Project Erlang: Key-Value Store

Multicore Programming 2025

Janwillem Swalens (janwillem.swalens@vub.be)

For this project, you will implement and evaluate a key-value store in Erlang. Clients connect to the application and send requests to store and retrieve key-value pairs. Your application should implement these operations in a scalable way: it should be able to handle an increasing number of requests by distributing the load over multiple processes. 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**:
  - For regular students, the deadline is Thursday, **10 April** 2025 at 23:59.
  - For students official registered as working student for this course, the deadline is Wednesday, **28 May** 2025 at 23:59, although you are recommended to submit earlier.
- **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 scalable key-value store

Erlang has been used to implement key-value and document stores, such as Riak, CouchDB, and Mnesia. For this project, you will implement a basic but scalable key-value store, that you will then make more scalable.

Like many similar systems, we will organize the key-value pairs into buckets. A typical session is shown in Figure 1. There is a server and two clients: Alice and Bob. First, Alice connects to the server and sends a request to create a new bucket called `shopping`. Next, she stores the value `1` for the key `milk` in that bucket. Then, Bob connects and stores the value `3` for the key `eggs` in the same bucket. When Bob retrieves the value for the key `milk`, the server responds with the value `1`, as set by Alice. Then, Alice deletes the value

for the key `eggs` in the `shopping` bucket. When Bob now retrieves the value for the key `eggs`, the server responds with `not_found`. Finally, Alice disconnects from the server.

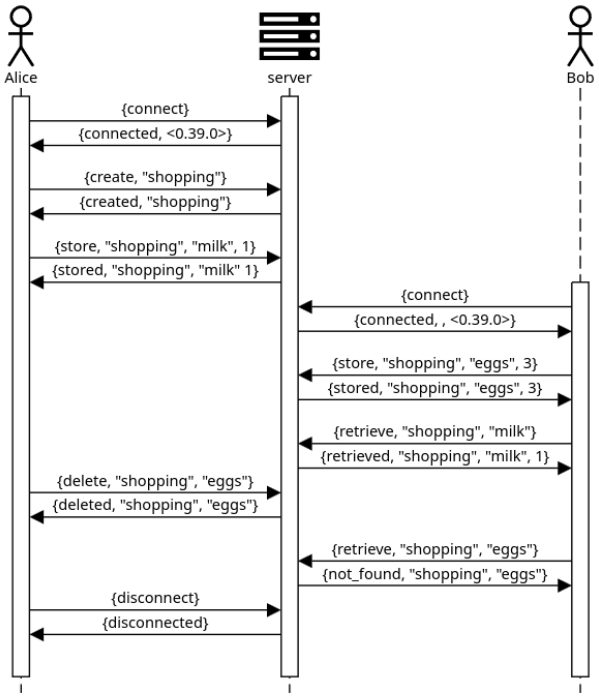

Figure 1: An example session between a client and the server. (Tip for your report: this sequence diagram was created using https://sequencediagram.org.)

Implementation

The aim of this project is to create a scalable implementation of this application in Erlang. We provide an example implementation in which there is only one process in which all data is stored. It is up to you to implement this in a more scalable manner.

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. In your benchmarks you will generate many processes, each of which represents a client that requests some information from the server.

Required operations

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

- **`{connect}`**: Connect to the server.

- **`{disconnect}`**: Disconnect from the server. (This is not required, but may be useful depending on your implementation.)
- **`{create, BucketName}`**: Create a new bucket.
- **`{store, BucketName, Key, Value}`**: Store a value for a key in a bucket.
- **`{retrieve, BucketName, Key}`**: Retrieve a value for a key in a bucket.
- **`{delete, BucketName, Key}`**: Delete a value for a key in a bucket.

The `connect` request 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 to handle the requests of a subset of the clients, which may be useful for scalability.

You may add other operations if you need them. For example, in some implementations it might be useful to have an operation that returns the pid of a process to be used for accessing a specific bucket.

Parallelization

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

You are free to choose how you want to parallelize your application. Below are some potential directions, which each have several options and trade-offs you can explore.

- Data can be *sharded* over multiple processes, i.e. each process contains a subset ('shard') of the data. You can assume a fixed number of processes or you can implement a protocol that allows for dynamic sharding ('auto sharding').
- Data can be *replicated* over multiple processes, i.e. processes contain a copy of (some of) the data. You are free to choose whether you want to implement a protocol that ensures strong consistency between the copies, or to only guarantee eventual consistency (i.e. allowing the data to be temporarily inconsistent across processes).
- Clients can be *load balanced* over multiple processes. This may assume a fixed number of processes and a static distribution of clients over processes, or you can implement a dynamic load balancing technique.
- A *caching mechanism* can improve the performance of the `retrieve` operation.

