argument.
   c. Close the connection socket after handling the request.

As it is a sequential web proxy, it handles one client connection at a time.

```
int main(int argc, char **argv) {
  int listenfd;
  int connfd;
  socklen_t clientlen;
  struct sockaddr_in clientaddr;
  pthread_t tid;

  if (argc != 2) {
    fprintf(stderr, "usage: %s <port>\n", argv[0]);
    exit(1);
  }
  listenfd = Open_listenfd(argv[1]);
  clientlen = sizeof(clientaddr);
  while (1) {
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
    doit(connfd);
    Close(connfd);
  }
}
```

Part 2. Dealing with multiple concurrent requests

To deal with multiple concurrent requests, threading is adopted. A thread function `void *thread(void *targ)` is designed to handle client connections concurrently by executing each client request in a separate thread. Here's a summary of what the code does:

1. Detach the thread to avoid memory leaks.
2. Obtain the connection file descriptor (`connfd`) is obtained from the `targ` argument, which is expected to be a pointer to an integer containing the file descriptor.
3. Free `targ` to release the memory allocated for the file descriptor.
4. Invoke the `doit` function to handle the client request using `connfd`.
5. Close the connection after the request is processed.
6. Return NULL to comply with the pthread library requirements.

```
void *thread(void *targ) {
  Pthread_detach(Pthread_self());
  int connfd = *((int *)targ);
  Free(targ);
  doit(connfd);
  Close(connfd);
  return NULL;
}
```

The main function is modified to handle multiple client connections using threads. Here's a summary of what is modified:

- Memory is dynamically allocated to store the file descriptor for the accepted connection. This is done to avoid a potential race condition where the value of `connfd` may be overwritten before the thread accesses it.
- The obtained file descriptor by `Accept` function is assigned to the memory location pointed to by `connfd`.
- A new thread is created using `Pthread_create`. The thread function used is the `thread`, and the `connfd` value is passed as an argument.

```
int main(int argc, char **argv) {
  int listenfd;
  int *connfd;
...
  while (1) {
    /* Accept connection (Malloc to avoid race condition) */
    connfd = Malloc(sizeof(int));
    *connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

    /* Create thread */
    Pthread_create(&tid, NULL, thread, connfd);
  }
}
```

Part 3. Caching web objects

There are some components to introduce a cache system for the proxy server.

1. `MAX_CACHE_SIZE`: Represents the maximum size of the cache in bytes. It is set to 1049000, indicating that the cache can hold up to approximately 1 MB of data.
2. `MAX_OBJECT_SIZE`: Represents the maximum size of an object that can be stored in the cache. It is set to 102400 bytes, indicating that objects larger than this size will not be cached.

3. `cache_blk` structure: Represents a cache block, which contains the following fields:

    a. `uri`: The URI associated with the cached object.
    b. `rsp`: The response headers of the cached object.
    c. `content`: The content (body) of the cached object.
    d. `content_length`: The length of the content in bytes.
    e. `next` and `prev`: Pointers to the next and previous cache blocks in the cache.
4. `cache_t` structure: Represents the cache itself, which contains the following fields:
    a. `size`: The current size of the cache in bytes.
    b. `head` and `tail`: Pointers to the head and tail cache blocks in the cache.
5. `proxy_cache`: Represents the cache object used by the proxy server. It is declared as a static variable to maintain its state across different function calls.
6. Mutex and semaphores: These are for synchronization mechanisms to ensure thread safety when accessing the cache. It includes:
    - `mutex`: A mutex lock used to protect critical sections of code that access the cache.
    - `w`: A semaphore used to control write access to the cache, allowing only one writer at a time.
    - `readcnt`: An integer variable used to keep track of the number of readers accessing the cache.

Furthermore, some functions are defined to interact with the cache, including:

- `init_cache`: Initializes the cache by setting the `size` to 0 and the `head` and `tail` pointers to `NULL`.
- `find_cache_blk`: Searches for a cache block with a given URI in the cache and returns a pointer to it if found. If not found, return `NULL`.
- `add_cache_blk`: Adds a new cache block to the cache with the specified URI, response headers, content, and content length.
- `enqueue`: Adds a cache block to the tail of the cache.
- `dequeue`: Removes the cache block at the head of the cache.

