

Coordonner sans se marcher dessus

Réserves de travailleurs, contre-pression et lois de la parallélisation

réserve de travailleurs

contre-pression

loi d'Amdahl

vol de travail

let it crash

Un coordinateur qui ne calcule rien, des travailleurs qui ne se connaissent pas, et une limite qui change tout.

Un cadeau de fin de cours, à lire par curiosité : rien n'est à rendre. Le projet « Ferme de rendu » reste décrit par sa fiche sur le site et par le README de son squelette ; ce fascicule raconte les idées qui l'entourent et ce qu'elles deviennent au-delà du cours.

1 Pourquoi ce fascicule

Le projet « ferme de rendu » découpe une image de Mandelbrot en bandes indépendantes, confie chaque bande à un processus et reconstruit l'image quand les résultats reviennent, dans le désordre. Vous y écrivez un *coordonateur* : un processus qui lance au plus `limit` travailleurs, garde la file des bandes restantes, range chaque résultat sous l'identifiant de son travail et transforme un travailleur arrêté en résultat d'erreur. Tout le reste, le calcul des pixels, le dessin SVG et la commande, vous est fourni.

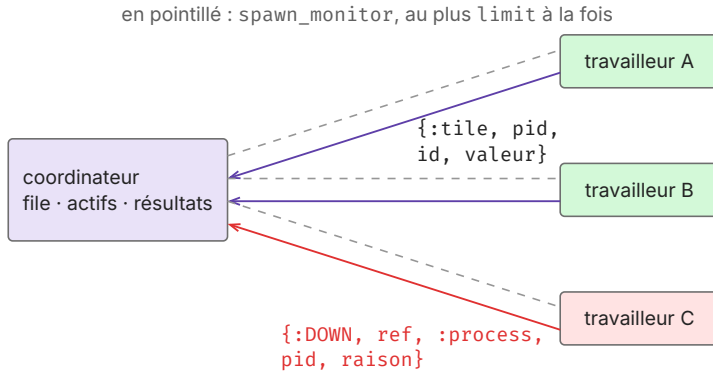
Ce fascicule ne contient pas la solution, ni en code ni en prose : le README du squelette vous donne les contrats et les invariants, et c'est à vous de les articuler. Il raconte plutôt d'où vient cette façon de travailler, et ce qu'elle devient hors du cours. Nous verrons pourquoi borner le nombre de calculs en vol est une idée centrale des systèmes concurrents (la *contre-pression*), pourquoi la machine virtuelle d'Erlang s'y prête particulièrement, quels outils d'Elixir résolvent déjà ce problème et ce qu'ils cachent, combien de fois plus vite on peut espérer aller (les lois d'Amdahl et de Gustafson, avec un calcul déroulé), ce que « laisser planter » veut dire au juste, et enfin comment les fermes de rendu de cinéma, MapReduce et les files de travaux reprennent les mêmes idées à une tout autre échelle. Lisez-le après avoir fait tourner votre première trace, un soir, sans rien à rendre.

2 Une réserve de travailleurs et une limite

Commençons par nommer ce que le projet vous fait construire. Une *réserve de travailleurs* (*worker pool*) est un ensemble de processus qui exécutent des travaux indépendants pour le compte d'un coordinateur. Un travail est *en vol* entre le moment où il est lancé et celui où son résultat, ou sa panne, est connu. La *concurrence bornée* consiste à fixer un nombre maximal de travaux en vol : c'est la `limit` de `RenderFarm.run/3`.

Pourquoi se limiter, alors que la BEAM sait faire tourner des milliers de processus ? Parce que la limite ne porte pas sur les processus mais sur ce qu'ils consomment. Sur une machine à huit cœurs, quarante bandes lourdes se partagent huit cœurs : les trente-deux autres attendent leur tour dans l'ordonnanceur, et chacune garde pendant ce temps sa pile et ses pixels en mémoire. À l'usage, la ressource rare est ailleurs encore : des connexions à une base de données, la bande passante d'un

disque, le nombre d'appels par seconde qu'un service distant accepte. La limite dit donc « pas plus de n consommateurs de cette ressource à la fois », et le coordinateur est l'endroit unique où cette règle s'applique.



Chaque travailleur ne connaît que son travail ; seul le coordinateur voit l'ensemble.

Les lancements partent du coordinateur (pointillés). Les résultats remontent sous forme de messages, et la fin d'un travailleur sans résultat remonte aussi, grâce au monitor, par un message `:DOWN`. Le coordinateur ne calcule rien : il reste libre pour recevoir.