You are free to choose any of these options, explore other approaches, or a combination of these.

Example implementation

We provide an example implementation, consisting of these files:

- **`server.erl`**: the server's API. Existing operations should remain unchanged, as this is the interface that will be used by benchmarking code, but this can be extended with other operations.
- **`server_centralized.erl`**: an example implementation that uses a single process to represent the server. You can use this as a starting point for your implementation and compare your approach to this one.
- **`benchmark.erl`**, **`run_benchmarks.sh`**: a starting point for benchmarks. These files demonstrate 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

- The aim of this project is to focus on parallelism, not distribution. You should therefore *not* experiment with distributing server processes over multiple machines. Instead, run everything in one Erlang VM and use regular message passing to communicate between processes.
- Your application only needs to support the minimal set of operations described above. Focus on the parallel and concurrency aspects. For instance, you do not need to implement authentication or failure handling.
- Do not use Erlang's `ets` (Erlang Term Storage) module: the point of the project is to implement your own data store.

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. retrieve, store, delete).
- The latency of storing a value: if a client stores a value, how long does it take for other clients to see it? When does it reach the first client and when does it reach the last (if relevant)?
- The speed-up (of a certain operation or scenario) 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? 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 processes, clients/buckets/keys per process, how buckets or clients are spread over processes.
- Parameters such as the number of clients, buckets, keys, and keys per bucket.
- Which (mix of) requests you execute. In particular, the read/write ratio may greatly influence the performance of your system.
- 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.

You are expected to perform your experiments and process the results in a scientific manner. This means that you should repeat each experiment multiple times, report averages or medians as well as measurement errors, and use graphical representations (e.g. box plots) to visualize the results. You should also describe the experimental set-up in detail, including all relevant parameters, and describe the methodology you used to measure the results. Refer back to what we've seen in class on how to perform benchmarks.

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 the text files with individual results, calculate statistics, and plot the results. You can use these scripts as a starting point, but you are free to use other tools to run and process your benchmarks.

Hardware

For this project, you should run your experiments on "Firefly", a 64-core server available at the SOFT lab. Its specifications are in Table 1. 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.

Next to the (required) experiments on the server, your report can also contain experiments that you ran on your local machine or the machines in the computer rooms, but make sure to use machines with at least four (physical) cores.

Include the specifications of the machines you used in your report, as in Table 1, 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/25/OTP_VERSION` to get the Erlang/OTP version. You may need to change the major version number in the path.

Table 1: Specifications of the machine "Firefly"

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 24.04.1 (Linux kernel 6.8.0-49-generic)
Erlang/OTP version	25.3.2.8

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. E.g. each of the 30 repetitions should spawn new, freshly

initialized, server processes.
- Does your CPU support Intel's *Hyper-Threading*? On these processors, 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. So, make sure to keep the power connected.

Also check out the relevant section in the Erlang documentation.

Resources

There are several widely used implementations of key-value stores in Erlang and other languages. You can read up on them to get inspiration for your implementation. In particular, these documents may be useful:

- The Chapter "Riak and Erlang/OTP" in the book The Architecture of Open Source Applications. Riak is a popular key-value store written in Erlang.
- The documentation of Elixir contains a tutorial in which a key-value store is implemented. Elixir is another language that runs on the Erlang VM.

You are not expected to implement all features these systems offer. These links are provided merely as inspiration.

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** (1 or 2 paragraphs): Briefly summarize your overall implementation approach and the experiments you performed.
2. **Implementation** (1 or 2 pages maximum, excluding figures):
    1. **Architecture**: Describe your project's software architecture on a high level. Describe the different aspects of your implementation, such as how the data or clients are distributed over processes. Highlight any special features such as automatic load balancing or sharding. 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 requests? When would the responsiveness decrease?
        - How do you ensure conceptual scalability to hundreds, thousands, or millions of clients or key-value pairs? Are there any conceptual bottlenecks left?
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 Table 1).
  - 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) *variables* did you measure? For example: speed-up, latency, throughput, degree of inconsistency.
  - What (independent) *parameters* did you vary? For example: number of scheduler threads, number of server processes, clients, buckets, keys per bucket, read/write ratio, number of simultaneous requests.
  - Describe the load that you generated and other relevant *parameters*. What are the proportions of different requests? 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. 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 `Threads`: how would this affect the performance and how could you improve this?
   2. 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?

Remember that shorter is better: edit your text to make it concise and to the point.