Computer Networks - II

Programming assignment - II

Saurabh (CS14BTECH11031)

1. Taking URL and number of connections as command-line arguments :

```
if len(sys.argv) != 3 :
    print "Usage : python client.py [URL] [n]"
    exit(0)

url, no_of_connections, cr_lf = sys.argv[1], int(sys.argv[2]), '\r\n\r\n'
```

An error check is performed if the command-line arguments are not appropriately given. The arguments are stored in variables **url** and **no_of_connections**.

2. Creating HTTP headers for sending HEAD and GET requests :

The program creates raw HTTP headers for sending HEAD request (in order to query for the availability of the server/site for file download, and getting the size of the file) and GET request (in order to download the file). The requests are created as below :

```
head_request = "HEAD " + url_path + " HTTP/1.1\nHost: %s\r\n\r\n" % (host_address)
```

The variable **url_path** contains the path of the file's location on the server and the **host_address** variable is the front part of the url, specifying the address of the parent host.

The question also asks to create **HTTP Range Request** to query for a specific chunk of the main file, and later append all the downloaded parts to get the original file. The range request is created as follows :

```
get_request = "GET " + url_path + " HTTP/1.1\nHost: %s\nRange: bytes=%s\r\n\r\n" % (host, byte_range)
```

The new variable, **byte_range**, is a string, which contains the specific range of bytes to be sent to the server in order to download that particular chunk in the GET request. The variable's value is of the form `'%s-%s' % (start, end)`, where **start** is the starting byte and **end** is the ending byte of the chunk.

The above raw HTTP headers are then sent to the server by creating a socket, and then invoking the **send** call, and the server appropriately responds with the status code (*200 - OK, 400 - Bad Request, 404 - Not Found*, etc.), content-type, content-length, connection-type, etc. in the headers, and in case of GET request, the content (in the form of text/html or text/javascript). Note that, the HEAD method is identical to GET except that the server MUST NOT return a message-body in the response and the meta information contained in the HTTP headers in response to a HEAD request is identical to the information sent in response to a GET request.

Also, the HEAD request also helps to see if the file download is feasible or not, by checking the response code, and exiting from the program is the status code is not 200.

```
if "200" not in request_line :
    print "Download failed. Please try a different URL, Error code :", request_line[9:]
    exit(1)

if 'content-length' not in headers.keys() :
    print "Nothing to download, please try a different URL."
    exit(0)

file_size = headers['Content-Length']
if file_size == 0 :
    print "Nothing to download (content-type : 'text/html' or similar), please try a different URL."
    exit(0)
```

The later two checks are for checking whether the header *'Content-Length*' is present in the response sent by the server (since for a file download, '*Content-Length*' is necessary) and if there is a *'Content-Length*', then there is no file, that is '*Content-Length*' is 0, which means there is mostly text/javascript present on the specified URL and hence can't be downloaded.

Socket creation and sending the HEAD Request :

```
head_request = "HEAD " + url_path + " HTTP/1.1\nHost: %s%s" % (host, cr_lf)
socket_id = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try :
    socket_id.connect((host, 80))
    socket_id.send(head_request)
    recv_head = socket_id.recv(1024)
except socket.error, e :
    print "Exception caught socket.error : %s" % e
```

The socket is created using the **socket.socket([family[, type[, protocol]]])** call, and the HEAD request is sent as a raw HTTP request in the socket.send() call.
The **socket.socket([family[, type[, protocol]]])** opens a socket of the given type. The family argument specifies the address family which defaults to AF_INET in our case. The type argument specifies whether this is a stream (SOCK_STREAM in our case since we are using TCP) or datagram (SOCK_DGRAM) socket. Socket.socket is basically a method of the **_socketobject(__builtin__.object)** which has other class methods such as *send()*, *sendall()*, *recv()*, *accept()*, *bind(), close()*, *shutdown()*, etc.

3. Get the file size from the server using *HTTP HEAD Request* :

The file size is captured using the **'Content-Length'** header of the response sent by the server. The headers (captured using *socket.recv()* system call) are converted into a 'key-value' pairs dictionary using *Mimetools* and *StringIO* package, and the file size is the value corresponding to the key **'Content-Length'**.

```
file_size = headers['Content-Length']
if file_size == 0 :
    print "Nothing to download (content-type : 'text/html' or similar), please try a different URL."
    exit(0)

print "The file size is %s bytes (%s)." % (file_size, parse_size(file_size))
```

The *file_size* is obtained as the value of the key, '*Content-Length*' in the *headers* dictionary.

