Uğur Erdem Seyfi
21801744

CS421 - PA2

For this assignment I have used Python3. The program starts with the code described in the **main** function. The use logic of the multithreading is put into **process_download** and **download_thread** methods. Other than that, the methods that are used to send and retreive data according to HTTP are also described below. **http_request** method uses Sockets to establish data transfer between the HTTP server. All of the other http methods I described uses **http_request** method as their core. I have also implemented methods to parsing http (**parse_http**) string and retreiving the urls (**extract_urls**) from content of the index file.

main()
This is where the program starts to run, it first checks the index file, and then starts to iterate through the urls in the index file.

```
def main(argv):
    argv_len = len(argv)
    if(argv_len == 0):
        return False  # throw an error
    index_url = argv[0]
    connection_no = int(argv[1])

    print("URL of the index file: {}".format(index_url))
    print("Number of parallel connections: {}".format(connection_no))

    response, _ = http_request_get(index_url, DEFAULT_RECEIVE)

    if('200' not in response['status']):
        raise Exception('Requested file is not found')
        sys.close()

    data = response['data']
    urls = extract_urls(data)

    print("Index file is downloaded")
    print("There are {} files in the index".format(len(urls)))

    # list the files
    count = 1
    for url in urls:
        download_status = process_download(url, connection_no)
        print("{}. {} {}".format(count, url, download_status))
        count = count + 1
```

process_download
This is th main function where the process of downloading is handled. Multi-threading is also handled here.

```
def process_download(url, connection_no):
    _res, head_len = http_request_head(url)
    # check status
    if('404' in _res['status']):
        return "is not found"
    else:
        content_length = int(_res['Content-Length'])
        # if file found,decide whether its downloaded or not
        data_per_thread = math.ceil(content_length / connection_no)
        prev_thread = None
        file_parts_str = ""
        for i in range(connection_no):
            start = i * data_per_thread
            end = min(content_length, (i+1) * data_per_thread)
            file_parts_str += "{}:{}({})".format(start, end, end-start) + \
                (", " if i != connection_no - 1 else "")
            prev_thread = threading.Thread(
                target=download_thread, args=(url, start, end, prev_thread))
            prev_thread.start()
        prev_thread.join()
        return "(size= {}) is downloaded\r\nFile parts: {}".format(content_length, file_parts_str)
```

download_thread(url, start, end, prev_thread)

This is the function that is called in different threads. First three arguments are obvious, prev_thread is taken to prevent starting to write to the file storage before the previous thread has completed its writing process.

```
def download_thread(url, start, end, prev_thread=None):
    _res, _ = http_request_get_range(url, start, end)
    if(prev_thread is not None):
        prev_thread.join()  # do not start writing file before the prev thread has been written
    # write to the file
    file_name = url.split('/').pop()
    with open(file_name, 'a') as f:
        f.write(
            _res['data'])
        f.close()
```

extract_urls(data)

This is an helper method. This method uses regex to find the urls in the given data string and returns the found urls as a list.

```
def extract_urls(data):
    urls = []
    url_pattern = re.compile(
        "((http|https)\:\/\/)?[a-zA-Z0-9\.\/\?\:@\-_=#]+\.([a-zA-Z]){2,6}([a-zA-Z0-9\.\&\/\?\:@\-_=#])*")

    for url in data.split('\\n'):
        if url_pattern.match(url):
            urls.append(url)
    return urls
```

http_request_get_range(url, start, end, head_len)

This method acts like http_request_get method described below but also specifies which range of bytes of the content to retreive.

```
def http_request_get_range(url, start, end, head_length=DEFAULT_RECEIVE):
    target_host, target_endpoint = url_to_target(url)
    # send some data
    request = "GET {0} HTTP/1.1\r\nHost:{1}\r\nRange:bytes={2}-{3}\r\n\r\n".format(
        target_endpoint, target_host, start, end)
    return http_request(target_host, request, end - start + head_length)
```

http_request_head(url), http_request_get(url, content_length, head_len)
These structurally very similar methods are using the **http_request** method in order to send a HEAD and GET requests to the server. These methods use a method called **url_to_target** which is created for the purpose of seperating the host and endpoint data.

```
def http_request_head(url):
    target_host, target_endpoint = url_to_target(url)
    request = "HEAD {0} HTTP/1.1\r\nHost:{1}\r\n\r\n".format(
        target_endpoint, target_host)
    return http_request(target_host, request, DEFAULT_RECEIVE)


def http_request_get(url, content_length, head_len=DEFAULT_RECEIVE):
    target_host, target_endpoint = url_to_target(url)
    # send some data
    request = "GET {0} HTTP/1.1\r\nHost:{1}\r\n\r\n".format(
        target_endpoint, target_host)
    return http_request(target_host, request, content_length + head_len)
```

http_request(target_host, request, recv_len)

This method is used for sending all kinds of different HTTP requests. It creates a socket for each request, it closes the socket after it receives the data.

```
def http_request(target_host, request, recv_len):
    client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # socket object

    # connect the client
    client.connect((target_host, HTTP_PORT))

    # send some data
    client.send(request.encode())

    # receive some data
    response = client.recv(recv_len)
    http_response = repr(response)
    http_response_len = len(http_response)

    client.close()
    return parse_http(http_response), http_response_len
```

parse_http(response_string)

This method parses the response http string in a way that properties and their content can be easily handled in a key pair structure. This is actually a helper method that is used by **http_request** method.

```
def parse_http(http):
    if("Content-Type: text/plain\\r\\n\\r\\n" in http):
        fields = http.split("Content-Type: text/plain\\r\\n\\r\\n")
        output = parse_http_header(fields[0])
        output['Content-Type'] = " text/plain"
        output['data'] = fields[1]
        return output
    return parse_http_header(http)
```

```
def parse_http_header(http):
    fields = http.split("\\r\\n")
    output = {}
    output['status'] = fields[0]
    for field in fields[1:]:
        if not ':' in field:
            continue
        key, value = field.split(':', 1)
        output[key] = value
    return output
```