Programare Concurenta si Distribuita

Tema 1

Nistor Marian-Sergiu

Requirements

URL: https://profs.info.uaic.ro/~adria/teach/courses/pcd/homework/PCD_Homework1.pdf

Write a program that measure the time to transfer various amount of data (500MB, 1GB (1.5 milion of WhatsApp messages, 4000 photos, 10.000 emails or reuse a large buffer)) under various conditions (Make sure you are sending bytes. Do not use higher-level functions that assume you are writing strings.)

Requirements for implementation:

- Supported protocols as parameters (for both client/server): UDP, TCP
- Message size: 1 to 65535 bytes
- Two mechanisms should be implemented: streaming and stop-and-wait (acknowledge is performed before the sending of the next message)

Requirements for output:

- After each server session, the server will print: used protocol, number of messages read, number of bytes reads
- At the end of execution, the client will print: transmission time, number of sent messages, number of bytes sent.

Language: you can use any language for implementation. The program must run on a Linux system.

Evaluation:

- Client/Server Code
- A pdf/word/plain text document with details regarding options for client/server and with statistics regarding performed tests.

Hint: Your system architecture can have a two-tier architecture or a multi-tier one. (e.g. a middle tier can perform queuing/scheduling of clients request, connection management, traffic monitoring, connection to a back-end database et.al.)

Repository

https://github.com/SergiuDeveloper/Programare-Concurenta-si-Distribuita/tree/main/Tema1/V2

Implementation

Architecture

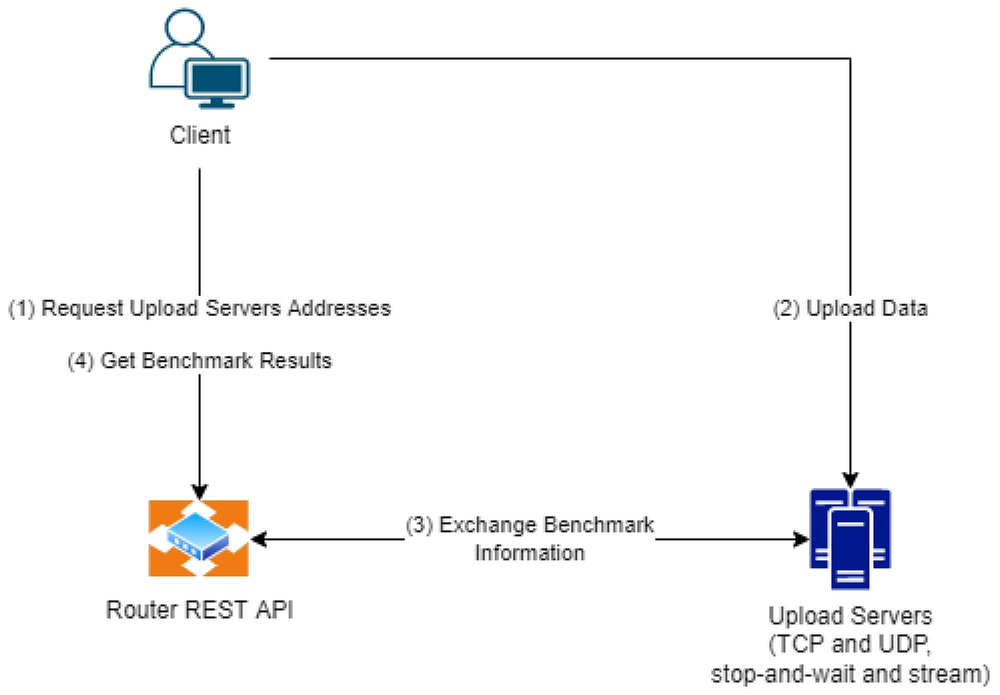
Client
(1) Request Upload Servers Addresses
(4) Get Benchmark Results
(2) Upload Data
(3) Exchange Benchmark Information
Router REST API
Upload Servers (TCP and UDP, stop-and-wait and stream)

Figure 1. Application architecture.

The architecture is a multi-tier one, being based around the following components:

- The Client, which reads binary data, in chunks, from a specified file, calls the Router REST API for retrieving the addresses for all the upload servers in the Upload Cluster, and its benchmark results. The client connects to each of the 4 servers, and uploads the file.
- The Router REST API
- The Upload Cluster.

The reason for separating the routing and the compute/storage entities is the intention of distributing the tasks across multiple servers, therefore allowing for more granular deployments, easier debugging, and increased security (a compromised router would not put the user data in danger).

The Router REST API

The Router REST API holds information regarding the servers that comprise the upload cluster. Additionally, the Router REST API holds the benchmark results, after the user files are uploaded on all servers in the upload cluster.