4. Create N parallel TCP sockets to the server and request specific chunk using *HTTP Range Request :*

Once the HEAD request returns with an OK 200 status code, and the file size is obtained from the server, the code creates **N** threads using the *threading* library in Python.

```
threadpool = [Thread(target = thread_download, args = (i, byte_range))
              for i, byte_range in enumerate(get_range(file_size, no_of_connections))]

for thread in threadpool :
    thread.start()

for thread in threadpool :
    thread.join()
```

The *threadpool* variable contains the list of the required number of threads, the target function is the **thread_download** function, with arguments being the thread_number and the specific format of *byte_range* as described earlier. *get_range(file_size, no_of_connections)* is

the list which contains the byte ranges for their corresponding threads and looks like this for a 1024 Byte file download with N (or no_of_connections) = 10 :

```
>>> get_range(1024, 10)
['0-102', '103-205', '206-307', '308-409', '410-512', '513-614', '615-716', '717-818', '819-921', '922-1023']
```

We can easily notice that all byte-ranges are equal in magnitude, that is, each thread queries an equal amount of chunks for the given file to be downloaded, as per the requirement of the question. The following is the thread_download function, where the specific thread downloads the chunk specified by the given byte_range argument to the function :

```
def thread_download(thread_no, byte_range) :
    socket_id = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    head_request = "HEAD " + url_path + " HTTP/1.1\nHost: %s\nRange: bytes=%s%s" % (host, byte_range, cr_lf)

    try :
        socket_id.connect((host, 80))
        socket_id.send(head_request)
        recv_head = socket_id.recv(1024)
    except socket.error, e :
        print "Exception caught socket.error : %s" % e

    headers = mimetools.Message(StringIO.StringIO(recv_head.split('\r\n', 1)[1]))
    get_request = "GET " + url_path + " HTTP/1.1\nHost: %s\nRange: bytes=%s%s" % (host, byte_range, cr_lf)
    socket_id.sendall(get_request)
    response, remaining = get_response_header(socket_id)
    cur_size, thread_data[thread_no], content_length = 0, remaining, int(headers['Content-Length'])
    while cur_size < content_length :
        thread_data[thread_no] += socket_id.recv(1000000)
        cur_size = len(thread_data[thread_no])
    socket_id.shutdown(1)
    socket_id.close()
```

First, a socket is created using the *socket.socket()* method. Then, a HEAD request is sent to first query the size of the chunk, then, the response of the server is parsed using *mimetools* and *StringIO* libraries in Python, and the file size is obtained as the value of the *'content-length'* key in the obtained dictionary.

Then, we keep receiving the chunk in subsequent *socket.recv()* calls until the total received size (stored in the *cur_size* variable, which gets updated in every iteration) is equal to the queried size. We need multiple *socket.recv()* calls because TCP/IP is a *stream-based* protocol, not a *message-based* protocol, and *socket.recv(buffer)* can receive at most *buffer* number of bits, but can receive as minimum as 1 or even 0 bits, and there is no guarantee how many bits will be received in a single *socket.recv()* call.

Also, the received data keeps appending to *thread_data[i]*, where *thread_data* is a global dictionary, with the thread numbers as indices (keys) and the received data as the corresponding value.

One important thing to do with the downloaded data/chunk is to separate the HTTP headers provided in the response, which are present alongside the content downloaded, in order for the data to be unaltered. So, we remove these headers first using the *get_response_header* function, as follows :

```
def get_response_header(sock) :
    data = ''
    while '\r\n\r\n' not in data :
        part = sock.recv(1024)
        if not part :
            # socket closed
            break
        data += part
    header, boundary, remaining = data.partition('\r\n\r\n')
    return header, remaining
```

It's clear from the code that we keep 'discarding' the data in the downloaded stream unless we encounter an CRLF (***'\r\n\r\n'***), marking that we have reached the end of the headers and now the real data content begins. The *while-loop* of sending multiple *socket.recv()* calls happens after this, where the real data is downloaded.

5. Combine the received chunks and create the original file :

Having said this, we create the threads, dispatch each one of them, and later join them with main thread once they finish their respective execution (download).

```
for thread in threadpool :
    thread.start()
for thread in threadpool :
    thread.join()

filename_ = url_path.split("/")[-1]
filename = filename_.split(".")[0] + "_download." + filename_.split(".")[1]
final_data = sorted(thread_data.iteritems())
with open(filename, 'wb') as fh :
    for _i, part in final_data :
        fh.write(part)
```

After downloading, we first sort the downloaded parts in order to retain the same order of bits in the original file, and then, open a new file in *binary* mode, and write the data of each of

the threads to the opened file, thus, making a complete downloaded file. The file name is same as that of the original file on the server (*url_path.split("/")[1]* would give "ubuntu16_04.iso" if the url path is "/files/ubuntu16_04.iso", which is the real file name) + "_download", to avoid name collision ("ubuntu16_04_download.iso" in this case).

