

Une place, un seul propriétaire

Courses, atomicité, jetons,
et ce que les bases de données en font

condition de course

atomicité

linéarisabilité

idempotence

consensus

Deux clients lisent « libre » au même instant. Lequel a raison, et surtout, qui décide ?

Un cadeau de fin de cours, à lire par curiosité : rien n'est à rendre. Le projet « Réservations concurrentes » 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 « réservations concurrentes » fait tenir une séance de cinéma dans un Gen-Server : chaque place est libre, retenue par un jeton ou confirmée avec ce jeton. Vous écrivez les trois transitions pures d'une place, puis les callbacks qui les appliquent dans le serveur, de sorte que vérifier et réserver se fassent dans le même traitement ; vous programmez l'expiration d'une réservation avec `Process.send_after`, et vous faites en sorte qu'une expiration périmée ne libère pas la place de quelqu'un d'autre. Le registre, le superviseur dynamique, l'API et les tests vous sont fournis.

Ce fascicule ne contient aucune des fonctions que vous devez écrire. Il explique en revanche pourquoi ce petit problème occupe les ingénieurs depuis les années 1970 : nous allons dérouler la course de deux clients sur une frise, donner le nom exact de la garantie que votre serveur offre (la *linéarisabilité*, définie en 1990), comprendre ce que sont les clés d'idempotence et pourquoi une minuterie n'est pas une horloge, lire les deux briques fournies, `Registry` et `DynamicSupervisor`, comme des patrons réutilisables, puis regarder comment une base de données et un système réparti répondent à la même question. Nous finirons par la billetterie réelle, où vendre plus de places qu'il n'en existe est un choix économique, et par les machines à états. Rien n'est à rendre : lisez par curiosité.

2 Posséder un état plutôt que le partager

2.1 La course « vérifier puis agir »

Une *condition de course* (*race condition*) est une situation où le résultat dépend de l'ordre dans lequel plusieurs acteurs s'exécutent, alors que cet ordre n'est pas maîtrisé. La forme la plus fréquente est « vérifier puis agir » : un client vérifie qu'une condition est vraie, puis agit comme si elle l'était encore, alors qu'un autre client a pu la changer entre-temps. La littérature l'appelle *TOCTOU*, *time of check to time of use*. Le souci n'est pas la rapidité des clients, ni l'absence de verrou au sens des langages à mémoire partagée. Il tient à la séparation en deux étapes. Reproduisons-le d'abord sans aucun processus, avec une map immuable que deux clients, Alice et Bob, consultent avant d'écrire :

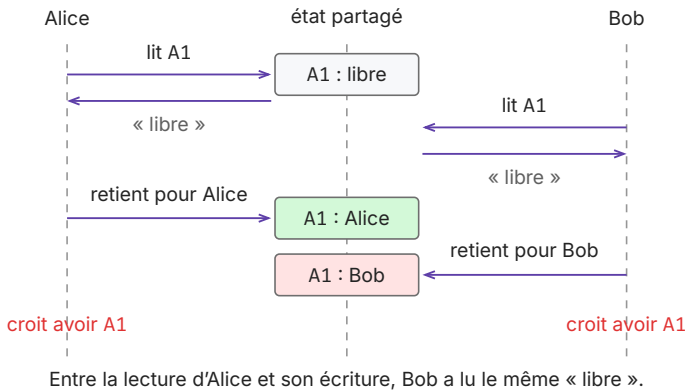
```

1 seats = %{ "A1" => :free }
2 free? = fn seats -> seats["A1"] == :free end
3 {alice_sees, bob_sees} = {free?.(seats), free?.(seats)}
4 seats = if alice_sees, do: %{seats | "A1" => {:held, :alice}}, else:
5   seats
6 seats = if bob_sees, do: %{seats | "A1" => {:held, :bob}}, else:
7   seats
8 {alice_sees, bob_sees, seats}

```

VALEUR {true, true, %{ "A1" => {:held, :bob}}}

Les deux ont vu « libre » ; la seconde écriture a gagné, et Alice ne le saura jamais. L’immuabilité d’Elixir n’y change rien : elle garantit qu’une valeur ne bouge pas, pas qu’une *décision* prise sur une ancienne valeur reste bonne. Suivons maintenant la même scène avec deux clients réellement concurrents, sur la frise ci-dessous ; le temps descend.

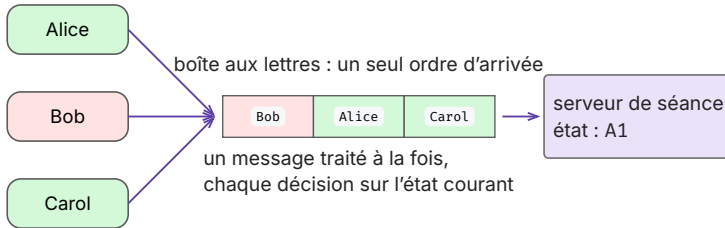


Les deux lectures précèdent les deux écritures. Chaque client a raisonné juste sur ce qu’il avait lu ; c’est l’état qui a changé sous ses pieds. La seconde écriture efface la première, et personne n’a reçu de refus.

