LINFO1104 TP12: Programmation avec Erlang.

Cette séance a pour but de vous familiariser avec le Erlang. Il a été créé pour la concurrence, la tolérance aux pannes et pour construire facilement des applications distribuées. OTP (Open Telecom Platform) est une collection de bibliothèque et de middleware prévus pour Erlang.

Installation d'Erlang

Ubuntu/Debian

Erlang n'est pas inclu dans les packages de bases de la distribution. Pour l'installer, vous devez importer le dépôt d'Erlang.

1. Importer la clé GPG du dépot Erlang:

```
$ wget -O- https://packages.erlang-solutions.com/$(echo $(lsb_release -si) |
awk '{print tolower($0)}')/erlang_solutions.asc | sudo apt-key add -
```

La commande s'écrit sur une seule ligne !

NB: `wget` doit être installé sur votre système. Si ce n'est pas le cas, la commande suivante devrait l'installer: `$ sudo apt-get install wget`

2. Ajout du dépot Erlang:

```
$ echo "deb https://packages.erlang-solutions.com/$(echo $(lsb_release -si) |
awk '{print tolower($0)}') $(lsb_release -cs) contrib" | sudo tee
/etc/apt/sources.list.d/rabbitmq.list
```

La commande s'écrit sur une seule ligne !

3. Installation d'Erlang :

```
$ sudo apt-get update && sudo apt-get -y install erlang
```

Fedora

À partir de Fedora 29, l'installation se fait directement depuis les repos officiels: `$ sudo dnf install erlang`

Arch Linux

Vous savez comment faire ;)

MacOS

Une manière de l'installer utilise Homebrew (`brew`). Si vous ne possédez pas cet utilitaire, installez le via un terminal avec la commande suivante :

```
$ /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

De plus amples informations se trouve sur le site officiel de Homebrew (`https://brew.sh/index_fr`)

Une fois le gestionnaire de paquets installé, la commande suivante installe Erlang : `$ brew install erlang`

Windows

Rendez-vous sur le site officiel d'Erlang (https://www.erlang.org/downloads) et téléchargez le binaire correspondant à l'architecture de votre PC (`Windows {32,64}-bit Binary File`). Installez-le en suivant les instructions affichées à l'écran. Pour utiliser `Erlang` depuis l'invité de commande Windows, vous devez explicitement ajouter le chemin vers les binaires d'`Erlang` dans la variable d'environnement `PATH`. Vous trouverez une vidéo expliquant les démarches à suivre sur cette vidéo YouTube (https://youtu.be/uhsIu-zP7Hs)

Exercices.

1. **Commencer avec Erlang**. Ce premier exercice a pour but de vous familiariser avec la syntaxe d'Erlang. Considérez le premier programme suivant écrit dans le fichier `/tmp/hello.erl`:

```
-module(hello).
-export([hello_world/0]).

hello_world() ->
        io:fwrite("Hello, World!\n").
```

N'oubliez pas d'insérer un point (‘.’) à la fin de **chaque commande**, un peu à la manière du point-virgule pour le `C` ou `Java`.

Le programme doit premièrement être compilé pour que la machine virtuelle Erlang puisse le comprendre. Il y a deux manière:

1. **Le compiler directement via le shell Erlang**. Pour ce faire, rendez-vous dans le dossier où se trouve votre code source. Dans l'exemple, il se trouve dans le dossier `/tmp`.

   ```
   $ cd /tmp
   ```

   Une fois dans le dossier `/tmp`, allumez le shell erlang: `$ erl`. Vous devriez avoir un nouveau shell comme montré ci-dessous:

   ```
   Erlang/OTP 22 [erts-10.6.4] [source] [64-bit] [smp:8:8]
   [ds:8:8:10] [async-threads:1] [hipe]