6. Verify that the received file is not corrupted using checksum of the actual file :

This is done using two methods, one using the Linux's *cksum* command, and the other being a complex algorithm, *sha256,* which in a similar way, generates a complex hashcode for the given file. The following is the code performing the checks using above mentioned methods :

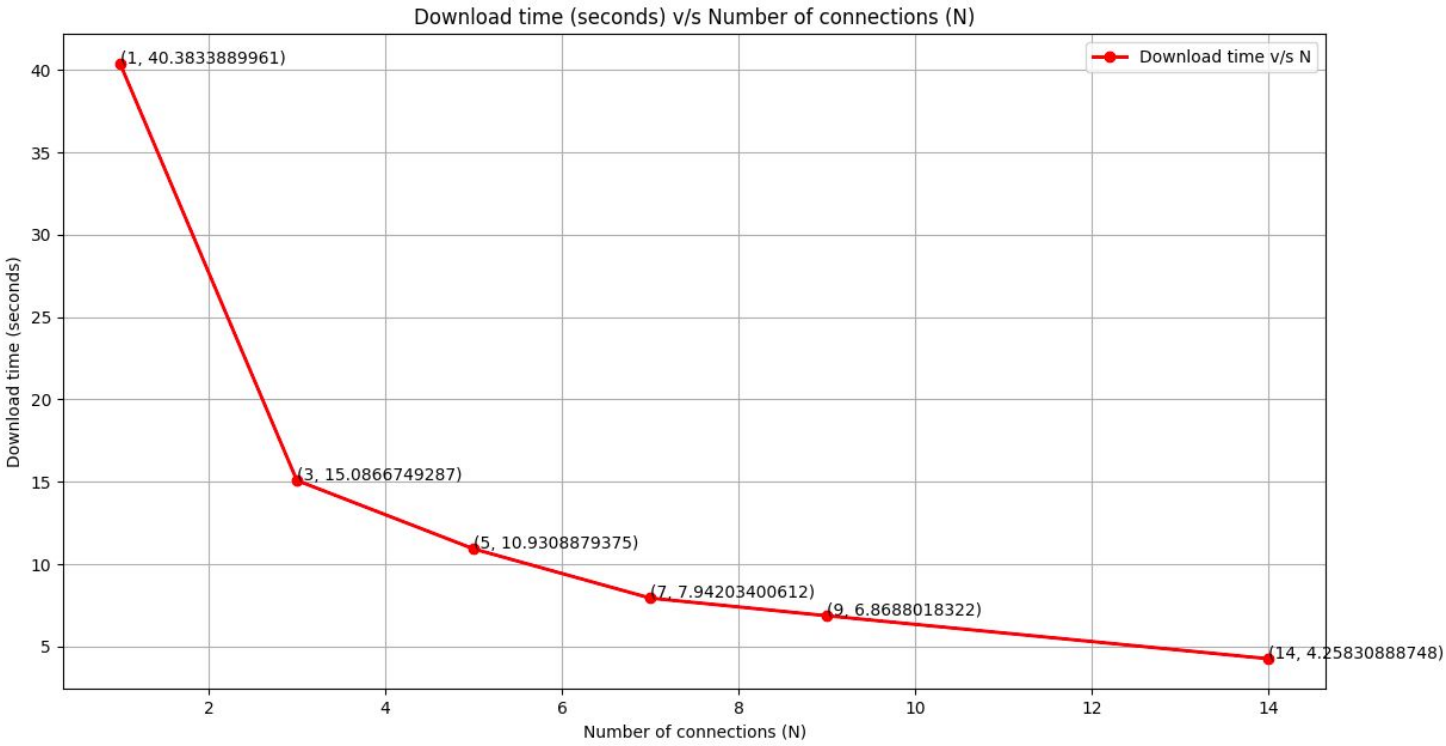
```
# perform checksum check using Linux's 'cksum' command
print "\nChecksum check using Linux's 'cksum' command :"
print "============================================="
cksum = subprocess.check_output("cksum " + filename_ + " " + filename, shell = True)
original, downloaded = cksum.split("\n")[0].split(" ")[0], cksum.split("\n")[1].split(" ")[0]
print "Downloaded file not corrupted !" if original == downloaded else "Downloaded file corrupted !"

print

# perform checksum check using 'sha256' secure hash algorithm
print "Checksum check using 'sha256' secure hash algorithm :"
print "================================================="
original, downloaded =  hashlib.sha256(open(filename_, 'rb').read()).digest(),  hashlib.sha256(open(filename, 'rb').read()).digest()
print "Downloaded file not corrupted !" if original == downloaded else "Downloaded file corrupted !"
```

Studies performed and inferences made :

The impact of using parallel threads and creating parallel sockets to the server is measured by calculating the time taken for a download to finish. This is done by running another code provided in the assignment directory, named, ***analyze.py***, which runs the main source code several times, increasing the number of connections, N, in every iteration, recording the download time, and then plotting a graph, showing how the download time varies with the number of connections (N) made.

