Communication par Messages

Go vs Erlang vs Akka

Sébastien Golouboff Bartosz Tulisz

Table des matières

1 Passage de messages

En informatique, le passage de messages est une technique de communication permettant aux composantes d'un programme ou d'un système informatique de communiquer entre elles sans connaître le fonctionnement interne de leurs pairs. Dans ce modèle, les composantes communiquent entre elles en s'envoyant des messages, et réagissent en fonction des messages reçus.

Nous nous intéressons en particulier au passage de message dans les programmes concurrents, en prenant comme exemple les langages Go et Erlang, ainsi que la librairie Akka.

2 Go

2.1 Communicating sequential processes

Communicating Sequential Processes est un langage formel conçu en 1978 par Tony Hoare pour la spécification et la vérification des systèmes concurrents. Ce langage permet de spécifier un nombre fini de processus exécutant en parallèle une séquence d'instructions et communiquant entre eux en passant des messages de manière strictement synchrone via des canaux, c'est-à-dire que la lecture et l'écriture de de données dans ces canaux sont des opérations bloquantes.

2.2 Présentation du langage

Go est un langage conçu par Google en 2007. Inspiré du C, il est conçu pour améliorer certains de ses aspects gênants, comme la gestion de la mémoire (intégration d'un Garbage Collector), la lisibilité du code ou encore la difficulté d'apprentissage du langage.

Le modèle de concurrence en Go est inspiré de CSP et repose principalement sur deux concepts : les goroutines et les canaux.

2.3 Les goroutines

En Go, il est possible d'exécuter une fonction en parallèle en utilisant le mot-clé **go**. Contrairement à un appel de fonction classique, un appel de fonction précédé de **go** va immédiatement créer un nouveau fil d'exécution traitant l'appel sans bloquer le fil appelant.

Comme les goroutines partagent un unique espace mémoire, elles peuvent communiquer entre elles et coordonner leur exécution en utilisant des variables partagées, notamment des structures de données adaptées à la programmation concurrente.

```
func f(c chan<- string) {
        time.Sleep(2 * time.Second)
        c <- "Hello, World!"
}

func main() {
        c := make(chan string)
        go f(c)
        msg := <-c
        fmt.Printf("received message: %v\n", msg)
}
```

Exemple de code asynchrone en Go

2.4 Les canaux

Pour se passer des messages entre elles, les goroutines peuvent utiliser des canaux. Ceux-ci disposent d'un tampon de taille fixe et de deux opérations bloquantes : la lecture et l'écriture de données. La taille du tampon et le type de données passés au canal sont fixés à l'initialisation de celui-ci. La lecture et l'écriture sont des opérations synchrones, c'est-à-dire que deux appels parallèles à la même opération seront exécutés séquentiellement.

3 Erlang

3.1 Modèle acteur

Le Modèle acteur est un procédé de communication par messages développé par Carl Hewitt en 1973 et repose sur des entités informatiques appellées acteur.Les acteurs prennent une suite de décisions à partir des messages qu'ils reçoivent et sont capables de réaliser les actions suivantes :

- Envoyer un nombre fini de messages à d'autres acteurs.
- Créer un nombre fini d'acteurs.
- Changer de comportement c'est-à-dire les actions réalisées suite à la réception de messages.

3.2 Présentation du langage

Le langage Erlang a été conçu en 1987 au sein de l'entreprise Ericsson pour programmer des appareils de télécommunication et est très adapté à la programmation concurrente et distribuée, il possède des fonctionnalités de tolérance aux pannes et de mise à jour du code à chaud, permettant le développement d'applications à très haute disponibilité. Il est compilé vers du code octets s'exécutant sur la machine virtuelle BEAM.

3.3 Les Processus

En Erlang, les acteurs sont représentés par des processus communiquant entre eux de manières asynchrones via des signaux (ATTENTION ! Il ne s'agit pas de processus et de signaux UNIX !). Les processus sont des unités d'exécution identifiées par un PID, disposant chacune de son propre espace mémoire de taille dynamique. Un groupe de fils d'exécution dédiés se charge de l'ordonnancement.

