Multicore Programming

Project Erlang: Twitter

Assistant: Janwillem Swalens
Mail: jswalens@vub.ac.be
Office: 10F719

Deadline: 27 April 2014

For the first programming project, you will implement and evaluate a scalable Twitter-like service in Erlang. Twitter's *Fail Whale* became famous symbolizing its struggle with scaling its infrastructure to the rising demand of users. Scalability, i.e. availability for an increasing number of users, is a common issue and always comes with trade-offs. For this project, you are explicitly allowed to sacrifice data consistency to increase scalability. You should also evaluate your system with a number of benchmarks, to simulate a variety of work loads.

Twitter is over capacity.

Please wait a moment and try again. For more information, check out **Twitter Status.**

Bahasa Indonesia Bahasa Melayu Deutsch English Español Filipino Français Italiano
Nederlands Português Türkçe Русский हिन्दी 日本語 简体中文 繁體中文 한국어
© 2012 Twitter About Help Status

1 Procedure Overview

This project consists of three parts: the implementation of a scalable Twitter-like service, an evaluation of this system, and a report that describes and the implementation and evaluation.

Deadline 27th of April at 23:59 CEST. The deadline is fixed and cannot be extended.

Deliverables Package the implementation into a single ZIP file, including the report as a PDF. The ZIP file should be named `Firstname-Lastname-erl.zip`.
On the PointCarré page of the course, `http://pointcarre.vub.ac.be/index.php?application=weblcms&go=course_viewer&course=3092`, go to *Assignments (Opdrachten)* > *Project Erlang* and submit the ZIP file.

Grading This project will account for one third of your final grade. It will be graded foremost based on the *quality of the report* and the *project defense* at the end of the year. However, the *code quality* will be taken into account too. So, please comment your code where necessary to guide the reader.

2 A Scalable Twitter-like Service

Your task is to implement a scalable Twitter-like service. To keep the conceptual overhead small, this service only needs to store the minimal amount of information and will not support any of the advanced features such services offer. Furthermore, concentrate on the backend functionality. An actual web front end is **not required** and **not desired**. Focus on parallel and concurrency aspects.

Your service needs to store information about users, and their subscriptions to other users ("followers"). The tweets of a user only contain a timestamp and the text, which typically is limited to 140 characters. Fig. 1 visualizes this data as a class diagram.

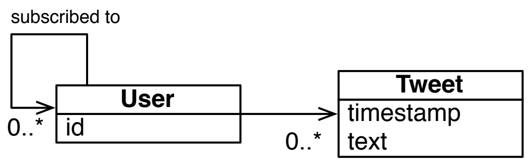

Figure 1: Required data to be maintained by the service.

Your application should encode this information somehow, but it does not need to encode it as depicted. In fact, the actual data representation probably should be different to realize the different goals. Furthermore, you can add other relations, for instance to directly represent the reverse of *subscribed to* (following vs. followed by): if the query for the subscribers is done often, it might pay off to keep this list as well.

2.1 Implementation Requirements

- Represent client connections as one Erlang process per client, for instance for load generation. In other words, in your load generation code you will create a number of processes, each of which is a client that requests some information from the server(s).

- Represent the server as one or more processes. Entry processes are the processes that communicate directly with client processes. In the given example implementation client and entry processes are mapped one to one. However, this is an implementation detail and you are free to change it.

- Only a subset of the conceivable operations is required. The set of operations that need to be supported is given as part of the public API definition in `server.erl`. Its semantics are supposed to remain unchanged:

`get_timeline` Get own timeline. A user can ask the server for his/her current timeline. The result includes a (perhaps partially stale) view on all own tweets, and all tweets of the users the user is subscribed to.

`tweet` Tweet. A user can submit a new tweet. The length is not restricted but should be around the typical size of 140 characters. The content may be generated randomly. The timestamp should represent the wall clock time when the tweet was received.

`get_tweets` Get tweets. A user can ask for all tweets of another user. The given interface allows to paginate the results to all user requests, thus, if you see it fit, you can restrict the amount of data of a response and require the client to make additional requests to obtain all data.

- Beyond these basic operations, you will need additional functionality to set up your evaluation experiments, like loading generated data sets, creating load, etc.
- This assignment is about modeling a scalable system using Erlang processes, message passing, etc. You are supposed to experiment with different strategies to see which impact they have on performance/consistency. Thus, do **not** use Erlang databases like Mnesia or Riak. They will obscure your results and make it harder to compare approaches.

