Project Report

Performance Analysis and Modeling of Software Systems, Fall Semester 2018

Ettore Puccetti, [ettore.puccetti@alumnos.upm.es](mailto:ettore.puccetti@alumnos.upm.es)

# 1. Implementation overview

## 1.1 Tag cleaning

For extracting the text from the HTML document, it was needed to ignore all the characters in between the angular parenthesis that identify an HTML tag.

I use a Boolean flag (tagOpened) that was set to *true* when I parsed a ‘<’ and to *false* when I parsed a ‘>’. So, while parsing the document char by char, I stored the result only if the Boolean flag was set to false.

Since the starting point of the document was set randomly, first of all it was needed to check if we already are inside an HTML block. And it can be done by starting parsing the document without saving the result, checking if we find first a ‘<’ or a ‘>’, setting the Boolean flag properly, and starting again.

## 1.2 Statistic collection

In the server, I use four *HashMaps<ClientID, arrayOfTimes>,* named properly, for storing the amount of time taken by the four tasks, for every client request processed. So, every client has an Array associated with it, where I can store the statistics associated to his request. This has been useful for the Multithread version of the Server, where each client was assigned always to the same thread worker (who also stored the statistics), and results could be written to the data structures without need of any synchronization mechanism, because it can never happen that two workers thread need to write to the same array associated to a client.

Then, when all clients have disconnected, I can aggregate them.

Times are calculated using the java function System.nanoTime() before and after every task, and subtracting the two times.

Both the client and the server use a *fileWriter* object to print the statistic on a file.   
The server produces 4 files, one for each task measured, containing the following information: total number of times the task has been done, average response time, 100 percentiles of the response time.

The client produces two files, in the first one (*ClientIntervalPID[[1]](#footnote-1).csv*), he writes the throughput measured in the last segment of time (~0.5 seconds), while in the last two line he reports the total time of the client run, the average throughput, and the average response time. In the second file (*clientPercentPID.csv*) he reports all the 100 percentiles of the response time.

## 1.3 Multithread implementation

### The Server

The main thread is the skeleton of the server. He initializes all the data structure and launch dispatcher thread. He accepts the requests from clients, he read the lines of text from sockets, he cleans them from HTML tag, and then he puts them in a *LinkedList, requestQueue, notifying* the dispatcher thread*.* He also aggregates all the statics and writes them to file.

### The dispatcher

The dispatcher creates and initializes the worker threads. To each worker is assigned an index.  
Then, he waits for new requests in the request queue, once he founds it non-empty, he extracts the first request (in FIFO order) and assign it to a worker thread. The assignment is done by calculating the modulus between the ClientID associated to the request and the number of workers.  
Requests are passed to the threads using an *HashMap<ThreadIndex ,Request>* called *requestMap*. If the choosen worker is busy (not waiting), the dispatcher waits until he gets a signal from him.

### The workers

The worker threads wait for a new request associated to their index in the HashMap. They process it, they send the result back to the client and then they *signal* the dispatcher in case the dispatcher he is waiting for assigning him a new request.

### RequestQueue – mutual exclusive access

The accesses to *requestQueue* are done in mutual exclusive way, by both the server and dispatcher thread.

### RequestMap – mutual exclusive access

The access to each element in the *requestMap* is done in mutual exclusive way, between the dispatcher and the only one thread who can access the request associated to his index. I implemented this behavior with an array of objects, one for each worker threads, representing the locks.