3.4 Les Signaux

Les signaux sont utilisés par les processus pour communiquer entre eux, et servent notamment à la gestion de l'exécution de processus (par exemple pour tuer un processus ou obtenir des informations sur lui). Les messages sont un type de signal permettant de transmettre un enregistrement commençant par un atome et suivi par une suite de valeurs quelconques.

Contrairement à Go et au CSP, la communication par signaux est totalement asynchrone. En effet, deux processus peuvent s'envoyer un message ou consulter les messages reçus en même temps sans que cela n'affecte leur exécution (pas de blocage). On peut voir un processus erlang comme un processus CSP disposant de son propre canal asynchrone faisant office de file de messages.

```
ping(N, Pong_PID) ->
  Pong_PID ! {ping, self()},
  receive
    pong -> io:format("Ping␣received␣pong~n")
  end,
  ping(N - 1, Pong_PID).

pong() ->
  receive
    fin -> io:format("pong␣finished~n");
    {ping, Ping_PID} ->
      io:format("Pong␣received␣ping~n"),
      Ping_PID ! pong,
      pong()
  end.
```

Exemple de code asynchrone en Erlang

4 Akka

4.1 Présentation d'Akka

Akka est un ensemble de librairies de programmation concurrente Open Source développées par la société Lightbend depuis 2009. Cet ensemble de librairies est disponible à la fois sous Scala et Java et implémente comme Erlang ou encore Elixir le modèle Acteur.

Différents modules sont égalemet proposés permettant de créer des applications dans le cloud ou des clusters de noeuds sous Akka.

4.2 Un acteur en Akka

La notion d'acteur sous Akka est assez similaire à celle de Erlang. Les acteurs disposent :

— D'une Mailbox (ou Message queue) qui est une structure Fifo servant à la réception des messages et à la séquentialisation du traitement de ces derniers par l'acteur. En akka, elle est utilisée pour recréer un acteur en situation d'échec et ainsi conserver ses messages car elle est indépendante de ce dernier.

— D'un State ("Etat") qui englobe les informations critiques sur l'état de l'Acteur et son fonctionnement. Ces données sont protégées par une implémentation légère d'un Thread des autres acteurs et du système bien qu'il soit possible en cas de mauvaise utilisation des acteurs de corrompre ces données.

— D'un comportement, un ensemble d'actions à réaliser suite à la réception d'un message, de plus ce comportment peut évoluer / changer selon les messages.

La particularité d'Akka dans l'implémentation de ces élements est leur cycle de vie. En effet contrairement à un Thread ou un processus Erlang qui s'arrête s'il n'est pas réexplicitement appeler, un acteur akka reste en mémoire tant qu'il ne reçoit pas de messages d'arrêt ou tant que son acteur père est lui aussi actif.

En comparaison avec un Thread, un acteur est beacoup plus léger et rapide.On peut ainsi développer des applications faisant communiquer plusieurs millions d'acteurs.

4.3 Un modèle en hiérarchie

Dans Akka, les acteurs sont supervisés par leur créateur. On retrouve ainsi une relation parent - enfant entre les acteurs.

Cette hiérarchie est entre autre utilisée dans la gestion des messages d'erreurs en Akka, le "let it crash" qui consiste à faire remonter au parent le message d'erreur rencontré par un acteur enfant jusqu'à trouver un superviseur, chargé de gérer ce message . On retrouve ce même système de gestion d'erreurs en Erlang.

5 Comparaison et Conclusion

Pour conclure, nous allons résumer les éléments notables permettant de comparer Go, Erlang et Akka Les différences notables sont les suivantes : Contrairement au modèle acteur, les goroutines doivent disposer d'un canal vide pour recevoir un message ce qui n'est pas le cas des mailboxes présentent en Erlang et sous Akka.

Go contrairement à Akka et Erlang n'est pas nativement pensé pour de la programmation distribuée, il faut installer pour cela une librairie.

6 Bibliographie

Cliquez sur un titre pour suivre le lien :
— *Communicating Sequential Processes*, T. Hoare, 1978
— *Spécification du langage Go* (sections Canaux et Goroutines)
— *A Universal Modular ACTOR Formalism for Artificial Intelligence*, Hewitt, Bishop et Steiger, 1973
— *Documentation d'Akka* (section Acteurs)
— *Introduction to the Actor Model for Concurrent Computation: Tech Talks*