CPDS-Conc Lab 5
Introduction to Concurrent Architectures

Authors: Jorge Castro and Joaquim Gabarro
Comments and suggestions to `gabarro@lsi.upc.edu` with header `CPDS-ConcLab`.

Goal: Recognize frequent concurrent architectures or program scheme patterns. Erlang implementation of several architectures.

Basic material:

- Course slides.
- *Programming Erlang: Software for a Concurrent World*, Joe Armstrong, 2007. Second edition 2013.
- Chapter 5 of the book *Erlang Programming*, Francesco Cesarini and Simon Thompson, O'Reilly, 2009. In particular the *Managing Radio Frequencies* is taken from this book.
- Following some complementary reading. In relation to *client-server* read Section 10.3 of *Concurrency, State Models & Java Programs*, Jeff Magee and Jeff Kramer, Wiley, Second Edition, 2006 (M&K for short). In relation to the *filter pipeline* read Section 11.1 of Chapters 11 of M&K.

Comment: Parts of this lab are not original and are taken directly from the references.

3.1 Training

The term *architecture* is used to mean the process structure of a concurrent program together with the way in which the elements of the program interact. Most concurrent software have program scheme patterns that correspond with those patterns defined by a particular architecture: client server, filter pipeline, supervisor workers, token ring ...

3.1.1 Client-Server

The client-server architecture is a structure consisting of one or more client processes interacting with a single server process. The interaction is bi-directional consisting of a request from a client to the server and a reply from the server to the client. This organization is at the heart of many distributed computing applications.

Erlang processes can be used to implement client/server solutions, where both clients and servers are represented as Erlang processes. A server could be a FIFO queue to a printer, window manager, or file server. The resources it handles could be a database, calendar, or finite list of items such as rooms, books, or radio frequencies. Clients use these resources by sending the server requests to print a file, update a window, book a room, or use a frequency. The server receives the request, handles it, and responds with an acknowledgment and a return value if the request was successful, or with an error if the request did not succeed. When implementing client/server behavior, clients and servers are represented as Erlang processes. Interaction between them takes place through the sending and receiving of messages. We borrow from the book *Erlang Programming* (chapter 5) the following exercise.

Managing Radio Frequencies

Let's walk through a client/server example and test it in the shell. The server will be responsible for managing radio frequencies on behalf of its clients, the mobile phones connected to the network. The phone requests a frequency whenever a call needs to be connected, and releases it once the call has terminated.

- In order to start the server you need to complete the following snippet code. The name of the server (the spawned process) should be `frequency`. This is an Erlang style rule (not mandatory): the name of the server process has to agree with the module name.

```
-module(frequency).
-export([start/0, stop/0, allocate/0, deallocate/1]).
-export([init/0]).

%% These are the start functions used to create and
%% initialize the server.

start() ->
  register(... , spawn(frequency, init, [])).

init() ->
  %%Frequencies is a tuple of two lists
  %%Fist list contains free frequencies
  %%Second list contains tuples of {busy frecuency, user Pid}
  Frequencies = {get_frequencies(), []},
  loop(Frequencies).

% Hard Coded
get_frequencies() -> [10,11,12,13,14,15].
```

- *Message passing is often hidden in functional interfaces.* For instance, when a mobile phone has to set up a connection to another subscriber, it calls the `frequency:allocate()` client

function. This call has the effect of generating a synchronous message which is sent to the server. When the client has completed its phone call and releases the connection, it needs to deallocate the frequency so that other clients can reuse it. It does so by calling the client function `frequency:deallocate(Freq)`. The call results in a message being sent to the server.

```
%%  The client Functions

allocate()       -> call(allocate).
deallocate(Freq) -> call({..., ...}).
stop()           -> call(stop).

%% We hide all message passing and the message
%% protocol in a functional interface.

call(Message) ->
  ... ! {request, self(), ...},
  receive
    {reply, Reply} -> Reply
  end.
```

- Now that we have covered the code to start and interact with the frequency server, let's take a look at its receive-evaluate loop. The main loop call the internal help functions `allocate()` and `deallocate()`

```
%% The Main Loop

loop(Frequencies) ->
  receive
    {request, Pid, allocate} ->
      {NewFrequencies, Reply} = allocate(Frequencies , ...),
      reply(Pid, Reply),
      loop(NewFrequencies);
    {..., ... , {deallocate, ...}} ->
      ... = deallocate(..., ... ),
      ...(Pid, ok),
      ...(NewFrequencies);
    {..., ..., stop} ->
      ...(Pid, ok)
  end.

reply(Pid, Reply) ->
  ... ! {reply, ...}.
```

- We complete this system by implementing the allocation and deallocation functions.

```
%% The Internal Help Functions used to allocate and
%% deallocate frequencies.

allocate({[], Allocated}, _Pid) ->
   {{[], Allocated}, {error, no_frequency}};
allocate({[Freq|Free], Allocated}, Pid) ->
```

```
    {{Free, [{Freq, Pid}|...]}, {ok, Freq}}.

deallocate({Free, Allocated}, Freq) ->
    NewAllocated=lists:keydelete(Freq, 1, ...),
    {[Freq|Free], ...}.
```

- You can see an example of the frequency allocator in action now:

```
1> c(frequency).
{ok,frequency}
2> frequency:start().
true
3> frequency:allocate().
{ok,10}
4> frequency:allocate().
{ok,11}
5> frequency:allocate().
{ok,12}
6> frequency:allocate().
{ok,13}
7> frequency:allocate().
{ok,14}
8> frequency:allocate().
{ok,15}
9> frequency:allocate().
{error,no_frequency}
10> frequency:deallocate(11).
ok
11> frequency:allocate().
{ok,11}
12> frequency:stop().
ok
```

3.1.2 Filter Pipeline

The Filter Pipeline architecture can be thought as the "computation equivalent" of a production line of a factory. In the production line of a car factory there are several agents working concurrently. The first agent process raw material and makes car chassis. Simultaneously, the second agent receives car chassis and put the doors, etc.

