Design Document

Author: Julia Ni
Student ID: 1657608

Project Description

Httpproxy acts as a http server and client that forwards received requests to the server and then forwards responses from the origin server to the original client. The httpproxy takes files from GET requests and stores them in a local cache to reduce response time and traffic on the network. If there is a GET request for an object in the cache, httpproxy will respond directly to the client if the cached version is not outdated. Essentially, we are implementing a http reverse proxy with a cache that can also handle persistent connections.

Program Logic

In httpproxy, we parse the command line for 2 port numbers and 3 optional arguments: the number of items the cache can hold, the maximum size for a file in the cache, and a flag that signifies the cache policy will be least recently used (LRU) instead of first-in-first-out(FIFO). A socket is created for listening to the client on the specified port, and a client socket is also created for communicating with the server. While there is a request from the client, we receive the request one byte and a time and parse into the different fields of the ClientReq data structure. If the request is a HEAD or PUT, we forward the request to the server and wait for a response. Then we receive it one byte at a time and send the response back to the client. If it is a GET request, we check if the file exists in the cache. The cache is an array of cache entries. Each cache entry keeps track of the file name, the last modified time, the ID number of the cache entry, and the contents of the file. If it is in the cache, we send a head request to the server to get the last modified time. Once we receive the time, we compare the server's and cache's last modified times of the file. If the cache's copy is outdated compared to the server's copy, we need to send a GET request to get the new contents of the file and new last modified time. We cache this version of the file and then send the GET response from the server to the client. If the cache's copy is up to date with the server's copy, then the proxy will use the information from the cache to create a response and directly respond to the client without asking the server. If the file doesn't exist in the cache, then we forward the GET request to the server and receive the GET response from the server. If the file's content length is within the maximum size stated in the command line, we cache the file. If the file is larger, we don't cache it. If the cache is full and we want to add another cache entry, we use the appropriate replacement policy (LRU or FIFO) depending on if there is a LRU flag or not in the command line. If there is an error, the server will send the appropriate error response to httpproxy, which will send the response back to the client. At the end, we close the connection socket. The server connection will remain open unless there is an error.

Data Structures

I am using a ClientReq data structure to hold all the components of the client requests. I am also using an array (char[]) as a buffer to store the client requests and as fields for the ClientReq data structure. I also use these to receive/send requests and responses to and from the client and server. There is also a cache and a cache entry data structure to keep track of each cache entry's data.

Functions in httpproxy.c

- struct ClientReq
  Defines a data structure that makes all the components of the http message accessible.

- void clearbuffer(struct ClientReq* request)
  This function loops through request->body and sets everything to the null character, which resets the char[].

- void parse_response(char* buf, struct ClientReq* request)
  This function parses responses received by httpproxy from the server. It checks for the content length and last modified headers and parses the length and time respectively. The written out time is converted to a time_t variable using strptime() and mktime(), which is a number representation of the written out version. These new values are then stored in the corresponding request fields.

- int parse_req(struct ClientReq* request)
  This function parses request→ buf into different components such as command, filename, etc. The GET and HEAD command only need the command and filename field, but the PUT command needs those two fields plus the header and content length. The backslash is also removed from the file name here. An integer is returned indicating whether the parsing is complete or if there is an error. Errors include invalid content length and issues with sscanf().

- void read_req(int socketfd, struct ClientReq* request)
  This function reads the client request one byte at a time until a \r\n is reached and stores it in request→ buf. It calls parse_req() to parse the buffer into the appropriate data structure fields. This process continues until the entire request is processed (when \r\n\r\n is reached).

  If there is a HEAD request, we directly send the request to the server.

If there is a PUT request, we directly send the request to the server. We also receive the body of the PUT request. We then store the content length into int tempbytes, which represents the total bytes in the file. If tempbytes is larger than the buffer size, then we will use a loop to continuously read in bytes from the body of the client request and send the body to the server until the bytes remaining is less than the buffer size. At this point, we will receive and send one more time.

If there is a GET request, we check if the file is in the cache. If it is not in the cache, we directly send the request to the server. If the file is in the cache, we construct(using string operations) and send a HEAD request to the server to get the last modified time.

- void handle_get(int socketfd, int serverfd, struct ClientReq* request)
  This function responds to all GET requests.

  If the file doesn't exist in the cache, we receive the GET response from the server in a while loop one byte at a time into a buffer. We call parse_response() to parse the buffer for the content length and last modified time. We continue reading until we reach \r\n\r\n. Then we need to receive the body of the response which contains the file contents of the requested file. We malloc a char* recv_content, which holds the file contents and will be passed as a parameter into create_entry(). We then store the content length into int tempbytes, which represents the total bytes in the file. If tempbytes is larger than the buffer size, then we will use a loop to continuously read in bytes from the body of the client request and send the body to the server until the bytes remaining is less than the buffer size. At this point, we will receive and send one more time. As we are receiving, we use memcpy() to copy each part of the body into recv_content. We use an int index to keep track of the indexing of recv_content. If the content length is within the maximum size stated in the command line, we create a cache entry with create_entry() and add this entry to the cache using add_cache(). If the content length is too large, we free recv_content because we don't end up using it.