   Eshell V10.6.4  (abort with ^G)
   1>
   ```

   Pour compiler notre premier module, il faut utiliser la commande intégrée `c` (comme `compile`). Si le fichier `hello.erl` est accepté par le shell, il devrait vous renvoyer `{ok, hello}`, comme le montre l'exemple:

   ```
   1> c(hello).
   {ok, hello}
   2>
   ```

   Pour exécuter la fonction `hello_world` du module `hello` (après compilation), entrez simplement `hello:hello_word().`:

   ```
   2> hello:hello_world().
   Hello, World!
   ok
   ```

   Pour ensuite fermer le shell, vous pouvez utiliser `init:stop()`

   ```
   3> init:stop().
   ok
   4> %
   user@PC /tmp %
   ```

2. **Le compiler sans passer par le shell Erlang**. Rendez vous dans le dossier contenant votre code source `hello.erl`. Ensuite, compilez le avec la commande `$ erl -compile hello`. Un fichier `hello_world.beam` vient d'être créé.

   Finalement, pour exécuter la fonction, vous devrez utiliser la commande suivante:

   ```
   $ erl -noshell -s hello hello_world -s init stop
   ```

   L'argument `-s` spécifie quel fonction de quel module, la VM doit exécuter. Notez que vous pouvez appeler plusieurs fois l'argument `-s`. Dans ce cas, la commande exécutera les fonctions de gauche à droite. De plus, pour terminer correctement la VM, vous devez reseigner à la fin de la commande `-s init stop`. Sinon, le programme ne s'arrêtera pas.

   Si la fonction prend des arguments en paramètre, comme celle-ci:

   ```
   -module(my_math).
   -export([my_add/2, start/0]).

   my_add(A, B) -> A + B.
   ```

   Il n'est pas possible de l'appeler directement depuis un shell bash. Vous devez alors créer une autre fonction, que l'on appelle `start` qui initialise les arguments pour vous:

```
-module(my_math).
-export([my_add/2, start/0]).

my_add(A, B) -> A + B.

start() ->
    io:fwrite("~p~n", [my_add(2, 3)]).
```

Puis de l'appeler comme avant :

```
$ erl -compile my_math
$ erl -noshell -s my_math start -s init stop
```

Toutefois, l'argument `-s` autorise à passer des arguments à une fonction sous certaines conditions. La fonction doit être d'arité 1 (ne prend qu'un seul argument) et son argument doit prendre une liste en paramètre. La commande suivante demande à Erlang d'exécuter une fonction `start` du module `wrapper` avec l'argument `[add, '1', '2']` (liste d'**atomes**) :

```
$ erl -noshell -s wrapper start my_add 1 2 -s init stop
```

Vous êtes en charge d'implementer une librairie qui manipule des entiers. Pour que les utilisateurs puissent utiliser facilement les fonctions que vous avez écrites, il est intéressant de les appeler directement depuis un shell bash, sans devoir utiliser le shell Erlang.

Votre tâche est double. **Tâche 1**, vous allez implémenter `my_fib/1` qui calcule la suite de fibonacci, `my_pow/2`, qui calcule la puissance d'un nombre et `my_add/2` qui calcule l'addition de deux nombres.

Tâche 2, implémentez la fonction `start/1` qui permet d'appeler les fonctions spécifiées en **Tâche 1** comme dans l'exemple du dessus. `start/1` appelle la fonction `my_add/2` en lui donnant ses deux arguments, 1 et 2. Puis, `start/1` affiche le résultat de `my_add/2` sur la sortie standard. Si start ne connaît pas la fonction, il affiche `unk` dans la console.

Astuces: pour transformer un atome en un `Integer`, utilisez les fonctions intégrées suivantes:

```
fun(Y) ->
    list_to_integer(atom_to_list(Y))
end.
```

Aidez-vous de la fonction `apply/3` pour appeler une fonction de votre module depuis la fonction `start`. (`apply(Module, Function, [Arg1, Arg2, ...])` appelle la fonction de la manière suivante: `Module:Function(Arg1, Arg2, ...)`).

La fonction `lists:member/2` pourrait également vous être utile. Rendez-vous sur la documentation d'Erlang (https://erlang.org/doc/man/lists.html).

2. **Erlang et la programmation multi-agent**. Pour créer un agent concurrentiel capable de recevoir des messages, Erlang fournit la fonction `spawn/3`. Cette fonction prend trois arguments:
    1. Le nom du module comprenant la fonction que l'on désire transformer en agent;
    2. La fonction "agent";
    3. Les arguments que l'on passe à la fonction;

    Imaginons la fonction `voiture/0` programmée pour être un agent:

    ```
    voiture() ->
        receive
            start -> io:format("Engine is started~n");
            move -> io:format("I'm moving~n");
            stop -> io:format("Engine is down~n");
            _ -> io:format("Aaah noooo! Kernel Panic~n")
        end.
    ```

    La voiture comprend trois messages: `start`, `move stop`. Elle attend jusqu'au moment où elle reçoit un message. Le morceau de code suivant crée la voiture en tant qu'agent.

    La ligne suivante lui envoie le message `start` grâce à l'opérateur `!`

    ```
    start() ->
        V = spawn(?MODULE, voiture, []),
        V ! start.
    ```

Notez le `?MODULE` qui indique à Erlang que la fonction `voiture` se trouve dans le même module que le code qui l'invoque.

L'implémentation actuelle de la voiture est telle qu'après un message, le processus s'arrête. Comment feriez-vous pour qu'elle puisse encore accepter des messages après qu'on lui aie envoyé un message ?

L'implémentation de la voiture est très basique, essayez de l'augmenter pour qu'elle se souvienne de l'état dans lequel elle se trouve. Si on lui envoie `move`, alors qu'elle n'a pas démarrée, elle ne doit pas afficher à l'écran `"I'm moving"`. Idem si la voiture a déjà démarrée, elle ne doit pas démarrer une nouvelle fois le moteur et ainsi de suite.

En considérant le code suivant, quels sont les messages valides selon l'état dans lequel la voiture devrait être ?

```
run_voiture() ->
    Voiture = spawn(?MODULE, voiture, []), % ou spawn(?MODULE, voiture, [idle])
    Voiture ! move,
    Voiture ! start,
    Voiture ! move,
    Voiture ! start,
    Voiture ! move,
    Voiture ! stop,
    Voiture ! stop.
```

3. **S'échanger des messages entre agents**. Maintenant que vous avez vu comment traiter des messages, il est également utile que deux agents puissent communiquer entre eux.

```
hal9000() ->
    receive
        {Pid, _} -> Pid ! {self(), "I'm sorry Dave, I'm afraid I can't do that"}
    end,
    hal9000().

dave(HalPid) ->
    HalPid ! {self(), "Open the Pod Bay Doors HAL"},
    receive
        {Pid, Msg} ->
            io:format("HAL (~p) says \"~p\"", [Pid, Msg]);
        _ -> ok
    end.

