Student Name: Pesch Nathan
Student ID: 0556396

Multicore Programming
Assignment 1: Erlang

VRIJE UNIVERSITEIT BRUSSEL

Contents

1 Overview

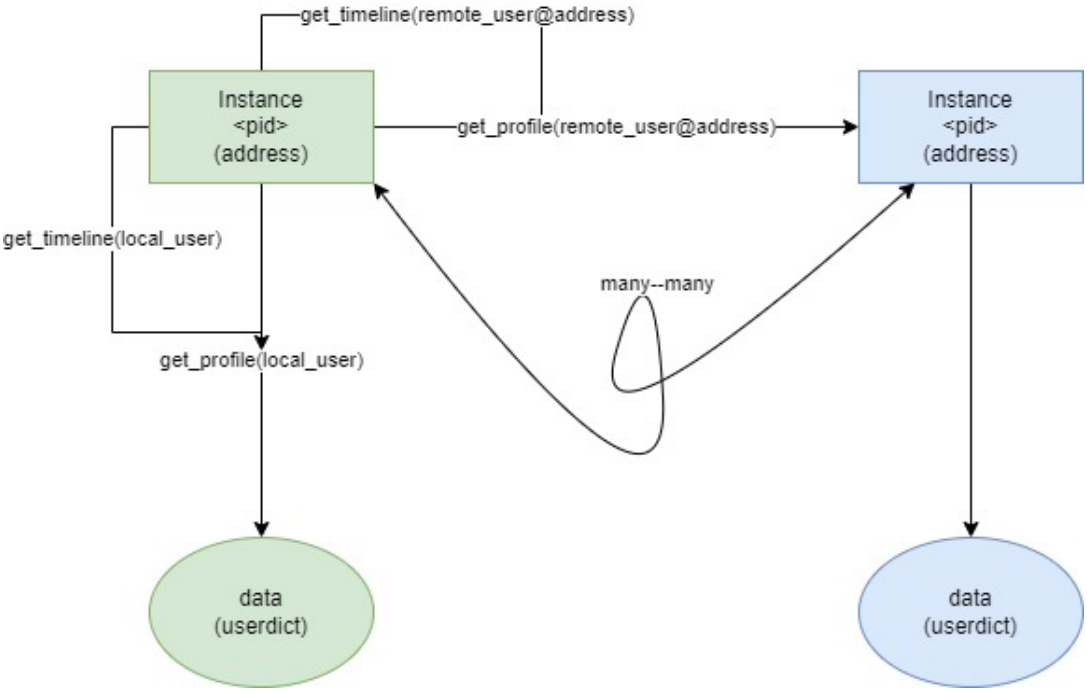
For this project, I was required to expand the implementation of a Twitter/Mastodon like software system so that it could run parallel instances that communicate with each other through Erlang messages. I decided to split each **server** by an address such as *vub.be*. Each of these servers runs in its own process and has local state that stores the data of all users in a dictionary. Users register with one of these servers, but can follow or get the profile of users from another server. When such a request is made, a messaging action is launched between these two instances. I will from now on refer to these servers with a unique address as an **instance**. When benchmarking, I performed three experiments where I measured the speedup when varying the thread count, the instance count and the user count per instance.

2 Implementation

2.1 Architecture

The project architecture is very similar to the initial implementation. I decided that each instance should be registered with their address, such that for example users from *vub.be* can follow each other just by username, but can also follow users from other instances by sending a follow request to their instance of *username@address*. The address of the remote instance is then parsed from the request. By doing this, the host instance can exchange data with the remote instance and a result can be presented. Another byproduct of this is that we have some sort of data locality, where user data is only stored on their host instance. The only operation on the server that had to be made significantly differently is **get_profile**. I implemented this operation so that when the request is made about a local user, it is classically returned, and else, a **get_profile** request is sent to the remote instance, and the profile is returned to the host. **Get_timeline** is implemented by means of this **get_profile** operation. If one of the followed users is on a remote address, it will automatically

handle sending the request.



2.2 Scalability

The scalability of this implementation is mostly the data locality. There is no duplication of any data anywhere, which means the best case scenario for this system is where there are a lot of small instances carrying their local data. In the worst case, there is a small amount of instances where each instance is a single process that has to handle all the requests of many users at once. This could very well be improved by expanding the implementation such that a single instances consists of many different message handling processes. This way, a lot of messages could be processed at once, even in a single instance. With such an implementation, a form of caching should be implemented, because Erlang processes do not share data. The limit of the scalability would then be dependant on the caching system. Some data inconsistencies will also be introduced there.

3 Evaluation

3.1 Set-up

For the benchmarks, I used the Firefly server. The server has the following specs:

	Hardware
CPU	AMD Ryzen Threadripper 3990X Processor (64 cores / 128 threads, at 2.9 GHz base, 4.3 GHz boost)
Ram	128 GB (DDR4 3200 MHz)
	Software
OS	Ubuntu 20.04.5, running Linux kernel 5.4.0-137-generic
Erlang/OTP version	22.2.7

3.2 Methodology

I decided to measure the wall clock time, and measure the speedup in each experiment.

I adjusted the benchmarking code so that the two bottom parameters vary as also the thread count varies, as each instance runs in a seperate process and thus it would we wise to find representative values experimentally. Each experiment was ran 30 times, with 500 requests per benchmark.

3.3 Experiments

I varied the following parameters:

- Thread count
- Instance count
- User count per instance

These parameters were varied for the **send_message** and **get_timeline** requests. The thread counts were as follows:

- 2
- 4
- 8
- 16
- 32
- 64

The user count per instance were varied from 1000 to 10000 users per instance, and set at 5000 when varying the instance count, which varied from 2 to 10 instances. The number of subscriptions is set to 30 per user and the number of starting messages to 20. Due to issues with adjusting the parser script, I was not able to process the results in time for the deadline of this project.