  If the file does exist in the cache, we receive the HEAD response from the server in a while loop one byte at a time into a buffer. We call parse_response() to parse the buffer for the content length and last modified time. We continue reading until we reach \r\n\r\n. We call get() which returns the existing entry with the same file name. Then we compare the server's and cache's last modified times.

  If the cache copy is outdated, we send a GET request to the server and read in the GET response the same way we did when the file didn't exist in the cache (as mentioned in the paragraph above). Once we have the new content length, last modified time, and file

contents, we free the existing entry's contents array and update the existing cache entry with the new information.
If the cache copy is up to date, we update the last modified time in the cache entry, create a response with sprintf() with the cache entry information, and send the reply to the client.

- void respond(int socketfd, struct ClientReq* request)
  If it is a GET command, we call handle_get(), which will send the appropriate response back to the client.

  If it is a PUT command, we receive the response from the server one byte at a time in a while loop until we see \r\n\r\n, the body, and the final \n. If the response is larger than 4096 bytes, we will receive and send the response in chunks, resetting the buffer every time we reach 4096 bytes until the full response has been forwarded to the client.

  If it is a HEAD command, we receive the response from the server one byte at a time in a while loop until we see \r\n\r\n. If the response is larger than 4096 bytes, we will receive and send the response in chunks, resetting the buffer every time we reach 4096 bytes until the full response has been forwarded to the client.

- void clear (ClientReq* request)
  This function resets all the ClientReq data structure fields. The char[] are set to null characters and ssize_t contentlen is set to 0.

- void handle_connection(int connfd)
  This function calls read_req(), respond(), and clear() until there is nothing left to read from the client. The connection socket is closed at the end.

- int main(int argc, char* argv[])
  This function parses the command line for the 3 optional arguments with getopt(): cache capacity, maximum file size, and the replacement policy (LRU or FIFO). We also parse the server port and listening port (for the client). There is a while loop that continuously accepts connections, and we keep processing requests until the server and client connections are closed.

- Rest of the functions were given in skeleton code and were not changed

Functions in cache.c

- cache cache_init(int size, int mode)

This function initiates the cache. We use malloc to create space for the cache based on the size parameter and the cache entries it will hold. We then initialize each cache entry to be null and also set the mode. We return the cache at the end.

- cache_entry* create_entry(char* file, time_t edit_time, char* content)
  This function initiates the cache entry. We use malloc to create space for the cache entry and for the file name. We then set the cache entry fields to the corresponding parameters and we return the cache entry.

- void free_entry(cache_entry* c)
  This function frees the file and content strings of the cache entry and then frees the entire cache entry.

- void add_cache(cache c, cache_entry* entry)
  This function adds to the cache. If either the cache or the cache entry is null, we return in the beginning. If there is space in the queue, we look for an empty spot and add the entry at that index. If the cache is full, we free the entry with the smallest ID and replace it with the new entry.

- cache_entry* get(cache c, char* filename)
  This function returns the cache entry if it exists in the cache and returns null if it doesn't. If the cache is empty, we return null. Otherwise, we loop through the cache with a for loop and strcmp() the file names. If there is a match, we return the cache entry. If we are implementing the LRU replacement policy, we also increment the ID of the cache entry before returning.

Questions

1. Without the cache, it took about 2 minutes to process the requests. With the cache, it only took about 40 seconds. There was a big improvement in speed with the cache.

2. Aside from caching, I can also use a reverse proxy to manage incoming and outgoing traffic and for security purposes. If there is a virus coming from one of the clients, the proxy will see it first and can stop the virus from reaching the server and potentially crashing the entire server. We can also monitor the traffic through the proxy and check that all the incoming requests are being processed properly. Having a reverse proxy will also improve performance because the proxy can directly respond to the client if it has sufficient information, which saves time and resources that would be consumed if you had to ask the server for the information.

Testing

Individual function testing

- Sending all types of requests to the server

- Receiving all types of responses from the server
- Caching with small/large binary and text files
- FIFO and LRU replacement policies are working properly
- Check cache entry's file, edit time, id, and contents are stored correctly into the data structure
- Handling outdated cache copy and up-to-date cache copy
- File is too large to be cached
- Parsing the command line arguments correctly
- All types of requests for 0 byte files

Whole system testing

- Testing if cache works with persistent connections
- Testing if combinations of requests work, one request at a time while the server connection is open
- Testing forwarding for a combination of all the requests
- Checking if the log file matches the requests sent to the server when using all types of requests
- Testing if there is a speedup in processing requests with the cache enabled
- Testing a lot of requests with a small sized cache
- Testing few requests with a large sized cache