HEIG VD
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD

Paradigme de concurrence

En utilisant Elixir

2025-04-16

Guillaume Dunant & Edwin Häffner
HEIG-VD
2025
ICSL

Table des matières

I. Introduction

L'idée derrière le paradigme de concurrence est de pouvoir efficacement gérer plusieurs tâches en parallèle sur une même machine, ou même sur plusieurs. L'objectif final est d'optimiser l'utilisation des ressources disponibles, de réduire les temps d'attente et d'accélérer l'exécution des programmes en répartissant les calculs ou les opérations entre plusieurs processus ou threads qui s'exécutent simultanément.

Un exemple simple serait d'avoir une application qui utilise un thread du CPU pour afficher la partie GUI (interface graphique) et un autre thread pour effectuer des calculs en arrière-plan. Cela éviterait de bloquer l'affichage graphique lorsqu'un calcul complexe est demandé, améliorant ainsi la fluidité et la réactivité de l'application.

II. Motivations du choix de la concurrence

La concurrence est devenue essentielle dans le développement de logiciels modernes. Les CPU actuels disposent d'un nombre croissant de cœurs, même dans le marché des processeurs abordables, par exemple, un processeur à moins de 100 francs offre déjà au moins 6 cœurs.

Donc il nous faut pouvoir utiliser ces cœurs maintenant si abondant pour pouvoir faire des applications efficace.

La concurrence est aussi très importante lorsqu'on interagit avec des API web et d'autres machines. La communication n'est pour l'instant pas instantanée et il faut alors avoir un mécanisme d'attente qui ne fait pas arrêter le programme. Les opérations d'entrée/sortie (I/O), comme les requêtes réseau ou les accès disque, sont particulièrement concernées car elles comportent des temps d'attente réel.

Pour exploiter efficacement le matériel moderne et gérer ces opérations non instantanées, il est indispensable d'adopter une approche de programmation adaptée : c'est le rôle du paradigme de concurrence.

III. Paradigme de concurrence

La programmation concurrente est donc une façon de programmer en tenant compte de l'existence de ces threads et processus. Elle implique la conception de programmes où plusieurs séquences d'instructions peuvent s'exécuter simultanément ou en alternance rapide (préemption de plusieurs processus entre eux par exemple).

Ce paradigme introduit des concepts spécifiques comme la synchronisation, les verrous, les sémaphores, les variables atomiques, et les files de messages, qui permettent de coordonner l'exécution des différents processus ou threads et d'éviter les problèmes classiques de concurrence comme les race conditions, les deadlocks ou les famines.

Gérer ces problèmes complexes a conduit à l'apparition de langages et de bibliothèques conçus spécialement pour les applications concurrentes. Parmi ces solutions, Erlang occupe une place importante et influente dans l'histoire de la programmation concurrente, ayant plus tard donné naissance à Elixir.

IV. Petit historique d'Erlang et Elixir

Erlang

Erlang (nom dérivé soit d'Ericsson Language, soit du mathématicien danois Agner Krarup Erlang) est né en 1986 dans les laboratoires d'Ericsson pour répondre aux défis croissants du monde des télécommunications. L'objectif était de créer un langage spécifiquement adapté à ces enjeux, capable de reproduire les propriétés des systèmes de commutation téléphonique. C'est de là que viennent ses caractéristiques distinctives : une haute concurrence et une tolérance aux pannes totale. Un système de commutation physique dans l'ère analogique de la téléphonie était par nature concurrent, il devait gérer de multiples appels simultanément. Il devait également être résilient, car une défaillance sur un appel ne devait en aucun cas compromettre les autres communications en cours !

Elixir

Elixir est un langage de programmation fonctionnel créé par José Valim en 2012, et dont la première version stable a été publiée en 2014. Le but de ce language était d'améliorer la syntaxe vieillissante d'Erlang (26 ans en 2012 !). Valim étant un contributeur du language Ruby On Rails, il a voulu ammener cette syntaxe et ces fonctionnalité aux outils de concurrence mis à disposition par Erlang/OTP(Open Telecom Platform).