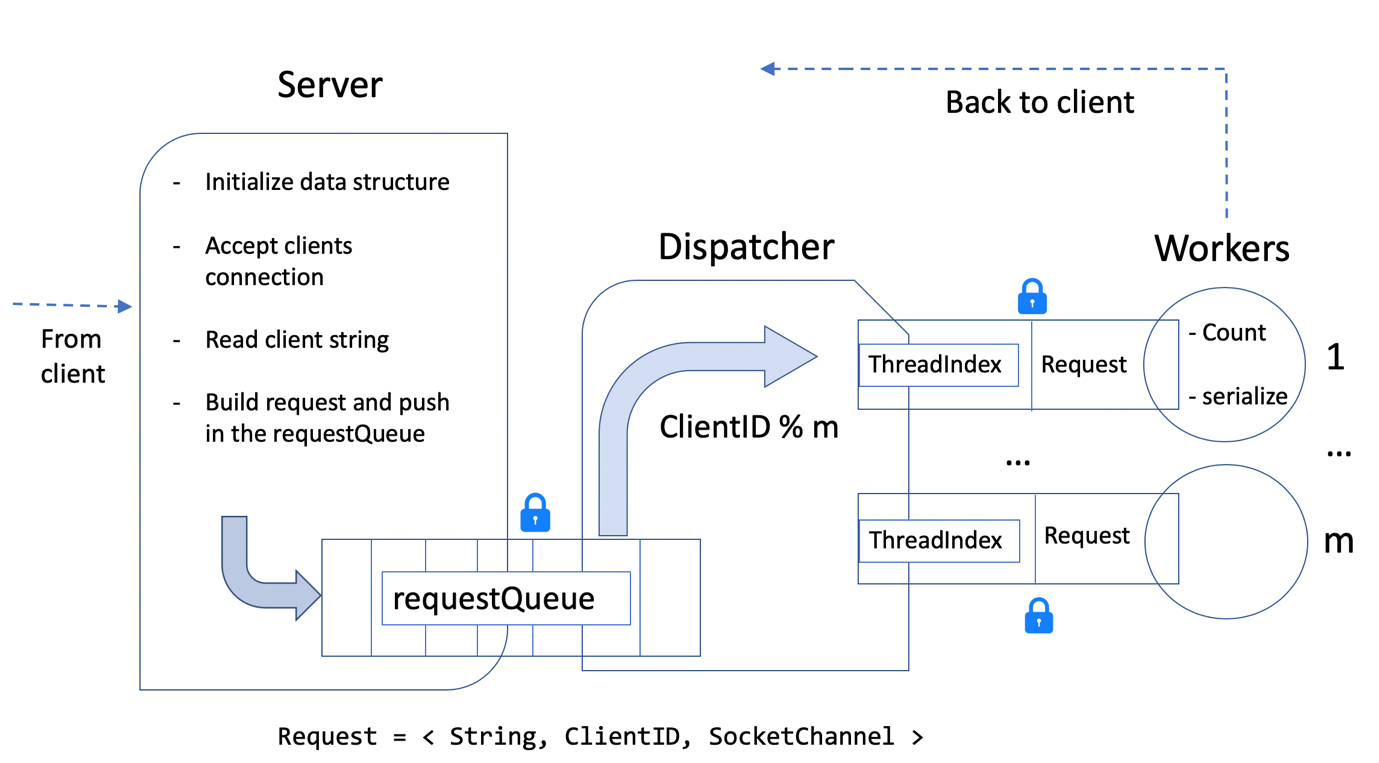


Figure 1: Multithread server structure

# 2. Baselines

## 2.1 Test Environment and Experimental Design

The system on which I run the experiment is an Ubuntu desktop, running on a virtual machine, hosted in a server with 16 cores CPU (Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz) and 16 GB of ram.

For the first set of experiment, with fixed document size, I set eight worker thread for the multithread version, while I variate the number of repetitions, decrementing them with higher number of clients, for having a total time a more similar over the experiments.  
Plotting the first 15 seconds of the average throughput over the clients, I decided to fix the *warm-up* time to 2 seconds.

For the second experiment, I use the multithread version of the server, with 8 worker, and the number of operations equals to 20k.

Considering that most of the experiments lasts more than one minute, removing or not the measurement taken during the warm-up time, does not change the plots.