The REST API exposes two methods:

- GET /addresses, which returns the host and port for each upload server from the cluster (the TCP Stop-And-Wait Server, the TCP Stream Server, the UDP Stop-And-Wait Server, the UDP Stream Server)
- GET /benchmark_data, which, based on the caller's address, returns information regarding the timestamp when the upload operation finished, and the number of bytes that were uploaded to each upload server in the cluster.

The Upload Cluster

The Upload Cluster is comprised of a series of 4 upload servers, used to benchmark the user's upload speed and the percentage of successfully transmitted data chunks. It is composed of 4 servers:

- The TCP Stop-And-Wait Server
- The TCP Stream Server
- The UDP Stop-And-Wait Server
- The UDP Stream Server.

The acknowledgement operation consists of sending a single byte to the client, in order to indicate the availability for receiving another chunk of data.

The flow diagrams shown below represent the actions executed by the TCP, and the UDP server, respectively, in order to benchmark the upload performance of the user. The only difference between the two is the fact that, after processing one chunk and, optionally, sending the acknowledgement, the server will not await other data chunks from the same sender. The reason for that is the fact the thread associated with a given client would be stuck indefinitely after the client finished sending all the chunks. For the TCP servers, the server is able to tell whether a client finished sending the chunks, by the fact that that particular client closed its connection.

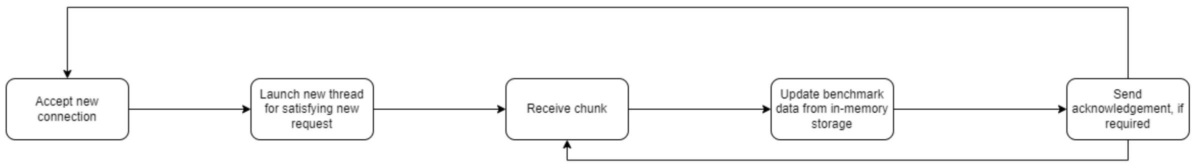
Accept new connection
Launch new thread for satisfying new request
Receive chunk
Update benchmark data from in-memory storage
Send acknowledgement, if required

Figure 2. TCP servers flow diagram

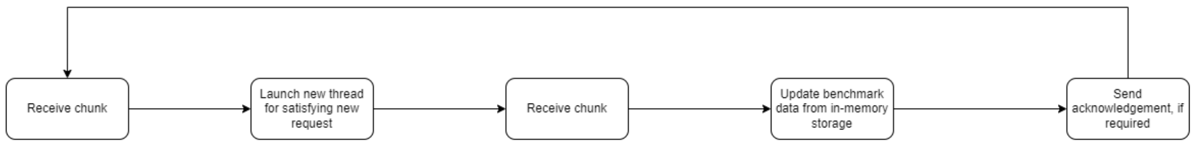
Receive chunk
Launch new thread for satisfying new request
Receive chunk
Update benchmark data from in-memory storage
Send acknowledgement, if required

Figure 3. UDP servers flow diagram

Data Storage

The data is stored either in-memory or on the disk, depending on its utility. The uploaded files are placed on the disk, in a specified folder. As for storing user benchmark results, the BenchmarkData class exposes a set of methods for accessing, setting and retrieving data from an encapsulated dictionary.

Configuration

Both the server application and the client application are configurable using the "configuration/config.ini" file, which specifies the following:

- The Router REST API host, port, and methods
- The path to the benchmark file
- The chunk size
- The ports for each server in the Upload Cluster.

The server or client application can be run using *python3 main.py*.

Technologies used

The programming language of choice was Python3, given the fact that it is currently the most popular programming language for prototyping. The REST API was created using Flask. ConfigParser was used for reading configuration files.

Results

The files used for testing the benchmarking functionality had a size of approximately 754 MB. The testing was performed by hosting both the servers and the clients on the local environment.

The TCP Stop-And-Wait method has an average execution time of 1.03 seconds, with a transfer rate of 100% (~12166 chunks, ~754 MB).

The TCP Stream method has an average execution time of 1.63 seconds, with a transfer rate of 100% (~12166 chunks, ~754 MB).

The UDP Stop-And-Wait method has an average execution time of 45.65 seconds, with a transfer rate of 99.5% (~12103 chunks, ~750 MB).

The UDP Stream method has an average execution time of 0.97 seconds, with a transfer rate of 3.28% (~754 chunks, ~24.75 MB).

The UDP Stop-And-Wait method takes a lot of time, due to the fact that the acknowledgement is awaited for a given period of time. The assumption is that some of the acknowledgement packages sent from the server are lost in the process.

The UDP Stream method has a low success rate, due to the fact that the client sends the packages in a timely manner, while the server launches a new thread and opens a file (in order to append the new chunks to its previous content), for each request.