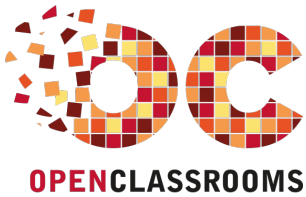
Avant-goût du langage : une todo-list distribuée

Par bluestorm
et rks`

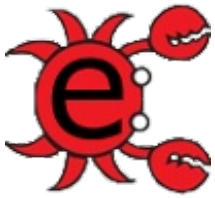


www.openclassrooms.com

Sommaire

Sommaire 2
Avant-goût du langage : une todo-list distribuée 3
Notes 3
Le langage Erlang 3
Le problème du parallélisme 3
L'histoire d'Erlang 4
Les concepts fondamentaux 4
Un exemple de programme 4
Et le client ? 7
Partager 7

Avant-goût du langage : une todo-list distribuée

Par  bluestorm et  rks`

Mise à jour : 01/01/1970

Ce tuto est un peu inhabituel car, au lieu de vous apprendre un langage de programmation, en partant des bases et en exposant toutes ses caractéristiques, il vise à vous faire découvrir un langage. Il est donc conçu pour être court, et incomplet : c'est un *avant-goût*.

Après une présentation succincte du langage et des problématiques qu'il aborde, vous serez confrontés à un exemple de code, qui met en valeur ses spécificités. Le but n'est pas d'apprendre la syntaxe, qui ne sera quasiment pas présentée (un minimum quand même, pour que vous puissiez lire le code) mais plutôt d'avoir un aperçu des *concepts* du langage. C'est à vous, programmeurs, de faire le travail d'adaptation ; espérons que cela vous donnera peut-être envie d'en savoir plus sur ce langage surprenant !

Notes

Ce tuto a été rédigé collaborativement ; inspiré d'une news de Poulet, il a été rédigé par Dark-Side et bluestorm.

Ce tuto a été écrit, non pas en zCode, mais en **mdown**, un petit langage de mise en forme agréable à utiliser (et qui peut produire du zCode), conçu par rz0; Vous pourrez trouver plus d'informations sur cette page.
Sommaire du tutoriel :

- Le langage Erlang
- Un exemple de programme

Le langage Erlang

Le problème du parallélisme

À l'heure actuelle, les possibilités d'augmentation des performances des processeurs sont assez limitées, et les fabricants ont donc décidé de se tourner vers une méthode différente d'augmentation des performances : la multiplication des processeurs. Le problème, c'est que les principaux programmes actuels sont écrits pour tourner sur un seul processeur à la fois, et souvent ne savent donc pas exploiter cette possibilité.

Malheureusement, il est beaucoup plus difficile de concevoir des applications tournant sur plusieurs processeurs (ou *threads*, ou plus généralement "parties") en même temps : la difficulté réside principalement dans l'interaction entre les différentes parties du programme. Si par exemple deux parties ont accès à une même variable, il y a un risque que l'une modifie cette variable pendant que l'autre en avait besoin (imaginez qu'entre la vérification du mot de passe et son stockage (chiffré, évidemment) dans la BDD, une autre partie de votre site web modifie cette variable pour y stocker le contenu d'un message !). Pour éviter ces problèmes on peut mettre en place un système de "verrous", qui bloquent une variable en empêchant la modification par les autres parties. On rencontre alors des problèmes encore plus complexes, où plusieurs parties se disputent le contrôle des variables dont elles ont besoin sans qu'aucune ne puisse avancer, jusqu'à bloquer complètement tout le programme.

Ce n'est pas un problème lié à la compétence des programmeurs (on dit souvent par exemple que les problèmes de sécurité en PHP sont en partie liés au niveau des webmasters) : même pour un excellent programmeur, il est très difficile, voir impossible, de programmer une application concurrente (dont différentes parties s'exécutent simultanément en interagissant) complexe sans erreurs, et ces erreurs sont très difficiles à corriger. C'est un problème d'*outils* : les langages de programmation principaux ne sont pas adaptés.

