Design Document:httpproxy

Author: Surya Rani

Student Id: 1653408

Program Description:

This program will act as an intermediary between a client and a server. Neither the client nor the server are able to communicate directly with one another, though they assume they are. Our proxy will receive client requests and forward them to the server, and receive server responses and forward them to the client. An added element to our proxy is that it will also have a cache element in it. The cache will only be invoked when our client sends a GET request to the server. On the first time the GET request is called for any file it will get it directly from the server, but after that our proxy will store the data received and cache it. The next time the client requests a GET request of the same file our server will make sure that the file data we have stored is not obsolete(by comparing last modified times of the file we have saved and the file on the server). If it is not obsolete then we will send the file data to the client without ever sending a GET request to the server. Our proxy will have multiple program argument options including choosing the size of the cache, how files will be deleted from the cache when the cache is full, and also the max size of the files that can be cached.

Program Logic:

We initially start our program and parse for our program arguments. We do this with switch statements having a case for each different program argument. When each case is countered we will store the value in its respective data structure. We must get both our client port to receive requests on and the server port to be able to relay the requests to. We then can listen in a infinite loop to receive incoming client connection. We get the client connection file descriptor and then handle the connection for it.

In our handle connection function we then receive the request from the client and parse it as we had done in our httpserver previously. We parse it for the file name, function name, and the content length. With all of this information we are able to choose between either our GET, PUT, or else method. If its GET or PUT we need to send and receive the data differently since their is added data of sending or receiving a file, but with HEAD and error requests we can directly send it to the server and send the client the response immediately.

Our entire cache is built upon the ADT Linked List. I retrieved it from the internet and will not be explaining its functions or implementation. I will only be referring to it as our cache and the functions that I can perform with it such as insert and delete.

Now that we have dealt with the three different cases we will go into our first case GET. With GET we are basically doing exactly what we did before but before calling GET we call cache() to check if we have the up to date file in cache and if we do then we send that straight to the client without sending the real request to the server. If cache returns a -1 we know that it did not successfully find or send the data to the client so we must actually send the entire request from the client to the server. We do this by just sending and receiving in chunks like we did before, but with this time we also send to the client when necessary. Since if we are in the GET function it means our cache has failed we must now store the file received from the server into the cache. We do this by calling storeFile() a function we created.

Before moving onto storeFile we will talk about cache(). Our cache function is called if we parse the request from the client and it wants a GET request from the server. In our cache function we first check if we have the file stored in our cache by calling find() a function included in our Linked List implementation of the cache. If we do not find it then we will immediately return -1 so that we know that we were unable to retrieve anything from the cache and for the GET function to continue as normal. If we do find the file in our cache then we will check that it is not obsolete by calling our checkTime() function. checkTime will check our stored last modified time and the last modified time we receive from a head request from the server and make sure that the file we have is as early or earlier than the one on the server. If so then we will send the file data that we have in our cache directly to the client. If not then we will return -1 so that way we run GET as normal.

In our checkTime function we will send a HEAD request to the server and compare it with our own last modified time from the file in our cache. We can then use time functions and parse the time from the two strings. We can then compare the times by calculating the difference between the two.

In storeFile we make sure that if we our cache is full we will remove the last object. After that we always insert into the first slot our new file with all the required parameters including filename, file data, response from the server, file last modified time, and the length of the data.

Now we go to our second case PUT, where we do almost the exact same thing as in GET, but we do not have to cache the file or anything. We just read and send in a loop until we have reached the contentLength. We must receive and send the initial 201 created from the server to the client.

Our last case will handle HEAD and error formatted requests. All I need to do is receive the request, send it to the server, and receive the response and send that to the client.

Of course after each request we must close our connection with the client.

Data Structures:

Linked List:

- This is for our cache and I copied it from online
- Has the functions
    - InsertFirst
    - Delete
    - deleteLast
    - printList
    - Find // finds the node with given name

Struct tm tm

- Object that we get from time.h
- Used to store time variables from a string
- Use this to calculate the last modified of all the files we store in our cache

Time_t time

- Object that will hold our time in seconds since the epoch
- We will use this to find the difference in seconds between the file in the cache vs the one on the server

Struct sockaddr_in server_addr :

- This struct holds our sin_family = AF_INET
- Holds our port sin_port = port
- Holds our sin_addr.s_addr = INADDR_ANY
- We will then use this when we want to bind to our socket that is open

Socklen_t addrlen = size_of(server_addr):

- When we want to bind our server to the socket we need to have both the server and the size of it as well

Struct sockaddr client_addr:

- Same thing as before but this is for the client side which we need to use when we call accept on incoming connections from the client

Socklen_t clients_addrlen:

- Same purpose, necessary for accepting connections

Uint8_t Buffer[fixed_length]:

- Used in multiple areas throughput the program for storing data
- Char buffer for http request

- Uint8 buffer when incoming or outgoing file data because it could be binary
- Used to store incoming client request
- Used for incoming and outgoing file data
- When reading and writing it needs to be done in a loop so we can keep our buffer to a fixed length

char * token

- Used to hold the string tokens after we call strtok with our delimiter on the http request
- This allows us to easily access data in the specific format we are looking for

Functions:

Int Main:

- Parses through program user arguments and stores cache size,type of cache deletion, etc if specified
- Gets both server and client ports and creates listening and sending sockets that we can send and receive to both of
- Starts an infinite loop that keeps accepting incoming client connections

```
void handle_connection(int clientSock,int serverSock)
```

- Handles all of our client connections
- Receive message from client
- Parse through message to see what is being called either put, get or head
    - Parsing will require to use strtok probably to be able to delimit each header by \r\n
    - Once we get the first header we need to find the function name and the file name
    - Check if each header after that follows the correct http conventions if not throw error
    - Get content length if request has it and store it
- If correctly formatted call that function method

```
int cache(char* fileName,int clientSock,int serverSock)
```

