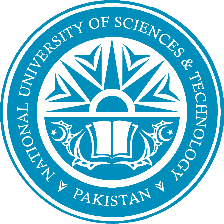
****

**School of Electrical Engineering and Computer Science,**

**NUST H-12 Islamabad**

**Department of Computer Science**

|  |
| --- |
|  |
| HTTP Downloader |
| Computer Networks Project  Multi Connection Downloader  Using TCP with  Resume Functionality |

|  |
| --- |
| **Ali Hassaan Mughal**  12-21-2018 |

The task of making a multi-connections TCP/UDP downloader was divided into the following specific sub-parts. Following are the divided sections for the downloader:

**Parsing Command Line Arguments:**

The parsing of command line arguments was done using a loop over all the arguments and checking the flag of each argument and assigning specific variables the values from the arguments into their corresponding required data type.

**Extracting Data from the Arguments:**

The parsed arguments after converting to specific types of data. The extraction of specific features from those parsed arguments like the name of file, the name of directory where the temporary files/chunks will be stored. Like the location of file to be stored on the PC, the specific Webhost address and the path of the webhost from which we can use the request to fetch the required resource.

**Make TCP IP Sockets:**

The TCP IP sockets are made according to the number of connections we require. We created an array of TCP IP Sockets.

**Connecting TCP Sockets:**

The TCP Sockets are connected to the web host using the port 80. These connections will be used for the downloading of the file.

**Making TCP Request for Header:**

Requesting the header of the file and extracting the size of the file using a Get request and splitting with the (“\r\n\r\n”) format and getting the first argument and then finding the “Content-Length” after splitting each row using new line and “:” to extract the size of file in bytes.

**Divide into Chunks:**

Setting the receive byte size, we divided the whole file into chunks using the formula

**[ j \* recvSize , (j + 1)\* recvSize - 1 ] for j in range( ( int(dataSize / recvSize )) + 1 )]**

**Creation of Threads:**

We created the threads, according to the number of connections the user wants. Each thread refers a function

**Assign Chunks to each Thread:**

Each thread is assigned the chunks using the formula **[ [ ( int ( len( fileChunksList ) / connections ) + 1 ) \* i , ( int ( len ( fileChunksList ) / connections ) + 1 ) \* ( i + 1 ) - 1 ] for i in range(connections) ].**

**Assign Thread to Connection:**

Threads are assigned to each connection with the starting and ending address of the chunk which it has to download.

**Making TCP Request for Data using Single/Multiple Connections:**

The TCP request were sent for each connection using the starting and ending address of the first and last chunks assigned to the thread. Each chunk before downloading is checked against two variables one is In Use and the other is whether it has been downloaded or not. This doesn’t request for the request of chunk again.

**Receiving the Data from Requests:**

After receiving the data, the data is split using “\r\n\r\n” to check whether the data is giving partial content headers with the data, if so removes that header and then stores the rest of data into the chunk file.

**Maintaining the Resume File:**

After every 100 chunks are downloaded, the \_resume.txt file is updated for the file under download so that, it can be resumed for the next time. The information we stored in the resume file is the number of connections used to download the file from the start, the size of file that was before and the number of chunks we created of the file.

**Joining the Chunks into one File:**

The join chunks function basically checks whether the all the chunks of the file have been downloaded. If they all are downloaded, then it checks where to create the output file, and opens all the chunks one by one and concatenates each chunk from starting chunk to ending chunk and creates a final file.

**Removing the Temporary Files:**

After joining the file into the final part, the temporary folder for storing the chunks and \_resume.txt file for the original file is removed using the rmtree function of python. It removed the whole tree of the file structure and is very efficient in performance also.

**About UDP Part:**

We tried implementing the UDP part of the assignment, but the HTTP server didn’t allow the connection and hence it didn’t reply to the request. It didn’t work out with the SOCK\_DGRAM.