Multicore Programming

Lab 3: Concurrent Programming with Erlang

Assistant: Janwillem Swalens
Mail: jswalens@vub.ac.be
Office: 10F719

26 February 2014

In this exercise session, you will use Erlang's actors to run several tasks concurrently. This will be applied to a Snake game, in which multiple players will be active at the same time. You will also learn how to deal with unreliable actors, and learn how to use Erlang's support for distribution to allow different programs, or even different machines, to communicate.

Send in your solutions by next Monday (at the latest at 23:59), by mailing them to `jswalens@vub.ac.be`.

1 Preparation

1.1 Material

The material for this lab is on Pointcarré.

For all exercises, you can use `make` to compile the Erlang source files, and `./run.sh` to execute them. They are the same as for the previous assignment session.

1.2 Links and Literature

- Erlang documentation: `http://www.erlang.org/doc/`
- Erlang course: `http://www.erlang.org/course/course.html`
- Alternative Erlang documentation (easier to search): `http://erldocs.com`
- Google usually helps best with a query like: `erlang man $searchTerm`
- Concurrent Programming in Erlang by J. Armstrong, R. Virding, C. Wikström, M. Williams: `http://www.erlang.org/download/erlang-book-part1.pdf`
- EUnit manual: `http://www.erlang.org/doc/apps/eunit/chapter.html`

2 Exercise 1: Concurrent Hello World

`hello_world.erl`

1. Check `hello_world.erl` or see listing **??**.
2. Use "`erlc hello_world.erl`" and "`./run.sh hello_world start`" to compile and execute it.
3. Serialize the output of “Hello World!”: the string should be printed in the correct order. However, do not remove the random sleeps. Instead, use message send and receives to pass the string from one actor to the next. Make sure the serialization is stable, and not a consequence of implementation details of the scheduler.

Listing 1: hello_world.erl

```
-module(hello_world).
-export([start/0, print/2]).

start() ->
    HelloWorld = "Hello World!",
    lists:map(fun printer/1, HelloWorld),
    timer:sleep(1000),
    io:nl().

printer(Char) ->
    spawn(?MODULE, print, [Char, random:uniform(500)]).

print(Char, Sleep) ->
    timer:sleep(Sleep),
    io:put_chars([Char]).
```

3 Exercise 2: Reliable Systems

`manager.erl, unreliable_worker.erl`

Joe Armstrong on reliable systems and how to handle errors in Erlang:

> I often think the way we do errors is the most misunderstood part of Erlang.
>
> If you have a system with large numbers of small processes, having a few die is no big deal. All you do is detect these errors *somewhere else* and let the process that detects the error perform the *corrective action*. *Somewhere else* being on a physically different machine is the basis for building fault-tolerant systems.
>
> In sequential languages error avoidance is a huge deal – if you have only one process and it crashes you are are screwed. Thus in C, avoiding errors is an enormous deal. The entire school of *defensive programming* comes from the idea

> that if you only have one process you'd better take enormous steps to make sure it doesn't crash.
>
> Not so in Erlang.
>
> The notion of an error-detection mechanism that works across machine boundaries is the key idea in Erlang. `link/spawn_link/trap_errors/monitor` does the magic stuff.
>
> I think this point should be hammered home early and often.
>
> > "To build a fault-tolerant system you need at least two machines"
> >
> > "Let it crash"

One of the typical ideas found in systems built with Erlang, is that errors are deferred to levels of higher authority. This helps to reduce the amount of error handling code and usually utilizes the actor approach, to just restart faulty processes.

In this exercise, you have to implement a manager process which makes sure all work is done properly.

The worker implementation is already given in `unreliable_worker.erl`. The basic structure of the manager is given in `manager.erl`.

Tasks:

1. Implement the manager module.
2. Start another process/actor for each item of work.
3. Processes can fail without completing their work, these processes should be restarted.
4. Make sure all work has been completed properly in the end.

`erlang:spawn_link/3` might be useful: `http://www.erlang.org/doc/man/erlang.html#spawn_link-1`. To learn more about links between processes, you can read `http://learnyousomeerlang.com/errors-and-processes`.

4 Exercise 3: Snakes, Actors, and Artificial Intelligences

In the directory `snake` is a simple implementation of the classic snake game. A snake is moving on a 10×10 field and grows when it eats apples. The game ends when the snake bites itself. It already comes with a simple AI, but all snakes are on their own field. New features you could add:

- Extend the game so multiple players/AIs can play on the same field. Currently, each player is on a separate field.
- Each player should be represented by an actor.

- The game is turn-based (with each turn lasting 500 ms). During each turn the players can submit the new direction in which they want to move (perhaps multiple times, only the last one is taken into account).
- There should be a timer/heartbeat actor. After the game master gets a time signal, the result of the next move is computed and the new board configuration is made available. The game master checks for collisions and eaten apples.
- The players should always have access to the latest board, i.e. the status after the last completed turn.

Start by opening all files and determining their function. To understand what each file does, read the comments at the top of the file. You might want to draw a diagram to understand the relations between the different actors.

Go Distributed (optional[1])

Erlang provides the basic components to connect distributed VMs easily. Extend the Snake game to enable multiple Erlang instances to connect to a common game master and allow multiple human players to interact with each other.

You can find documentation on distributed Erlang here:

- `http://www.erlang.org/doc/reference_manual/distributed.html`
  Official documentation on distributed Erlang.
- `http://learnyousomeerlang.com/distribunomicon#setting-up-an-erlang-cluster`
  Section “Setting up an Erlang cluster” from the book “Learn You Some Erlang for Great Good!”

For this game, one solution would be to let the master node register the game master under a name which then can be used via an RPC call to retrieve the process id from connecting players.

[1]If you finish this, feel free to include it in your solutions. It is however not required.