4/26/2015

Proxy Server ArtOz

Arturo Parrales-Salinas and Ozan Erat

1. Introduction

The aim of the proxy server ArtOz is to provide basic GET requests functionality. On top of it features such as caching, concurrency, content filtering and a search engine for cached files were developed. The whole system was programmed using C++11 on Mac OSX Yosemite.

Section two discusses the structure of the proxy server as a system integrated by different classes. Section three emphasizes in the additional features developed for the ArtOz proxy server and how they were integrated to the basic structure. Section four explains how to start the proxy and how to configure Mozilla FireFox 36.0 to use the proxy. Finally, section five states the conclusion and lessons learned from the design of the proxy server.

2. ArtOz basic structure

The implementation of a proxy is not an easy task to achieve since it has to work both as client and server. In order to achieve such functionality and considering only a GET request functionality, the structure shown in figure 1 was implemented.

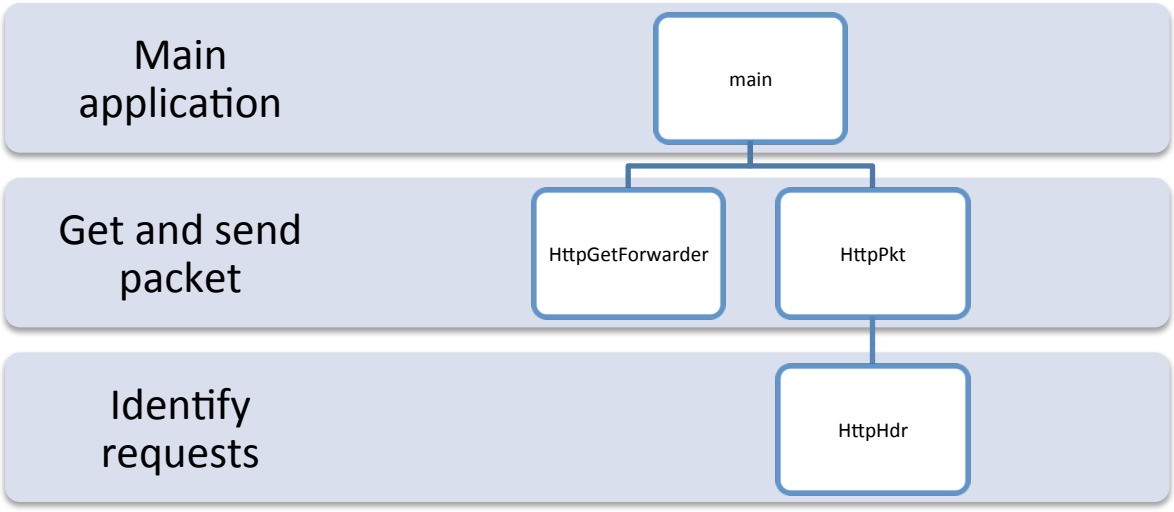

Figure 1. Basic structure of the proxy server ArtOz

The structure used implements the classes HttpHdr, HttpPkt and HttpGetForwarder in a main.cpp.

In the main.cpp the basic socket programming is implemented as well as the logic of attending requests. Maybe the most important parts in the main are the system calls: socket(), bind(), listen(), accept(), followed by an infinite while loop, the instantiation of HttpHdr, HttpPkt and HttpGetForwarder.

The three classes HttpHdr, HttpPkt and HttpGetForwarder are described below.

- **HttpHdr.** It is a class to extract the type of requests, the host and the object/filename. It also checks whether the HTTP header is complete or not. Additionally, it has functions to print the header.
- **HttpPkt.** It retrieves the packet from the HTTP response. It also checks whether the packet is complete or not, by using the 'Content-Length' field from HTTP 1.1. It can also print the packet.
- **HttpGetForwarder.** This is the essential class to implement ArtOz as a proxy since it implements the other two classes by processing the GET request, taking the information of URL, connecting to the server being requested, getting the response, making the packet and sending it to the client.

These classes together can identify GET request from the client, connect to the server and send the response to the client. All of this happens in the main, which implements the classes.

3. Additional features

Additional functionalities are achieved using different classes. The functions build on top of the basic GET request proxy are concurrency, caching, content-filtering and a search engine for cached files.

