Project Erlang: Decentralized Microblogging

Multicore Programming 2023

Janwillem Swalens (janwillem.swalens@vub.be)

For this project, you will implement and evaluate a decentralized "microblogging" application in Erlang, like Mastodon. Users can connect to the application, send messages, follow each other, and read each other's messages. Your application should consist of multiple "instances": separate Erlang processes that are responsible for a subset of the users. Users can follow each other and receive messages across instances. Finally, you should evaluate your system with a set of benchmarks, simulating a variety of workloads.

Overview

This project consists of three parts: an **implementation** in Erlang, an **evaluation** of this system using benchmarks, and a **report** that describes the implementation and evaluation.

- **Deadline**: Sunday, **9th of April** 2023 at 23:59.
- **Submission**: Package the implementation, your benchmark code, and the report as a PDF into a single ZIP file. Submit the ZIP file on the Canvas page of the course.
- **Grading**: This project accounts for one third of your final grade. It will be graded based on the *submitted code*, the accompanying *report and its evaluation*, and the *project defense* at the end of the year. If you hand in late, two points will be deducted per day you were late. If you are more than four days late, you'll get an absent grade for the course.
- **Academic honesty**: All projects are individual. You are required to do your own work and will only be evaluated on the part of the work you did yourself. We check for plagiarism. More information about our plagiarism policy can be found on the course website.

A decentralized microblogging application

Several popular messaging applications are written in Erlang (Whatsapp [1] [2] [3]), use the Erlang VM (Discord uses Elixir), or have used Erlang (Facebook Chat started with Erlang but later switched to C++).

For this project, you will implement a basic but scalable "microblogging" application, similar to Twitter or Mastodon. The application allows users to send messages, follow each other, and read each other's messages. The application is decentralized (or "federated"): it consists of multiple server instances, each of which is responsible for a subset of the users. Users can follow each other and receive messages across instances.

Your service needs to store information about the users, the messages they have sent, and the users they follow. This is depicted in the figure below. Each message has a sender, text, and a timestamp.

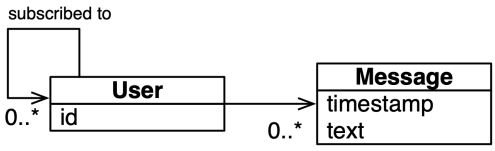

Figure 1: The data stored by one server instance.

A typical session is shown in the figure below. First, users register their user name on a specific instance, e.g. `alice@vub.be` and `bob@ulb.be`. Next, they log in, and Bob decides to follow Alice. When Alice sends a message, it is saved on her instance. When Bob fetches his timeline, he sees Alice's message. How his server `ulb.be` fetches the message from the server `vub.be` is left unspecified and up to you to implement: it may be fetched on demand, it may be cached, or it may be prefetched. Lastly, the user Carol can request all of Alice's messages when visiting her profile page.