hal_dave_conversation() ->
    Hal = spawn(?MODULE, hal9000, []),
    dave(Hal).
```

Dans ce petit programme, `dave` demande à `hal` de lui ouvrir les portes. Dave s'attend qu'il lui réponde. Pour que Hal puisse renvoyer sa réponse, il faut que Dave lui donne son identifiant de processus. C'est que que Dave fait en lui envoyant un tuple ayant comme premier élément, son Pid (Process Identifier). Erlang peut lui fournir grâce à la fonction `self/0`. Le second élément du tuple est alors le message que Dave veut lui transmettre.

C'est seulement en envoyant le Pid, que deux agents peuvent établir un cannal de communication isolé des autres processus.

Maintenant que vous savez établir un canal de communication entre deux agents, ré-implémentez la fonction `my_pow/2` de l'exercice 1 pour qu'il accepte des messages d'autres agents lui demandant de calculer une puissance. Lorsque votre agent `my_pow/0` (**pourquoi l'arité vaut zero maintenant ?**) a trouvé la réponse, il envoie le résultat du calcul au processus appelant. Servez vous de l'agent `my_pow/0` pour calculer:

$$\sum_{x=1}^{100} x^2 = \text{(spoiler) } 338\,350$$

4. **Le retour de Charlotte**. Souvenez vous de l'exercice 3 de la séance 10. Vous étiez en charge d'écrire Charlotte, qui réalise un étude sur la consommation de bière. Nous allons modifier l'exercice pour ajouter

un nouvel agent qui s'occupe de stocker les informations recueillies par Charlotte, comme le montre la figure du dessous.

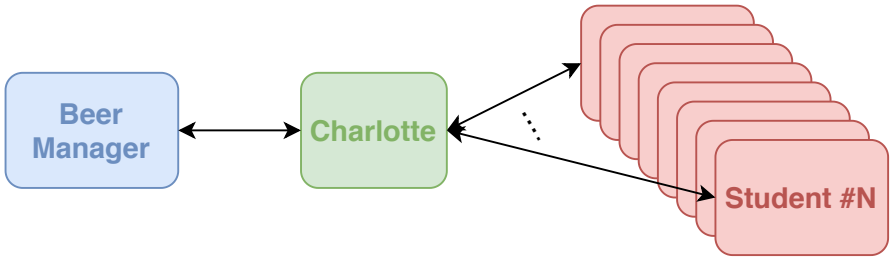

Figure 1: Charlotte

Charlotte est en charge de communiquer avec les étudiants pour leur demander leur consommation de bières. Quand Charlotte reçoit la réponse d'un étudiant, elle le communique au manager qui sauvegarde le résultat dans son état interne.

Lorsque Charlotte a fini d'interroger les étudiants, elle demande au manager de générer les statistiques (et de lui renvoyer le résultat), pour qu'elle puisse continuer son étude.

Votre mission, si vous l'acceptez, est d'implémenter les différents agents décrit sur la Figure 1:

- **Beer Manager**: il accepte les messages suivants :
  - `{Pid, stats}`: le processus `Pid` demande à **Beer Manager** de lui renvoyer les statistiques de consommation de bière grace à l'atome `stats`. Dans le cas de l'exercice, `Pid` sera uniquement Charlotte.
  - `{Pid, BeerNb}`: reçoit de `Pid` (Charlotte) une nouvelle consommation de bière.
  - `shutdown`: lorsque l'atome **shutdown** est reçu, **Beer Manager** termine son execution.
- **Charlotte**: doit implémenter les messages suivants :
  - `{Pid, NbBeer}`: reçoit d'un étudiant `Pid` sa consommation de bière.
  - Si Charlotte reçoit un autre message, elle renvoie `"error"`
- **Student**: génère un nombre aléatoire de bière lorsqu'on lui envoie le message `{Pid, how_much_beers}`. Ici, `Pid` sera Charlotte qui lui enverra l'atome `how_much_beer`.

Le code qui suit vous aidera pour tester votre implémentation. Les `TODO` doivent être remplacés par vous:

```
% Mise à jour de l'état interne du manager
update_stats({Nb, Min, Max, NbStud}, NbBeer) ->
    { Nb + NbBeer,
      if Min == -1 -> NbBeer;
         Min > NbBeer -> NbBeer;
         true -> Min
      end,
      if Max < NbBeer -> NbBeer;
         true -> Max
      end,
      NbStud + 1}.

% Génère les statistiques pour un message "stats"
do_stats({Nb, Min, Max, NbStud}) ->
    {Nb,
     (Nb * 1.0) / NbStud,
     Min, Max, NbStud}.

beer_manager({Nb, Min, Max, NbStud}) -> % TODO.

charlotte([], BeerManager) -> % TODO;
charlotte([_Head | _Tail], BeerManager) -> % TODO.

student() -> % TODO rand:uniform(N) génère un entier aléatoire dans l'intervalle [1; N].

build_students(Tot) ->
```

```
    [spawn(?MODULE, student, []) || _ <- lists:seq(1, Tot)].