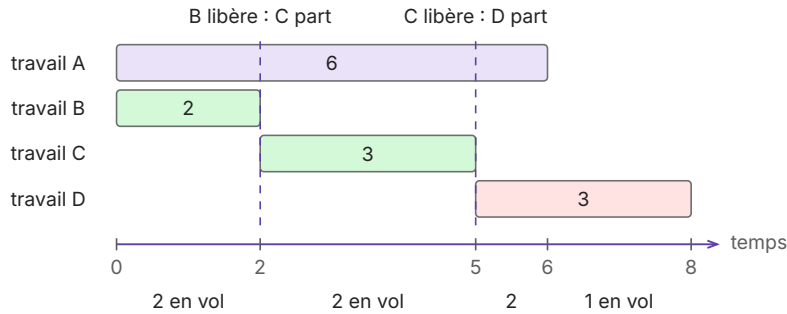
2.1 L'invariant et pourquoi la collecte se termine

Deux quantités décrivent l'état du coordinateur à tout instant : la file des travaux qui restent à lancer et l'ensemble des travaux actifs. L'*invariant* du projet, c-à-d. la propriété vraie avant et après chaque message, est que le nombre d'actifs ne dépasse jamais la limite. Chaque événement, un résultat ou un `:DOWN`, retire exactement un actif, après quoi le coordinateur relance des travaux de la file jusqu'à retrouver la limite, ou jusqu'à ce que la file soit vide.

Pourquoi cette boucle se termine-t-elle ? Esquissons l'argument, car c'est celui que votre compte-rendu doit tenir. Supposons que chaque travail finit par produire exactement un événement, un résultat ou une fin de processus : c'est l'hypothèse du socle, et nous verrons plus loin ce qu'elle exclut. Chaque lancement retire un élément de la file, qui ne reçoit jamais rien : il y a donc au plus autant de lancements que de travaux. Chaque lancement produit un événement, et chaque événement est traité une fois. Le nombre total d'événements est donc fini, et après le dernier, la file est vide et plus aucun travailleur n'est actif. Remarquez que les deux conditions sont nécessaires : une file vide ne dit rien des calculs encore en cours, et un ensemble d'actifs vide ne dit rien de ce qui reste à lancer. C'est pourquoi le README vous demande de les énoncer séparément.

2.2 Un déroulé à la main

Prenons quatre travaux dont nous connaissons, pour l'exemple, les durées : A coûte 6 unités, B en coûte 2, C et D en coûtent 3 chacun, et la limite est de deux en vol. Le coordinateur lance A et B. À l'instant 2, B répond ; une place se libère et C part. À l'instant 5, C répond et D part, alors qu'A calcule toujours. À l'instant 6, A répond enfin : la file est vide, mais D est encore actif. La collecte ne termine qu'à l'instant 8.



Deux travaux en vol au plus. Les pointillés marquent les deux instants où un résultat libère une place. La file est vide dès l'instant 5, mais la collecte ne finit qu'à l'instant 8, quand D répond.

instant	événement	actifs	file
0	lance A et B	A, B	C, D
2	B répond ; lance C	A, C	D
5	C répond ; lance D	A, D	vide
6	A répond	D	vide
8	D répond : terminé	vide	vide

Séquentiellement, les quatre travaux auraient coûté 14 unités ; sans limite, 6 unités, la durée du plus long. Avec deux places, 8. Notez que l'ordre des réponses, B, C, A, D, ne ressemble ni à l'ordre de lancement ni à l'ordre des bandes dans l'image : c'est bien l'identifiant qui range chaque résultat.

Elixir fournit `Task.async_stream`, un outil que nous détaillons plus loin et qui applique précisément une telle limite. Servons-nous-en d'abord comme d'un instrument de mesure : un `Agent` compte les travaux en vol et retient le maximum observé. Avant de lire la valeur, prévoyez-la.

```

1 { :ok, gauge } = Agent.start_link(fn -> {0, 0} end)
2 work = fn n ->
3   Agent.update(gauge, fn {now, peak} -> {now + 1, max(peak, now +
4     1)} end)
5   Process.sleep(10)
6   Agent.update(gauge, fn {now, peak} -> {now - 1, peak} end)
7   n * n
8 end
9 squares =
10   1..8
11   |> Task.async_stream(work, max_concurrency: 3)
12   |> Enum.map(fn { :ok, v } -> v end)
13 { squares, Agent.get(gauge, fn _, peak} -> peak end) }

```

```
VALEUR { [1, 4, 9, 16, 25, 36, 49, 64], 3 }
```

Huit travaux, trois places : le pic vaut 3, jamais 8. Le tuple `{now, peak}` est mis à jour dans le processus de l'Agent, en une seule opération, pour les raisons vues au chapitre 10 : lire puis écrire en deux requêtes pourrait perdre une mise à jour.

