Project: Measurement & Modeling

Imagine you have a company with customers using one of your products, a client-server application. When looking at the performance of such an application, one is typically interested in two problems:

- What response time do my customers experience? If the response time is high, what is the reason for this? Is the CPU or the disk in the server too slow? Or maybe the network connection? What about memory usage?
- Assuming I get new customers in the future, how many customers can my application handle before the response time becomes too high?

In this project, you will analyze the performance (e.g response time) of a simple client-server application that you will implement in **Java**. The details of the application are given in the next section. We ask you to write two versions of the server, a simple version and an optimized version. The way how the server is optimized is left to you. The goal of this work is to measure the performance of the server in a systematic way and to show how the optimizations you have chosen impact that performance. You will also compare the measurement results with results obtained from a queueing station model.

Warning: In your experiments, the clients and the server **MUST** run on different physical computers with a network connection between them. Alternatively, you can have clients and the server as virtual machines on the same physical machine **ONLY** if you explain how you emulate the network (with tools like netem). If you are emulating network conditions, you have to choose parameters of the emulation like the latency. The value of those parameters depend on many things like how far (geographically) the two hosts are from each other. For example, the latency between two hosts in Belgium will be lower (e.g ~10ms) than the latency between a host in Belgium and one in the United States (e.g ~100ms).

Application specifications

The application that you have to implement works similarly to a read-only database. The server stores data and the clients make requests to retrieve some of them. To keep things simple, the data are human readable sentences from which clients attempt to find the ones that match specific regular expressions.

When the server is started, it loads the data from a dataset file into main memory. Each line in the file has the following format:

<type>@@@<sentence>

The type is an integer that indicates what kind of sentence follows, depending on the context it could be the language of the sentence (english, french,...), its origin (movie, book,...) or something else. The “@@@’ is the delimiter. The sentence can be any character string. Note that the file is only read once when the server starts. After that, all requests are served from main memory.

The requests sent by the clients have the following format:

<types>;<regex>

<types> is a comma-separated list of integers. <regex> is a string containing a regular expression. When receiving such a request, the server will search for all rows that match one of the provided types and the regular expression and send them back to the client. The client can also provide an empty list of types if they want to look for all types.

The dataset file is provided here: dataset.

Task 1: Implementation

Write a simple server that accepts requests from a client through the network (similar to a web server). The server should open a socket on a certain TCP port and wait for incoming requests from the client. In Java, this can be implemented with just a few lines of code thanks to the java.net.ServerSocket class. See here for a short introduction:

http://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html

Hint: Note that the constructor of the `ServerSocket` class allows you to define a so-called *backlog*. This is basically the maximum queueing capacity of the operating system's queue for incoming TCP connections. That could be important for task 3.

Implement the requests and the responses as a simple text based protocol, i.e. the client sends the request as a string terminated with a newline ("\n"). Similarly, the server sends the result rows as a list of strings separated by newlines, followed by an empty line to indicate the end of response.

Your server should be multi-threaded, i.e., it should be able to process several requests at a time. The choice for the number (at least two) of threads used by the server is left to you. To achieve this, you should look at how to use thread pools in Java.

In this project, you have to write two versions of the server:

- The basic version uses a very simple data structure to store the data read from the file in main memory and perform a linear search: a two-dimensional array with 2 columns (one for the type, one for the text) and *N* rows where *N* is the number of lines in the text file.
- For the second version, we ask you to modify the basic version to improve its performance. You are allowed to modify the server as you see fit as long as it satisfies the specifications.

Task 2: Measurements

Measure the average response time of your system. For the measurements, we define the *response time* as the delay (as seen by the client) between sending the request and receiving the complete response. Obviously, the response time will depend on how many requests the server receives and how "difficult" the requests are. For example, requests with complex regular expressions need more CPU time on the server. And some requests might result in big responses with a lot of data sent back to the client.

To do measurements, you should not send requests to the server by hand (that would take too much time for you). Instead, you should implement a small client application that sends requests at random times. For the inter-request time, choose some distribution. To send multiple requests to the server at the same time, you can either write a client application that opens multiple TCP connections, or the client only opens one TCP connection and you start the client application several times in parallel. The client should not wait for a response before sending the next request. It's perfectly possible that a request is sent while another request is still being processed.

Try to go up to 50 to 100 client connections. Hint for task 3: Do you remember what we said in the course about the stochastic properties of many independent clients?

Show measurement results for combinations of

- different request rate,
- different difficulties

Think about different difficulties and how they would affect the performance of the server in different ways (e.g more CPU intensive vs more network intensive). Choose reasonable values for the parameters: obviously, a request rate of 1 request/day does not produce interesting response time results.

Show plots and discuss the results. What is the most important factor (CPU, network,...) in the response time, depending on the chosen parameters? Here are some tools to measure the network and CPU load on Linux:

http://www.binarytides.com/linux-commands-monitor-network/

https://www.tecmint.com/command-line-tools-to-monitor-linux-performance/

The measurement must be done for both versions of the server, you have to show that the optimized version is (hopefully) better than the normal one and explain why. We expect you to explain what modifications you have done to the basic version, why you chose to make the said modifications and what you did expect to improve by making these modifications.

Task 3: Modeling

In this task, you have to analyze your server using a queuing station model seen in the course. To select the most appropriate model think about the properties of your experiment (arrival rate of requests, number of server threads, etc.). With those information, you should be able to assess what kind of model is the most appropriate (M|M|1, M|M|m, M|G|1, M|G|m,...). Then use the model to calculate the theoretical mean response times using the same parameters as in task 2. Compare the model results (plots!) with the measurement results and discuss the results. How well does the model work?

Some of the models seen in the course require the moments of the service time distribution. To determine them, you have to know how much time your system needs to process a single request without any queueing delay. That's relatively easily done: Just send a single request and wait for the answer *before sending the next request*. Obviously, the service time will depend on the difficulty of the requests.

General remarks

- Show results and discuss them. Don't just dump plots into your report!
- Repeat your measurements several times. You cannot calculate an average from one value. The danger when showing average values is that the reader does not know what the variance of the data behind the average is. Therefore, do not only show averages but also minimum and maximum values for each point in your plots. Let's say you have measured an average response time of 100ms (minimum 80ms, maximum 150ms) for an arrival rate of 5 requests/s. You can write in a file

  ```
  5	100	80	150
  ```

  and plot errorbars with gnuplot:

  ```
  plot "file.txt" with errorbars
  ```

- Be careful about interferences with other programs running on your system or in your network.

Expected output

You should hand in the source code of your implementation and a written report in a zip file.

The report should be 4 pages (A4, 11pt, pdf format) and contain

- a description of the implementation, including the client
- a description about how you optimized the server and plots showing the impact of the optimization process.
- a description of your measurement setup: The used hardware, network, and software to do the measurements
- a description of the *workload* you used in your experiments, i.e. how the clients generate the requests

- the results (plots) and discussions of the measurements and modeling (task 2+3).

Don’t forget to include your names, email addresses and NOMA. Keep your report concise. Don’t waste space on long introductions, a table of contents, etc.

Evaluation criteria

- Quality of code (comments, correctness, etc.)
- Quality of report (reasoning, language, readability, clearness of presentation)