For each access to the cache, 1st reader-writer problem is used, which means this web proxy favors the reader. The following is the implementation of the functions:

```
void init_cache(cache_t *cache) {
  cache->size = 0;
  cache->head = NULL;
  cache->tail = NULL;
}

cache_blk *find_cache_blk(cache_t *cache, char *uri) {
  P(&mutex);
  readcnt++;
  if (readcnt == 1) {
    /* First reader */
    P(&w);
  }
  V(&mutex);
  cache_blk *blk = cache->head;
  while (blk != NULL) {
    if (!strcmp(blk->uri, uri)) {
      break;
    }
    blk = blk->next;
  }
  P(&mutex);
  readcnt--;
  if (readcnt == 0) {
    /* Last reader */
    V(&w);
  }
  V(&mutex);
  /* Re-insert cache block */
  if (blk != NULL) {
    P(&w);
    /* Remove cache block */
    if (blk->prev != NULL) {
      blk->prev->next = blk->next;
    } else {
      cache->head = blk->next;
    }
    if (blk->next != NULL) {
      blk->next->prev = blk->prev;
    } else {
      cache->tail = blk->prev;
    }
    blk->prev = NULL;
    blk->next = cache->head;
    if (cache->head != NULL) {
      cache->head->prev = blk;
    }
    cache->head = blk;
    V(&w);
  }
  return blk;
}

void enqueue(cache_t *cache, char *uri, char *rsp, char *content,
             int content_length) {
  cache_blk *blk = (cache_blk *)Malloc(sizeof(cache_blk));
  blk->uri = (char *)Malloc(strlen(uri) + 1);
  strcpy(blk->uri, uri);
  blk->rsp = (char *)Malloc(strlen(rsp) + 1);
  strcpy(blk->rsp, rsp);
  blk->content = content;
  blk->content_length = content_length;
  blk->next = NULL;
  blk->prev = NULL;
  if (cache->size == 0) {
    cache->head = blk;
    cache->tail = blk;
  } else {
    blk->next = cache->head;
    cache->head->prev = blk;
    cache->head = blk;
  }
  cache->size += content_length + strlen(rsp);
}

void dequeue(cache_t *cache) {
  if (cache->size == 0) {
    return;
  }
  /* dequeue cache block */
  cache_blk *blk = cache->tail;
  cache->tail = blk->prev;
  cache->tail->next = NULL;
  cache->size -= blk->content_length + strlen(blk->rsp);
  Free(blk->uri);
  Free(blk->rsp);
  Free(blk->content);
  Free(blk);
}

void add_cache_blk(cache_t *cache, char *uri, char *rsp, char *content,
                   int content_length) {
  cache_blk *blk = find_cache_blk(cache, uri);
  if (content_length > MAX_OBJECT_SIZE) {
    Free(content);
    return;
  } else if (blk != NULL) {
    Free(content);
    return;
  }
  P(&w);
  while (cache->size + content_length > MAX_CACHE_SIZE) {
    dequeue(cache);
  }
  enqueue(cache, uri, rsp, content, content_length);
  V(&w);
}
```

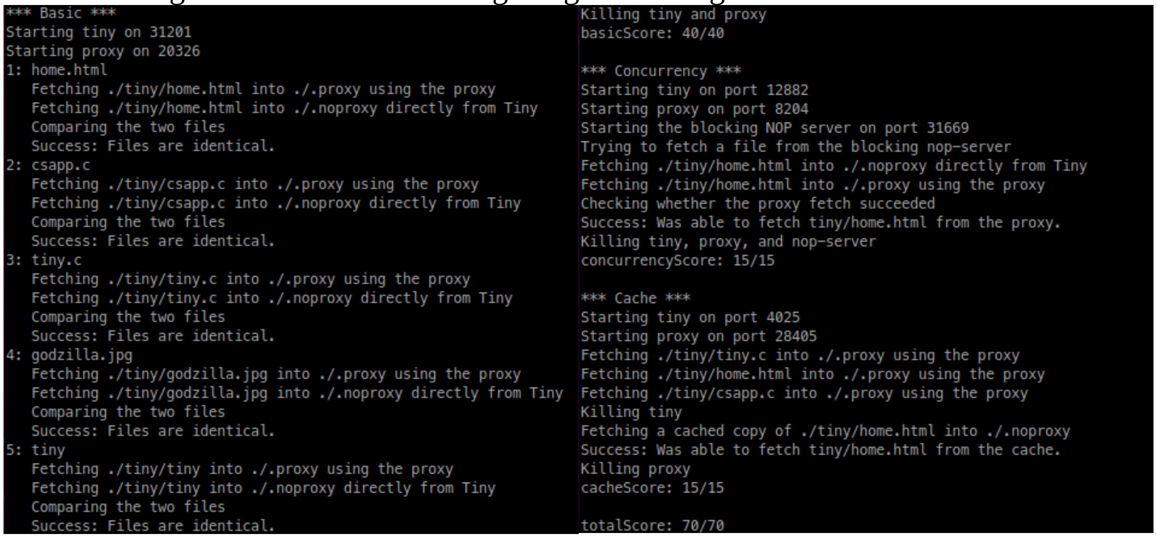
The above functions are used in the `doit` function. It performs the following new steps after modification:

- Check the cache for a cached response before connecting with the server. If found, send the cached response back to the client and return.
- If the requested object is not in the cache, establish a connection with the server.
- Add the response from the server to the cache after sending it to the client.

```
void doit(int fd) {
...
  if (parse_uri(uri, host, port, path) == -1) {
    printf("Not http protocol\n");
    return;
  }
  cache_blk *blk = find_cache_blk(proxy_cache, uri);
  if (blk) {
    Rio_writen(fd, blk->rsp, strlen(blk->rsp));
    Rio_writen(fd, blk->content, blk->content_length);
    return;
  }
  while ((serverfd = Open_clientfd(host, port)) < 0)
    ;
  sprintf(req_buf, "%s %s %s\r\n", method, path, server_version);
  Rio_writen(serverfd, req_buf, strlen(req_buf));
...
  Rio_writen(fd, content_buf, content_length);
  add_cache_blk(proxy_cache, uri, rsp, content_buf, content_length);
  Close(serverfd);
}
```

Result

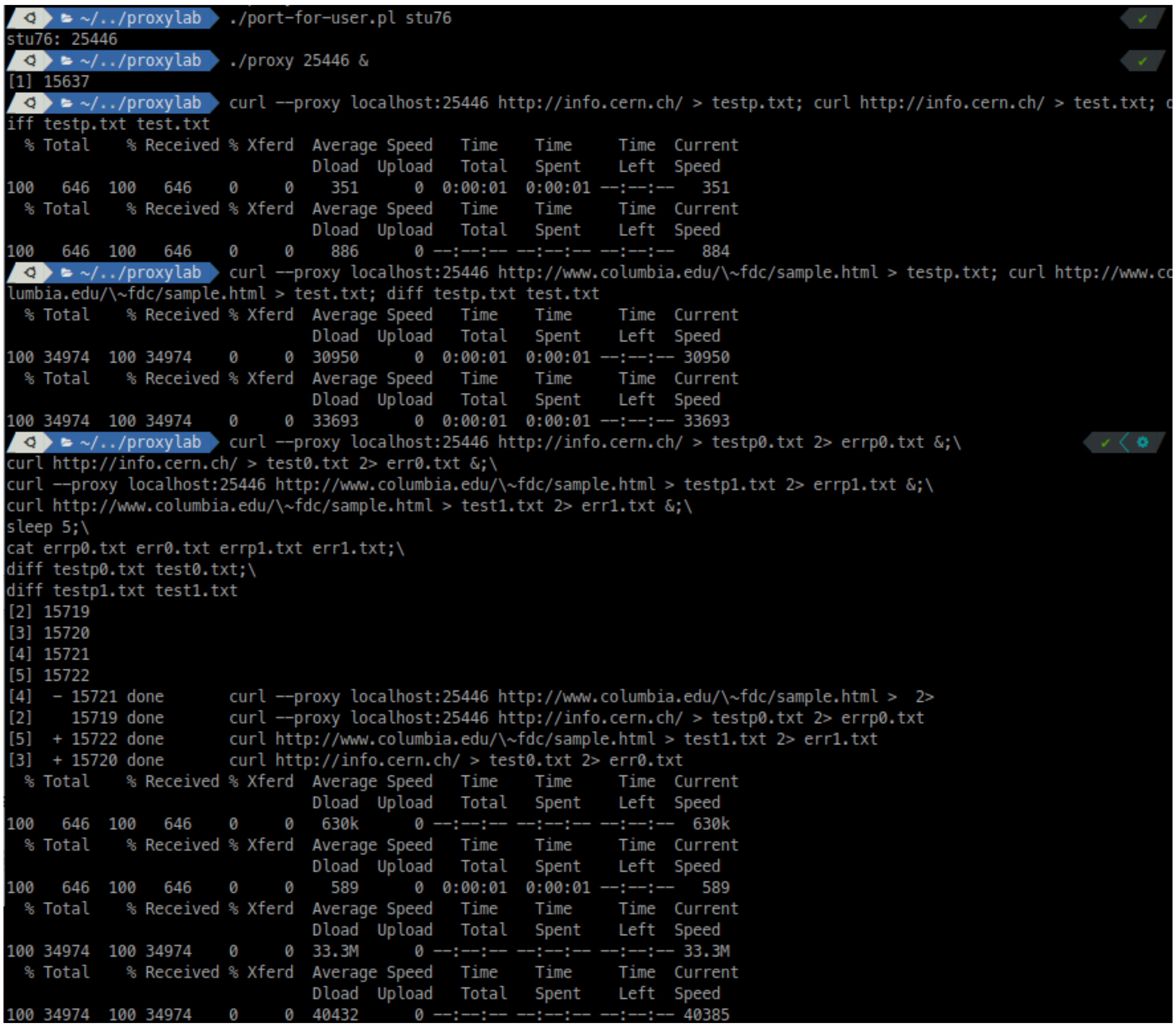
The following is the result of executing the given auto-grader.



```
*** Basic ***
Starting tiny on 31201
Starting proxy on 20326
1: home.html
   Fetching ./tiny/home.html into ./.proxy using the proxy
   Fetching ./tiny/home.html into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
2: csapp.c
   Fetching ./tiny/csapp.c into ./.proxy using the proxy
   Fetching ./tiny/csapp.c into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
3: tiny.c
   Fetching ./tiny/tiny.c into ./.proxy using the proxy