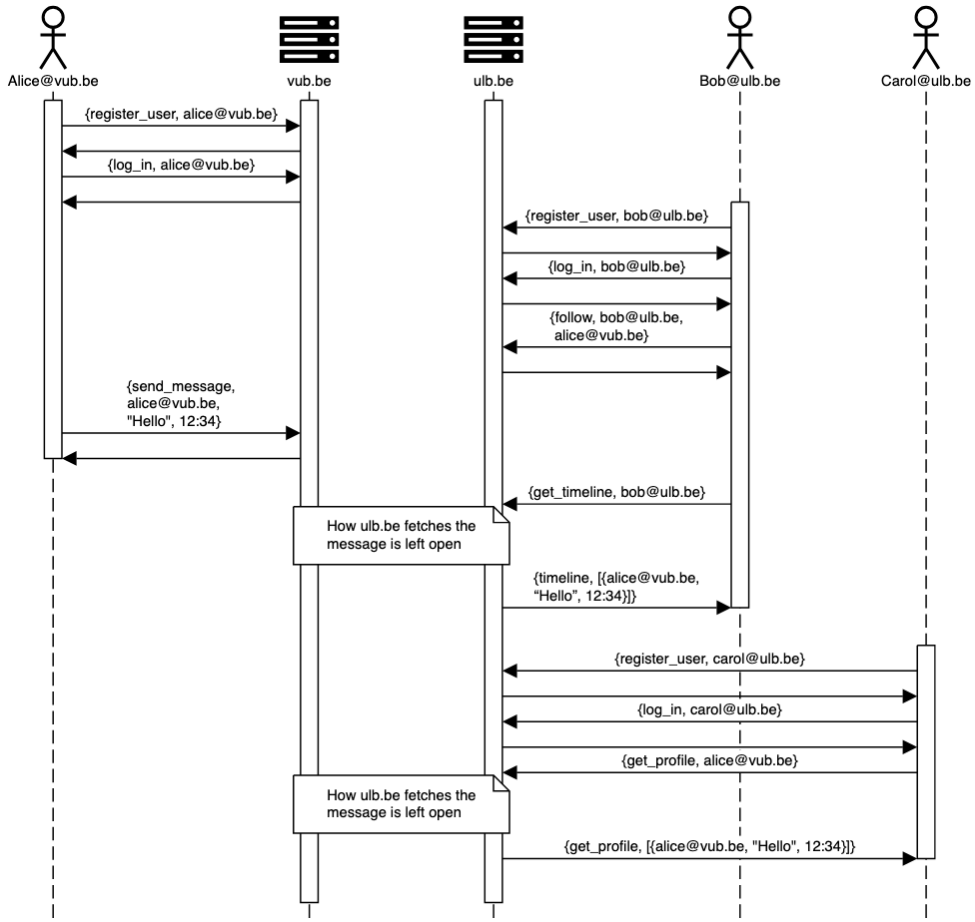

Figure 2: Typical session between three users. Alice and Bob log in, and Bob follows Alice. Alice then sends a message, which Bob can see on his timeline. Carol will also see this message when she fetches Alice's profile. How messages are synchronized between the two instances is left unspecified. (Tip for your report: this sequence diagram was created using https://sequencediagram.org.)

Implementation

The aim of this project is to create an implementation of this application that uses several

instances that run in parallel, with a focus on scalability. We provide a starting point implementation in which there is only one instance, and users can only communicate within that instance. You need to change so that there are multiple instance and users can communicate across them.

Server instance and clients

The application is separated into several server instances and several clients.

Each server instance should be a separate Erlang process. However, it may be beneficial to spawn several processes within an instance to increase performance!

The client-side consists of several client processes, one per connected user. In your benchmarks you will generate several client processes, each of which represents a user that requests some information from a server instance.

Required operations

A server should support the operations listed in the API definition, in `server.erl`. Their semantics are supposed to remain unchanged:

- **`register_user`**: Register a new user. This creates the user on that instance.
- **`log_in`**: Log in as a certain user. You can assume that a user will only log in on their own instance.
- **`follow`**: Follow another user. After this request, the timeline of the current user should contain the messages from the followed user.
- **`send_message`**: Send a message. The message is saved on the server. The timestamp of the message should represent the wall clock time when the message was received.
- **`get_timeline`**: Get the user's timeline. The result includes a (perhaps partially stale) view of all tweets of the users that this user is following.
- **`get_profile`**: Get all messages of a user.

The goal of this project is to increase scalability: your application should be able to handle a large number of users sending these requests.

The `register_user` and `log_in` requests return a pid that can be used by the client for further communication. This is not necessarily the pid of the sender of the message. This allows you to spawn new processes on the server, which handle the requests of a subset of the users, which can be useful for scalability.

> Note: these operations are inspired by the ActivityPub protocol that is used by Mastodon, but are heavily simplified. You can find information about this protocol in the following articles: 1, 2, 3, 4, 5. Watch out though, the ActivityPub protocol confusingly uses the word "actor" to refer to a user, and not what we refer to as an "actor" (from the actor model) in this course.

Scalability over consistency

Your application should contain all information shown in the figure 1, but it does not need to encode it as depicted or as in the given implementation. Instead, you should choose the actual

data representation to be suitable to realize the goal of this project. You can add other relations, for instance to directly represent the inverse of the *follow* relation: it might pay off to keep the list of "followers" as well as (or instead of) the list of "following" for each user.

Moreover, as this project focuses on scalability, it is explicitly *allowed and encouraged to sacrifice data consistency in order to increase scalability*. For example: you may choose to have some requests return data that is stale (e.g. recently followed users missing in a timeline), in the wrong order (e.g. in timeline), or inconsistent (e.g. Bob sees Alice's message while Carol not yet), or, in case you keep both sides of the follow relation, these might not always be consistent.

