Assignment Etienne LAFARGE

1. How to get the program running

The program can be launched with default parameters using the following command :

```
python main_monitor.py /path/to/server/access_log
```

Calling it with the -h flag gives further information about the command line parameters available to custiomize the behaviour of the application (updating frequencies of different components, alert threshold…).

```
python main_monitor.py –h
```

To exit the program, just press Ctrl-C at wait a couple of seconds for all the threads to terminate (the time you have to wait depends on the update intervals you gave).

Tests

You can run the PyUnit test testing the logic of the OverloadMonitor by launching the following command in the root of the project:

```
python –m unittest
```

Wait for a couple of seconds for the test to execute (once again because of the update intervals). It shouldn't take more than 10seconds.

Format of the log file

The program is compatible with the default Apache2 log file format (see http://httpd.apache.org/docs/2.2/en/logs.html#accesslog).

In case it doesn't suit the W3C standars, I can quickly make the program compatible with this format as well.

Access log example :

```
127.0.0.1 - - [27/Jan/2015:06:45:53 +0100] "GET /test/test HTTP/1.1" 404 1082
```

```
127.0.0.1 - - [27/Jan/2015:06:45:55 +0100] "GET / HTTP/1.1" 200 14517
```

You can as well use the http_spammer.py script to launch requests to a server and test the program from there (call it with -h for details, you can update the requested paths by editing the script) :

```
python http.spammer.py –h –i [SPAM_INTERVAL] –d [DOMAIN]
```

2. Design choices

Modularity

I chose on purpose to think about the future of this application as I developed it.

It could have been writer faster, with fewer classes, less files but I did it the way I would have done it in a professional company : as modular as possible so that improvements or modifications would be easy, fast, and wouldn't have a negative impact on the code readability and cleanness.

In brief, I coded it as a program that would keep on living, not as a script fullfiling a very specific purpose.

Commonly used design patterns

- As in most applications, the **Observer** design pattern is omnipresent : GUI components observing status changes in some monitors, monitors listening for new requests to appear… It doesn't strictly respect MVC, at least, not in the folder structure but it is strongly inspired from it. I judged that the program wasn't big enough for a complete MVC approach and sticking to it would have resulted in a longer development.
- A very light variant of the **Dependency Injection** pattern is used, nothing fancy and big as you can have in web frameworks. For instance, the submonitors are a dependency of the `RequestMonitor` and are injected in it through the `MainMonitor`. It is a simple constructor-based dependency injection, without any “officially declared” dependency manager but it still results in a flexible way of adding new submonitors.

Technical constraints

The application must be able to **run forever**. That's why it doesn't store all the requests but only requests for a given timeframe (ex: the last two minutes). Therefore, the **amount of memory** needed is limited and there is no risk of having the server crash.

Some parts of the application (the `OverloadMonitor` for instance) try to modify the same data in different threads. I payed attention to using locks accordingly to avoid issues due to **concurrent thread access to the same resource**.

3. Application structure

The global application structure is described below. The scheme is just intended to give a better understanding about the way the application has been built.

