Distributed Systems
TP n°1 - Erlang for distributed systems

Hadrien Croubois
hadrien.croubois@ens-lyon.fr

4/2/2016

All documents are available on my website: `http://hadriencroubois.com/#Teaching`

References

Erlang is full of features, and there is no way we will be able to cover everything here. Therefore, you will have to learn most of it by yourself. But don't worry ! Erlang's online documentation is very complete, and the teaching assistant (me) is here to help you. Just make sure to use google before asking elementary question.

Online manuals are available here: `http://erlang.org/doc/`

For informations about the function provided, they are all referenced by module. For example, details about the function `erlang:spawn/3` (function `spawn` of the module `erlang`, in it's 3 parameter version) can be found at `http://erlang.org/doc/man/erlang.html#spawn-3`. The following module will be very useful in this class (you should find almost everything you are looking for in those) :

`erlang`, `io`, `math`, `lists`, `random`

Data-types in Erlang

In order not to be completely lost, the first thing you have to learn is what data-types exist in Erlang and how to recognize them ! Many pages list them. Try to find a description of each of those:

Number, Atom, Variable, Tuples, Lists

Question 1

a) Can an atom contain one or more space ?

b) Can an atom start with a capital letter ?

c) Can a list contain elements of different types ?

d) How are string represented ?

Erlang, a functional language

Unlike C, Erlang is a fully functional language. This means that the all the logic is done through pattern matching and recursion. This can feel tricky at first, but it just comes done to changing your point of view.

Matching can be done on structure (empty lists), numeric values, atoms and even variables. Matching can also be used to affect values to unused variable. Note that as variable can be modified (they are in fact constant), matching on a variable has very different behaviour depending on whether or not a this variable is used or not.

Matching on structure:

```
length([])    -> 0;
length([_|T]) -> length(T) + 1.
```

Matching on atoms:

```
receive
  {PID, ping} -> PID!pong;
  {PID, pong} ->
    io:fwrite("pong from ~p!~n", [PID])
end.
```

Matching on numeric value:

```
isnull(0)   -> true;
isnull(0.0) -> true;
isnull(_)   -> false.
```

Matching on a variable:

```
equals(V, []   ) -> [];
equals(V, [V|T]) -> [true |equals(V,T)];
equals(V, [_|T]) -> [false|equals(V,T)].
```

Figure 1: Example of matching in erlang

Question 2

a) Write a function `max(L)` which returns the maximum elements of the list `L`.

b) Write a function `perimeter(Form)` which computes the perimeter of the form. A form can be any of the following:

- `{square, Side}`
- `{circle, Radius}`
- `{triangle, A, B, C}`

c) Write a function `sort(L)` which sort `L`, a list of integer.

Functions are objects !

Question 3

a) Write a function `map(F,L)` which returns `[F(L1), F(L2), ..., F(Ln)]` with `L=[L1, L2, ..., Ln]`.

b) Write a function `partial(F,P1)` where `F` is a two-parameter function. `partial(F,P1)` must return a one parameter function which, if given `P2`, would return `F(P1,P2)`.

c) Use `partial` and `map` to write `multimap(F,L)` which does the same as `map` but takes and returns a list of lists.

Spawning and communication for distributed systems

Question 4

a) Write a function `multispawn(L)` where `L` is a list of tuples of the form `{M,F,L}` where:

- `M` is a module name (an atome)
- `F` is a function name (an atom)
- `L` is a parameter list for `F`

`multispawn` must, for each tuple, spawn one process which computes `M:F(L)`. Multispawn must return the list of PIDs for those spawn processes.

b) Write a function `dreturn(Pid, T)` that takes a `{M,F,L}` tuple `T` as above, computes `M:F(L)` and then sends back `{Result, self()}` to `Pid`.

c) Write a `delegate(L)` which is the same as `multispawn` but returns a list of results (rather then a list of spawned PIDs).

Distributed algorithms

Question 5

a) Write a function `dmax(L)` which computes a distributed maximum

- `L` is a list of integers
- if the list is bigger then a certain length then
  (a) split the list into smaller sub-lists
  (b) spawn one process for each sub-lists and make this process compute the maximum element of the sublist and send it back
  (c) gather the results in a list
  (d) compute the maximum of that list

b) (Bonus) Using the same approach, write an efficient distributed sort `dsort`. Benchmark the results.