Elixir a été conçu pour offrir tous les avantages d'Erlang tout en ajoutant des fonctionnalités modernes sans sacrifier la compatibilité avec l'écosystème Erlang/OTP existant.

V. Présentation de BEAM

Elixir tourne sur la machine virtuelle d'Erlang qui est nommée BEAM. Plus précisément le code Elixir est compilé dans un bytecode qui tourne sur BEAM. Il faut alors présenter ce que c'est BEAM avant de se focaliser sur Elixir.

Qu'est ce que c'est BEAM ?

BEAM, qui veut dire en français "La machine abstraite Erlang de Bogdan" est la machine virtuelle qui exécute le bytecode compilé des programmes Erlang et Elixir.

Vu que cette machine virtuelle a été développé pour l'ecosytème Erlang/OTP, elle est spécialement conçue pour répondre aux exigences liées à la télécommunication, la haute disponibilité, la tolérance aux pannes et la concurrence.

Voici ses caractéristiques :

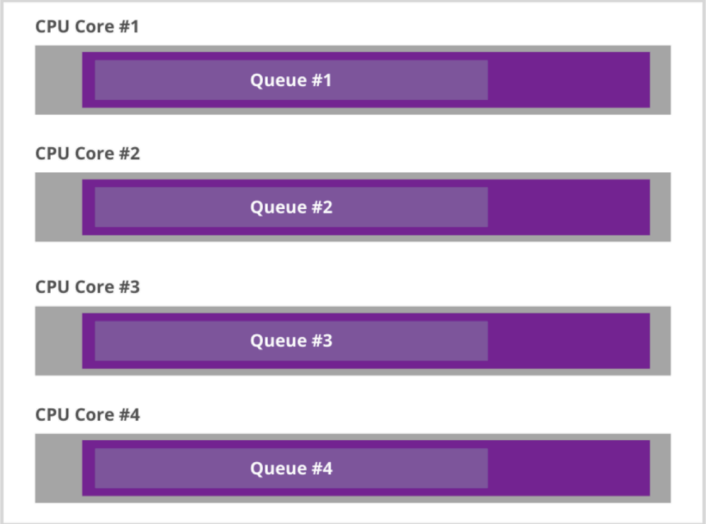
Gestion de la concurrence

BEAM exécute des processus légers qui ne partagent pas de mémoire. Ces processus communiquent uniquement via un passage de **messages asynchrone**, ce qui permet d'éviter les problèmes courants liés aux accès concurrents comme les conditions de course et les verrous. Cette isolation entre les processus est la base de la tolérance aux pannes : si l'un des processus échoue, il n'impactera pas les autres.

Planificateurs multicœurs efficaces

Initialement, BEAM utilisait une seule file d'attente d'exécution (run queue). Aujourd'hui, elle attribue une file d'attente à chaque cœur de processeur disponible, ce qui permet de paralléliser les programmes de manière optimale dynamiquement selon le nombre de cœurs de la machine.

Figure 1 — Files d'executions sur chaque cœurs.



Tolérance aux pannes grâce à la gestion des erreurs

Contrairement à un language comme Java ou l'erreur est fatale si non traitée dans un bloc try catch, Erlang prone la philosophie du "let it crash". Comme dit précédemment, cela signifie que les processus sont conçus pour échouer de manière isolée et sans impact sur le reste du système. Lorsqu'un processus rencontre une erreur, il peut simplement se planter sans essayer de corriger l'exception, tandis qu'un autre processus, souvent supervisé par un superviseur OTP, prendra en charge sa relance. Nous reparlerons de ce superviseur lorsque nous parlerons d'Elixir !

Collecte de déchets par processus

Chaque processus dans BEAM possède son propre tas et sa propre pile, alloués dans un même bloc mémoire et croissant l'un vers l'autre. Lorsque la pile et le tas se rencontrent, le collecteur de déchets (garbage collector) est déclenché pour récupérer la mémoire inutilisée. Si la mémoire récupérée s'avère insuffisante, la taille du tas est augmentée pour répondre aux besoins croissants du processus.