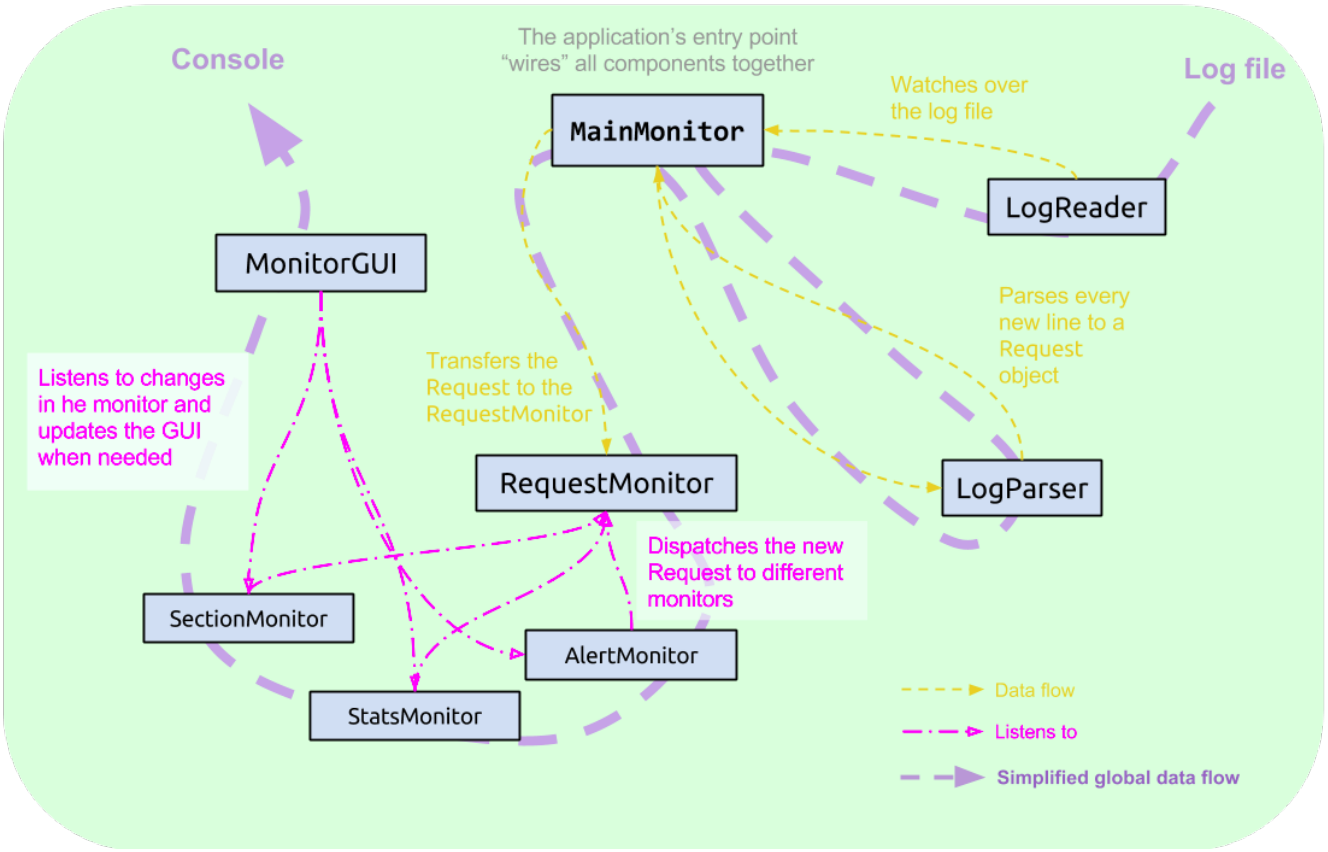

Figure 1 - Application structure

The entry point of the application resides in the MainMonitor object. A `__main__` routine parse the console arguments and launches and instance of MainMonitor passing it the given console parameters and launching the loop waiting for new lines in the `access_log` file.

- **`MainMonitor`** : the main component of the application, taking care of instantiating all the components and binding them as well (using the Observer design pattern or a very similar process). This is where dependency injections into different components is handled, passing dependencies in the different component's constructors.
- **`LogReader`** : detects when a line is appended to the given `access_log` file and then calls the `add_request` callback function of `MainMonitor`.
- **`LogParser`** (or **`ApacheLogParser`**) : parses every new line and instanciate a Request object out of it. `ApacheLogParser` (the only implementation of `LogParser` so far) is

compatible with the default Apache2 format. Of course, it would be very straightforward to create other implementations for different log file formats.

- **`RequestMonitor`** : is called every time a new request is added to the log file. It dispatches the submonitors. This class can be instanciated with an arbitrary number of submonitors (instances of classes implementing an `add_request()` method). This way, adding a new functionnality is often just a matter of creating a new submonitor, the corresponding View part of the program (here, it's just about printing the output of this monitor somewhere in the console) and to declare the new monitor in the `MainMonitor` constructor.
- **`SectionMonitor`** : a submonitor listing the different sections of the visited website and storing the number of hits for each one. It is refreshed every X seconds (by default, X=10)
- **`StatsMonitor`** : manages different statistics about the requests since the program is launched (number of requests, number of request that cannot be satisfied based on the HTTP response status code...)
- **`AlertMonitor`** : stores the request that occurred during the past two minutes and triggers an alert if a certain threshold is reached. When the traffic becomes normal again, it triggers another notification.
- **`MonitorGUI`** : the word GUI is very pretentious here. It is just a class listening to the different submonitors and printing what it gets on the console. Actually a console program is smart since our monitoring service will probably accessed via SSH but of course, a nicer interface would be essential for a professional use (something a bit Vim-like, with different tabs containing different stat panels...).

4. Possible improvements

In terms of user experience

Since the program is intented for a remote access (via SSH for instance), a console app is perfect. Yet, even in a console app, the layout could be improved, for instance it could be responsive (i.e. adapt to the size of the console). It could contain different tabs displaying different stats.

A web interface could also be a good alternative.

The current console program could also be refreshed less frequently so that less data is sent over the network when used remotely using SSH.

Adding new LogParsers to make the program compatible with the widest possible range of servers is also essential since it would widen the customer segment drastically. What's more it is easy to do with the chosen structure for the program.

