Project Erlang: Slack

Multicore Programming

Assistant: Janwillem Swalens
Mail: jswalens@vub.ac.be
Office: 10F737

Deadline: 16 April 2017

For the first programming project, you will implement and evaluate a chat room application (similar to Slack) in Erlang. Several users can connect to your application, join different channels, and send messages. As the number of users increases, you need to scale the application to handle the increased load. For this project, you are allowed to sacrifice data consistency to increase scalability. You should also evaluate your system with a set of benchmarks, simulating a variety of workloads.

1 Procedure Overview

This project consists of three parts: the implementation of a scalable chat application, an evaluation of this system, and a report that describes the implementation and evaluation.

Deadline 16th of April at 23:59 CEST.

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, go to *Assignments (Opdrachten)* > *Project Erlang* and submit the ZIP file.

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.

2 A Scalable Chat Room

Your task is to implement a scalable chat room application, similar to Slack[1] (Figure 1). This chat application allows users to create several "channels" (or chat rooms), which they can join and send messages to.

[1] https://slack.com/

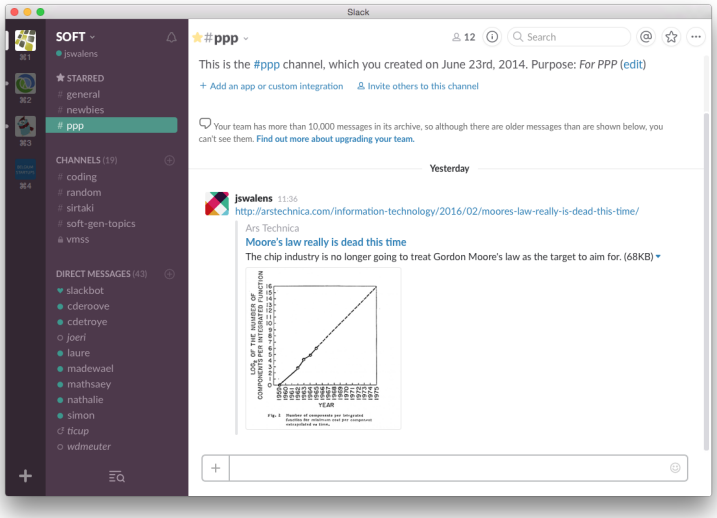

Figure 1: Slack

Your service needs to store information about the users, the channels they have joined, and the messages they have sent. This is depicted in Figure 2. Each message has a sender, a channel it was sent to, text, and a timestamp.

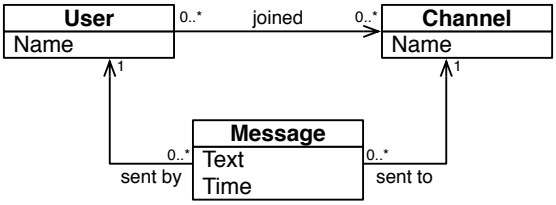

Figure 2: Data maintained by the chat application.

A typical session is shown in Figure 3. First, users need to register their user name, e.g. Alice and Bob. Next, they can log in and join channels: Alice and Bob join the home channel. When a message is sent to a channel, it is saved on the server, and broadcast to all other members of that channel *that are currently logged in*. Thus, in the figure, Alice's message is sent to Bob. Lastly, the operation `get_channel_history` retrieves all messages previously sent to a channel.

3 Implementation

The aim of this project is to create a scalable implementation of this application. We provide an example implementation in which the server consists of a single process. It is up to you to change this so as to make this scalable when there are many more users.

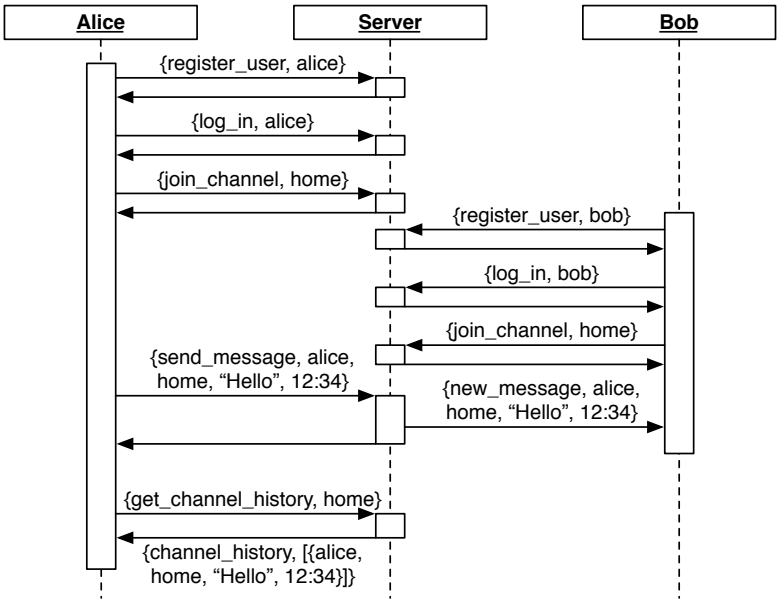

Figure 3: Typical session in which two clients, Alice and Bob, create accounts, log in, and join a channel. Alice then sends a message to that channel, which is forwarded to Bob.