2.3 La demande remonte : la contre-pression

Dans le projet, la liste des travaux est connue d'avance. Dans un serveur, les travaux arrivent au fil de l'eau, et une question nouvelle se pose : que faire quand ils arrivent plus vite que les travailleurs ne les terminent ? Si le coordinateur accepte tout, sa file grandit sans borne, et sa boîte aux lettres avec elle, jusqu'à épuiser la mémoire. La *contre-pression* (*backpressure*) est le mécanisme qui transmet la limite vers l'amont : c'est le consommateur qui annonce combien de travaux il peut prendre, et le producteur n'en envoie pas davantage. En clair, la demande circule dans le sens inverse des données. Votre coordinateur en est une version minuscule : une place libre est une demande de un.

À RETENIR

Borner les travaux en vol protège une ressource, pas le nombre de processus. L'invariant « actifs ≤ limite » se maintient parce qu'un seul processus lance et compte. La terminaison demande deux conditions : file vide *et* aucun actif. Enfin, quand les travaux arrivent en continu, la limite doit remonter jusqu'au producteur : c'est la contre-pression.

3 Pourquoi la BEAM s'y prête

Le chapitre 9 vous a présenté les processus comme des unités légères qui ne partagent rien. Précisons maintenant ce que fait la machine virtuelle en dessous, car c'est ce qui rend votre coordinateur si simple à écrire.

3.1 Un ordonnanceur par cœur, et un budget de réductions

Au démarrage, la BEAM crée par défaut un *ordonnanceur* (*scheduler*) par cœur logique de la machine ; `System.schedulers_online()` en donne le nombre. Chaque ordonnanceur est un fil d'exécution du système d'exploitation qui tient sa propre file de processus prêts et les fait avancer à tour de rôle. Un processus BEAM n'est donc jamais un cœur à lui seul : lancer quarante travailleurs sur huit cœurs ne multiplie pas la puissance par quarante, et c'est l'une des raisons de la limite précédente.

La répartition du temps est *préemptive* : la machine virtuelle compte, pour chaque processus, des *réductions*, en gros un appel de fonction, et le retire du cœur après un budget fixé, 4 000 réductions dans le code source actuel de la VM, même si le processus est en plein calcul. Pour votre ferme, cela a une conséquence directe : un travailleur qui calcule une bande pendant une seconde ne peut pas empêcher le coordinateur de lire sa boîte aux lettres. Ce n'est pas le cas de tous les environnements concurrents. Dans une boucle d'événements coopérative, comme celle de JavaScript dans Node.js ou celle d'`asyncio` en Python, un calcul long bloque tout le monde jusqu'à ce qu'il rende la main de lui-même. Soyons précis cependant : la préemption de la BEAM s'arrête à la frontière du code Erlang. Une fonction native écrite en C et appelée depuis Elixir, une *NIF*, n'est pas découpée en réductions, et une NIF longue retient bel et bien son ordonnanceur ; la VM prévoit pour cela des ordonnanceurs séparés, dits *dirty*. Vos pixels sont calculés en Elixir, et ce souci ne vous concerne pas.

3.2 Des processus réellement légers

Combien coûte un processus ? La documentation d'Erlang indique qu'un processus fraîchement créé occupe 338 mots de mémoire, dont un tas initial de 233 mots ; sur une machine 64 bits, un mot fait huit octets, soit environ 2,7 Ko. Un fil d'exécution du système d'exploitation réserve, lui, une pile de l'ordre du méga-octet. La création d'un processus prend quelques microsecondes, et la VM en fait vivre des centaines de milliers sur une machine ordinaire. Voilà pourquoi personne n'hésite à créer un processus par bande, par requête HTTP ou par connexion. Un processus par pixel resterait pourtant une mauvaise idée, non parce que la VM n'y survivrait pas, mais

parce que chaque message coûte une copie : la BEAM copie les termes d'un tas à l'autre lorsqu'elle les envoie, à l'exception des binaires de plus de 64 octets, qu'elle partage par référence. Des milliers de petits résultats coûtent plus de copies qu'une poignée de bandes.

