CS 421 - Computer Networks
Fall 2021 Programming Assignent 2
ParallelFileDownloader
Emrecan Kutay 21702500 Mehmet Berk Şahin 21703190

1. Introduction

In this assignment, we are asked to implement a program that downloads an HTML file to get a list of text file URLs and download these files in parallel. To achieve this, we implemented multithreading by using the Python programming language. Our program receives the URL of the HTML file and the number of connections for parallel downloading as parameters establishes parallel connections to get the content of each URL in the HTML file, and then does parallel downloading to get the content of each file. We got familiar with the internals of the HTTP protocol and we did not use any (third party, core, or non-core) API providing any level of abstraction specific to the HTTP protocol. However, we used the Socket package in default Python distribution, threading, and queue libraries. We used the latter two libraries to create multiple threads and receive the data from threads.

2. Implementation

In the implementation part, URL and connection number are taken from the user. Entered inputs were controlled to prevent undesirable errors like connection count is entered as 0. After verification, the main index URL connection was established. To communicate through the HTTP protocol, a TCP connection was established by using the Python Socket library. Firstly, an HTTP GET message was sent to get the base HTML file in the following format

“GET “ + requestURL + " HTTP/1.1\r\nHost: " + serverURL + "\r\n\r\n"

For instance, to have the HTML file of the first example, one should send

“ GET /~cs421/fall21/project1/index1.txt HTTP/1.1\r\nHost: www.cs.bilkent.edu.tr \r\n\r\n”

After the sender sees **200 OK** messages in the message’s header section, it extracts the URLs from the message, otherwise, it ignores the message and stops. For the extraction, the sender goes over each URL sequentially. Before creating multi-threads, the program sends a **HEAD** message to get the size of the content in the file. After acquiring the content size, we initialized multi-threads according to the number of connections entered by the user. If a number of connections are greater than the content length, it was set to content length since each connection will download a single byte, not more. Then, we calculated the data size that each thread should download simultaneously. The calculation is the following:

Let there be n connections for parallel downloading. Then, for every ith thread, we get

$$thread_i = roundup\left(\frac{\text{message size}}{\text{\# of connections}}\right) \; for \; i \in \{1, 2, \ldots, n-1\} \quad (1)$$

For $i = n$, we calculated the data size that should be sent by the last thread as

$$thread_n = \text{message size} - \sum_{i=1}^{n-1} thread_i \quad (2)$$

After calculating the data size for each thread, we initialized the thread objects and appended them to a list. The index of each thread object represents the message component. For example, if the data size that thread downloads is 200 Bytes, then the first thread downloads the range of 0–200. The second thread downloads the range of 200-400, and so on. We put the threads in order because if we did not do that, we would not be able to determine the order of the messages. Due to parallel downloading, message arrivals may come out of order, which is a very likely scenario. To prevent this issue, we ordered the threads and their ranges in the message, and after the **parallel downloading** was done, we concatenated the messages and saved the final message into a .txt file. Additionally, we have a thread function that has the link, interval, result, and count parameters. Link can be any URL under the base HTML file. The Interval parameter represents the range of content that should be extracted by a thread. The count is the index of the thread and it helps us to put the message into the correct place in the result list. That way, we maintain the correct order of messages coming from different threads.

3. Conclusion

All in all, we implemented **multithreading** to be able to download non-overlapping parts of a file in parallel. We take the URL and number of connections as input from the user. Links were taken from the main URL and head requests were sent to each to control their content. After checking threads were created to have parallel downloading. In the threads, sockets were created and range requests were sent with proper intervals. During the experimentation, it was observed that there are unordered packets due to variable delay. To solve this issue, a count variable was created and given to thread functions to several save received content in the proper indexes. Having the implementation, results were checked and verified. A detailed explanation was tried to be made in this report. For further explanation, one can check the comments in the code. These implementations are beneficial in understanding and observing parallel downloading and how parallel sockets are created and used.