Development of a Recursive Web Crawler Application with Interruptible Download Capability

Butnaru Vlad Andrei

Abstract. This paper details the development of a web crawler application designed to download files from specified URLs with an adjustable depth of search and the capability to pause and resume downloads. Utilizing HTTP protocols and client-server architecture, this application aims to provide an efficient and robust tool for data retrieval from the web.

1 Introduction

This project focuses on developing a web crawler type of application capable of downloading files from a given URL to a user-defined depth. It features the ability to pause and resume downloads. It is implementing using a TCP protocol and HTTP type of header in requests. The downloading works by the client first making a request to the server that retrieves the correct information either from its resources or from the web. The client is than using the information to reconstruct the files.

2 Applied Technologies

The primary technology employed in this project is the Hypertext Transfer Protocol (HTTP). HTTP is chosen for its and reliability in web data transfer and ease to communicate between the server and the client. Alongside HTTP, the project extensively uses C sockets for implementing the client-server architecture.

C sockets provide a low-level mechanism for network communication and are instrumental in handling the transmission of data between the client and the server. This choice is motivated by the flexibility and control offered by sockets in C, allowing for fine-tuned management of data streams and a more thorough understanding of the underlying communication processes.

This choices are complying well with the design of the client server infrastructure because we are making sequential requests for each of the files or folders that we are needing. We are using a TCP design because we want to be able to request files as fast as possible and to continuously make requests to the server for new ones, utilizing its full power.

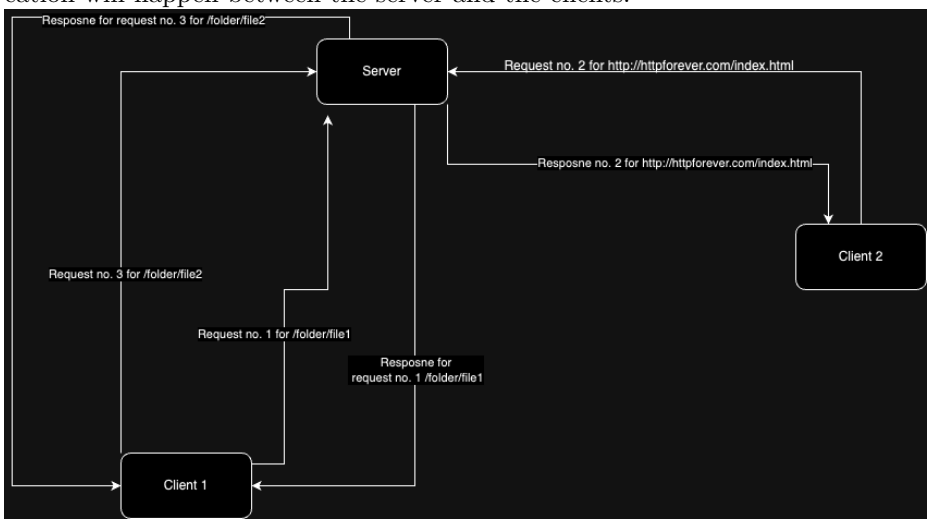
The integration of HTTP with C sockets forms the backbone of our client-server architecture, ensuring efficient management and processing of data requests.

3 Application Structure

The application is divided into two main components: the server and the client. Both will be explained in the next section.

All the communications between the server and the client will be based on TCP in order for the HTTP requests to be processed in real time and the server to be utilized at its fullest. The server is going to have multithereading for two reasons: one is the better usage of its resources and the other one being the possibility of handing multiple downloads if one the clients is pausing its one.

In the following diagram will present a small example of how the communication will happen between the server and the clients.



4 Implementation aspects

In the schema presented above we can we can see how the server is communicating with both clients "at the same time" by multithreading being able to process requests continuously while the clients are busy creating and parsing their files.

4.1 Server

The server receives requests from different HTTP GET requests and does the follwing steps in order to get the information:

- 1. It first checks if the request HOST is the server itself - that meaning it has to get the resources from the local files and to retrieve them. It first checks if the required path exists and if yes it retrieves the content of the file.
- 2. If the url points to a folder from the localhost it returns to the client a list of all the available files into that folder with a value that corresponds to the ENODE being a folder or a normal file.

– 3. If the host of the ask is not the server, it makes a HTTP GET request to the actual server and gets the actual file that was requested (here it acts like a proxy server).

The server can also use the socket to communicate with the clients for pause and unpause of the downloading (sending of new data) informs. Here is a more detailed implementation of the server and clinet:

4.2 Client

The client is responsible for reading the URL and the depth the user asks for and then making the correct requests to the server to retrieve the data.
It asks for the content of the first URL and if it receives a file that is not HTML, it simply creates a file for it on the local machine. If it is a HTML or a folder, it searches for the next items to fetch (the files of the folder or the resources of the HTML file) and makes the asks in order to fetch that data and it continues recursively until it either has nothing more to fetch or it reaches the maximum depth. In the case of folders being fetched it also creates them on the local machine.
It will also be able to control the download (to pause and continue them) by also having a special socket with the server in which it can send a inform to start or stop sending data.