Les programmeurs, et surtout les concepteurs de langages de programmation, se sont donc mis à la recherche de manières

différentes d'aborder le problème. Les idées qu'ils ont alors tenté d'exploiter ne sont pour la plupart pas *nouvelles*, mais plutôt *retrouvées* : elles viennent de langages moins utilisés par la majorité des programmeurs, conçus par des "scientifiques", principalement des universitaires, qui mettent en place des langages basés sur un support théorique solide, avec des approches parfois radicalement différentes des langages "grand public".

Erlang, le langage que nous allons présenter ici, est un de ces exemples de langages créés par une rencontre de la théorie (programmation logique, et programmation fonctionnelle) et de la pratique (une équipe d'ingénieurs avec un problème précis : la télécommunication).

L'histoire d'Erlang

Le langage Erlang est né du besoin d'un grand opérateur téléphonique, Ericsson, d'un langage adapté à la programmation de systèmes complexes : certaines parties de l'infrastructure d'un opérateur gèrent en continu (24 heures sur 24, 7 jours sur 7, et une panne est très grave) un grand nombre de connexions simultanées.

Les recherches qui ont finalement donné naissance au langage Erlang ont commencé en 1981. Il s'agissait à l'époque de versions de Prolog, un langage de programmation logique, spécialisées dans la programmation concurrente (on appelle *concurrente* la programmation tenant compte de la communication et du partage des ressources entre différentes parties d'un programme qui s'exécutent en même temps (ou dans un ordre indéterminé)). Petit à petit, les ingénieurs d'Ericsson ont fait naître un nouveau langage, qui reprenait un grand nombre d'idées de Prolog, mais pas seulement (on observe par exemple une grande influence des langages fonctionnels). Erlang est longtemps resté interne à Ericsson, qui s'est mis ensuite à le vendre à des clients spécialisés.

En 1998, Ericsson a décidé de se recentrer sur l'*utilisation* de langages de programmation (Erlang était alors devenu un des points forts de l'entreprise, avec la mise en place de projets critiques dans ce langage), plutôt que sur l'innovation en matière de langages. L'équipe qui avait créé Erlang a donc quitté Ericsson, qui a alors décidé de rendre Erlang Open Source : l'implémentation du langage, ainsi que les bibliothèques l'accompagnant, ont été rendues disponibles à tous.

Erlang est spécialisé dans la concurrence, mais plus particulièrement la *gestion des erreurs* : un programme Erlang peut fonctionner dans un environnement très hétérogène (plusieurs machines connectées par une liaison peu fiable, par exemple), et doit savoir gérer les erreurs ou les problèmes de communication. Tout est mis en place pour qu'un programme puisse surmonter chaque erreur (au lieu de s'arrêter tout simplement), et même se réparer lui-même : il est possible de modifier un programme Erlang en direct, pendant son fonctionnement, sans que les utilisateurs observent de discontinuité.

Toutes ces qualités d'Erlang sont principalement dues à un modèle particulier de communication entre les différentes parties du programme, par "envoi de message". C'est cette idée fondamentale que nous allons présenter par la suite, avec une mise en application.

Les concepts fondamentaux

Les processus

Un programme Erlang est constitué, pendant son exécution, d'un ensemble de *processus* parallèles (qui s'exécutent en même temps, par exemple sur des processeurs différents). La communication entre ces processus s'effectue par transmission de *messages* : chaque processus peut envoyer des messages à un autre, qui peut les recevoir et effectuer des actions en conséquence.

Le nombre des processus et leurs relations dépendent du programme : on peut mettre en place une multitude de "schémas" différents, avec par exemple un simple canal de communication (deux processus qui échangent des messages), un modèle client-serveur (un serveur central qui reçoit des messages de tous les clients, qui ne communiquent pas directement entre eux), ou même une architecture *pair à pair* (les clients se parlent entre eux). Le langage lui-même n'impose aucun choix à ce sujet, car la méthode utilisée (le *passage de message*) est très flexible.

