C++ HTTP Caching Proxy Server

Members: Boyi Wang, Qiheng Gao, Yutong Zhang, Zilin Yin

1. Motivation

Nowadays, people visit various websites everyday. A lot of times, people could visit the same website multiple times within a short time. Each visit to the same website comes with a HTTP request to the web server. Knowing that the response the web server provides to the same HTTP request is going to be consistent, making a new HTTP request to the web server to retrieve the response causes waste in network bandwidth and latency. Hence, something in the middle of clients and web servers playing as an intermediate to cache HTTP responses from web servers and serve them for clients when it is needed could be significantly helpful, as it helps reduce network bandwidth and latency.

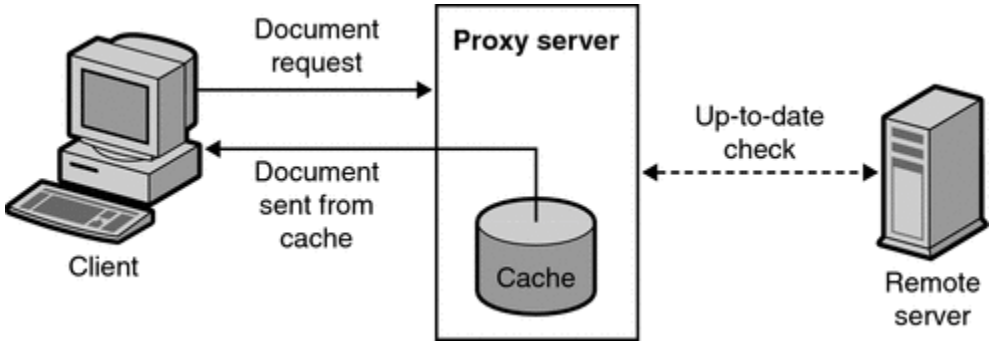

Figure 1. Proxy Caching

Hence, we want to introduce proxy caching, as shown in Figure 1. Proxy caching allows a server to act as an intermediary between a user and a provider of web content. When a user accesses a website, proxies interpret and respond to requests on behalf of the original server. And this effectively solves the situation mentioned before.

2. Project Description

Our goal is to build an HTTP caching proxy server that handles GET, POST and CONNECT requests from clients via TCP socket, and caches responses following rules of expiration time, re-validation and cacheability according to RFC 7234 specification https://www.rfc-editor.org/rfc/rfc7234.

The HTTP caching proxy server uses multi-threading to handle concurrent requests from clients, and to increase its performance under a heavy load environment.

3. Implementation

A high-level overview of the implementation of the HTTP caching proxy server is shown below in the diagram:

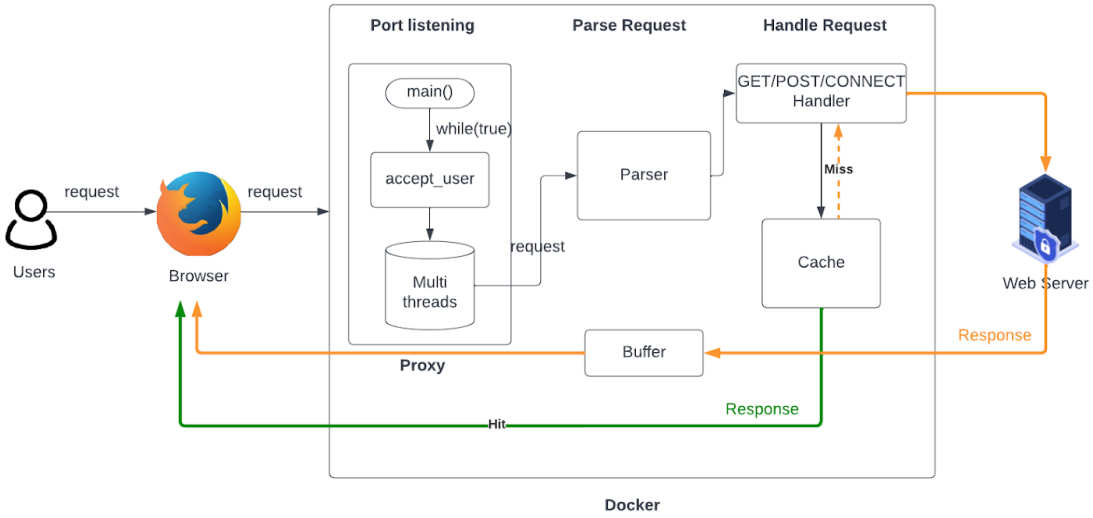

Figure 2. Implementation Diagram

As shown in Figure 2, we separate and abstract our proxy server into several components.

3.1 Multithreading in Proxy

Our proxy server would listen on and receive traffic on port 12345 when it is running, and we use thread-per-request concurrent strategy to handle requests. If users' requests are valid, the proxy would set the tcp connection with both users and web servers, receiving and forwarding messages.

```
void Proxy::startService() {
  int listen_fd = initServer(this->portNum);
  if (listen_fd == -1) {
    return;
  }
  int connect_fd;
  while (true) {
    std::string ipReader;
    connect_fd = acceptClient(listen_fd, ipReader);
    if (connect_fd == -1) {
      continue;
    }
    pthread_t thread;
    Client* client = new Client(this->clientCount, connect_fd, ipReader);
    ++this->clientCount;
    pthread_create(&thread, NULL, serveClient, client);
  }
}
```

3.2 Parser

We would parse the HTTP request and response into request class and response class, and their constructors take the parsing task. To be more detailed, we parse the text line by line to extract the keywords to fields of the objects.

```
class Request {
 private:
  size_t id;
  std::string ip;
  MyTime receiveTime;
  std::string msg;
  std::string firstLine;
  std::string method;
  std::string url;
  std::string hostname;
  std::string port;
  std::string protocol;
  int insert_pos;
};

class Response {
 private:
  std::vector<char> msg;
  std::string max_age;
  std::string date;
  bool chunked;
  bool save_in_cache;
  bool no_cache;
  std::string content_length;
  size_t content_size;
  size_t header_size;
  std::string expire_time;
  std::string etag;
  std::string last_modify;
  std::string line;
};
```

3.3 Request Handler

Given the method specified in the request, we pass our generated request objects to the corresponding handler, i.e. getHandler, postHandler and connectHandler.

