PCPP IT University of Copenhagen, F2024

Exercises week 13

Last update: 2024/11/24

Goal of the exercises

The goals of this week's exercises are:

- Design and implement an Actor system with dynamic topology.
- Design and implement an Actor monitoring mechanism to handle failures.
- Design and implement a dynamic load balancing mechanism for a server.

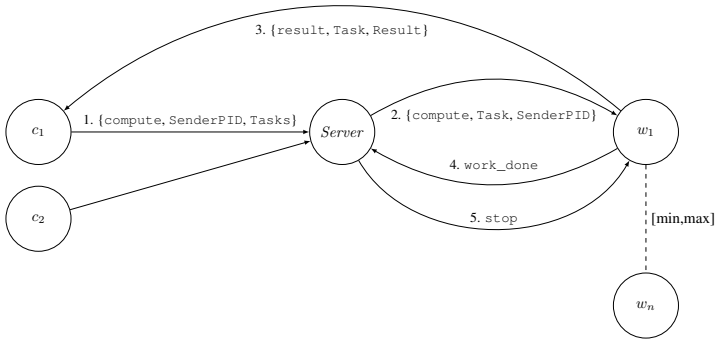
Exercise 13.1 Your task in this exercise is to implement an elastic fault-tolerant system using Erlang. In particular, the goal is to implement a prototype Function as a Service (FaaS) server for serverless computing in the style of AWS Lambda, Google Cloud Functions, or Azure Functions. The system consists of three types of actors:

- *Client*: These actors send a list of operation for the server to compute.
- *Server*: This actor manages a set of workers (see below) in a fault-tolerant and efficient manner. If a worker fails during the execution of an operation, the server restarts it. The server keeps a minimum number of workers active, but new workers may be created on-demand for high workloads. If the workload decreases, the number of workers is reduced to the minimum again.
- *Worker*: These actors simply compute operations sent by the Server. When they are finished they send the result directly to the client, and tell the server that they are done with the computation.

The directory `code-exercises/week13exercises` contains the implementation of the client and worker modules. The code skeleton also defines all the messages exchanged by client, server and worker actors. *Your task in this exercise focuses on implementing the server.*

We predefined a `task` record so that there is a common interface to write tasks; you can find it in the header file `defs.hrl`. It is important to use the `task` record for sending tasks, as this is the format that the workers can process. The format for tasks is straightforward. They have two elements `function` and `arguments`. The first element, `function`, must be bind to an Erlang function, and the second, `arguments`, must be bind to a list with the arguments for the function. For instance, to model a function that computes the multiplication $2 \cdot 3$, we can define the following task `#task{function=fun (X,Y) -> X*Y end, arguments=[2,3]}`.

The figure below shows an example of the basic behavior, and the type of messages that the different actors can exchange.



In this system we have two clients c_1 and c_2, the $Server$, and n workers $w_1, \ldots, w_n$. The basic behaviour of the system is as follows:

1. The client sends a list of tasks to the server.
2. The server sends each task to a worker.
3. After the worker finishes the computation, it sends the result directly to the client, and sends a message to the $Server$ notifying that the computation is finished.
4. Upon receiving the notification from the worker, the server may decide to stop the worker.

This basic behaviour does not explicitly show several complications that the server must deal with. In the following exercises, we discuss them one by one and you should find solutions to them. The first exercise starts with an implementation with only a limited set of features, and the following exercises build on the previous exercise to build the complete fault-tolerant elastic server.

Important: In your submission, include only the final version of the code for `server.erl`. That is, the code after solving all exercises. If there are parts of the code that you remove from one exercise to the other, please describe it and explain why in the written answer.

Note: Remember that the only file you should modify for the mandatory exercises is `server.erl`, and `client.erl` only for exercise 13.1.5.

Mandatory

1. Implement the server actor so that it works with a fixed number of workers. To this end, the server must be able to keep track of idle workers, busy workers and pending tasks. The starting function for the server takes a parameter indicating the number of workers for the server. Upon receiving a list of tasks to compute, the server should iterate through each of them and, for each task, it should:

   (a) If there are idle workers, then it must send the task to one of the idle workers.

   (b) If there are no idle workers, then it must place the task in the list of pending tasks.