Cette approche, où chaque processus dispose de son propre collecteur de déchets, présente plusieurs avantages. Tout d'abord, elle permet d'isoler les pauses dues à la collecte de déchets à un seul processus, sans affecter les autres. Cela réduit les interruptions globales du système, améliorant ainsi la réactivité et la fluidité des applications.

De plus, les pauses induites par la collecte de déchets sont généralement très courtes, car elles se limitent à la mémoire utilisée par un seul processus.

Cependant, il faut faire attention aux problèmes de duplication des informations entre les processus vu que la mémoire n'est pas partagée.

VI. Pourquoi Elixir est la solution ?

Paradigme fonctionnel

Outre le paradigme de concurrence, il nous semblait important de, pour une fois, effectuer un projet en entier en utilisant un language fonctionnel. Dans le cadre de notre cours de "Paradigme et Language de Programmation", on a pu se mouiller les mains avec Haskell, mais outre les mini programmes que nous avions pu réaliser, nous n'avions jamais fait de "grand" projet du début à la fin en utilisant un language fonctionnel.

Dans la programmation fonctionnelle, une variable est toujours immuable, il n'y a pas de système de reassignments de valeurs, donc certains problèmes liés à la synchronisation disparaissent. Pas tous par contre, notamment lors des problèmes de coordinations entre différentes entités.

Les processus légers

En évoquant BEAM, nous avons mentionné l'utilisation de processus légers, mais que sont-ils exactement ?

Les processus légers sont des entités gérées directement par la machine virtuelle, contrairement aux processus natifs ou aux processus virtuels mappés sur des threads réels du système d'exploitation. Dans le cas d'Elixir, ces processus ne sont pas liés à un thread OS, mais sont entièrement administrés par la VM BEAM.

Grâce à l'absence d'appels système lors de leur création, ces processus peuvent être générés en très grand nombre et de manière extrêmement rapide.

Cependant, cette gestion par BEAM n'est pas sans inconvénients. Pour des tâches purement computationnelles, comme des calculs complexes, la surcharge introduite par l'ordonnancement et la gestion des processus légers peut ralentir l'exécution par rapport à une approche utilisant des threads natifs optimisés pour le CPU.

De plus, bien que ces processus soient légers, ils ne sont pas gratuits en termes de mémoire. Chaque processus dispose de sa propre pile, ce qui nécessite une gestion attentive de l'espace mémoire disponible.

Malgré ces limitations, les processus légers restent l'une des principales forces d'Elixir, offrant une solution puissante et flexible pour la gestion de la concurrence.

VII. La concurrence dans Elixir

Les outils de base

Les processus

En elixir, il est assez simple de créer un nouveau processus. Pour cela, il faut appeler la méthode `spawn` qui prend en paramètre une fonction à exécuter et qui retourne son pid (Processus ID).

```
pid = spawn(fn -> 1 + 2 end)
```

Cela crée un processus indépendant, c'est à dire que si celui-ci crash, le processus l'ayant créé (son parent) continuera sans s'en soucier. Il est donc également possible de créer un processus lié à un autre un appelant la méthode `spawn_link`. Cela aura pour conséquence que si un processus crash, le signal de terminaison sera transmis à son parent.

Les messages

Afin de pouvoir communiquer entre les processus, il est possible d'envoyer et de recevoir des messages. La méthode `send(dest, msg)` permet d'envoyer un message, où `dest` est la destination du message (celui-ci peut être un pid local ou distant, un port local, ou autre ...) et `msg` le contenu du message. Cette méthode est non bloquante. Le message est ajouté à la "boîte aux lettres" (mailbox) du processus. Ces messages peuvent être récupérés via la construction `receive`. Celle-ci permet de récupérer un message selon un pattern ou d'attendre qu'un message arrive.

```
pid = spawn(fn ->
  receive do
    {:hello, msg} -> msg
    {:world, _msg} -> "Not a match"
  after
    1_000 -> "Timeout"
  end
end)

send(pid, {:hello, "World"})
```

Comme `receive` est bloquant, il est possible de définir un timeout grâce au mot-clé `after`.