- This is to check if we have the file already located in our cache
- If we do then we want to check if the file is obsolete or not
- If it is then we will delete the file from our cache and go through the entire process of GETting and storing the file into our cache
- If it is not obsolete then we can send it directly to the client
- If we did not even have the file in our cache at all then do GET as normal
- We will also maintain the order of the cache if it is accessed by moving it up to front of the linked list if and only if LRU has been activated

```
bool checkTime(char* fileName,int serverSock,char* tme)
```

- Will compare the last modified time of the file on server and the file we have stored
- If it is the same we will return true

- If it is newer on the server then we will return false

```
char* getLastModified(char* fileName,int serverSock)
```

- This is a small sub function that helps us get the last modified header from the server
- It forms a head request, sends it to the server and returns the last modified header

```
void get(char* fileName, int clientSock, int serverSock){
```

- Does all of our get function like it did in httpserver
- Sends all data to and from client and server
- We store all the file data so that way we can cache it
- Store the ok response as well to send to the client later as well

```
void put(long length, int clientSock,int serverSock)
```

- Requires the client socket to be able to receive the data it wants to put into file
- Requires content length, filename
- Dont need to worry about creating or opening a file as the server will do it for us
- We just need to continually in a loop send the data from the client to the server

```
void storeFile(char* fileName, uint8_t* file,char* response,int dataLen,char* tme)
```

- Called in our get function at the end to store the GET file into our cache
- Takes all these parameters and creates and stores a file at the front of our linked list cache

```
int create_listen_socket(uint16_t port)
```

- Prewritten for us
- Gives us the ability to create a socket to listen for requests from

```
int create_client_socket(uint16_t port)
```

- This is able to create the socket that we want to send stuff to
- This is to talk to the server and send and receive from it

```
int setsockopt(int socket, int level, int option_name,
               const void *option_value, socklen_t option_len);
```

- **Allows you to avoid "Bind:Address Already in use" error**

```
int bind(int sockfd, const struct sockaddr *addr,  socklen_t
addrlen);
```

- **Bind server address to socket that is open**

```
 int listen(int sockfd, int backlog);
```

- **Listen for incoming connections**

```
int accept(int sockfd, struct sockaddr *restrict addr,
```

```
                    socklen_t *restrict addrlen);
```

- Accepts incoming requests and now that is our new client socket to receive and send our information to for that request

```
char *strtok(char *restrict str, const char *restrict delim);
```

- Used to break up request into each header
- Each token is now each header and we can parse through it much easier

```
int sscanf(const char *str, const char *format, ...);
```

- Used to get our data such as filename, filesize etc.
- We are looking for data in a specific format from our token and this finds it and puts it into whatever data we want it in

```
ssize_t send(int sockfd, const void *buf, size_t len, int
flags);
```

- Used to send data to our client socket
- We use this when we send our messages to the client, and also when we want to send file data over to the client

```
close (int sockfd)
```

- Closes the socket connection

Questions:

Q: Start your proxy with no cache and request the file ten times. How long does it take?

A: It takes about a few seconds to get a 50 mb file. With a 10 mb file it was about a second so it scales up from there.

Q: Now stop your proxy and start again, this time with cache enabled for that file. Request the same file ten times. How long does it take?

A: You are definitely able to notice the speedup slightly as there is no latency in between the server and the client to be sending and receiving in between. Since we have the file already stored we can send directly to the client so there is much less time

since we do not have to wait to receive from the server and send to the client. I would say it probably helps it by a second or so with a 50mb file and is not very noticeable at all with a 10mb file. I could definitely see this helping as the file size scales up, but the tradeoff here is that now the file needs to be stored in two places, once on the server and another time in our own cache in our proxy.

Q: Aside from caching, what other uses can you consider for a reverse proxy?

A: There are many uses for a reverse proxy, one of them off the top of my head is being able to load balance and pick and choose between servers. If we were to have multiple servers, then potentially if we were to receive a huge amount of requests from the clients we could delegate the requests based off of how loaded each server is and act as an intermediary who checks the health amongst all the servers. Another use for a reverse proxy could be for malicious attacks. If we had more time we could have parsed through the requests themselves and made sure that the requests are both syntatically and semantically correct. The proxy could take the fall between the server and the client, and so the server is not going to shut down because of it. Since the server would have all the stored data it would be vital to protect it.

Testing:

1. First initially wanted to test parsing algorithm
    a. Made sure that all arguments could be in any order
    b. First port number would be our client port, second the server
2. Test request parsing algorithm
    a. Print out filename, request type, content length etc
3. Test out GET, function
    a. Pipe in GET data to file and compare file using diff
4. Test PUT function
    a. Do same thing as with GET function just PUT into a different file and compare the two files for differences
5. Check that proxy handles HEAD and error requests
    a. Compare mal formatted requests when i send it to the proxy and directly to the server
    b. Same thing with HEAD
6. Start testing cache
    a. Add, delete and print out cache before and after every change
7. Test that we store file at the end of our GET function
8. Test that cache checks time and compares properly
    a. Use saved last modified header and the one on the server and compare the two using time functions
    b. Print out time difference in seconds

9. Check that when file is newer on server that we dont use the cache
10. Make sure that we are updating the cache according to the replacement policy
    a. If fifo we just insert and delete from our linked list like normal
    b. If LRU we make sure to delete and re insert into the front after everytime it is accessed from the cache
11. Make sure that altering cache size and max file size works
    a. Make sure that cache does not add if file size is greater than the max file size
    b. Make sure that cache deletes if it hits the max length
12. Test that everything works with larger files