3.3 D'où vient tout cela : Ericsson, 1986

Erlang est né au laboratoire d'informatique d'Ericsson, en Suède, où Joe Armstrong, Robert Virding et Mike Williams cherchaient, à partir de 1986, un langage pour programmer des commutateurs téléphoniques : des milliers d'appels simultanés, isolés les uns des autres, et un système que l'on corrige sans jamais l'éteindre. Les premières versions étaient un interpréteur écrit en Prolog ; plusieurs machines virtuelles lui ont succédé, dont la BEAM, qui s'est imposée à la fin des années 1990 et que nous utilisons toujours. OTP, la bibliothèque des serveurs génériques et des superviseurs, a été assemblée vers 1996 à partir de l'expérience des projets internes. En 1998, Ericsson a rendu Erlang public sous licence libre. Joe Armstrong a raconté cette histoire lui-même, en 2007, dans un article de la conférence HOPL sur l'histoire des langages de programmation, cité en fin de fascicule.

La formule « *let it crash* » résume la philosophie qu'il a formalisée dans sa thèse, soutenue en 2003 à l'institut KTH de Stockholm. Elle ne dit pas « ignorez les erreurs » ; elle dit que le code qui fait le travail ne doit pas être aussi celui qui répare. Un processus qui rencontre une situation qu'il ne sait pas traiter s'arrête, proprement et vite, et c'est un *autre* processus, qui l'observe, qui décide quoi faire : relancer, abandonner, signaler. Deux primitives rendent cela possible. Le *lien* est symétrique et relie les destins de deux processus : la fin anormale de l'un termine l'autre, sauf s'il a choisi de recevoir ces signaux comme des messages. Le *monitor* est asymétrique et sans conséquence pour l'observé : l'observateur reçoit simplement un message : *DOWN*. Votre coordinateur observe ; il ne veut pas mourir avec un travailleur. Un superviseur, lui, est lié à ses enfants et piège leurs sorties. Elixir, créé par José Valim à partir de 2011 et publié en version 1.0 en 2014, n'a rien changé à ce modèle : il a apporté une syntaxe et des outils à une machine virtuelle qui avait déjà vingt ans de production derrière elle.

POUR EN PARLER

Le coordinateur reste disponible pendant les calculs grâce à la préemption par réductions, pas grâce à une astuce de votre code. Un processus coûte quelques kilo-octets, mais chaque message coûte une copie. Un monitor informe sans lier les destins ; c'est le choix d'un observateur qui veut survivre à ce qu'il observe.

4 Les outils qui existent déjà

Le README vous demande de construire la boucle à la main, puis seulement de comparer avec les outils d'Elixir. Voici ce que vous retrouverez chez eux, et ce qu'ils décident à votre place.

4.1 Task.async_stream et max_concurrency

`Task.async_stream(enum, fun, options)` crée une tâche par élément de l'énumérable, en gardant au plus `max_concurrency` tâches en vol, et rend un *flux* (`stream`), c-à-d. une suite paresseuse de résultats que l'on consomme avec `Enum`. Par défaut, la limite vaut `System.schedulers_online()`, les résultats sont rendus dans l'ordre des entrées, et chaque tâche dispose de cinq secondes. Avec quatre travaux qui dorment des durées différentes, l'ordre de fin est d, b, c, a, mais le résultat suit l'ordre de la liste :

```
1 jobs = [a: 30, b: 10, c: 20, d: 5]
2 jobs
3 |> Task.async_stream(fn {id, ms} ->
4   Process.sleep(ms)
5   {id, ms * 2}
6 end, max_concurrency: 2)
7 |> Enum.map(fn {:ok, value} -> value end)
```

VALEUR [a: 60, b: 20, c: 40, d: 10]

Chaque résultat arrive enveloppé dans `{:ok, valeur}`. Que se passe-t-il si une tâche dépasse son délai ? Par défaut, l'appelant lui-même sort avec une erreur. L'option `on_timeout: :kill_task` demande au contraire de tuer la tâche en retard et de rendre `{:exit, :timeout}` à sa place, ce qui ressemble à votre `{:error, raison}` :

```
1 [fn -> :vite end, fn -> Process.sleep(500); :lent end]
2 |> Task.async_stream(& 81.(), timeout: 50, on_timeout: :kill_task)
3 |> Enum.to_list()
```

VALEUR [ok: :vite, exit: :timeout]

Vous reconnaissez, sous ces options, les décisions que votre coordinateur prend explicitement : combien en vol, dans quel ordre rendre, quoi faire d'un travailleur qui ne répond pas. La différence est que `Task.async_stream` les a figées, et qu'il lie les tâches à l'appelant : si l'appelant meurt, les tâches meurent avec lui, ce que votre socle ne fait pas.