In terms of functionnalities

Following the ideas developped in the previous section, making a View (a class having the same role as the `MonitorGUI`) sending the data as JSON over a secured connection could also be a big plus : it could be accessed remotely by GUI-based programs without having to send a big amount of data over the net. Providing such an interface could also enable system adminisrators to write scripts that respond to alerts (like redirecting trafic to another machine if the current one is overloaded for instance).

I thought that writing a monitor to keep a list of the URL throwing an error code (404, 500...) could be a good idea as well : the system administrator or webmasters could see which calls display a 404 and adapt their websites to show a more explicit error message or to redirect to the page where the required resource has been moved.

As well it would be good to keep track of all the request objects, but not in RAM (since it would result in a server crash when the memory is full). A standard SGBDR would be perfect for this. We can of course keep on storing information about recent requests in RAM to ensure that our monitoring system is fast. But giving access to any past request in any time interval would be something users would probably like to do.

Finally, a faster shutdown process would be great. I is probably rather simple to do but I really didn't have time to look into it so far.

Application design improvements

The rigor of the code could also be improved : using `type` and `isinstance` assertions for example, or declaring explicitly some interfaces (even if not mandatory in Python). For instance an interface holding an `add_request` method for all the monitors (`RequestMonitor` as well as the submonitors). Or even better, we should have used our `Observable`/`Observer` classes for the `LogReader`/`RequestMonitor`.

All this would make the code a bit more verbose but also makes things a bit more clear for somebody discovering the code. Many other languages would actually have forced us to do so (Java for instance).

Of course, we could as well use an explicit MVC (or at least MV) approach in the program. As well, a more structured approach to Dependency Injection would be essential if the program starts to go big.

5. Details about some technical decisions

Storage type for the most visited sections

Since I decided to make the timeframe during which to store the list of the most visited questions tunable (and even potentially infinite), it may be important that we consider the way we sort the list of the most visited sections. Indeed, this list can be big and furthermore, it has to be sorted since we only want the **most visited**.

First of all, dictionnaries do not support ordering, especially on the values. The `{section : views} dict` is therefore a bad idea. We decided to use a `list` of tuples `(section, views)` on which we apply our specific ordering : a descending one based on the value of `views`. Since we control the list from the very beginning, we can **order** it **on insertion** only. In addition, we can **rely on the ordering of the existing list** when inserting the element to **optimze the complexity**.

Here is what we did. When a section is hit, we check whether it was already in the list or not (complexity : O(n)). If not, we append it to the list since it has only one hit, which is the minimum number of hits we can have for a section (so it's logical to put it at the end).

If the section exists, we update the number of hits. But now, the element(s) before may have a fewer number of hits (if it was/they had the same before). So we swap our element, little by little, until the element on his left has a bigger (or equal) number of hits. In theory, this **bubble-sort like operation** has a complexity of n at most but in most cases, it will only have to perform a couple of left shifts or even none.

So what we managed to do is to have an insertion cost of O(n) and if the section list is maintained for a long while, the likelyhood to have an unexisting section (O(n)) or to shift the updated element a lot, (O(n)) will go down close to zero and most of our insertions will be done with a **constant complexity**. The complexity when researching if the newlyhit section is present in the list and where could be improved if we also add a sorting on the section's names. It could be done by storing in a separate list the section name and the position of the section in the list (exactly like when you declare an **index** on a specific field of a SQL table). The complexity when updating an existing section would then tend to O(ln(n)). I didn't do it here since in the described case because we drop this list every 10 seconds so that n is never very big.

The access to the list is also extermely fast, we just return it sorted. If we just want the ten most visited section, we just return the first ten elements and we're done.

Storage of the requests in the OverloadMonitor

The topic is more or less the same for the list of last requests. Actually we just store the `dateimes` of these in a sorted list that we also sort on insertions.

We need to perform two major operations on this list : inserting new queries and deleting the old ones. The fastest way to insert an element into a list is to append it, so a ascending order on the dates sounds

good.

When it comes to deleting, I made the assumption that Python truncating operations (`mylist[n::]`) were done in a constant time. It is probably not true, the complexity may definitely be closer to n but since this list is not supposed to get huge it is not a problem. In my view, optimizing the truncation would require to use a custom container : a linked list where you always have a pointer or reference on the first and the last element. Then the deleting starts from the beginning of the list (the oldest elements) and stops as soon as it finds a datetime younger than two minutes. So again, the worst complexity would be O(n) but in real life, just a few operations would be performed in 99,9 % of the cases.

A similar approach is implemented in my program to find up to which point we have to truncate the list to remove the old dates from it.

Etienne LAFARGE