3.1. Concurrency

Multiple connections are handled using the select call. This allows ArtOz to attend the requests of every client one by one successfully. The implementation was achieved creating an array of clients that have a socket initialized to the value -1. As soon as the call FD_ISSET recognizes that the original socket (from listen call) repeats it initializes a new client. Then, another FD_ISSET checks for the sockets that made a request. This worked efficiently for serialized requests. However, when trying to process a client the select call has to wait until the operation of the client has been completed to attend a new one. As a result, threads were implemented to allow the process of attending a request parallel while another client can ask for another request. A struct named _sock _vec, which contains a socket, a pointer to a char vector and a thread. This

struct is used to keep track of the clients. Another struct called _sock _hdr contains the HTTP header and a copy of the client socket. The latter is essential for the thread since it is passed as a struct to the thread function named doRequest. The doRequest function only activates when ArtOz receives a GET request. Once this happens, doRequest connects to the corresponding server of the request, ask for data, generates a packet for the client and sends the data to it. This doRequest function also implements the caching and search engine. These functions are described in sections 3.2. and 3.4.

3.2. Caching

A simple caching function was implemented on top of ArtOz. The classes involved in the caching of a file are CacheStorageHelper and CacheManagement.

- **CacheStorageHelper.** Checks for the hostname and object in order to determine which directories and file to create to cache content. It also determines if the path from which the file must be retrieved is valid and if the file exists. It reads and writes to a file. Finally, it retrieves the length of the file for the case of reading.
- **CacheManagement.** For the caching function this class is used to store the names of the cached host and file (concatenated as a path) into a vector, so that they can be used for search purposes.

The integration of these classes is simple and inside the doRequest function. Basically, when the proxy receives a GET request, the doRequest function is called, and the proxy checks whether the host and file exist in the cached directory and if it has a valid host and file name (isFileExisted() and isPathValid()). If this is true, the cached file is read from memory and returned to the client (getFileLength() and readFileContent()). Otherwise, the proxy verifies that it received a host and a file name (isValidPath()) to connect to the server (sendToURL()), cache the file (writeIntoFile()), store it in a CacheManagement pointer to object (insertIntoCachedFiles()) and send it to the client (sendToClient()). If the path where the file is supposed to be saved is not valid (want to access to root), then the proxy only connects to the server and sends the response to the client.

3.3. Content-filtering

This is maybe the simplest additional function on ArtOz. The content to be filter is written into the filterURL.txt file and it will be read in the main before starting the infinite while loop. An object from the class FilterHelper stores the URL's. In every iteration of the while loop after extracting the header of any request, the host will be compared to the list of blocked URL's generated from the *.txt file. If there is a match (isHostValid()) then the request is dropped. Maybe the most important part of this class is the library <regex> which stands for regular expressions. This library allows ArtOz to find matches that are not exact words, but that contain the name of the host. The previous functionality generates a robust filtering system.

3.4. Search Engine

The implementation of the search engine is related to the properties of the HTTP request. In other words, it takes advantage of the QUERY_STRING. This feature uses two HTML files: search.html and 404.html. The classes involved in the search engine are CacheManagement and CacheSearchHelper.

- **CacheManagement.** It has a list that contains the cached files. It uses that list to look for all the URL's or content that match with a provided keyword (searchForURL() and searchForContent()). It returns a vector with the matches found.
- **CacheSearchHelper.** It is initialized with a keyword and a type (URL or content). After have been initialized, it is able to perform a search by calling CacheManagement to search for URL or content. The vector returned is used to create an HTML body to display the results in an HTTP response.

The complex part of the search engine is handling the concurrency of the threads. This is solved using mutex locks and unlocks to allow only one thread at a time to look for URL or content in the CacheManagement methods.

The search engine is implemented in the doRequest function. When this function starts, if the string "localhost/search.html" or "127.0.0.1/search.html" is found the search.html will be displayed on the browser. The client then can enter the keyword and select the type of search. Once the button "I'm feeling lucky" is clicked the html form generates a get that contains the keyword and the type in the QUERY_STRING in the form keyword=…&type=... This string is parsed in the CacheSearchHelper constructor to initialize an object. Then, the search is performed and the results are shown.

The logic of the search engine is explained in the flow chart of figure 2.

It is important to notice that the search engine only looks for files that have been cached.

3.5. Integration

The integration of all these classes happens in the main. The flow chart shown in figure 3 describes the logic of the proxy ArtOz. The function doRequest manages the call for the search engine, the caching and the GET responses.