Server and clients The application is separated into a server and several clients.

The server should consist of one or more processes. In the given example implementation, in `server_centralized.erl`, the server is a single process that contains all data discussed in the previous section. You will need to change this to increase scalability.

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 the server processes.

Required operations The 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 the server.

`log_in` Log in as a certain user. The client process will be notified of all messages that are sent to the channels that this user has joined.

`log_out` Log out. After logging out, the user will not get notifications of new messages anymore.

`join_channel` Join a channel. Joining a channel subscribes the user to notifications of new messages in that channel. (Even after the user has logged out and logged in again.)

`send_message` Send a message to a channel. The message is saved on the server, and broadcast to all other members of that channel that are currently logged in.

`get_channel_history` Get a channel's history. This returns a list of all messages sent to that channel.

The `register_user` and `log_in` requests return a pid, that should be used by the client for further communication. This is not necessarily the pid of the sender of the message.

Furthermore, client processes can receive the following message:

`new_message` A new message was sent to one of the channels that you joined.

Beyond these basic operations, you may add functionality to set up your experiments, like loading generated data sets, creating load, etc.

Your application only needs to support the minimal feature set described above, and should not support any of the advanced features that similar applications offer. Furthermore, you should concentrate on the back end functionality. An actual web front end is *not required* and *not desired*. Focus on the parallel and concurrency aspects.

Scalability over consistency Your application should contain all information shown in Figure 2, but it does not need to encode it as depicted. In fact, the actual data representation probably should be different to realize the goal of this project. You can also add other relations, for instance to directly represent the reverse of the *joined* relation: it might pay off to keep the list of members for each channel as well as the list of joined channels 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 stale data, messages may arrive in the wrong order, you might duplicate data over several processes that is not always consistent, or, in case you keep both sides of the joined relation, these might not always be consistent.

Example implementation We provide an example implementation, consisting of three files:

`server.erl` The 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 data actor.

`benchmark.erl` A starting point for benchmarks.

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. They will obscure your results and make it harder to compare approaches.

4 Evaluation

Next to your implementation, you should perform a thorough evaluation of your application. The goal of the evaluation is to show how your theoretical design performs in practice.

Metrics You should set up experiments to evaluate the performance of the system in best- and worst-case scenarios. There are several 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. send message, get channel history).
- The latency of broadcasting a message: if a user sends a message, how long does it take to arrive at other users? When does it reach the first user and when does it reach the last (if relevant)?
- 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)`).

Compare these metrics in different settings by varying parameters such as the number of processes, users, channels, members per channel, etc. Try to adapt your experiments to show where your design shines and where it does not perform well. Between worst-case and best-case scenarios, there is typically a wide range of different loads.

Sharing To have a wider range of experiments, you may share your benchmarks with other students. It is *explicitly permitted, and encouraged* to share benchmarking code. You can do this through the forum on the course page on PointCarré. However, remember that every student should have their own implementation and report.

Available hardware You can run your experiments on your own machine, or one one of the machines in the computer rooms. Make sure to use a machine with at least four cores.

Additionally, we give you the opportunity to run your experiments on "Serenity", a 64-core server available at the lab. Its specifications are in Figure 4. 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 the server as well as on your local machine.

Machine	
Model	HP Proliant DL585 G7
CPU	4 × AMD Opteron 6376
RAM	128 GB
Each processor has:	
Number of cores	16
Base clock speed	2.3 GHz
L1 cache	768 KB
L2 cache	16 MB
L3 cache	16 MB
Software	
OS	Ubuntu 16.04
Erlang/OTP version	18.3

Figure 4: Specifications of Serenity

5 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. (max. 250 words)

2. Implementation

2.1 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.

2.2 Scalability and Responsiveness 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?

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 experimental set-up, including all relevant details:
  - Which CPU, tools, platform, versions of software, etc. you used (even if you used Serenity, see Figure 4).
  - All details that are necessary to put the results into context.
  - All the parameters that influenced the run-time 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[2] or Turbo Boost[3]?
  - Describe the different loads that you generated for your benchmarks. What are the proportions of different requests? How many users/channels/members per channel does your system contain, how many clients are connected simultaneously, and how many requests are there per connection?
- Describe your experimental methodology:
  - What variables did you measure? For example: speed-up, latency, throughput, degree of inconsistency.
  - What parameters did you vary? For example: number of processes, users, channels, members per channel, number of simultaneous requests.
  - How did you measure? Good starting point: `http://erlang.org/doc/efficiency_guide/profiling.html#benchmark`.
  - How often did you repeat your experiments, and how did you evaluate the results?
- 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.
- 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.

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 each):

1. Your current implementation uses multiple cores on a single machine. Imagine now that the load is too high for one machine, and you want to distribute the

[2] 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.

[3] Turbo Boost 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.

load over multiple machines. How would you change your architecture, and 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[4]. 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 `Threads`: how would this affect the performance and how could you improve this?

[4] http://www.erlang.org/doc/efficiency_guide/processes.html