Communication asynchrone

Chaque processus envoie et reçoit des messages. Il existe différents modèles de messages, celui choisi par Erlang est asynchrone : quand un processus A envoie un message à un processus B, il ne "se passe rien" : le processus B n'est pas perturbé dans son fonctionnement.

Le message est en effet stocké dans un espace spécifique au processus B, une sorte de "boîte aux lettres", où il attend. Périodiquement, le processus B peut consulter le contenu de sa boîte aux lettres (plus il le fait souvent, plus vite il sera au courant des messages qu'on lui envoie).
Cette méthode "détendue" de communication, par sa grande souplesse, permet d'éviter certains bugs liés aux problèmes de synchronisation.

Un exemple de programme

À titre d'exemple nous allons mettre en place une todo-list (liste de tâches). Une todo-list se présente comme une liste de tâches à effectuer ("acheter des tomates", "prendre une douche", "manger"...), à laquelle on peut ajouter des tâches (quand le devoir nous appelle) ou en retirer (quand on les a effectuées).
Notre todo-list a la particularité d'être *multi-utilisateurs*, c'est-à-dire qu'elle est accessible par plusieurs personnes 'en même temps' : tout le monde peut y ajouter ou en enlever des tâches, ou consulter la liste.
On peut aussi imaginer des modèles plus précis (par exemple dans une école : les professeurs ne font qu'ajouter des tâches, et vous, vous devez les retirer), mais celui-là est suffisamment simple pour mettre en oeuvre la plupart des *outils de base* du langage, tout en restant compréhensible.

Voici le code :

Code : Erlang

```
-module(todo_list).
-export([start/0, loop/1]).

loop(Liste) ->
    receive
        {add, X} -> loop([X|Liste]);
        {del, X} -> loop([Y || Y <- Liste, Y =/= X]);
        {From, show} -> From ! Liste,
                        loop(Liste);
        close -> io:format("Fin de la connexion~n")
    end.

start() ->
    io:format("Création d'un processus~n"),
    spawn(todo_list, loop, [[]]).
```

Les deux premières lignes servent en fait à *déclarer* notre programme. Par cette dénomination pour le moins surprenante, je veux signifier que ces deux lignes vont nous permettre de réutiliser le code qui va être écrit ensuite.

Code : Erlang

```
-module(todo_list).
-export([start/0, loop/1]).
```

Ces deux directives ne sont pas intéressantes pour une première approche : elles s'occupent de la portée des variables et de l'interaction de ce fichier avec le reste du code, un peu comme l'inclusion de `.h` en C, et les déclarations de portée (publique / privée).

Comme cela a été expliqué dans la première partie, de multiples schémas de communication sont possibles. Ici, nous allons mettre en oeuvre une architecture *client-serveur* (simplifiée). Plus exactement, ce code ne contient que le comportement du serveur (fonction loop), c'est-à-dire la partie qui gère l'accès et la modification par tous les utilisateurs (les "clients") de la liste ; les clients sont très simples, et on peut passer directement par la console Erlang pour cela.

Le serveur fonctionne d'une manière assez spécifique aux programmes Erlang, que l'on peut décrire de la manière suivante : on donne la liste à un "employé", on lui dit "garde-là tant que tu ne reçois pas de message". S'il reçoit un message, différents cas se présentent, selon le contenu du message ; on gèrera ici trois types de messages : "ajouter la tâche machin", "retirer la tâche bidule", "montrer la liste à la personne truc". Il agit en conséquence, et la partie spécifique se déroule à ce moment-là : au lieu de *modifier* la liste des tâches, il donne une *autre* liste à un nouvel employé, qui est alors chargé de répéter le processus. Par exemple, si le message était "ajoute la tâche 'manger'", il va donner à un autre employé sa liste, ainsi que le message "manger" (ce qui constitue donc une nouvelle liste plus grande), et c'est ce nouvel employé qui s'occupera des messages suivants.
C'est une mise en oeuvre particulière de la *récursivité*, un concept décrit dans un des tutos du SDZ.