4.2 GenStage et Flow : la demande qui remonte

En 2016, José Valim a présenté *GenStage*, une bibliothèque qui organise un calcul en *étages* reliés par des flux d'événements. Un étage *producteur* émet des événements, un étage *consommateur* les traite, et un étage intermédiaire fait les deux. Le point essentiel est le sens de la demande : c'est le consommateur qui demande au producteur un nombre d'événements, et le producteur n'en envoie jamais davantage que ce qui lui a été demandé. La contre-pression de la section précédente est ainsi le contrat même de la bibliothèque, et non une option. *Flow*, construit au-dessus de *GenStage* et publié début 2017, applique ce modèle aux collections : on écrit un `Flow.map` et un `Flow.reduce` comme avec `Enum`, et la bibliothèque répartit le travail entre des étages parallèles, partitionne les données par clé et règle la demande entre eux. *Broadway*, en 2019, a repris le même socle pour consommer des files de messages externes. Dans chacun de ces outils, votre coordinateur est devenu un producteur, vos travailleurs des consommateurs, et `limit` une demande.

4.3 Le vol de travail

Une file centrale n'est pas la seule manière de répartir. Robert Blumofe et Charles Leiserson ont étudié, dans un article présenté en 1994 et publié dans sa forme définitive en 1999, une stratégie décentralisée : le *vol de travail* (*work stealing*). Chaque processeur garde sa propre file de tâches et y travaille seul ; lorsqu'elle se vide, il choisit un autre processeur au hasard et lui *vole* une tâche. Ils ont montré que le temps d'exécution attendu reste proche de $T_1/P + T_\infty$, où T_1 est le travail total, P le nombre de processeurs et T_∞ la longueur du plus long chemin de dépendances ; en clair, on ne perd ni à répartir ni à attendre, à une constante près. Ce résultat est la base du langage Cilk, développé au MIT par le groupe de Leiserson à partir de 1994, et il est aujourd'hui partout : le `ForkJoinPool` de Java, livré avec Java 7 en 2011, ou la bibliothèque `rayon` de Rust reposent sur ce principe. La BEAM elle-même, depuis la fin des années 2000, donne à chaque ordonnanceur sa propre file et laisse un ordonnanceur désœuvré voler des processus aux autres.

Votre coordinateur, lui, est une file centrale : c'est plus simple à écrire et à prouver, et chaque décision passe par un seul processus, ce qui est aussi sa limite quand les travaux deviennent très nombreux et très courts.

5 Combien de fois plus vite ?

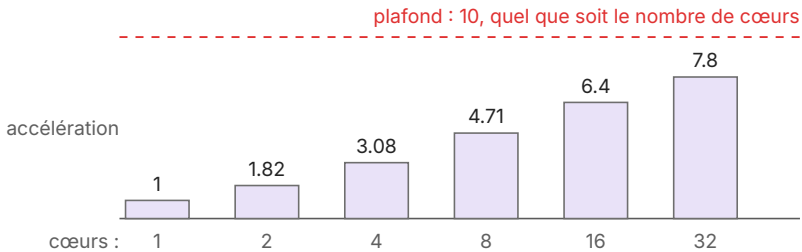
Le README vous prévient qu'une petite image peut être plus lente en parallèle. Deux lois anciennes disent pourquoi, et jusqu'où l'on peut espérer aller.

5.1 La loi d'Amdahl

En 1967, Gene Amdahl, l'architecte des grands ordinateurs d'IBM, a présenté un argument simple contre les machines à nombreux processeurs. Appelons p la fraction du temps d'un calcul qui peut être parallélisée, et $1 - p$ la fraction qui reste séquentielle, quoi qu'il arrive. Sur N processeurs, la partie parallèle est divisée par N et la partie séquentielle ne change pas. L'accélération, c-à-d. le rapport entre le temps sur un processeur et le temps sur N , vaut donc

$$S(N) = \frac{1}{(1 - p) + p/N}.$$

Déroulons le calcul avec $p = 0,9$, soit un calcul parallélisable à 90 %, sur quatre cœurs. La partie séquentielle vaut 0,1 ; la partie parallèle, 0,9 divisé par 4, vaut 0,225. Leur somme, 0,325, est le temps sur quatre cœurs si le temps sur un cœur vaut 1 ; l'accélération est donc 1/0,325, soit environ 3,08 : quatre cœurs, trois fois plus vite. Lorsque N grandit, le terme p/N tend vers zéro et l'accélération plafonne à $1/(1 - p)$, ici 10. Avec dix pour cent de séquentiel, mille cœurs ne donneront jamais mieux que dix fois plus vite.



