2013

DIT027 – Exercises on Distributed Erlang

The exercises are supposed to be done individually. However, it is allowed to discuss the problems with other students and share ideas with other students. A good rule of thumb is to only use pen and paper when discussing the exercises, in that way you won't cheat “by accident”.

There are two separate problems to solve, you should submit them in a file named **dist_erlang.erl**. There is a skeleton-module provided, **do not change the list of exported functions!**

Just like for the first set of assignments, I will provide some test cases for you at a later time (probably around the workshop the 14th).

Deadline is **17/10 – 2013 at 8:00**.

GOOD LUCK!

Problem 1 – Client-Server

You are supposed to implement a simple **Term storage server.** The server is supposed to be able to store Erlang terms associated with keys. The server (process) should be registered under the name **`sts`**, thus there is no need to pass around its Pid. Also, the term storage is ***per process***, i.e. two different processes are able to store different values for the same key. (From the perspective of a single process, there is little difference to the DB implemented in the sequential assignment.) The process calling store/fetch/flush is automatically passed to the server.

A shell interaction with such a server could look like (note, here we are storing terms for a single process – the shell process):

```
34> dist_erlang:start().
{ok, <0.97.0>}
35> dist_erlang:start().
{ok, <0.97.0>}
```

I.e. if it is already started, do nothing but return the already started Pid.

```
36> dist_erlang:store(key1, val1).
{ok,no_value}
```

store/2 returns ok and the old value (or no_value, if the key has not been used before!).

```
37> dist_erlang:store(key2, {a, [more, complex], thing}).
{ok,no_value}
38> dist_erlang:fetch(key2).
{ok,{a,[more,complex],thing}}
39> dist_erlang:fetch(key3).
{error,not_found}
```

fetch/1 returns either {ok, Value} or {error, not_found}. fetch/1 does not change the stored term.

```
40> dist_erlang:store(key1, val1_changed).
{ok,val1}
41> dist_erlang:flush().
{ok,flushed}
42> dist_erlang:fetch(key1).
{error, not_found}
43> dist_erlang:stop().
stopped
44> dist_erlang:stop().
already_stopped
```

Stopping the server twice should not be a problem, and flush/0 removes all terms for the **calling** process (i.e. **not for all processes**!).

The skeleton-module export the functions: start/0, stop/0, init/0, store/2, fetch/1, and flush/0. You should not change this. Your task is to implement the functions and correctly spawn a process running the init function (in turn calling a local server-loop function) in start/0. start/0 should spawn a separate server process, stop/0 should stop the server, and store/2, fetch/1, and flush/0 are wrapper-functions for the message passing between the server and the caller/client. Make sure that stop/0 is synchronized (i.e. waiting until the server is actually stopped).

You also need to write some code to handle the actual storage of terms. Since this exercise is about implementing the server, you may use any back-end you like (a list, a tree, an ETS-table, etc.). For example, it is ok to re-use the DB from the sequential assignment.

A more complex interaction with the term storage, using multiple processes, and showing how they interact (or rather don't interact, as the term storage should be solely on a per process base!) is provided by the example below. Be aware that several processes are run in parallel, and the order (but not the content!) of the printouts will vary between different executions!

```
-import(dist_erlang, [fetch/1, store/2, flush/0, start/0]).

test() ->
  start(),
  spawn(fun() ->
            io:format("Process ~p started!\n", [self()]),
            store(key1, val1), store(key2, val2),
            io:format("~p fetch(key1): ~p\n", [self(), fetch(key1)]),
            io:format("~p fetch(key2): ~p\n", [self(), fetch(key2)]),
            flush(),
            io:format("~p fetch(key1): ~p\n", [self(), fetch(key1)])
        end),
  spawn(fun() ->
            io:format("Process ~p started!\n", [self()]),
            store(key1, val2), store(key2, val1), %% Values swapped!
            io:format("~p fetch(key1): ~p\n", [self(), fetch(key1)]),
            io:format("~p fetch(key2): ~p\n", [self(), fetch(key2)]),
            store(key1, blafooo),
            io:format("~p fetch(key1): ~p\n", [self(), fetch(key1)])
        end),
  ok.
```

And running the test/0 function:

```
46> dist_erlang_test:test().
Process <0.126.0> started!
Process <0.127.0> started!
ok
<0.126.0> fetch(key1): {ok,val1}
<0.127.0> fetch(key1): {ok,val2}
<0.126.0> fetch(key2): {ok,val2}
<0.127.0> fetch(key2): {ok,val1}
<0.126.0> fetch(key1): {error,not_found}
<0.127.0> fetch(key1): {ok,blafooo}
48> dist_erlang_test:test().
Process <0.171.0> started!
Process <0.170.0> started!
ok
<0.171.0> fetch(key1): {ok,val2}
<0.170.0> fetch(key1): {ok,val1}
<0.170.0> fetch(key2): {ok,val2}
<0.171.0> fetch(key2): {ok,val1}
<0.170.0> fetch(key1): {error,not_found}
<0.171.0> fetch(key1): {ok,blafooo}
```

Problem 2 – Process handling

In the new era of computation, where normal PC's and laptops are having multi-core processors, parallel computation is crucial for program efficiency. In this problem, we will experiment with one simple way of doing this.

Imagine that we have a (very) long list of computations that we need to perform. (Or a list of data where we need to perform a computation for each element in the list.) A simple way to parallelize this would be to split the list in two, and give half of the list to a separate process. (Or in four if you happen to have four cores, etc...) With the lightweight processes in Erlang, we can go a step further: We can create a process for each element in the list. (**Note:** this is overly naïve, and if each computation is small the overhead is too much even for Erlang processes!)

In the skeleton module you are given the mysterious function **task/1**, that we use as an example. **task/1** takes an integer parameter (range 0..100), and performs a long computation, and returns an integer value.

The first task is to implement a function **dist_task/1**. The function should, given this list of numbers, spawn a process for each computation and collect all the computation results. Using this function could look like:

```
133> Data = [17, 9, 3, 12, 11, 24, 27].

[17,9,3,12,11,24,27]

134> dist_erlang:dist_task(Data).

[375,460,358,511,494,494,324]
```

Here 7 computation processes were spawned and each process calculated one task.

Hint: Make sure that you collect the results in the right order (i.e. that you maintain the same order for the results as for the input).

This computation pattern is a specific example of a general pattern, it is often called ***parallel_map*** or ***pmap.*** It works very much like the **lists:map/2** in Erlang. With the difference that each function application is evaluated in a separate process. (**Hint:** the similarity with lists:map/2 could be used to test your implementation!)

Your second task is to implement the more general **pmap/2**, where the first argument is the function to apply and the second argument is the list of data. You should take into account the possibility of a function crashing, i.e. your implementation should work also for the provided **faulty_task/1**, and not hang.

```
135> dist_erlang:pmap(fun dist_erlang:task/1, Data).
[375,460,358,511,494,494,324]
136> dist_erlang:pmap(fun dist_erlang:faulty_task/1, Data).
[375,460,358,511,494,494,unexpected_error]
137> dist_erlang:pmap(fun dist_erlang:faulty_task/1, Data).
[375,460,358,511,494,494,324]
138> dist_erlang:pmap(fun dist_erlang:faulty_task/1, Data).
[375,unexpected_error,358,511,494,494,324]
```

Hint: Make sure that you actually do things concurrently, that is, dist_task/1 should be more efficient that sequentially calculating the tasks. You could use **timer:tc/3** to measure the performance (be aware that you need to have a multi-core processor in order to actually gain any performance).