Group work 2:

The goal of this work is to build a simple client-server system and measure its performance, identify how the different components contribute to its behavior, and do a model-based evaluation.

Task 2.1: A simple server for mathematical computations

In the first task you should measure and analyze the performance of a simple server. The server should receive requests from clients to perform mathematical computations and send the computation results back to them.

Server

The server should accept computation requests from clients through the network (similar to a web server). You can choose what type of computation you want to implement in your server and how the application protocol should look like (hint: keep it simple). Some ideas:

- The client sends an image to the server, and the server applies a Kernel[1] and returns the result image, or
- The clients sends a two-dimensional matrix M with real values and a natural number p, and the server calculates and returns M^p, or
- The clients sends the coefficients of a system of linear equations and the server calculates and returns the solution,
- …

In order to allow the server to receive requests from clients, the server should open a socket on a certain TCP port and wait for incoming computation requests. In Java (and also in C with BSD sockets), this can be implemented with just a few lines of code thanks to the java.net.ServerSocket class. The server should be *single threaded*, i.e., process one request at a time. Note that the constructor of the ServerSocket class allows you to define a so-called *backlog* (maximum number of pending requests).

Client

Write a client that allows you to send requests of adjustable "difficulty" to the server. For the examples given above, the difficulty would depend on the size of the picture, the size of the matrix, the exponent, etc.

Measurement #1

Measure the time your server needs to handle an individual client request. Since the difficulty of a request is adjustable in your client, make a plot showing the average time needed as function of the difficulty. Your plot should also show how the time is split between calculation time (time to calculate the answer) and network time[2] (time to send the request and receive the answer). If your application accesses the disk, also show those times.

Measurement #2

Extend your client to a *load generator*. The load generator should send random requests of varying[3] difficulties to the server. Simulate the behavior of many independent clients by sending requests

[1] https://en.wikipedia.org/wiki/Kernel_%28image_processing%29
[2] Obviously, you need two computers for that.
[3] Up to you to define.

with exponentially distributed[4] inter-request times. Measure and show as a function of the request rate

- the average CPU load of the machine hosting the server. On Linux, an easy way to measure the CPU load is to use the `top` command.
- the average network load. On Linux, network utilization can be measured with tools such as `NetHogs` or `iftop`.
- the average response time.

Discuss the results. For example: What is the most important contributor to the response time? What are possible bottlenecks, i.e., what is limiting the performance?

Modeling #1

Make a queueing station model for the above system. Explain your model and how you have parametrized it. Calculate the mean response time as function of the request rate. Compare the results with the ones from measurement #2 and discuss. How well does the model work?

Measurement #3

How could you increase the performance of the system? We have seen several methods in the course, for example caching and the usage of job priorities (a third method, namely multi-threading, is the objective of task 2.2). Implement an improvement and show its impact on the response time.

Task 2.2: A multi-threaded server

Extend the server from task 2.1, so it can process an adjustable number of requests in parallel (multi-threading).

Repeat your response time measurements #2 for the multi-threaded system. Show how CPU load, network load and response times change with the request rate and the number of server threads. Discuss the results. Again, what are possible bottlenecks? Any ideas for improvements? (this time, you don't have to implement them)

Compare your measurement results with the results obtained from a queueing model. Explain your model and discuss the results. Does the model work well? If not, what could be possible reasons?

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.
- Be careful that your measurement does not disturb the system (e.g., do not use a CPU-demanding measurement tool if you want to measure the CPU usage of a system)
- 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 4-5 pages (A4, 11pt, pdf format) and contain

- a brief description of the server:

[4] Use the *inversion method* to generate exponentially distributed random numbers.

    - The computation performed by your server
    - The request and response format
- a description of your measurement setup:
    - The used hardware and software (top, etc.)
    - The load generator, especially how the random requests are generated.
- the results and discussions of the measurements and modeling, as described in task 2.1.
- the results and discussions of the measurements and modeling, as described in task 2.2.

Evaluation criteria

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