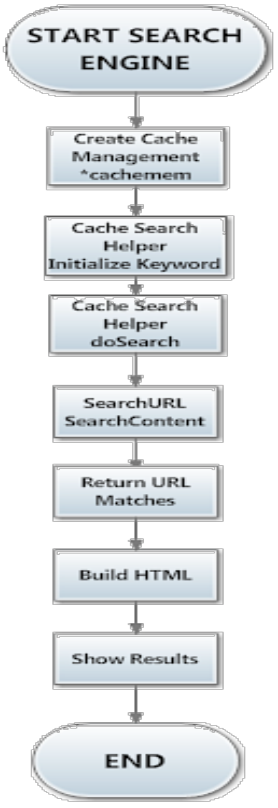

Figure 2. Flow chart of the search engine

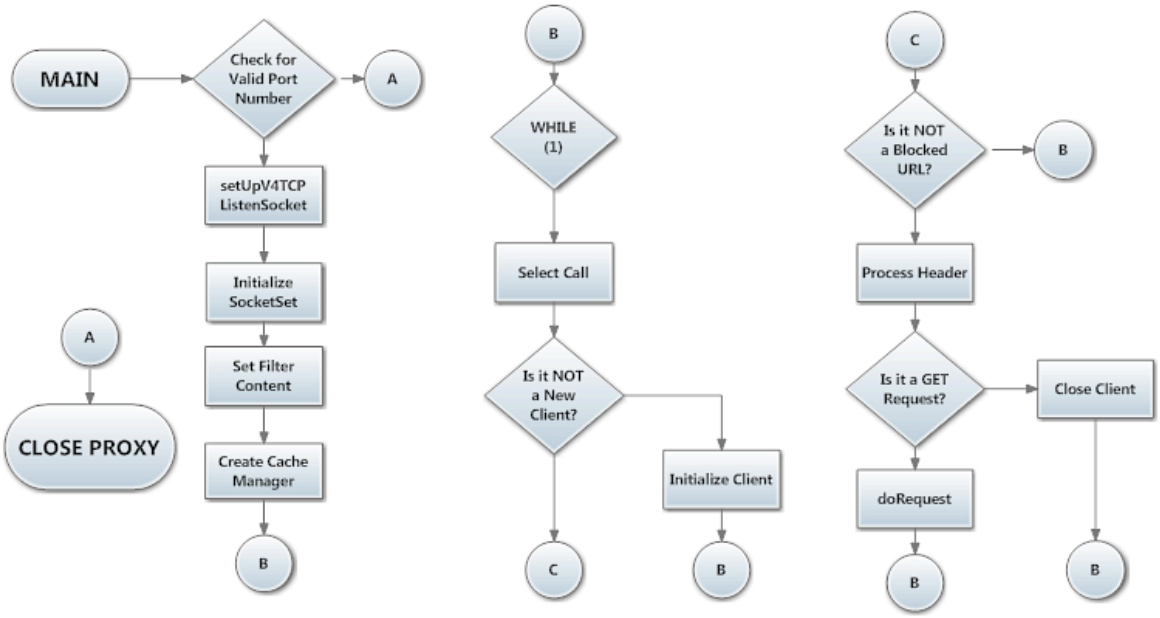

Figure 3. Flow chart of the proxy ArtOz

4. Compile and run

In order to compile the program, it is highly recommended to use Mac OSX Yosemite. It is very simple to start the proxy since a script was written. ArtOz has a default start in the port 6000. To make any change open the run_proxy.sh and modify the port.

A 'Makefile' was created as part of designing the script. This eases the compiling procedure. To start ArtOz type on the terminal the following command:

$ ./run_proxy.sh

Afterwards, configure Firefox 36.0 to use the IP address of ArtOz as a proxy. To do this, open Firefox> = > Preferences > Network > Settings. Then, choose Manual proxy configuration and set it.

In case the reader does not know how to look for its IP. Type the following command on the terminal:

$ ifconfig | grep inet

This should display all the IP addresses in use. Avoid using 127.0.0.1 in Firefox since that triggers the search engine.

5. Conclusion

After designing ArtOz the knowledge of the protocol HTTP 1.1 and HTML was highly increased. Including interaction between the two of them.

The skills in C++11 were boosted and improved. The use of classes ease the coding and gives it a better structure. In addition, OOP makes it easier to understand and identify what are the pieces that integrate a proxy.

Knowledge about networks is required to understand how to approach the design and debug.

Concurrency was a big issue and the knowledge of threads was acquired while designing the proxy as a better concurrent service for multiple clients.

Surprisingly, the socket programming was simple with the previous knowledge about it.