Voyons maintenant le code. La structure `receive .. end` permet d'examiner les messages, et d'agir en fonction de leur contenu. Si on prévoit de recevoir deux types de messages différents, le code aura cette tête-là (où `expression` désigne un bout de code qui renvoie une valeur) :

Code : Erlang

```
    receive
        premier_type -> expression;
        deuxieme_type -> expression
    end
```

Code : Erlang

```
loop(Liste) ->
    receive
        {add, X} -> loop([X|Liste]);
        {del, X} -> loop([Y || Y <- Liste, Y =/= X]);
        {From, show} -> From ! Liste,
                        loop(Liste);
        close -> io:format("Fin de la connexion~n")
    end.
```

On peut remarquer que le point-virgule (;) ne sert pas à séparer les instructions, mais à séparer les différents cas possibles. Pour exécuter deux instructions à la suite, on utilise une simple virgule.

Ici, on reçoit quatre types de messages :

- l'ajout d'un élément à une liste ;
- la suppression d'un élément de la liste ;
- la demande d'affichage de la liste à quelqu'un ;
- un message de fin, en cas de fermeture de l'application.

Le message de fermeture est "simple" : c'est le message `close` : quand on le reçoit, on envoie un message de fermeture, et on s'arrête. Les autres messages sont un peu plus délicats parce qu'ils contiennent de l'information : quand on envoie le message "ajoute à la liste", il faut préciser l'élément à ajouter : il est contenu dans le message. De même, le message "montre la liste à machin" doit contenir l'adresse de machin, pour que le serveur puisse lui envoyer la liste. Pour faire cela, on utilise des messages en plusieurs parties : `{..., ...}` est un message en deux parties. Certaines parties sont fixes (par exemple `add` et `del`) : on appelle ça des *atomes*, et on peut voir cela un peu comme des constantes définies par le programmeur. D'autres parties sont variables : le `X` dans les deux premiers messages est une variable qui contient la valeur donnée (et dépend donc du message reçu). Les parties fixes sont en minuscules, et les parties variables commencent par une majuscule.

Le message d'ajout fonctionne simplement : si l'on reçoit `{add, X}` (on sait que c'est un message d'ajout grâce à la présence de l'atome `add`), on rappelle `loop` avec la nouvelle valeur `[X|Liste]`, c'est-à-dire une liste qui contient tous les éléments de la liste initiale, plus le contenu de la variable `X`. (c'est là qu'on doit imaginer que l'on donne cette nouvelle liste à un nouvel employé).

La nouvelle liste donnée en cas de message de suppression est un peu particulière : la syntaxe `[Y || Y <- Liste, truc(Y)]` signifie "tous les éléments Y de la liste, qui vérifient 'truc'". Ici, on sélectionne tous les éléments de la liste qui sont *différents* de `X` : à la fin, on a donc la liste, sauf la valeur de `X`, qui a donc bien été supprimée. C'est ce qu'on appelle une *compréhension de liste* (expression maladroite venant de l'anglais "list comprehension").

Enfin, le message "montrer la liste à machin" met en oeuvre une deuxième structure essentielle à la communication inter-processus en Erlang, l'envoi de messages : `!`. La syntaxe est `Pid ! Msg`, et cela envoie le message `Msg` au processus dont l'adresse est `Pid`. Ici, cette adresse a été donnée dans le message, c'est la valeur `From` (vous pouvez remarquer que contrairement aux deux premiers messages, la partie variable a été placée en premier : c'est la convention quand on envoie son adresse dans un message). La valeur que l'on envoie est `Liste` : on envoie bien le contenu de la liste de tâches au processus dont l'adresse est `From`. Ensuite (après la virgule) on rappelle `loop(Liste)` : le serveur continue à tourner, avec la même liste.

