# Internet Networking Homework 4 - Report

Yosef Goren & Ori Evron

June 24, 2023

## Contents



## 1 Running The Experiments

The full source code to our solution in our GitHub Repository under the folder 'Homework4/loadBalancer'.
To run the simulation go to the 'loadBalancer' folder in the provided zip file or in the repo and run `python loadBalancerLab.py`.
To run the emulator, go to the same folder and run `./emulator <sched_alg_name>` where `<sched_alg_name>` should be one of '`first, rr, greedy`'.
Note these same parameters can be passed to the 'load_balancer' executable to select it's scheduling algorithm.

# 2 Software Design

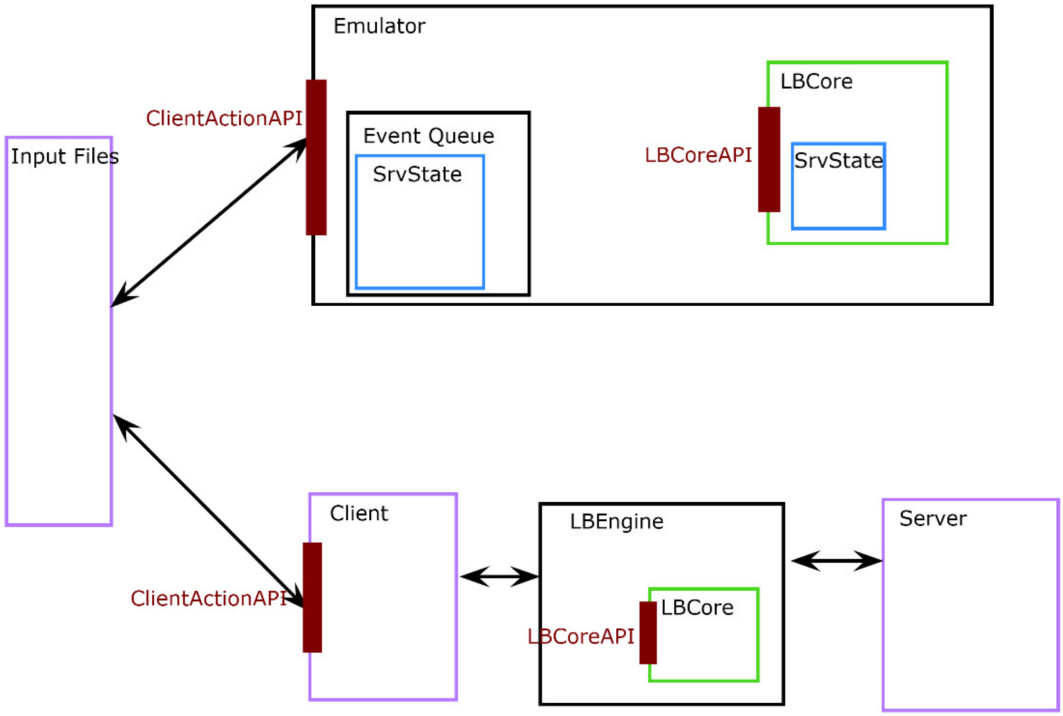
## 2.1 Overview

Before starting to implement the required server, we first examine how the requred functionality can be split into distinct tasks.
Notably - it is possible to distinguish between the pure algorithmic task of deciding which server should be handle each incoming request, and the task of implementing a forwarding proxy server that can simultaniously communicate with all of the parties involved.
Moveover, a modular API for the algorithmic section should allow us to seemlessly switch between different scheduling algorithms, and to use the exact same code within both a real proxy server, and within code that would emulate the behaviour of the system - without actually involving any networking communications.

After the analysis, we conclude that our solution should have the following software modules:

- `LBCore`: This module implements the various scheduling algorithms, and is independent of both the implementation of the actual server.
  This module is essentially a static library.
- `LBEngine`: This module accepts a scheduling algorithm and uses it implement a full-fledged load balancing proxy server.
  It is an executable.
- `Emulator`: This module accepts a scheduling algorithm and uses it to run an event based simulation of the networking system described in the homework. This simulation has a few notable advantages:
  - The simulation runs almost instantly[1].
  - The simulation should be able to run on any platform since it does not actually make use of any socket API.
  - It makes it easier to debug componenets used by it since it does not involve any asynchrony or multiprocessing[2].