Open Telecom Platform (OTP)

OTP est un ensemble d'outils permettant de simplifier la création de système distribué. A l'origine développé pour le langage Erlang, les bibliothèques d'OTP sont également disponible en Elixir.

Supervision Tree

Un des concepts clé de la gestion de panne dans OTP est l'arbre de supervision. Ce concept défini deux types de processus:

- Les **workers** dont le but est d'effectuer des tâches
- Les **supervisors** dont le but est de gérer les workers. Ils peuvent donc les redémarrer si l'un d'eux crash.

Cela permet de gérer aisément les erreurs des processus en définissant des politiques de redémarrage lorsqu'un des enfant d'un supervisor crash (plus de détail dans Supervisor). Les enfants des supervisors peuvent être des workers mais également un autre supervisor, permettant ainsi de créer une hiéarchie en arbre.

Agent

Les Agents permettent une gestion simple d'un état partagé. En plus, la concurrence est directement géré par le module, il est donc possible d'accéder et de modifier la valeur de l'Agent depuis plusieurs processus de manière concurrente. Voici un exemple[1] d'Agent implémentant un compteur:

[1] https://hexdocs.pm/elixir/Agent.html

```elixir
defmodule Counter do
  use Agent

  def start_link(initial_value) do
    Agent.start_link(fn -> initial_value end, name: __MODULE__)
  end

  def value do
    Agent.get(__MODULE__, fn state -> state)
  end

  def increment do
    Agent.update(__MODULE__, fn state -> state + 1)
  end
end
```

Les Agents définissent simplement deux fonctions, `get` qui permet de passer une fonction à l'agent qui prend en paramètre l'état actuel et dont le résultat sera retourné à l'appelant. Et la fonction `update` qui est similaire à `get` sauf que le résultat n'est pas retourné, mais utilisé comme nouvelle valeur de l'état interne. A noter également que lors du démarrage de l'agent, la macro `__MODULE__` est donné comme nom pour l'Agent. Cette macro retourne le nom du module et permet donc de s'adapter si le nom que l'on veut donner au module change, sans devoir nous même le modifier partout dans le code (le premier paramètre des fonctions `get` et `update` est également le nom que l'on a passé lors du lancement de l'agent).

GenServer

GenServer (Generic Server) est une abstraction faisant partie de la famille des workers. Celle-ci permet à un processus de pouvoir gérer un état interne et à des processus externe de communiquer avec lui via des messages synchrones ou asynchrones. Les messages sysnchrones signifient que l'envoyeur attend la réponse alors les messages asynchrones n'attendent pas de réponse. Cette abstraction va implémenter toute la logique de l'écoute de message et de gestion de l'état, laissant au développeur la seule résponsabilité d'implémenter les callbacks nécessaire. Voici un exemple[2] de GenServer implémentant une stack:

```elixir
defmodule Stack do
  use GenServer

  # Callbacks

  @impl true
  def init(elements) do
    initial_state = String.split(elements, ",", trim: true)
    {:ok, initial_state}
  end

  @impl true
  def handle_call(:pop, _from, state) do
    [to_caller | new_state] = state
    {:reply, to_caller, new_state}
  enddefmodule Counter do
  use Agent

  def start_link(initial_value) do
    Agent.start_link(fn -> initial_value end, name: __MODULE__)
  end
```

[2] https://hexdocs.pm/elixir/GenServer.html

```
  def value do
    Agent.get(__MODULE__, & &1)defmodule Counter do
  use Agent

  def start_link(initial_value) do
    Agent.start_link(fn -> initial_value end, name: __MODULE__)
  end

  def value do
    Agent.get(__MODULE__, & &1)
  end

  def increment do
    Agent.update(__MODULE__, &(&1 + 1))
  end
end
  end

  def increment do
    Agent.update(__MODULE__, &(&1 + 1))
  end
end

  @impl true
  def handle_cast({:push, element}, state) do
    new_state = [element | state]
    {:noreply, new_state}
  end
end
```