3.3.1 getHandler

Handling the GET method logic is one of the most complicated part in our project, and the process is shown in the follow flow chart:

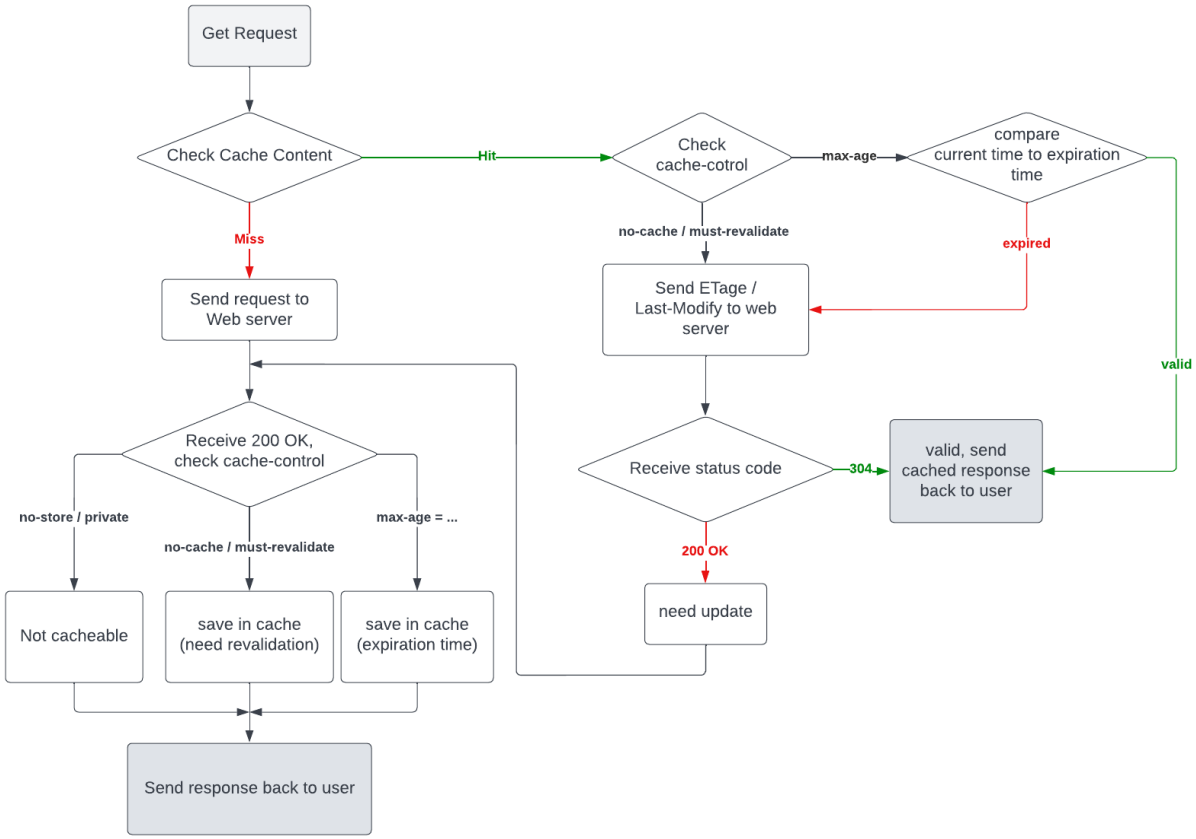

Figure 3. Get Request Process Logic

3.3.2 postHandler

In the post handler, we will forward user requests to the server, then we store the response in a buffer. After the response is fully received, we send it back to the user.

3.3.3 connectHandler

The CONNECT is to set up a tunnel between users and web servers, and we utilize a file descriptor set to monitor the sockets efficiently, transfer messages till the session ends.

```
fd_set toMonitor;
while (true) {
    FD_ZERO(&toMonitor);
    FD_SET(toServer_fd, &toMonitor);
    FD_SET(client->getSocket(), &toMonitor);
    select(maxSocketNum, &toMonitor, NULL, NULL, NULL);
    int numBytes;
    if (FD_ISSET(toServer_fd, &toMonitor)) {
    ...
    }
    if (FD_ISSET(client->getSocket(), &toMonitor)) {
```

```
    ...
    }
}
```

3.4 Cache

Our cache implementation is basically a hashmap that maps a specific url to a response object, our cache will start to clear stale responses if the size exceeds the preset maximum size. Additionally, we also add read-write locks to circumvent race conditions.

```
class Cache {
 private:
  const size_t capacity;
  std::unordered_map<std::string, Response> data;

 public:
  Cache() : capacity(100000) {}
  Cache(int _cap) : capacity(_cap) {}
  bool has(const std::string& url);
  void add(const std::string& url, const Response& response,
           const MyTime& addTime);
  Response& get(const std::string& url);
  bool isFull();
  size_t getCapacity();
  void clearStales(const MyTime& time);
};
```