README.md

Assignment 3: File Transfers in parallel using TCP and HTTP

Note: The assignment and guidelines below are not in its final version. Final version will be available soon.

A robust client to download content from multiple HTTP servers

In this Assignment and its Guidelines we will learn how to use Sockets to program client/server applications using the Java Language. For the assignment the final goal is to implement a Client/Server application for File Transfer based on HTTP. In the developed application, clients can download files from several web-servers used in parallel, in order to maximize the file transfer rate. For the assignmemt the implementation of the web-servers that will be provided as initial materal and the assignmemt consists in the developmet of the required client.

Backgorund

Reference for programming with TCP Sockets in Java

- You can find a **tutorial on programming with Sockets in Java Language** in https://docs.oracle.com/javase/tutorial/networking/sockets/**
- Remember that for the work assignmemt you will be particularly focused on the develpment of clients using TCP sockets (supporting HTTP Requests/Responses) because the HTTP
  servers to be used are provided in advance.
- You also have a convenient explanation in the text book of the course: **https://legatheaux.eu/book/cnfbook-pub.pdf, see chapter 5, section 5.3**.
- Complementarily you have these examples for your preliminary tests: **echo-client.java** and **echo-sever.java**: a very simple client/server application implementing an ECHO protocol.

Reference for programming with HTTP using TCP Sockets

As you know, HTTP is supported by the TCP transport protocol, and it operates in two basic variants (HTTP/1.0 - implementaing HTTP Request/Response with non-persistet connections, and hTTP/1.1: using persistent connections). Clients that interact with HTTP servers must send correct HTTP requests (with the proper HEADERS), sent as formatted requests sent in the TCP connection previous estabished with the server. Clients must be able to receive HTTP responses, processing them according to the HTTP protocol (interpreting the HEADERS and CONTENTS in the RESPONSE).

For the operation of the HTTP protocol you must consider the explanation in the theoretical classes.

- You can also study the HTTP protocol in the course textbook: **https://legatheaux.eu/book/cnfbook-pub.pdf, see chpter 12**, paying special attention to HTTP requests/responses using RANGE REQUESTS.

Assignment Motivation

In today's internet, most of the users consumed content is carried over the HTTP protocol. In the specific case of multimedia contents, the volume of the consumed information varies from a few Mbytes (in the case of photographs), up to several Gbytes (in the case of movies). It is not realistic to think of such bulky objects being transferred in a single HTTP request/reply interaction and using a single TCP connection. Inevitably, due to the high volumes of data, momentary anomalies in the network, or problems in the servers, it is necessary to resort to more than one interaction among the client and the server(s). In addition, in the case of movies, as they can take hours to play, it is not mandatory or interesting to transfer in only one chunk the full content, or from the same server. Also, a faster download may be achieved if transferring in parallel from several servers, using different HTTP ranges.

Goals

To complete this assignment you must program an HTTP client that must be able to transfer a voluminous file (e.g. above 100 Mbytes) from a set of HTTP "tricky" servers, in the shortest time. These "tricky" servers, whenever they receive a request of an object, may only send part of the requested object or break the connection in the middle of the transfer. Also, each server can exhibit variable transfer performances. Servers accept ranges requests from clients.

FIRST Delivery (or Minimal Goal)

Your program should be an HTTP client and mustbe called GetFile (GetFile.java). It must be able to correctly download a (huge) file from a HTTP "tricky" server. For the implementation and testing purposes of this minimal goal, you can use the provided server HttpTrickyServer.java.

Use the following command to start this server (it runs by default in port 8080 (TCP): $java HttpTrickyServer

As an example, for downloading the IFB.mp4 movie trailer in your computer, your client will be run in the following way: $java GetFile http://localhost:8080/IFB.mp4

SECOND Delivery

For the second delivery, your program should be able to correctly download a (huge) file, now from a set of HTTP "tricky" servers. For testing, you can start a pool of several servers in the same machine, running in parallel. For this purpose you must use a different port for each one. On Unix-like system you can use a command sequence like the following (all servers launched in background):

```
$java HttpTrickyServer 8080 &
$java HttpTrickyServer 8081 &
$java HttpTrickyServer 8082 &
$java HttpTrickyServer 8083 &
```

Note: You can also use the available script to launch the four servers. See the script serverclusterstart.sh

As an example, for downloading the IFB.mp4 movie trailer in your computer, from the above servers, also in your computer, your client will be run in the following way:

```
  $java GetFile http://localhost:8080/IFB.mp4 http://localhost:8081/IFB.mp4
http://localhost:8082/IFB.mp4 http://localhost:8083/IFB.mp4
```

In your implementation you can implement whatever solution you prefer:

```
  a) You can send a set of successive requests to the same server;
  b) You can send requests to several different servers, for example in a round-
robin way;
  c) You can send requests to several different servers, in parallel;
  d) Any other policy, according to the evolution of your experimental
observations and decided optimizations.
```

Note: each server only serve one client after the previous one (it's not a concurrent server).

Output statistics

It is mandatory that your programs (delivery 1 and delivery 2 clients) ust collect and output the following statistics:

- Time elapsed to complete the full transfer (in seconds)
- Total number of bytes downloaded (in bytes)
- End-to-end average bitrate of the full transfer (in bytes/sec)
- Number of requests performed by the client during the file transfer
- Optional: average size of the payload of each HTTP reply (in bytes)
- Optional: average time spent in each request/reply (in milliseconds)

Use the following output format (where the values are only indicative):

```
Total elapsed time (s): 41.668
Download size (bytes):  14744835
End-to-end debit (Kbytes/s):  353.865
Number of requests: 4
```

Delivery Rules:

The client can be completed and tested by students/groups (max t2o students). Groups will deliver the results and files using the submission form for thsi assignmemt 3.

Additional materials

All the needed materials (testing files, programs, scripts, ...) for the assignmet are available in this GitHub repositpry.