2.2 Scalability over consistency

As this project focuses on scalability, it is explicitly allowed and encouraged to sacrifice data consistency in order to increase scalability. For example, some requests might return stale data, or, in case you keep both a list of subscribers and subscriptions, these lists might not always be consistent.

In your report, describe the situations in which data might be in an inconsistent state. Describe your design, and detail your motivations: how do your design decisions improve the scalability of your system?

Furthermore, during the evaluation, try to quantify the degree of inconsistency a response contains. Look at typical use cases as well as worst case scenarios: in which cases can inconsistent data negatively affect the user experience?

2.3 Example Implementation

The given implementation consists of two files:

`server.erl` An interface definition to interact with an implementation of a Twitter-like system. It defines the semantics and should remain unchanged.

`server_single_actor.erl` A simplified example implementation providing a basic service based on a single data actor (Erlang process).

3 Evaluation

The goal of the evaluation is to show how your theoretical design performs in practice.

You should set up experiments that allow you to benchmark the system for best and worst case scenarios, using a number of metrics. Definitely measure the latency and throughput of the different types of requests (timeline, tweet, get tweets). Compare them in different settings, e.g. in situations with a varying number of users, subscriptions per user, tweets, etc. You can also add other metrics, e.g. to quantify the degree of inconsistency or stale data that a response might contain.

Try to adapt your experiments to show where your design shines and where it does not perform well. Between worst case and best case scenario is typically a wide range of different loads. To better understand the performance results you measure, consider sharing your load generation code with other students, to have a wider range of experiments. It is *explicitly permitted, and encouraged* to share your load generation code. However, every student should have their own implementation and report.

We will give you the opportunity to run your experiments on Serenity, a 64-core server available at the lab. We will allocate time slots of four hours to each of you, during which you can remotely log in on the machine and run your benchmarks. In one of the lab sessions you will get a demo on how to do this. More details will follow.

However, before running your experiments on the server, you should first run them on your machine, or one of the machines in the computer room. You should use a machine with at least four cores. You must make sure your experiments run correctly and provide you with the necessary results before you run them on the server, as you only have limited time available on the server.

Your report can consist of both experiments that you ran on the server, as well as on your local machine.

4 Reporting

Please follow the following outline for your report and concentrate on answering the posed questions:

1. **Overview:** Briefly summarize your overall implementation approach, and the experiments you performed. (max. 250 words)

2. **Implementation:**

    2.1 Implementation Strategy: Describe the implementation strategy on an abstract level. Depict your implementation strategy graphically.

    2.2 Scalability and Responsiveness: Discuss the scalability of your implementation. Answers amongst others the following questions:

    - Which trade-offs did you make? Where did you sacrifice data consistency to improve scalability? How does this decision improve scalability, and how does it affect data consistency?

- What is the worst case in terms of the mix of different user requests for your design? When would the responsiveness decrease, or when would the overall data consistency suffer significantly?
- How do you ensure conceptual scalability to hundreds, thousands, or millions of users? Are there any conceptual bottlenecks?

2.3 Implementation in Erlang: Give a brief explanation of how your approach maps onto its implementation in your Erlang source code.

3. Evaluation

- Describe your experiment setup including all relevant details:
  - which CPU, tools, platform, versions of software, etc. you used
  - all details that are necessary to put the results into context
  - all the parameters used to influence the runtime behavior of Erlang
  - watch out for the common benchmarking pitfalls: garbage collection issues, did you use the Erlang JIT compiler (HiPE), does your CPU support fancy features like Intel's Hyper-Threading[1] or Turbo Boost[2]?
  - Describe the different loads that you generate for your benchmarks. What are the proportions of different requests (tweet, get timeline, get tweets)? How many users/tweets/subscriptions does your system contain, how many clients are connected simultaneously, and how many requests are there per connection?
- Describe your experimental methodology:
  - What did you measure? For example: speed-up, latency, throughput, degree of inconsistency.
  - How did you measure?[3]
  - How often did you repeat your experiments, and how did you evaluate the results?
- Report results appropriately: use diagrams, report average or median and measurement errors, possibly use box plots. Graphical representations are preferred over large tables with many numbers.
- Interpret the results: make clear what the diagrams depict, and explain the results. How did your design decisions influence these results? If the results contradict your intuition, explain what caused this.

[1]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.

[2]Turbo Boost allows your processor to run at a higher clock rate than normal. It will be enabled in certain situations, when the work load 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.

[3]First starting point: http://www.erlang.org/documentation/doc-5.8.3/doc/efficiency_guide/profiling.html#benchmark.