On peut voir dans l'exemple les trois principaux callbacks:

- `init(init_args)` : Permet de définir le comportement du GenServer lors du démarrage du serveur.
- `handle_call(request, from, state)` : Permet de gérer un appel synchrone. Il est possible d'en définir plusieurs pour des `request` différentes. `state` correspond à l'état interne maintenu par le serveur. La valeur retournée à l'appelant est la valeur de `to_caller` et le nouvelle état qui sera transmis au prochain callback est `new_state` .
- `handle_cast(request, state)` : Comme `handle_call` mais pour les appels asynchrones. La principal différence est que le résultat est un `:noreply` , c'est-à-dire qu'aucune valeur n'est retournée à l'appelant.

Supervisor

Le Supervisor est l'outils servant à gérer le cycle de vie des autres processus. Voici un simple exemple reprenant notre GenServer:

```
children = [
      {Stack, "1,2,3"}
    ]

opts = [strategy: :one_for_one, name: Stack.Supervisor]
Supervisor.start_link(children, opts)
```

Description des éléments:

- **children :** Contient une liste avec les modules enfants qui seront supervisé. Dans notre cas, c'est un tuple avec le nom du module (Stack qui est le GenServer de l'exemple précédent) ainsi que ces paramètres d'initilaisation. Il est aussi possible de spécifier uniquement le nom du module.

- **opts :** C'est ici qu'on peut décrire les paramètres du supervisor. Il y a son nom ainsi que la stratégie de redémarrage des enfants. Il en existe 3
  - `:one_for_one` : Redémarre seulement le processus qui a crash
  - `:one_for_all` : Redémarre tous les processus quand un crash
  - `:rest_for_one` : Redémarre le processus qui a crash et tous ceux qui ont démarrer après lui
- Enfin, le supervisor est démarré avec les enfants et les options en paramètre. A noté que seul l'option `strategy` est obligatoire. `name` est optionnel est il en existe également d'autre comme, le nombre de redémarrage, etc...

Pour qu'un module puisse être passé au supervisor, il faut qu'il ait la fonction `child_spec` afin d'indiquer son comportement. Dans notre exemple, il n'est pas nécessaire de le faire car le GenServer le fait pour nous. Cependant, il est possible de directement passer une map contenant nos child_spec dans la liste d'enfant du supervisor.

```
children = [
  %{
      id: Stack,
      start: {Stack, :start_link, "1,2,3"},
      shutdown: 5_000,
      restart: :permanent,
      type: :worker
    }
]
```

Description des éléments:

- **id :** Identifiant qui sera utilisé par le supervisor en interne pour identidfier le processus
- **start :** Définit comment démarrer le processus, donc le nom du module, la fonction à appeler pour démarrer et les arguments
- **shutdown :** Définit comment arrêter le processus. Il y a 3 valeurs possibles:
  - N'importe quelle entier positif : Définit le temps en milliseconde que le supervisor laisse au processus pour s'arrêter. Si le processus est toujours vivant après, il est kill
  - `:brutal_kill` : Kill le processus immédiatement
  - `:infinity` : Attend jusqu'à ce que le processus s'arrête, sans timeout
- **restart :** Définit comment redémarrer le processus. Il y a 3 politiques disponibles:
  - `:permanent` : Le processus sera toujours redémarré
  - `:temporary` : Le processus n'est jamais redémarré, peut importe la startégie de supervision. Toutes terminaisons, même anormale, est considérée comme réussie.
  - `:transient` : Le processus est redémarré seulement s'il est terminé de manière anormale.
- **type :** Définit le type du noeud dans l'arbre de supervision. Il peut être soit `:worker`, soit `:supervisor`