Example implementation

We provide an example implementation, consisting of these files:

- **`server.erl`**: a server instance's API. This should remain unchanged, as this is the interface that will be used by benchmarking code.
- **`server_centralized.erl`**: an example implementation that uses a single process to represent a single server instance.
- **`benchmark.erl`, `run_benchmarks.sh`**: a starting point for benchmarks. These files demonstrates how you can set up your benchmarking environment and how to measure common metrics.
- **`benchmarks/process_results.py`**: a Python script that processes the results of the benchmarks: it parses the result files, calculates statistics, and plots the results.

Notes

This assignment is about modeling a scalable system using Erlang processes and message passing. You are supposed to experiment with different strategies to see what impact they have on scalability and consistency. Hence, do not use Erlang databases like Mnesia or Riak, or frameworks like Erlang's OTP or RabbitMQ. They will obscure your results and make it harder to compare approaches.

The aim of this project is to focus on parallelism, not distribution. You should therefore *not* experiment with distributing different server instances over different machines, instead, run everything in one Erlang VM and use regular message passing to communicate between instances.

Furthermore, your application only needs to support the minimal set of operations described above, and should not support any of the advanced features that real applications (or the real ActivityPub protocol) offer. Focus on the parallel and concurrency aspects.

Finally, you do not need to take into account authentication or privacy. Our system for instance allows users to send messages in the name of other users. For the purposes of this project, you should not take these concerns into account.

Evaluation

Next to your implementation, you should perform an evaluation of your application in which you test your design in practice, in three experiments.

Experiments

You should set up experiments to evaluate the performance of the system in different scenarios. There are several (dependent) variables you can measure:

- The latency (time it takes for a request to complete) and throughput (number of requests that can be processed in a specified time interval) of different types of requests (e.g. timeline, send message, get profile).
- The latency of broadcasting a message: if a user sends a message, how long does it take for other users to see it in their timeline? When does it reach the first user and when does it reach the last (if relevant)?
- The speed-up (of a certain operation) when increasing the number of Erlang scheduler threads.
- If you return inconsistent or stale data, can you provide a measurement for the degree of inconsistency or staleness? E.g. how often are messages in the wrong order? How often do inconsistencies appear, and how long does it take for an inconsistency to disappear?
- The load on the system, e.g. the length of the message queues of processes (you can measure this using `erlang:process_info(Pid, message_queue_len)`).

There are also several (independent) variables you can vary:

- The number of Erlang scheduler threads.
- The number of server instances, users per server instance, how users are spread over instances.
- Parameters such as the number of users, messages per user, followers per user. Do users mostly follow other users on the same instance or on other instances?
- Which (mix of) requests you execute.
- The number of clients that are connected simultaneously, the number of requests per client, and the number of simultaneous requests.
- Which implementation you use: the given centralized implementation vs yours.

For this project, you should perform **three** experiments, but you are free to choose the scenarios you want to evaluate. For each experiment, choose one (or at most two) variable that you vary, fix the others to representative values, and measure one variable. You should choose at least one experiment in which the number of Erlang scheduler threads is increased and the speed-up of an operation is measured. Furthermore, we recommend for your second experiment to choose a scenario that highlights the best-case performance of your design.

As part of the assignment, we provide a file `benchmark.erl` that demonstrates how you can run your benchmarks and measure common metrics. *This is only a starting point*, you should adapt the experiments to implement and measure the scenarios above. This file can be ran with `make benchmark` and writes results to text files. We also provide `benchmarks/process_results.py`: a Python script that uses numpy and matplotlib to parse these files, calculate statistics, and plot the results. You can use these files as a starting point, but you are free to use other tools to run your benchmarks and measure the metrics you are interested in.

Hardware

For this project, you should run your experiments on "Firefly", a 64-core server available at the SOFT lab. Its specifications are in figure 3. Before running your experiments on the server, you should first run them on your machine or one of the machines in the computer room. Make sure your experiments run correctly and provide the expected results on your own hardware before you run them on the server, as you only have limited time available on the server.

Your report can consist of experiments that you ran on your local machine or the machines in the computer rooms as well as those that ran on the server, but make sure to use machines with at least four cores.