Accélération pour 90 % de travail parallélisable. Les barres s'approchent du plafond rouge sans l'atteindre : seize cœurs n'en valent que 6,4 et trente-deux cœurs 7,8.

```
1 amдахl = fn p, n -> 1 / ((1 - p) + p / n) end
2 for n <- [1, 2, 4, 8, 16, 1000], do: Float.round(amdahl.(0.9, n), 2)
```

VALEUR [1.0, 1.82, 3.08, 4.71, 6.4, 9.91]

Où est la partie séquentielle de votre ferme ? Dans le découpage, dans la collecte et l'assemblage du SVG, dans la création des processus et, nous l'avons vu, dans la copie de chaque bande vers la boîte du coordinateur. Pour une petite image, ces coûts fixes dominent le calcul lui-même, et la version parallèle perd. Il y a une seconde raison, propre à Mandelbrot : les bandes ne coûtent pas toutes le même prix. Un point hors de l'ensemble s'échappe en quelques itérations ; un point dedans en consomme le maximum. Les bandes qui traversent l'ensemble sont donc bien plus

lourdes, et la durée totale ne peut pas descendre sous celle de la bande la plus lourde, même avec cent cœurs. Découper plus finement réduit ce déséquilibre, au prix de plus de messages. Votre compte-rendu gagnera à nommer ces deux effets plutôt qu'à promettre une accélération.

5.2 Le contrepoint de Gustafson

En 1988, John Gustafson a publié, dans les *Communications of the ACM*, une relecture de cet argument à partir de son expérience sur une machine à 1 024 processeurs. Son observation est que personne n'achète mille processeurs pour résoudre plus vite le problème d'hier : on les achète pour résoudre un problème mille fois plus gros dans le même temps. Si l'on fixe le temps d'exécution plutôt que la taille du problème, et que la part séquentielle mesurée *sur la machine parallèle* vaut s , l'accélération obtenue en ramenant le même travail sur un seul processeur vaut $s + (1 - s)N$: elle croît linéairement avec N . Avec $s = 0,1$ et seize cœurs, cela donne 14,5, là où Amdahl annonçait 6,4. Les deux lois ne se contredisent pas, elles ne mesurent pas la même chose : Amdahl répond à « combien de temps gagnerai-je sur cette image ? », Gustafson à « quelle image pourrai-je rendre dans le même temps ? ». Pour une ferme de rendu, la seconde question est souvent la bonne : on n'y accélère pas un film, on y rend un film plus grand.

UNE ACCÉLÉRATION N'EST PAS UN CRITÈRE DE RÉUSSITE

Comparez d'abord les deux images, octet par octet si possible ; elles doivent être identiques. Puis comparez les temps, plusieurs fois, en notant le nombre d'ordonnanceurs. Un temps parallèle plus long sur une petite image est une observation valable, à condition de l'expliquer : coûts fixes, copies, déséquilibre des bandes. Un chiffre seul, sans ces explications, ne prouve rien.

6 Quand un travailleur tombe

Le socle du projet transforme un `:DOWN` en `{:error, raison}` et libère la place. C'est la plus petite politique de panne qui soit. Regardons les autres, et ce qu'elles supposent du travail lui-même.

6.1 Stratégies de supervision

Un `Supervisor` d'OTP connaît pour chaque enfant une recette de démarrage et une *stratégie*. Avec `:one_for_one`, la stratégie d'AP7, seul l'enfant tombé est relancé. Avec `:one_for_all`, tous les enfants sont arrêtés puis relancés ensemble, ce qui convient quand ils ne peuvent pas fonctionner l'un sans l'autre. Avec `:rest_for_one`, l'enfant tombé et ceux démarrés après lui sont relancés, dans l'ordre : on l'utilise quand les enfants suivants dépendent du précédent. Chaque enfant porte aussi un type de redémarrage : `:permanent` est toujours relancé, `:transient` seulement s'il s'est terminé anormalement, `:temporary` jamais. Enfin, le superviseur se protège contre un enfant qui échoue en boucle : par défaut, plus de trois redémarrages en cinq secondes le font abandonner et se terminer lui-même, et c'est alors *son* superviseur qui décide.

Que donnerait un tel superviseur pour des travailleurs de rendu ? Un enfant `:permanent` dont la recette contient sa bande serait recalculé indéfiniment après chaque panne, ce qui est une politique de réessai sans limite ; un enfant `:temporary` ne serait jamais relancé, et c'est le coordinateur qui déciderait, comme dans votre socle. Le `Task.Supervisor` du prolongement démarre justement des tâches `:temporary`, qu'il isole et compte sans les reprendre. La supervision est faite pour remettre un *service* en état ; la reprise d'un *travail* est une décision applicative, qui a besoin de savoir ce que le travail a déjà fait.