[1] no 'sleep', the involved code is entierly C++
[2] as opposed to the `LBEngine`




## 2.2 Design Stages

In order to progressively implement our design we came up with the following incremental stages:

1. Define the API for `LBCore`.
2. Create a proof-of-concept application [3] to experiment with using the Linux socket API for our purposes.
3. Create `LBEngine`.
4. Create a basic implementation of `LBCore` with trivial algorithm [4].
5. Create `Emulator`.
6. Design and implement better scheduling algorithms (`LBCore`).
7. Test performance of the schedulers using the Emulator.

[3] can be found under `loadBalancer/SockHelloWorld`
[4] at this point we can debug `LBEngine`

# 3 Load Balancing Engine

## 3.1 Socket Manager

The `SocketManager` [5] played a crucial role in managing the multitude of sockets in our code. Specifically, we needed to maintain three active sockets to the servers, alongside a dynamically changing number of sockets to the clients.
To achieve this, the Socket Manager maintained a list of sockets for each working server. Upon receiving a finish signal from a working server, we could retrieve the corresponding socket from the Socket Manager. This socket was then used to send the finish signal to the client before closing the socket.
By utilizing the Socket Manager, we could ensure that all sockets were appropriately closed once we had completed our tasks. This streamlined approach helped us manage the complex socket interactions efficiently and maintain proper control over the communication process.

## 3.2 Async Communications

The hardest technical problem we have had to face was giving `LBEngine` the ability to asynchronously communicate with both the servers and the clients.
The Engine has to main types of communication sockets:

1. Worker Sockets: These sockets are used to communicate between the proxy server (a.k.a `LBEngine`) and each of the worker servers. These connections are established once when the proxy server starts running and continue untill it is done.

2. Client Sockets: These sockets are used to communicate between the proxy server and the end clients. For each request from a client - a new connection is established - which lasts which the request is being handled. Once a response is produced by a worker - that response is sent back over the session - and the session is closed.

In order to be able to handle both types of requests from a single process - we have implemented a main loop that continuously checks for either a new incoming client connection - or a new response message from one of the worker servers. If either one is sent - it is handled by the proxy server in that same iteration of the main loop.
In order to be able to keep track of which response should be returned to which client [6] - the proxy servers saves a fifo queue of client socket id's for each of the worker servers.

[5] can be found at `loadBalancer/LBEngine/LBEngine.cpp`
[6] since this information is not contained within the response messages from the workers

# 4 Emulator

## 4.1 Overview

As mentioned previously - the emulator is event based, and does not involve any actual networking communications.

## 4.2 Event Types

There are two possible events in our simulation:

1. `clientRequestEvent`: This event emulates the behaviour of a client makeing a new request to the proxy server. It causes a later event of type `serverResponseEvent`.

2. `serverResponseEvent`: This event emulates a server sending a response back to the proxy server. If the associated client has any more requests - it will trigger an immidiate event of type `clientRequestEvent`.

## 4.3 Event Queue

The event queue is initialized with an event of type `clientRequestEvent` from each of the clients (with a non-empty requests list).
The simulation lasts as long as there are any new events to pull from the evnet queue.

# 5 Scheduling Algorithms

## 5.1 Always First Worker

Initially, we attempted to direct all client requests to the first worker server, which, of-course resulted very poor performance.

## 5.2 Round Robin

We then proceeded to implement a round-robin approach to evaluate its effectiveness. The round-robin mechanism involved initializing a counter (modulus 3) to 0.
Each request was dispatched to a worker server using the current value of the counter, after which the counter was incremented.
The performance achieved using this round-robin approach was comparable to the performance of the code provided by the course staff.

## 5.3 Greedy

Upon further exploration, we discovered that employing a greedy algorithm proved to be the most effective solution for our problem. Our algorithm treats

each client's request as if it were the last and calculates the optimal server to dispatch it to. By doing so, we aim to complete the execution of all requests as quickly as possible, optimizing the overall performance of the system.

# 6 Results

As previously mentioned, our experiments with the greedy algorithm yielded the best results. In certain runs, we observed that it outperformed our round-robin approach by a factor of two,
resulting in significantly faster execution times.