   Fetching ./tiny/tiny.c into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
4: godzilla.jpg
   Fetching ./tiny/godzilla.jpg into ./.proxy using the proxy
   Fetching ./tiny/godzilla.jpg into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
5: tiny
   Fetching ./tiny/tiny into ./.proxy using the proxy
   Fetching ./tiny/tiny into ./.noproxy directly from Tiny
   Comparing the two files
   Success: Files are identical.
Killing tiny and proxy
basicScore: 40/40

*** Concurrency ***
Starting tiny on port 12882
Starting proxy on port 8204
Starting the blocking NOP server on port 31669
Trying to fetch a file from the blocking nop-server
Fetching ./tiny/home.html into ./.noproxy directly from Tiny
Fetching ./tiny/home.html into ./.proxy using the proxy
Checking whether the proxy fetch succeeded
Success: Was able to fetch tiny/home.html from the proxy.
Killing tiny, proxy, and nop-server
concurrencyScore: 15/15

*** Cache ***
Starting tiny on port 4025
Starting proxy on port 28405
Fetching ./tiny/tiny.c into ./.proxy using the proxy
Fetching ./tiny/home.html into ./.proxy using the proxy
Fetching ./tiny/csapp.c into ./.proxy using the proxy
Killing tiny
Fetching a cached copy of ./tiny/home.html into ./.noproxy
Success: Was able to fetch tiny/home.html from the cache.
Killing proxy
cacheScore: 15/15

totalScore: 70/70
```

It passed all the tests given by the auto-grader, testing the basic functionality and the concurrency of the web proxy and caching the web contents.