In the Filter Pipeline architecture agents are call filters. Each filter has as input a stream of data, makes a computation and outputs a stream of data results that will be the input of the next filter. In order to smooth the transition of data between a filter and the next one a pipe process is used, and often the pipe is implemented as a buffer. In the example below we implement the filters as Erlang processes and pipes are their mailboxes.

Eratosthenes Sieve

To illustrate the use of filter pipelines, we develop a program with this architecture that computes the set of prime numbers up to some given bound N. The program is a concurrent implementation of the Primes Sieve of Eratosthenes. We remind you that we implemented a sequential version of this algorithm in the last Lab session.

Function `erato` below has as input the bound N and generates the list of primes in the range $2 \ldots N$ following the filter pipeline pattern. It starts by spawning the first filter and a generator process. This last process generates as output stream the sequence of numbers in the range $2 \ldots N$ and finishes the stream with an `eos` mark. The generator output is the input stream of the first filter. Elements of the input stream are received by the filters as messages

```
-module(erato).
-export([erato/1]).
-export([filter/1, generator/2]). %reason: to spawn

% N >= 2
erato(N) ->  FirstFilterPid = spawn(erato, filter, [[]]),
             spawn(erato, generator, [N, FirstFilterPid]),
             ok.
```

- Complete the `generator` code below:

```
% N >= 2
generator(N, Pid) -> generator(2, N, Pid).

%N >= 2, K <= N + 1
generator(K, N, Pid) when K == N + 1 -> Pid ! eos;
generator(K, N, Pid) -> ....,
                        .....
```

- Each filter has as a parameter the list of primes already computed and does the following computation. It declares as a prime the first input stream number and subsequently filters out multiples of that value from the input stream. Non ruled out numbers form the output stream of the filter. The last filter in the chain is the filter that receives the `eos` mark as first input element. Complete the following code:

```
% Primes is the list of primes  already generated
filter(Primes) -> receive
                    eos -> io:format("~p~n",[lists:reverse(Primes)]);
                    P   -> NextFilterPid = spawn(erato, filter, [[.....]]),
                           loop(P, NextFilterPid)
                  end.


loop(P, NextFilterPid) -> receive
                            eos                 -> NextFilterPid ! ....;
                            M when M rem P /= 0 -> NextFilterPid ! .....
                                                   ..............
                            _                   -> ..............
                          end.
```

- Finally, try `erato(100000)`.

3.2 Homework

3.2.1 Token Ring Architecture

A "control" process has to offer a user function `go(N,M)` that generates a lists `L` of `M` random integer numbers in the set {`1,2,...,M`}, sets up ring of `N` processes (so-called "workers") and sends a token to the first worker. Token possesion grants the worker possesor permission to eat. When worker `k` receives a token, it sends a message {`self(), eat`} to control and sends the token to next worker.

When control receives a message {`Pid, eat`}, it withdraws the head `H` of the list `L` and appends to a result list the tuple {`Pid, H`}. When list `L` is empty, control sends a stop message to the ring that terminates the workers and prints the result list.

There are two ways to solve this problem:

1. control spawns all the workers in the ring.
2. control spawns only the first worker in the ring. Every new worker in the ring, but the last one, spawns the next worker.

Avoid unnecesary synchronization between processes (it is desired a *concurrent* solution). Pay attention to border cases, for instance `N = 1`, etc..., and be sure that all the processes end in a convenient way. Run several consecutive times your code in order to check everything is ok.

Execution examples:

```
11> control:go(3,7).
[5,4,1,2,2,1,1]
<0.68.0> eats
<0.69.0> eats
<0.70.0> eats
<0.68.0> eats
<0.69.0> eats
<0.70.0> eats
<0.68.0> eats
[{<0.68.0>,1},{<0.70.0>,1},{<0.69.0>,2},{<0.68.0>,2},{<0.70.0>,1},{<0.69.0>,4},{<0.68.0>,5}
stop
12> control:go(1,4).
[4,2,1,1]
<0.72.0> eats
<0.72.0> eats
<0.72.0> eats
<0.72.0> eats
[{<0.72.0>,1},{<0.72.0>,1},{<0.72.0>,2},{<0.72.0>,4}]
stop
13> control:go(4,6).
[1,4,4,1,2,1]
<0.74.0> eats
<0.75.0> eats
```

```
<0.76.0> eats
<0.77.0> eats
<0.74.0> eats
<0.75.0> eats
[{<0.75.0>,1},{<0.74.0>,2},{<0.77.0>,1},{<0.76.0>,4},{<0.75.0>,4},{<0.74.0>,1}]
stop
```

Have fun!