Voici enfin la dernière fonction du programme, qui joue un peu le rôle du "main" en C : c'est la fonction de départ, qui est appelée au lancement du programme.

Code : Erlang

```
start() ->
        io:format("Création d'un processus~n"),
        spawn(todo_list, loop, [[]]).
```

La fonction `start` contient deux instructions séparées par une virgule (Erlang utilise le point-virgule pour dénoter un autre type de séparation, c'est donc la virgule que l'on utilise pour séparer deux instructions ; les fonctions sont séparées par des

points). La première instruction, `io:format`, affiche du texte sur la sortie standard.
La deuxième instruction est plus intéressante : ll s'agit de la dernière des trois structures principales de gestion de la concurrence en Erlang : c'est la fonction `spawn`, qui lance un nouveau processus, et renvoie un identifiant le concernant.
Les arguments contiennent le module à utiliser (ici `todo_list`), le nom de la fonction à appeler (`loop`), et enfin une liste d'arguments à donner à cette fonction : avec `[[]]`, on donne un seul argument qui est `[]`, la liste vide : au départ, notre todo-list sera vide.

Et le client ?

Le client ne présente que peu d'intérêt : il suffit d'envoyer au serveur les bons messages, et cela marche tout seul.

Pour une mise en oeuvre rapide de cette todo-list, on peut utiliser la console Erlang. C'est un environnement interactif (un peu comme la ligne de commande sous GNU/Linux) qui permet de manipuler des modules Erlang de manière simple, pour faire des tests par exemple.

Le résultat se présente ainsi : les lignes qui commencent par un nombre suivi de > sont les lignes de code que l'utilisateur a entrées. Les lignes qui les suivent sont les résultats renvoyés par la console. Ici, l'utilisateur manipule notre module `todo_list`, en envoyant une tâche au serveur, avant de récupérer la liste des tâches. Les phrases après `%%` sont des commentaires : elle servent d'explications mais ne sont pas lues par l'interpréteur.

Code : Erlang

```
1> c(todo_list). %% cette commande sert à compiler le module
{ok,todo_list}
2> Serv = todo_list:start(). %% on initialise la variable Serv avec
le pid (l'adresse)
Creation d'un processus      %% du processus créé dans start (le
serveur)
<0.38.0>
3> Serv ! {add, "faire mes devoirs"}. %% on ajoute un élément à
notre todo-list
{add,"faire mes devoirs"}
4> Serv ! {self(), show}. %% self() permet d'obtenir le pid du
processus courant,
{show, <0.31.0>}          %% nécessaire pour que le serveur puisse
répondre
5> receive Liste -> Liste end. %% reçoit la réponse du serveur et
on l'affiche
["faire mes devoirs"]
6> Serv ! close. %% on ferme la connexion
Fin de la connexion
close
7>
```

Voilà une brève présentation du langage. J'espère que vous comprenez à peu près comment fonctionne la communication entre processus.

Certains se demanderont peut-être ce qu'apporte Erlang par rapport à un langage généraliste comme le C dans ce cas précis. L'exemple est peut-être un peu trop simple pour exposer véritablement les avantages de cette méthode, mais on peut déjà constater que cette todo-list est potentiellement accessible de n'importe où (même à travers le réseau, si on met en place le client Erlang correspondant), et ce sans surcoût, alors que l'ajout de cette fonctionnalité demanderait dans un autre langage un effort important. C'est là la grande force de l'Erlang.

Quoi qu'il en soit, nous espérons vous avoir donné envie de découvrir un peu plus profondément l'Erlang. Une simple pré-connaissance de la diversité des langages de programmation vous servira sûrement, même si vous ne vous lancez pas immédiatement dans un nouveau langage, mais si par hasard c'était votre souhait, vous pouvez aller consulter le site web dédié au langage.
Si vous voulez directement un cours complet, et que lire en anglais ne vous gêne pas (trop), vous pouvez essayer ce livre (en ligne).

Ce tutoriel a été corrigé par les zCorrecteurs.