6.2 Réessayer suppose l'idempotence

Reprendre un travail échoué semble naturel : la bande est perdue, on la relance. Mais relancer n'est sûr que si exécuter deux fois produit le même effet qu'une fois, ce que l'on appelle être *idempotent*. Calculer des pixels l'est : la fonction est pure, et un second calcul donne les mêmes octets. Envoyer un courriel, débiter une carte ou ajouter une ligne dans une base ne l'est pas. Le bout de code suivant relance naïvement deux fonctions ; la première rend la même valeur, la seconde a envoyé deux messages.

```
1 { :ok, sent } = Agent.start_link(fn -> 0 end)
2 send_mail = fn -> Agent.update(sent, &(&1 + 1)); :ok end
3 pixel = fn -> Enum.sum(1..100) end
4 retry = fn fun -> fun.(); fun.() end
5 { retry.(pixel), (retry.(send_mail); Agent.get(sent, & &1)) }
```

VALEUR {5050, 2}

Pourquoi ce souci est-il inévitable ? Parce qu'un `:DOWN` ou un délai dépassé ne disent pas *où* le travail s'est arrêté. Un travailleur a pu finir son calcul, envoyer son

résultat, puis mourir ; ou mourir avant ; ou être simplement lent. Le coordinateur qui relance peut donc recevoir deux résultats pour le même identifiant, d'où le *numéro de tentative* que le README suggère pour rejeter un résultat périmé. Les systèmes de messagerie nomment ces garanties : *au plus une fois* (on ne relance jamais, on peut perdre), *au moins une fois* (on relance, on peut dupliquer) et *exactement une fois*, qui n'existe pas sans un travail idempotent ou un registre des travaux déjà faits. Les files de travaux que nous allons voir choisissent « au moins une fois » et exigent l'idempotence. Deux raffinements complètent la panoplie : une *échéance* par travail, puisqu'un monitor détecte une mort et non un calcul qui ne progresse plus, et un *délai croissant* entre deux tentatives, pour ne pas assaillir une ressource qui vient justement de tomber.

EN PRATIQUE

Quand vous décrivez une panne dans votre compte-rendu, distinguez trois questions : comment l'avez-vous *détectée* (monitor, délai), ce que vous avez *décidé* (erreur locale, reprise, abandon), et ce que cela *suppose* du travail (pur ou non, résultat en double possible ou non). Les trois réponses tiennent en quelques lignes et valent plus qu'un graphique.

7 Au-delà du projet

Les idées de ce fascicule se retrouvent, presque telles quelles, dans trois familles de systèmes. Ce qui change à l'échelle, c'est que le coordinateur devient lui-même un service à protéger, et que les travailleurs vivent sur d'autres machines, dont les pannes se détectent moins bien qu'avec un monitor.

7.1 Les fermes de rendu, les vraies

Le mot vient du cinéma. Une *ferme de rendu* est un ensemble de machines qui calculent les images d'un film, image par image, souvent tuile par tuile, pendant des heures chacune. Pixar a publié en 1988 la spécification de l'interface RenderMan, qui décrit une scène pour un moteur de rendu, et *Toy Story*, en 1995, premier long métrage entièrement en images de synthèse, a été rendu avec le moteur correspondant sur une ferme de stations de travail. Aujourd'hui, Pixar vend avec RenderMan un gestionnaire de file nommé Tractor, qui répartit les tâches de rendu sur une ferme et que d'autres studios d'effets visuels utilisent aussi ; des outils libres comme OpenCue, rendu public en 2019 par Sony Pictures Imageworks et Google, jouent le

même rôle. On y retrouve une file de travaux avec des priorités, des dépendances entre tâches, une limite par machine et par licence de logiciel, la détection des machines mortes et la reprise des tâches interrompues. Votre coordinateur, avec sa file, sa limite et ses `:DOWN`, en est le modèle réduit.

7.2 MapReduce

En 2004, Jeffrey Dean et Sanjay Ghemawat, chez Google, ont décrit *MapReduce*, le cadre qui faisait alors tourner l'indexation du web sur des milliers de machines ordinaires. Le programmeur fournit deux fonctions pures : `map`, appliquée à chaque morceau de l'entrée, et `reduce`, appliquée à toutes les valeurs d'une même clé. Un processus maître découpe l'entrée, attribue les tâches aux travailleurs, et les réattribue quand une machine ne répond plus, exactement parce que les fonctions sont pures et donc idempotentes. L'article décrit aussi les *tâches de secours* : vers la fin d'un calcul, le maître relance les tâches les plus lentes sur d'autres machines et garde le premier résultat qui revient, car quelques machines à la traîne allongeaient sinon l'ensemble. C'est la bande la plus lourde de Mandelbrot, à l'échelle d'un centre de données. Hadoop, son implémentation libre, a suivi en 2006, et Flow, en Elixir, en reprend le vocabulaire.

