PCPP IT University of Copenhagen, F2022

Exercises week 12

Last update: 2022/11/24

Goal of the exercises

The goals of this week's exercises are:

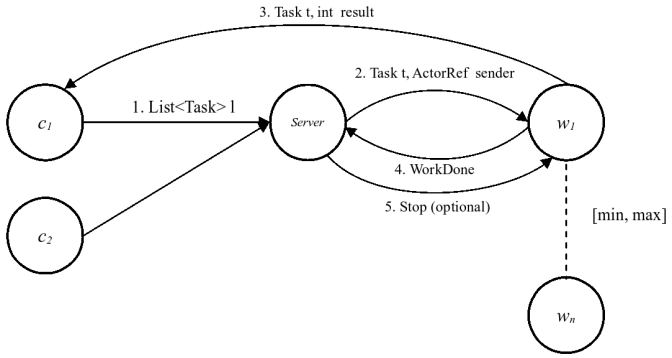
- Design and implement an Actor system with dynamic topology.
- Design and implement an Actor supervision mechanism to handle failures.
- Design and implement a dynamic load balancing mechanism for a server.

Exercise 12.1 Your task in this exercise is to implement an elastic fault-tolerant mathematics system using Akka. 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 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/week12exercises` contains the implementation of Client and Worker—as well as the standard Guardian and Main classes. The code skeleton also defines all the messages exchanged by Client, Server and Worker. *Your task in this exercise focuses on implementing the server*.

We use an auxiliary `Task` class to encapsulate the operations in a single object. The class contains an enumeration (`BinaryOperation`) with 4 operations corresponding to summation (`SUM`), subtraction (`SUB`), multiplication (`MUL`) and division (`DIV`). A task object contains two parameters `x` and `y`, and the `BinaryOperation` to perform on them.



The figure above shows an example of the basic behaviour. 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.

Note: Remember that the only file you should modify for the exercise is `Server.java`.

Mandatory

1. The *Server* should only send tasks to workers that are idle. If a task needs to be processed and there are no idle workers, then it should be placed in a list of pending tasks. Define the state of the *Server* so that it can keep track of idle workers, busy workers and pending tasks. Explain why the elements of the state that you added are sufficient to keep track of busy workers and pending tasks.
2. As mentioned above, the *Server* should always have a minimum number of active workers, and a limit in the number of workers it can hold active at the same time. Extend the state of the *Server* with two variables containing the number of minimum `minWorkers` and maximum workers `maxWorkers`. Also, write the constructor of the *Server* so that it spawns `minWorkers` workers. The constructor must properly initialize all the elements of the state, i.e., idle and busy workers, pending tasks, and min/max workers. Explain the implementation of the constructor and why the new elements of the state are sufficient for this exercise.
3. Now we move to handling a list of tasks sent from a client. Write the message handler for messages of type `ComputeTasks`, which contains the list of tasks sent by the client. For each task in the list, the *Server* must proceed as follows:
   (a) If there are idle workers the task is sent to a worker—using a `ComputeTask` message.
   (b) If there are no idle workers, but the number of busy workers is less than `maxWorkers`, then spawn a new worker and send the task.
   (c) If none of the above conditions hold, then the task must be placed in the list of pending tasks.

   Explain how your implementation addresses these cases.
4. If you run the system now, you will notice that two actors will crash due to a division by zero exception. Your task now is to make the server fault-tolerant. The server must *watch* all the workers it spawns, and, in case any of them fails, a new worker should be spawned and added to the list of idle workers. Explain how your implementation handles this situation. Hint: It might be useful to revisit `getContext().watch(...)`, the class `ChildFailed`, and `onSignal(...)`.
5. As you probably have already noticed, workers send a `WorkDone` message to the *Server* when they finish the computation. Here you have to implement the handler for messages of type `WorkDone`. In particular, when the *Server* receives this type of message, it must proceed as follows:
   (a) If there are pending tasks, the *Server* takes one and sends it to the worker.
   (b) If there are no pending tasks, the *Server* updates the status of the worker (i.e., move it from busy to idle).

   Explain how your implementation handles the `WorkDone` message and addresses these cases.

Challenging

6. 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.
7. This exercise sheet started by mentioning that we will implement an *elastic* and fault-tolerant server. However, we have not taken care of elasticity. In this context, elasticity means that the server keeps a number of workers proportional to the workload. Extend the system so that, when the *Server* receives a `WorkDone` message, it tells the worker to terminate if the number of idle workers is greater than `minWorkers`.

   Hint I: Note that workers can handle messages of type `Stop` that stop the worker.

   Hint II: When an actor executes `Behaviours.stopped()` it automatically sends a signal of type `Terminated` to the actor watching it. It is important to note two things:
   (a) `ChildFailed` extends from `Terminated`

(b) The message handler tries to find a handler in the same order as defined in `newReceiveBuilder()`. Consequently, if you place the handler for `Terminated` before the one for `ChildFailed`, the latter will never be executed.