   Note that, when a worker finishes computing a task, it sends a `work_done` message to the server. You must also implement the behavior upon receiving this message, which is specified as follows:

   (a) If there are pending tasks, then the server selects one and sends it to the worker.

   (b) If there are no pending tasks, it must set the worker as idle.

   Provide the code for implementing this behavior, and an explanation of all the elements in the actor (i.e., state, start/init functions and message handling).

   Note: After finishing this implementation, you should be able to run the function `send_tasks(Tasks, NumWorkers)` in `client.erl`. We recommend running this function with different inputs and to check that the number of idle workers after finishing the execution is as expected. To check the list of idle workers,

you may add debugging messages to the server actor, e.g., you may add a message `idle_workers` in the server so that, when it is received, the server prints the list of idle workers. Since communication in the system is asynchronous, it is not possible to determine when all computation has finished.

To avoid unnecessary complications, we recommend that you wait to send debugging messages until you observe all the printing of the `result` messages received by the client in the shell, and then send a debugging message to the server. Recall that using `timer:sleep(Millisec)` is generally not a good idea, as it is not possible to reliably set `Millisec`.

2. Update the server actor so that it has a minimum and maximum number of workers. During initialization the server must start the minimum number of workers. Upon receiving a list of tasks, for each task, the behavior of the server is updated as follows:

   (a) If there are idle workers, then it must send the task to one of the idle workers (same as in the previous exercise).

   (b) If there are no idle workers, but the number of busy workers is less than the maximum number of workers, then start a new worker and send the task to the worker.

   (c) If there are no idle workers and the number of busy workers equals the maximum number of workers, then it must place the task in the list of pending tasks.

   The behavior upon receiving a `work_done` message must be the same as in the previous exercise. Make sure that in your implementation you never have more workers busy than the specified maximum number of workers. Provide the updated code for implementing this behavior, and an explanation of all the updated elements in the actor (i.e., updates in the state, updates in start/init functions and updates in message handling).

   Note: After finishing this implementation, you should be able to run the function `send_tasks(Tasks, MinWorkers, MaxWorkers)` in `client.erl`. As before, we recommend to run this function with different input parameters and check that the number of idle workers after finishing the execution is as expected.

3. Now we turn the focus to making the server fault-tolerant. Update the server code so that it is notified every time an actor crashes or terminates normally. Update the server message handling code so that, if an worker crashes, the server prints an error message with the reason, creates a new worker and places the new worker as idle. The task that caused the crash is dismissed. Explain the modifications you made to your implementation.

   Hint: It might be useful to revisit the process monitoring mechanism in Erlang:

   - https://learnyousomeerlang.com/errors-and-processes#monitors
   - https://www.erlang.org/docs/26/reference_manual/processes#monitors.

   Note: After finishing this implementation, you should be able to run the function `crashing_example(MinWorkers, MaxWorkers)` in `client.erl`. If your implementation is correct, you should observe (in the shell) the message to be printed by the server upon receiving a failure signal from a worker.

4. The last feature for this exercise is to make the server elastic. In this context, elasticity means that the server keeps a number of workers according to the workload. To this end, extend the server so that, when it receives a `work_done` message, it instructs the worker to terminate if the number of workers in the system is greater than `minWorkers`. Note that workers handle messages of type `stop`. Upon receiving this message, worker actors terminate their execution normally. Make sure that your implementation guarantees that the server maintains the minimum number of workers in the system. Explain the modifications you made to your implementation.

   Note: After finishing this implementation, you should be able to run the function `send_tasks(Tasks, MinWorkers, MaxWorkers)` in `client.erl`. As before, we recommend to run this function with different input parameters. However, after finishing the execution, in this case, you should observe that the number of idle workers is `MinWorkers`.

Challenging

5. Is there any other situation (different than those covered above) where the $Server$ could check the list of pending tasks and pick one? If so, explain it, and implement a solution that handles the list of pending tasks as you describe. Furthermore, add an example function in the `client.erl` module that triggers the error you describe.
6. Is your implementation deadlock free? Recall that in lecture 12 we showed an example of deadlock in the bounded buffer system. In this task, you must argue whether a deadlock situation can occur in your implementation. In an actor system, a deadlock may be reached if there is a state of the entire system where no process can make progress because they are waiting for messages from each other.