7.3 Les files de travaux

Entre les deux, les applications web ordinaires utilisent des *files de travaux*. RabbitMQ, un courtier de messages publié en 2007, implémente le protocole AMQP : un producteur dépose des messages, des consommateurs les retirent, et un réglage, le *prefetch*, borne le nombre de messages confiés à un consommateur sans accusé de réception, ce qui est très exactement votre `limit`, appliquée par le courtier. Un message non acquitté quand le consommateur meurt est remis dans la file, « au moins une fois ». En Elixir, Oban, publié en 2019, garde ses travaux dans PostgreSQL : chaque file a une limite de concurrence, chaque travail un nombre maximal de tentatives, un délai croissant entre elles et un état visible, disponible, en cours, réessayable, terminé ou abandonné entre autres, que l'on peut consulter après coup. Les travaux survivent au redémarrage de l'application, ce que votre coordinateur ne promet pas.

Un dernier mot sur ce qui change quand les travailleurs sont sur une autre machine. La distribution d'Erlang étend les moniteurs d'un nœud à l'autre, mais une coupure réseau se présente alors comme un `:DOWN` de raison `:noconnection`, sans que l'on sache si le travailleur est mort ou seulement injoignable. La question « le travail a-t-il été fait ? » n'a plus de réponse sûre, et c'est tout le sujet du fascicule voisin sur les réservations.

8 Pour aller plus loin

Making reliable distributed systems in the presence of software errors, Joe Armstrong, thèse de doctorat, KTH (2003).

https://erlang.org/download/armstrong_thesis_2003.pdf

La source de « let it crash », écrite par son auteur, avec l'histoire du central AXD301. Les chapitres 2 à 5 se lisent sans connaître Erlang.

A history of Erlang, Joe Armstrong, HOPL III (2007).

<https://doi.org/10.1145/1238844.1238850>

Le récit de 1986 à 2006, du prototype en Prolog à la publication en logiciel libre, avec les choix qui ont été regrettés.

Validity of the single processor approach to achieving large scale computing capabilities, Gene Amdahl, AFIPS (1967).

<http://inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf>

Quatre pages, et la loi n'y est même pas écrite sous forme de formule : on la lit entre les lignes d'un argument commercial.

Reevaluating Amdahl's law, John Gustafson, Communications of the ACM (1988).

<https://doi.org/10.1145/42411.42415>

Deux pages pour renverser la question : ne pas accélérer le problème d'hier, mais agrandir celui de demain.

Scheduling multithreaded computations by work stealing, Robert Blumofe et Charles Leiserson, Journal of the ACM (1999).

<https://doi.org/10.1145/324133.324234>

L'article du vol de travail et de sa borne ; l'introduction et le modèle suffisent pour comprendre ce que font Cilk, Java et Rust.

MapReduce: simplified data processing on large clusters, Jeffrey Dean et Sanjay Ghemawat, OSDI (2004).

https://www.usenix.org/legacy/event/osdi04/tech/full_papers/dean/dean.pdf

Un maître, des travailleurs, des pannes et des tâches de secours : votre projet, à l'échelle d'un centre de données.

The BEAM Book, Erik Stenman.

<https://blog.stenmans.org/theBeamBook/>

Un livre libre sur la machine virtuelle ; le chapitre sur l'ordonnancement explique les réductions, les files par ordonnanceur et le vol de processus.

Task et GenStage, documentation, hexdocs.pm.

<https://hexdocs.pm/elixir/Task.html>

Les options d'`async_stream`, puis https://hexdocs.pm/gen_stage/GenStage.html pour voir la demande remonter d'un étage à l'autre.

9 Une question pour la route

Votre coordinateur lance les bandes dans l'ordre de la liste, et celles qui traversent l'ensemble de Mandelbrot sont bien plus lourdes que les autres. Si la plus lourde est lancée en dernier, que devient la durée totale, quelle que soit la limite ? Dans quel ordre lanceriez-vous les bandes si vous connaissiez leurs coûts, et que risqueriez-vous à découper l'image en bandes de plus en plus fines pour gommer la différence ? Ronald Graham a montré en 1969 que lancer d'abord les travaux les plus longs ne dépasse jamais l'optimal de plus d'un tiers ; trouver l'ordre optimal, lui, est un problème NP-difficile, et personne ne vous demande de le résoudre ce soir.

Pour toute question, vous pouvez me contacter par email, comme pour le reste du cours.