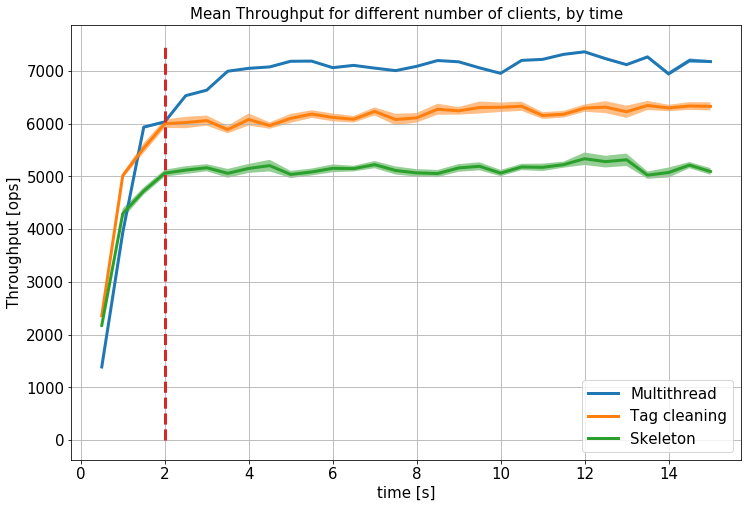


Figure 2: mean throughput by time and warm-up threshold

## 2.2 Throughput and Response Time

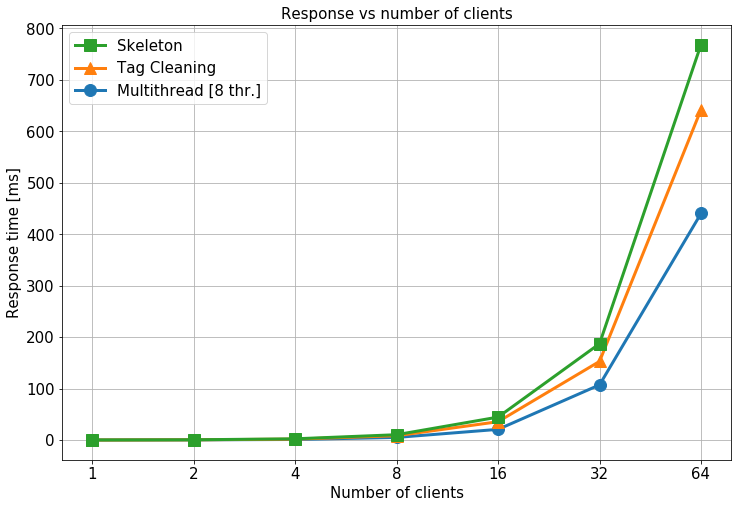
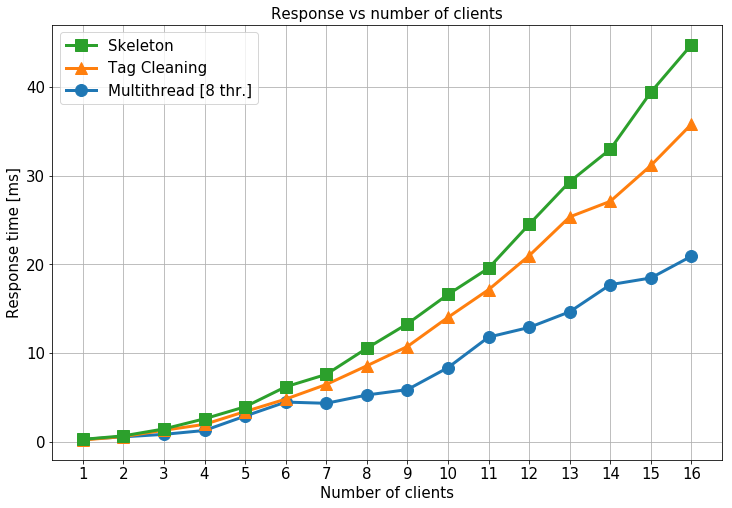
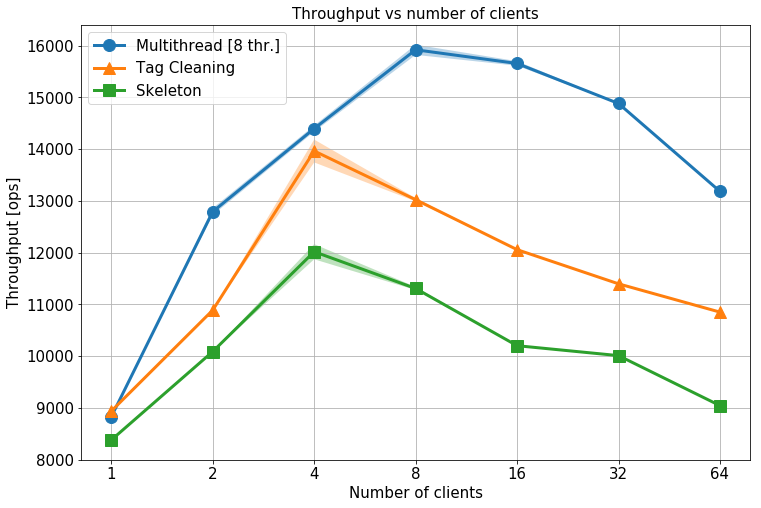
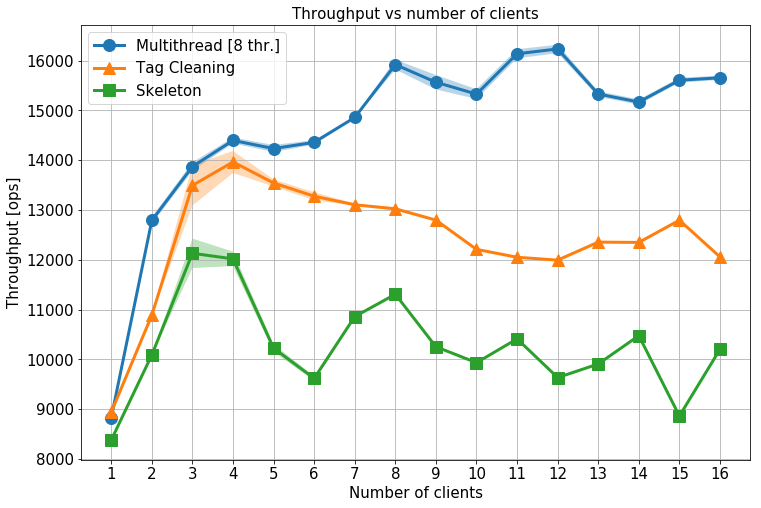