4.3 Key Code Segments

We present code snippets demonstrating command parsing and the recursive download function.
First we present the way of how the client handles a received file in order to find out the next files (or folders) to ask for and how it adds the files to the file system.

```
void fetchFileAndContinueDown (struct enod* ENOD, int maxDepth)
{
    //enod is the file we have to fetch now
    if (ENOD->depth > maxDepth)
        return;

    fileFetcher(ENOD);
    printf("the fetched file url is %s is a %d and the content is: \n %s
        \n", ENOD->url, ENOD->isFile, ENOD->fileContent);
    if (isFileNotFound(ENOD)) // no cotent to use further
        return;

    createFileOrFolder(ENOD); // creating the file on the files system
    if (isEnodAFolder(ENOD))
```

```
    {
        char *savePtr;
        char *childFolder = strtok_r(ENOD->fileContent, "\n", &savePtr);
        childFolder = strtok_r(NULL, "\n", &savePtr); // skipping the .
            folder
        childFolder = strtok_r(NULL, "\n", &savePtr); // skipping the ..
            foder
        while (childFolder != NULL)
        {
            struct enod nextNode;
            nextNode.id = globalENODId++;
            nextNode.depth = ENOD->depth + 1;
            strcpy(nextNode.url, "\0");
            strcat(nextNode.url, ENOD->url);
            strcat(nextNode.url, "/");
            strcat(nextNode.url, childFolder);
            strcpy(nextNode.baseUrl, ENOD->baseUrl);
            // cleaning the url of the file type receives
            for (int index = 1; index < strlen(nextNode.url); ++index)//
                optimize this
                if (nextNode.url[index] == ' ' && nextNode.url[index - 1]
                    != '\\')
                    nextNode.url[index] = '\0';
            fetchFileAndContinueDown(&nextNode, maxDepth);
            childFolder = strtok_r(NULL, "\n", &savePtr);
        }
    }
}
```

The code above is just a recursive function that iterates through the received response and detemines what the next requests will be whie also keeping track when it has to stop for the depth.

Next we are going to present the how the server is able to fetch a file from the internet

```
void returnFilesContent(char *url, char *result)
{
    struct addrinfo hints, *res;
    int sockfd;
    char buf[10056];
    char protocol[10];
    char domain[2056];
    char path[2056];
    char port[6];

    extractProtocol(url, protocol);
    extractDomain(url, domain);
    extractPath(url, path);
    extractPort(url, protocol, port);
```

```
// const char *port = strcmp(protocol, "https") == 0 ? "443" : "80";

printf("the protocol is %s \n", protocol);
printf("the domain is %s \n", domain);
printf("the path is %s \n", path);
printf("the port is %s \n", port);

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;

if (getaddrinfo(domain, port, &hints, &res) != 0)
{
    perror("getaddrinfo failed");
    return;
}

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
if (connect(sockfd, res->ai_addr, res->ai_addrlen) < 0)
{
    perror("Connection failed");
    close(sockfd);
    freeaddrinfo(res);
    return;
}

// Send the request
char header[5128];
if (strcmp(port, "80") != 0 && strcmp(port, "443") != 0)
    sprintf(header, "GET %s HTTP/1.1\r\nHost: %s:%s\r\n\r\n", path,
        domain, port); // if it is a special port
else
    sprintf(header, "GET %s HTTP/1.1\r\nHost: %s\r\n\r\n", path,
        domain); // here not
// sprintf(header, "GET ls /"); // here not
send(sockfd, header, strlen(header), 0);

// Initialize a flag to indicate whether headers are still being read
int headers_finished = 0;
while (1)
{
    int byte_count = recv(sockfd, buf, sizeof(buf) - 1, 0);
    if (byte_count <= 0)
    {
        break;
    }

    buf[byte_count] = '\0'; // Null terminate the buffer

    // Check if we are still reading headers
```

```
        if (!headers_finished)
        {
            char *header_end = strstr(buf, "\r\n\r\n");
            if (header_end)
            {
                // Found the end of headers
                headers_finished = 1;

                // Calculate the length of the headers
                int header_length = header_end - buf + 4;

                // Append headers to result
                strncat(result, buf, header_length);

                // Check if there's more data after headers
                if (byte_count > header_length)
                {
                    strncat(result, buf + header_length, byte_count -
                        header_length);
                }
            }
            else
            {
                // Headers not finished, append all read data
                strcat(result, buf);
            }
        }
        else
        {
            // Headers are done, append body
            strcat(result, buf);
        }
    }

    close(sockfd);
    freeaddrinfo(res);
}
```

In the code above we build a HTTP request and send it to the server and then fetching the result. After that we send it as a HTTP response to the client that is able to handle it.

4.4 Usage Scenarios

This section presents real-world scenarios demonstrating the application's functionality, emphasizing its ability to handle standard download processes and manage interruptions effectively.

Scenario 1: Download from an External URL This is a basic application that is able to fetch resources from the entire internet, a activity that web browsers are doing on a daily basis.
Scenario 2: A web crawler A Web crawler, sometimes called a spider or spiderbot and often shortened to crawler, is an Internet bot that systematically browses the World Wide Web and that is typically operated by search engines for the purpose of Web indexing. The part of fetching resources from the internet is what we have implemented in our project and that could be extended for the purpose above.

5 Conclusions

The application successfully implements a robust web crawling mechanism with interruptible download functionality. Future improvements could include better error handling, adaptive download speeds, and cloud storage integration.

6 Bibliographic References

References

L. Alboaie *Computer Networks Course*.
Available: `https://profs.info.uaic.ro/~eugennc/teaching/ga/`.

Wikipedia contributors. *Wikipedia, The Free Encyclopedia*.
Available: `https://en.wikipedia.org/wiki/`.

Umangshrestha *Implementing HTTP from socket*
Available: https://medium.com/geekculture/implementing-http-from-socket-89d20a1f8f43