Furthermore, it was tested for real web pages. The basic functionality of the web proxy, the concurrency of the web proxy, and caching are tested. The following is the result of applying the web proxy on the real webpage. It shows that the results using the proxy were slightly slower than those without the proxy, but after caching, it was much faster than not caching. Furthermore, we can check concurrency, if it were not the case, the result of *http://www.columbia.edu/\~fdc/sample.html* would not have appeared before the result of *http://info.cern.ch/*.



```
 ~/../proxylab  ./port-for-user.pl stu76
stu76: 25446
 ~/../proxylab  ./proxy 25446 &
[1] 15637
 ~/../proxylab  curl --proxy localhost:25446 http://info.cern.ch/ > testp.txt; curl http://info.cern.ch/ > test.txt; d
iff testp.txt test.txt
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   646  100   646    0     0    351      0  0:00:01  0:00:01 --:--:--   351
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   646  100   646    0     0    886      0 --:--:-- --:--:-- --:--:--   884
 ~/../proxylab  curl --proxy localhost:25446 http://www.columbia.edu/\~fdc/sample.html > testp.txt; curl http://www.co
lumbia.edu/\~fdc/sample.html > test.txt; diff testp.txt test.txt
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 34974  100 34974    0     0  30950      0  0:00:01  0:00:01 --:--:-- 30950
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 34974  100 34974    0     0  33693      0  0:00:01  0:00:01 --:--:-- 33693
 ~/../proxylab  curl --proxy localhost:25446 http://info.cern.ch/ > testp0.txt 2> errp0.txt &;\
curl http://info.cern.ch/ > test0.txt 2> err0.txt &;\
curl --proxy localhost:25446 http://www.columbia.edu/\~fdc/sample.html > testp1.txt 2> errp1.txt &;\
curl http://www.columbia.edu/\~fdc/sample.html > test1.txt 2> err1.txt &;\
sleep 5;\
cat errp0.txt err0.txt errp1.txt err1.txt;\
diff testp0.txt test0.txt;\
diff testp1.txt test1.txt
[2] 15719
[3] 15720
[4] 15721
[5] 15722
[4]  - 15721 done       curl --proxy localhost:25446 http://www.columbia.edu/\~fdc/sample.html >  2>
[2]    15719 done       curl --proxy localhost:25446 http://info.cern.ch/ > testp0.txt 2> errp0.txt
[5]  + 15722 done       curl http://www.columbia.edu/\~fdc/sample.html > test1.txt 2> err1.txt
[3]  + 15720 done       curl http://info.cern.ch/ > test0.txt 2> err0.txt
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   646  100   646    0     0   630k      0 --:--:-- --:--:-- --:--:--  630k
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   646  100   646    0     0    589      0  0:00:01  0:00:01 --:--:--   589
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 34974  100 34974    0     0  33.3M      0 --:--:-- --:--:-- --:--:-- 33.3M
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 34974  100 34974    0     0  40432      0 --:--:-- --:--:-- --:--:-- 40385
```

Discussion

What was Difficult?

The most challenging aspect of this assignment was that I was not familiar with web-related terminology, despite covering topics like requests, responses, headers, and content during class. As a result, understanding the requirements of the problem itself was also challenging. I particularly struggled with understanding the size of the response in the cache and where it should end. Eventually, I realized that it referred to the size of the entire request, including the headers, and used it to implement the cache.

I also had a lot of considerations regarding the data structure for the cache. Initially, I attempted to implement it as a hash table, but I encountered difficulties in designing it to follow the RLU (Recently Least Used) policy. Determining the most recently accessed item would require additional bits and updating these bits every time posed challenges. Therefore, I decided to modify a queue data structure, which provides a clear understanding of the access order. I inserted new items at the head, deleted items from the tail when necessary, and when a cache block corresponding to a specific URI was found, I moved that block to the head to indicate the access. This approach made cache management easier.

Something New and Surprising

I thought I had a decent understanding of the C language, but it turns out there were many string-related methods that I was not familiar with. While parsing the URI this time, I came across functions like `strstr` and `strcasecmp`, which are commonly used for string manipulation.

Furthermore, I was surprised by the difficulty of finding websites that use HTTP/1.0 on the internet. To test the performance of the implemented web proxy, I tried to find various HTTP/1.0 websites. However, most of the websites were using HTTP/2.0, and many of the websites using HTTP/1.x were using chunked encoding, which made it challenging to handle the responses. This was because HTTP/1.0 does not support receiving chunked content. Fortunately, after conducting more thorough searches, I was able to find a couple of websites that met the criteria.

Conclusion

In this lab session, we discussed the implementation of a web proxy that can handle multiple concurrent requests and utilize caching to optimize performance. The proxy supports HTTP/1.0 GET requests, establishes connections with web servers, and forwards client requests while retrieving server responses. It incorporates threading for concurrent request handling and employs a caching mechanism to store frequently used web objects. The cache has size and object size limits, and an eviction policy is implemented to manage the cache efficiently. Overall, this web proxy offers an efficient solution for handling concurrent requests, reducing server load, and improving browsing experiences through caching.