Figure 3: Throughput and response time for different numbers of clients

## 2.3 Discussion

Since the system do not look saturated, even with 16 clients. I double the number of clients up to 64. In the graphs to the left, reporting every experiment from 1 to 16, the performances do not change drastically. The system starts to saturate from 32 clients connected.

The tag cleaning operation, even if add an overhead to the skeleton, result in an overall improvement due to the less words to count.

# 3. Effect of Document Size

## 3.1 Time Spent in Server

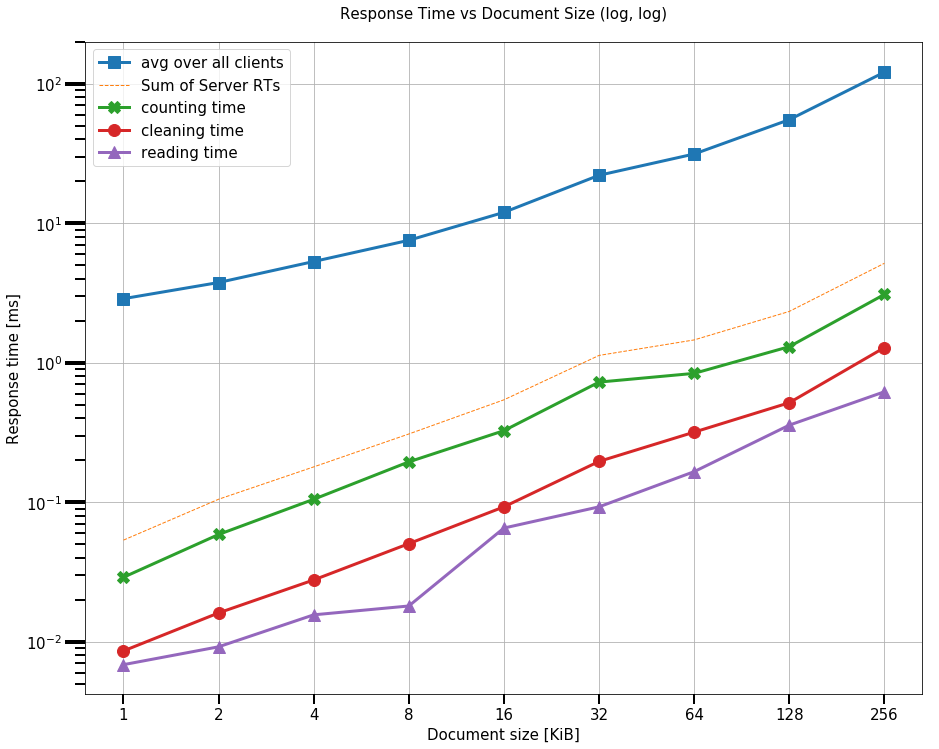


Figure 4: response time vs document size in logarithmic scale

## 3.2 Discussion

Note that there are two order of magnitude of difference between the response time measure by the client and the response times of the four server tasks. The counting operation is the most expensive one, as expected. The graph underline how the client request spent most of the time waiting in the queue.

# 4. Modeling

## 4.1 M/M/1 and N.o.Q.

Using the experimental results in Section 2.2 variant 3) and the insights you gathered in Section 3, build two models of your system (one using an M/M/m model, and one using a network of queues).

Plot the predicted response time and throughput as a function of load and compare this to the real-world results.

4.2 Discussion

Elaborate on how well the models match the real-world behavior[[2]](#footnote-2) and how these results relate to your design decisions.

Please make sure that in the explanations you don’t just describe what is in the graphs, but instead put the results in the context of your design decisions and implementation details.

1. Every client read his PID using the process API provided from Java 9, to diversify his log files. [↑](#footnote-ref-1)
2. Note that you will have to transform the numbers from number of clients into arrival rate to be able to compare to a model. [↑](#footnote-ref-2)