Include the specifications of the machines you used in your report, as in figure 3, even if you used Firefly. On a Linux machine, you can get information about the machine using the following commands:

- `cat /proc/cpuinfo` to get the CPU model and its details. You can then look up the CPU model on the manufacturer's website.
- `cat /proc/meminfo` to get information about the available memory. (Or a command like `top` or `htop`.) You cannot retrieve the memory type or frequency without root access.
- `lsb_release -a` to get information about the Linux distribution.
- `uname -a` to get information about the Linux kernel.
- `cat /usr/lib/erlang/releases/22/OTP_VERSION` to get the Erlang/OTP version. (You may need to change the major version number in the path.)

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

Figure 3: Specifications of the machine "Firefly"

> Watch out for some common benchmarking pitfalls:
>
> - Garbage collection issues: spawn new processes for each benchmark run, and make sure that they terminate after the benchmark. I.e. each of the 30 repetitions should spawn new, freshly initialized, server instances.
> - Does your CPU support Intel's Hyper-Threading: on processors with *Hyper-Threading*, several (usually two) hardware threads run on each core. E.g. your machine might contain two cores, which each run two hardware threads, hence your operating system will report four "virtual" (or "logical") cores. However, you might not get a 4× speed-up even in the ideal case.
> - Does your CPU support *Turbo Boost*? This allows your processor to run at a higher clock rate than normal. It will be enabled in certain situations, when the workload is high, but it is restricted by power, current, and temperature limits. For example, on a laptop it might only be enabled when the AC power is connected. Hence, make sure to keep the power connected!
>
> Also check out the relevant section in the Erlang documentation.

Report

Finally, you should write a report on your design, implementation, and evaluation. Please follow the outline below, and concentrate on answering the posed questions:

1. **Overview**: Briefly summarize your overall implementation approach and the experiments you performed. (1 or 2 paragaphs)

2. **Implementation**:

    1. **Architecture**: Describe your project's software architecture on a high level. Use figures to show how data is distributed over several processes, and how messages flow between them.
    2. **Scalability**: Discuss the scalability of your implementation. Make sure that the following questions are answered:
        - How does your design ensure scalability? What is the best case and the worst case in terms of the mix of different user requests? When would the responsiveness decrease?
        - How do you ensure conceptual scalability to hundreds, thousands, or millions of users? Are there any conceptual bottlenecks left?
        - Where did you sacrifice data consistency to improve scalability? How does this decision improve scalability? In which case would the overall data consistency suffer most?

3. **Evaluation**:

    - Describe your **experimental set-up**, including all relevant details (use tables where useful):
        - Which CPU, tools, platform, versions of software, etc. you used (even if you used Firefly, see Figure 3).
        - All details that are necessary to put the results into context.
        - All the parameters that influenced the run-time behavior of Erlang.
    - Describe your **experimental methodology**:
        - How did you measure? E.g. using wall clock time or CPU time, at client side or server side.
        - How often did you repeat your experiments, and how did you evaluate the results?
    - For your **three experiments**, describe:
        - What (dependent) *variable(s)* did you measure? For example: speed-up, latency, throughput, degree of inconsistency.
        - What (independent) *parameter* did you vary? For example: number of scheduler threads, number of server instances, users, messages, followers, number of simultaneous requests.
        - Describe the load that you generated and other relevant *parameters*. What are the proportions of different requests? How many users/messages/followers per user does your system contain, how many clients are connected simultaneously, and how many requests are there per connection?
        - *Report* results appropriately: use diagrams, report averages or medians and measurement errors, possibly use box plots. Graphical representations are

preferred over large tables with many numbers. Describe what we see in the graph in your report text.
- *Interpret* the results: explain why we see the results we see. Relate the results back to your architecture: how did your design decisions influence these results? If the results contradict your intuition, explain what caused this.

4. **Insight questions**: In this section you answer some insight questions. They relate to extensions to the problem and how you would change your implementation to deal with them. You should *not* actually implement these extensions or perform any experiments. Briefly answer the questions below (max. 250 words, or ~2 paragraphs, each):

    1. Your current implementation uses multiple cores on a single machine, but in reality, a similar application would be distributed over multiple machines. Imagine such a set-up. How would you change your architecture? How would you distribute the processes over multiple machines? Which bottlenecks might appear? (You might want to draw a diagram of your distributed architecture.)
    2. Spawning a process in Erlang is a very lightweight operation: a newly spawned Erlang process only uses 309 words of memory. How has this influenced your solution? Imagine the cost of creating a process was much higher, e.g. if you were using Java and created new `Thread`s: how would this affect the performance and how could you improve this?