DynamicSupervisor

Le DynamicSupervisor, contrairement au Supervisor simple, permet de démarrer et de stopper de manière dynamique ses processus enfants. Il est typiquement démarrer par un Supervisor sans enfant, pour ensuite les ajouter de manière dynamique. L'unique startégie possible pour un DynamicSupervisor est `:one_for_one` comme la gestion est dynamique. Voici un

exemple[3]:

```
children = [
  {DynamicSupervisor, name: MyApp.DynamicSupervisor, strategy: :one_for_one}
]

Supervisor.start_link(children, strategy: :one_for_one)

{:ok, stack1} = DynamicSupervisor.start_child(MyApp.DynamicSupervisor, {Stack, "1,2,3"})

{:ok, stack2} = DynamicSupervisor.start_child(MyApp.DynamicSupervisor, {Stack, "4,5,6"})

DynamiqcSupervisor.terminate_child(MyApp.DynamicSupervisor, stack1)
```

VIII. Cahier des charges prévisionnel

Nom du projet : Space Capitalism

Description

Space Capitalism est un jeu de gestion d'un empire galactique jouable sur le web. Le joueur incarne un dirigeant interstellaire qui doit gérer ses ressources, coloniser des planètes, miner des astéroïdes et développer son économie pour prospérer dans un univers sans foi ni lois.

Gameplay

Le joueur commence sa partie avec des ressources de bases, une planète, des travailleurs et quelque vaisseaux de minage. Le joueur peut améliorer sa troupe en améliorant de façon verticale ou horizontale pour pouvoir amasser le plus de ressources possible.

Utilisation des ressources

Les ressources collectées peuvent être utilisées de deux manières principales :

1. **Investissement dans la bourse galactique** : Maximiser les profits en spéculant sur les fluctuations économiques interstellaires.
2. **Amélioration des infrastructures et des unités** : Acheter des éléments pour renforcer sa flotte, améliorer les capacités de production ou développer de nouvelles technologies.

Interactions et défis

Le joueur devra également faire face à des défis tels que la concurrence avec d'autres empires, des crises économiques, ou des événements aléatoires qui peuvent influencer en bien comme en mal le déroulement de la partie!

But final du jeu

Le but ultime est de prospérer le plus longtemps possible en évitant la faillite. Dès que l'argent rentre dans le négatif, c'est la fin !

Utilisation de la concurrence dans le projet

Le but de ce jeu est de simuler cette gestion, chaque travailleur, vaisseau ou même planète sera son propre processus. La bourse elle même sera un procesus, donc le défi sera la communication entre ces centaines de processus de façon concurrente !

[3] https://hexdocs.pm/elixir/DynamicSupervisor.html

Technologies utilisées

Le framework Phoenix, basé sur Elixir, sera utilisé pour développer le jeu. Ce choix garantit une gestion optimale de la concurrence et une scalabilité adaptée à un jeu en ligne.

IX. Bibliographie

https://www.erlang-solutions.com/blog/comparing-elixir-vs-java/

https://medium.com/flatiron-labs/elixir-and-the-beam-how-concurrency-really-works-3cc151cddd61

https://medium.com/@ck3g/introduction-to-otp-genservers-and-supervisors-cf1358d545

https://medium.com/elemental-elixir/elixir-otp-basics-of-processes-d3437607d12b#:~:text=Elixir%20processes%20are%20similar%20to,internally%20by%20the%20Beam%20VM.

https://hexdocs.pm/elixir/

https://elixirschool.com/en/lessons/advanced/otp_concurrency

Histoire Erlang / Elixir

https://thechipletter.substack.com/p/ericsson-to-whatsapp-the-story-of https://www.erlang-solutions.com/blog/twenty-years-of-open-source-erlang/ https://elixir-lang.org/blog/2013/08/08/elixir-design-goals/ https://ouidou.fr/2019/01/31/une-breve-histoire-delixir-et-erlang-otp

BEAM

https://en.wikipedia.org/wiki/BEAM_(Erlang_virtual_machine)

https://www.erlang.org/blog/a-brief-beam-primer/

https://www.erlang.org/blog/beam-compiler-history/

https://www.erlang-solutions.com/blog/the-beam-erlangs-virtual-machine/

https://elixirschool.com/en/lessons/advanced/otp_supervisors

https://www.erlang.org/doc/apps/erts/garbagecollection.html