2.2 Un seul propriétaire décide

La réponse du cours, depuis AP6, est de ne pas partager l’état : un seul processus le *possède*, et les autres lui envoient des demandes. Le chapitre 10 l’a dit pour un compteur ; disons-le maintenant comme un invariant. Appelons *atomique* un traitement qui lit l’état, décide et écrit le nouvel état sans qu’aucun autre traitement du même état puisse s’intercaler. Un GenServer offre cette atomicité gratuitement, pour une raison très simple : il exécute ses callbacks l’un après l’autre, dans l’ordre de sa boîte aux lettres. La propriété que nous voulons, « à tout instant, une place est

retenue par au plus un jeton », se démontre alors en deux lignes. Avant la première demande, elle est vraie : tout est libre. Supposons-la vraie avant un traitement. Celui-ci voit l'état courant, et non une copie ancienne ; s'il retient une place, c'est qu'il l'a trouvée libre à cet instant même, et aucun autre traitement ne s'est glissé entre sa lecture et son écriture. La propriété reste donc vraie après. Par récurrence sur les messages, elle est vraie toujours.



Bob est arrivé le premier : Alice et Carol recevront un refus.

Trois demandes pour `A1` arrivent dans la boîte du serveur. Il les traite dans l'ordre d'arrivée ; la première trouve la place libre, les deux autres la trouvent retenue. L'ordre n'est pas prévisible, l'exclusivité l'est.

Un `Agent` suffit pour voir la différence sur un exemple voisin. Voici un stock d'un billet, et trois clients qui tentent de l'acheter. La décision (« reste-t-il un billet ? ») et la mise à jour (« il en reste un de moins ») tiennent dans la même fonction, exécutée dans le processus de l'Agent :

```
1  {:ok, stock} = Agent.start_link(fn -> 1 end)
2  buy = fn who ->
3    Agent.get_and_update(stock, fn
4      0 -> {{:error, :sold_out}, 0}
5      n -> {{:ok, who}, n - 1}
6    end)
7  end
8  replies =
9    [:alice, :bob, :carol]
10   |> Task.async_stream(buy)
11   |> Enum.map(fn {:ok, reply} -> reply end)
12  {Enum.count(replies, &match?({:ok, _}, &1)), Agent.get(stock, &1)}
```

VALEUR {1, 0}

Un seul `{:ok, _}`, quel que soit l'ordre des trois tâches. Si nous avions écrit `Agent.get` pour lire le stock, puis `Agent.update` pour le diminuer, les deux requêtes d'un client auraient pu encadrer celles d'un autre, et le bout de code aurait repro-

duit la frise. Retenez bien que l'atomicité est une propriété du *traitement*, pas des données : une map immuable lue par deux clients ne protège rien.

2.3 Le nom de la garantie : linéarisabilité

En 1990, Maurice Herlihy et Jeannette Wing ont donné un nom précis à ce que fournit un tel propriétaire unique. Considérons l'*histoire* d'un objet partagé : la suite des appels, chacun avec son instant de départ et son instant de réponse. L'objet est *linéarisable* si l'on peut placer, pour chaque appel, un instant unique entre son départ et sa réponse, son *point de linéarisation*, tel que les résultats soient exactement ceux qu'aurait produits une exécution séquentielle des appels dans l'ordre de ces instants. En clair, chaque opération semble avoir eu lieu d'un coup, à un moment précis de sa durée. Votre serveur de séance est linéarisable, et le point de linéarisation est l'instant où le callback s'exécute. Les appels d'Alice et de Bob se chevauchent ? Les deux ordres sont permis, et le gagnant peut changer d'un essai à l'autre, ce que les tests du projet acceptent. Deux réponses « ok » pour la même place ne sont jamais permises, car aucune exécution séquentielle ne les produirait.

Herlihy et Wing ont montré une propriété précieuse : la linéarisabilité est *locale*. Si chaque objet, pris séparément, est linéarisable, alors le système entier l'est, sans coordination supplémentaire. C'est ce qui vous autorise une séance par processus : « cinéma » et « théâtre » n'ont pas besoin de se parler. Mais cette localité a une contrepartie : la garantie porte sur *une* opération sur *un* objet. « Deux places dans une même séance, les deux ou aucune » reste atomique, parce qu'un seul callback peut décider des deux ; « une place au cinéma et une au théâtre, les deux ou aucune » ne l'est plus, parce que deux propriétaires sont impliqués. Les bases de données nomment cette garantie-là, qui porte sur des groupes d'opérations, la *sérialisabilité*, définie par Christos Papadimitriou en 1979 : une exécution de transactions est sérialisable si elle produit le même résultat qu'une exécution des mêmes transactions l'une après l'autre, dans un ordre quelconque. Nous y revenons plus loin.