start() ->
    BM = spawn(?MODULE, beer_manager, [{0,-1,0,0}]),
    Students = build_students(98956),
    Result = charlotte(Students, BM),
    BM ! shutdown,
    case Result of
        {Nb, Avg, Min,Max, NbStuds} ->
            io:format("Total Beers: ~p~nAverage: ~p~nMin: "
                      "~p~nMax: ~p~nStudents: ~p~n",
                      [Nb, Avg,Min,Max,NbStuds]),
            Result;
        _ -> "error"
    end.
```

5. **Gestions des erreurs dans les processus**. Pour certaines raisons, il est possible qu'un processus meurt. Si le résultat de ce-dernier était attendu par un autre processus, le programme ne pourra pas continuer son exécution correctement.

   Imaginons le programme suivant :

```
-module(hello).
-export([start/0, divisor_checker/1]).

divisor_request(Pid, Nb, Divisor) ->
    Pid ! {self(), Nb, Divisor},
    receive
        {Pid, Response} -> Response
    after 2000 ->
        timeout
    end.

random_kill(P) ->
    case rand:uniform() < P of
        true -> exit(killed);
        _ -> ok
    end.

divisor_checker(KillProb) when KillProb < 1.0 ->
    ok = random_kill(KillProb),
    receive
        {Pid, Nb, Divisor} ->
            Res = case Nb rem Divisor of
                      0 -> true;
                      _ -> false
                  end,
            Pid ! {self(), Res}
    end,
    divisor_checker(KillProb).

start() ->
    Pid = spawn(?MODULE, divisor_checker, [0.35]),
    % Checks the divisor of two from 1 to 10
    Lst = [ divisor_request(Pid, X, 2) || X <- lists:seq(1,10)],
    io:format("~p~n", [Lst]).
```

   Le processus qui regarde si un nombre `Nb` est divisible par `Divisor` a une probabilité `KillProb` de se faire tuer pour chaque appel à `divisor_checker/1`.

   Un exemple d'output, lorsque `start/0` est lancé peut être celui-ci:

```
1> c(hello).
2> hello:start().
```

```
[false,true,false,timeout,timeout,timeout,timeout,timeout,timeout,timeout]
ok
```

À votre avis, quel est le temps minimal que l'on a attendu avant d'avoir la réponse à l'écran, pour l'output produit ci-dessus ?

Afin d'éviter d'obtenir des timeouts, parce que le processus a planté, il serait intéressant de le redémarrer automatiquement. Il y a plusieurs concepts à prendre en compte:

1. **Les liens**: la fonction `link/1` permet d'établir un lien entre plusieurs processus. Si un processus de la chaine meurt, tous les autres meurt également. La mort se propage à travers tous les processus, jusqu'à celui du début de la chaine. Si rien n'est fait, alors il est impossible de redémarrer les processus défaillants. Avant de mourir, Erlang envoie un signal caché au processus lui indiquant qu'il s'apprête à mourir. Ce signal fonctionne comme du message passing, sauf qu'ils ne sont pas transférés au processus s'il n'est pas explicitement capturé. La fonction `process_flag(trap_exit, true)` permet de récupérer ce signal qui se traduit par un message du type `{'EXIT', Pid, Reason}`. Grâce à cette fonction, le processus peut choisir si oui ou non il meurt.

   Grâce aux liens et la capture du signal de mort, il est possible de créer un processur qui détecte si `divisor_checker/1` est en vie ou non.

   La fonction suivante permet de redémarrer `divisor_checker/1` s'il est mort :

   ```
   start_divisor_checker(KillProb) ->
       spawn(?MODULE, restarter, [KillProb]).

