Université de Liège

Introduction to computer networking

Second part of the assignement : Report

3rd engineering bachelor

Authors :
Tom Crasset
Antoine Louis

Professor :
G. Leduc

Assistant :
S. Hiard

Academic year 2017-2018

Software architecture

Like the first project, we implemented a `Webserver` class as well as a `WebServerWorker` class. The former being the one launching the threadpool and placing a connection on one of the threads, the threads being `WebServerWorker` instances.

A thread is assigned for each HTTP request, and not for each different user. In this `WebServerWorker` class, the request is parsed using another class, `HTTPParser`, which basically parses the requests by extracting the request method, the path requested, the cookie, the http version and puts the remaining header fields into a map for easy access. Furthermore, the `WebServerWorker` class consists mainly of if-else statements to check the validity of the different request types, the paths and, in case of error, send an error page to the client using the `generateError()` function.

At the very start of the project, we already had one wish : modifying the classes from the first project as little as possible. From the original project, we only reused the `Worker` class, the thread that was running the server before.
We went ahead and, instead of using the sockets out- and inputstreams, we used a `PipedInput-` and `PipedOutputstream`, which lets two threads communicate between each other using streams. Thus, we barely touched the previous class, apart from removing all the socket interaction and unnecessary functions such as listing all the previous choices.

Thus, we needed a way to translate the output of the previous project and the requests from the `WebServer`. The solution was to use another class, `GameInterface`, which does the relay between the `WebServerWorker` and the `Worker`. It also creates a Mastermind game and starts it. This class is static, because we didn't have the need to create an instance each time because a map can hold all the game threads. We can therefore store every running game/Worker with the corresponding input and outpustreams in a map. To differentiate between the different users and their games (as the `WebServerWorker` class only handles connections, independently of users), a cookie is being passed as an argument to the methods and that cookie is also used as a key in the different maps.

To dynamically generate the HTML page, we used a class called `HTMLCreator`, which creates the mastermind web pages. In this class, we generate all the HTML, CSS and JavaScript code necessary to the game and format it into strings that will be sent to the client either by chunked encoding or by gzip compression.

Multi-threading coordination

We implemented a threadpool to handle all connections to the `WebServer` in a measure to prevent DDoS attacks on the server. These threads do not need to be coordinated. Also, as the threads in the previous project needn't be coordinated, we didn't have to do this here either. One thing that we payed attention to is to use a `ConcurrentHashmap`, in the case that the map is accessed exactly at the same time.

Limits

The cookie mechanism doesn't work perfectly. If one refreshes the page, the game is not saved. The game is only saved in one session, that means between POST and AJAX requests, the game state is saved (as should be).

Furthermore, we don't delete the game map entries if the game is still running (that means the player hasn't won or lost the game) even when the user has closed (or refreshed) the page. That leads to some unused memory that is only reallocated when the server is powered off.

In addition, we don't do chunked encoding AND gzip compression, we do either one or the other. This is due to the fact that compressing chunks is a waste of time and is actually counter productive and the opposite, namely chunking a compressed file, also goes against the chunking efficiency, because one first has to generate the whole page, compress it and then chunk it.

Possible improvements

The possible improvements are mainly the solutions to the things listed in the Limits section above, means for example saving the session of a game if somebody refreshes the page. This could be useful if someone switches from JavaScript enabled to Javascript disabled, then he could refresh the page and resume his game by keeping the same cookie and so his previous guesses.

In addition, we could also have done some improvements in relation with the difficulty level of the game. For example, we could have set a timer for a session. The client chooses a limited time for which he must find the color combination and as soon as time runs out and he didn't find the combination, he looses the game.
Furthermore, we could have let the client decide of the number of colors he must guess.
A last thing we could have done is to display the solution in case of a game over.