Another metric involved is by measuring the throughput of each thread/stream. Each stream's estimated throughput is more or less the same as compared to that in a single download, but also suggests that multiple parallel downloads affect one another. We observe this when the number of connections is increased too much (around 30), and the observed throughput is decreased as well as the download time increases. This may happen due to the problem of resource competition and thus the conflict arising, or a good bit of overhead to establishing a lot of TCP, and then the HTTP connections, which may increase the download time as well as decrease the throughput. Also, maintaining state information in TCP causes overhead as well.

The following is the graph obtained when downloading an image of size 3.4 MB, showing the download time taken v/s the number of connections (N) made :



Clearly, we can observe that the download time decreases drastically when the number of parallel connections is increased from 1 to 3, and then, a gradual decrease in download time is observed as the number of connections increase upto a maximum of 14.

Thus, a general inference can be drawn that in case of parallel sockets downloading data from the server, there exists a higher aggregated downloading throughput and therefore shorter downloading time experienced by the client(s). However, the number of parallel connections, N, shouldn't be too large, as theoretically, it may suggest that download would be fast but in practical situations, this creates a huge overhead, since TCP has to maintain state information, and the server may also not respond well to too many parallel GET requests.

The following are the screenshots of the code's working, when downloading an image of size 5.31 MB. The only parameters changing in every run of the code is the number of connections (N), and again, we can observe that the download time varies inversely with the number of connections made. The number of connections is provided as the second command-line argument, where the first command-line argument is the download URL :

N = 2, Download time = 33.14 seconds :

```
saurabh@saurabh-Inspiron-3542:~/Desktop$ python client.py https://pixabay.com/get/e83cb4062af6083ecd1f4102e54c4692e56ae3d111b9194990f0c
879/homberg-1959229.jpg?attachment 2
The file size is 5566586 bytes (5.31 MB).

Finished downloading file 'homberg-1959229_download.jpg'.
Downloaded file size : 5566586 bytes.
Time taken : 33.1435039043 seconds.

Checksum check using Linux's 'cksum' command :
==============================================
Downloaded file not corrupted !

Checksum check using 'sha256' secure hash algorithm :
=====================================================
Downloaded file not corrupted !
```

N = 5, Download time = 14.61 seconds :

```
saurabh@saurabh-Inspiron-3542:~/Desktop$ python client.py https://pixabay.com/get/e83cb4062af6083ecd1f4102e54c4692e56ae3d111b9194990f0c
879/homberg-1959229.jpg?attachment 5
The file size is 5566586 bytes (5.31 MB).

Finished downloading file 'homberg-1959229_download.jpg'.
Downloaded file size : 5566586 bytes.
Time taken : 14.6119418144 seconds.

Checksum check using Linux's 'cksum' command :
==============================================
Downloaded file not corrupted !

Checksum check using 'sha256' secure hash algorithm :
=====================================================
Downloaded file not corrupted !
```

N = 8, Download time = 12.16 seconds :

```
saurabh@saurabh-Inspiron-3542:~/Desktop$ python client.py https://pixabay.com/get/e83cb4062af6083ecd1f4102e54c4692e56ae3d111b9194990f0c
879/homberg-1959229.jpg?attachment 8
The file size is 5566586 bytes (5.31 MB).

Finished downloading file 'homberg-1959229_download.jpg'.
Downloaded file size : 5566586 bytes.
Time taken : 12.1607789993 seconds.

Checksum check using Linux's 'cksum' command :
==============================================
Downloaded file not corrupted !

Checksum check using 'sha256' secure hash algorithm :
=====================================================
Downloaded file not corrupted !
```

N = 12, Download time = 9.66 seconds :

```
saurabh@saurabh-Inspiron-3542:~/Desktop$ python client.py https://pixabay.com/get/e83cb4062af6083ecd1f4102e54c4692e56ae3d111b9194990f0c
879/homberg-1959229.jpg?attachment 12
The file size is 5566586 bytes (5.31 MB).

Finished downloading file 'homberg-1959229_download.jpg'.
Downloaded file size : 5566586 bytes.
Time taken : 9.66265583038 seconds.

Checksum check using Linux's 'cksum' command :
==============================================
Downloaded file not corrupted !

Checksum check using 'sha256' secure hash algorithm :
=====================================================
Downloaded file not corrupted !
```