   restarter(P) ->
       process_flag(trap_exit, true),
       Pid = spawn(?MODULE, divisor_checker, [P]),
       link(Pid),
       receive
           {'EXIT', Pid, normal} -> % divisor_checker has termined correctly
               ok;
           {'EXIT', Pid, shutdown} -> % not a crash, manual temrination
               ok;
           {'EXIT', Pid, _} -> % this is a crash
               restarter(P)
       end.
   ```

   Notez que dans ce code, la création du processus et le lien se fait en deux instructions. Cependant, il est possible que le processus crash avant d'exécuter le `link(Pid)`. Préférez donc la fonction `spawn_link/3` (s'utilisant comme `spawn/3`) pour lancer et lier le processus de manière atomique.

   Il y a encore un problème à régler, quand un processus crash, `restarter/1` le relance. Mais, le Pid associé à `divisor_checker/1` a changé aussi ! Donc `divisor_request/3` ne fonctionne plus du tout! Il faut alors nommer le processus `divisor_checker/1`

2. **Les processus nommés**. Un processus peut être nommé à atome. Ainsi, il n'y a plus besoin de connaitre le Pid du processus pour lui envoyer un message. Grâce à la fonction `register/2`, on peut lier un atome à un Pid.

   ```
   Pid = spawn(?MODULE, my_agent, []).
   register(running_agent, Pid).
   ```

   Pour récupérer le Pid lié à un atome, la fonction `whereis/1` peut être utilisée :

   ```
   Pid = whereis(running_agent).
   ```

   Finalement, pour dissocier un atome à un Pid, la fonction `unregister/1` permet de le faire manuellement. Si un processus meurt, l'atome est **automatiquement** dissocié du processus.

   ```
   unregister(running_agent).
   ```

   La fonction `divisor_request/3` devient maintenant `divisor_request/2`:

   ```
   divisor_request(Nb, Divisor) ->
       % assume div_checker is the atom bound to divisor_checker function
       div_checker ! {self(), Nb, Divisor},
       Pid = whereis(div_checker),
   ```

```
receive
    {Pid, Response} -> Response
after 2000 ->
    timeout
end.
```

Utiliser `whereis/1` pour connaître le Pid sur lequel il faut envoyer le message peut être dangereux. Si le processus lié à l'atome crash juste après qu'un autre processus appelle `whereis/1`, alors `divisor_request/3` s'attend à recevoir un message d'un ancien Pid de `divisor_checker/1`. À la place, on uilise des références (`make_ref/0`) pour associer un message à un identifiant unique. Le processus envoie en plus de son Pid, la référence unique créée. Ainsi, quand l'autre processus renvoie sa réponse, il renvoie la référence en plus.

3. **Les moniteurs**. Un concept semblable au lien qui permet à un processus de surveiller un autre. Contrairement aux liens, le processus invoquant le moniteur n'est pas tué, mais simplement notifié du statut du processus surveillé via un message. Le moniteur est unidirectionel, seul le processus qui crée le moniteur est informé. Les liens sont bidirectionnel, si le lien casse entre les deux processus, les deux processus liés tombent ensemble. Plus d'informations dans la documentation d'Erlang (https://erlang.org/doc/reference_manual/processes.html#monitors).

Modifiez les fonctions définies plus haut pour que `divisor_request/3` puisse communiquer avec `divisor_checker/1` même après un redémarrage.

Votre programme doit réussir à passer ces instructions :

```
busy_wait(Atom) ->
    case whereis(Atom) of
        undefined ->
            timer:sleep(500),
            busy_wait(Atom);
        _ -> ok
    end.

must_pass() ->
    start_divisor_checker(0.35),
    % check divisor_checker has been correctly restarted
    ok = busy_wait(div_checker),
    exit(whereis(div_checker), must_restart),
    ok = busy_wait(div_checker),
    divisor_request(4, 2),
    ok = busy_wait(div_checker),
    divisor_request(3, 2).
```

6. **Nombre Premier** Réalisez un programme qui permet de trouver les nombres premiers de 1 à N.

   Implémentez une fonction `prime_range/2` qui permet de trouver les nombres premiers dans un range particulier [X;Y[.

   Utilisez la fonction précédente pour contruire la fonction `get_prime` qui crée P processus `prime_range` pour trouver tous les nombres premiers entre 1 et N. Les P processus devront se partager de manière égale les sous-ranges de 1 à N. Par exemple, si P vaut 2, le premier processus gère le sous-range [1; N/2[et le second [N/2; N+1[.

   Après avoir calculé si un nombre est premier, chaque processus P a une probabilité `p` de se faire tuer. S'il est mort, vous devez gérer son redémarrage.