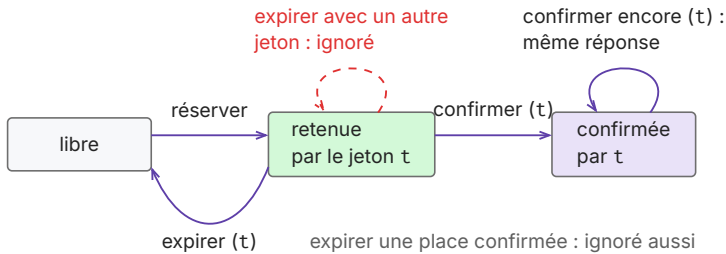
À RETENIR

Une course « vérifier puis agir » ne se corrige pas en allant plus vite, mais en réunissant la vérification et l'action dans un traitement que personne ne peut interrompre. Un processus propriétaire l'offre parce qu'il traite un message à la fois. La garantie obtenue s'appelle la linéarisabilité ; elle porte sur une opération et un objet, et elle se compose d'un objet à l'autre sans les coordonner.

3 Jetons, minuteriers et horloges

3.1 Ce qu'un jeton identifie

Le projet crée un jeton par réservation avec `make_ref()`. Pourquoi ne pas retenir simplement le nom du client ? Parce que le jeton ne désigne pas une personne mais une *tentative* particulière, et c'est ce qui permet de reconnaître un message en retard. Le scénario décisif est celui du README : T1 retient la place, T1 expire, T2 retient la place, puis une vieille expiration de T1 arrive. Si l'expiration ne portait que le nom de la place, elle effacerait T2. Avec le jeton, le serveur compare, constate que la place n'est plus retenue par T1, et ne fait rien.



L'automate d'une place. Les flèches pleines sont les transitions attendues ; la flèche rouge et la note grise sont les messages qui doivent être reconnus et ignorés. Expirer ne mène nulle part depuis « confirmée », et une seconde confirmation avec le bon jeton rend la même réponse.

Cette dernière propriété, une confirmation répétée qui rend la même réponse sans rien changer, est l'*idempotence* : appliquer l'opération deux fois revient à l'appliquer une fois. Elle a pris une importance considérable avec les API web, où un client qui n'a pas reçu de réponse ne sait pas si sa requête a été traitée, le chapitre 10 vous l'a montré avec le timeout de `GenServer.call`. L'API de paiement de Stripe, par ex., demande à ses clients d'envoyer avec chaque requête de paiement une *clé d'idempotence*, une chaîne unique choisie par le client ; si la même clé revient, le serveur rend la réponse mémorisée sans débiter une seconde fois. Un brouillon de standard de l'IETF, en cours de rédaction, propose d'en faire un en-tête HTTP commun, `Idempotency-Key`. Remarquez que ce n'est pas la même chose que votre jeton : le jeton est créé par le serveur et identifie une réservation acceptée ; la clé d'idempotence est créée par le client et identifie une *demande*, qu'elle ait été acceptée ou non. Le prolongement du README qui associe un identifiant de requête aux demandes est exactement cela, et sa difficulté est la mémoire : combien de temps le serveur doit-il se souvenir des clés déjà vues ?

3.2 Une minuterie n'est pas une horloge

`Process.send_after(pid, message, delay)` demande à la machine virtuelle de déposer `message` dans la boîte de `pid` après `delay` millisecondes. C'est un *délai d'envoi* : le message arrive dans la boîte après le délai, puis attend son tour derrière les messages déjà présents. Une boîte encombrée retarde donc le traitement de l'expiration, et une confirmation arrivée juste avant sera traitée avant. Le projet vous le fait dire dans le compte-rendu. Voici le même mécanisme sur un exemple voisin : une lampe qui s'éteint d'elle-même après un délai, et dont chaque pression du bouton relance le compte. Une seule génération compte, la dernière ; la minuterie de l'ancienne pression arrive quand même, et il faut la reconnaître :

```

1 press = fn delay ->
2   ref = make_ref()
3   Process.send_after(self(), {:off, ref}, delay)
4   ref
5 end
6 old = press.(20)
7 current = press.(40)
8 for _ <- 1..2 do
9   receive do
10     {:off, ^current} -> :extinction
11     {:off, ^old} -> :minuteur_perime
12   after
13     500 -> :timeout
14   end
15 end

```

VALEUR [:minuteur_perime, :extinction]

`Process.cancel_timer(ref)` aurait pu annuler l'ancienne minuterie, mais l'annulation ne retire pas un message déjà déposé dans la boîte : reconnaître le jeton reste nécessaire. Cette discipline, « rejeter par le jeton plutôt que compter sur l'annulation », est la bonne, parce qu'elle ne dépend pas de la vitesse relative des messages.

Reste la question de ce que mesure le délai. Une machine possède deux temps. L'*heure du système*, celle de `DateTime.utc_now/0`, est réglée par l'administrateur ou par le protocole NTP, et elle peut sauter, en avant comme en arrière, quand elle est corrigée. Le *temps monotone*, celui de `System.monotonic_time/1`, ne recule jamais et ne sert qu'à mesurer des durées ; il n'a aucun sens d'une machine à l'autre. Les minuterie de la BEAM reposent sur ce second temps, ce qui est la bonne décision : une correction d'horloge ne fera pas expirer vos réservations en avance. Mais la

documentation d'Erlang sur la correction du temps prévient d'un cas surprenant : sur beaucoup de systèmes, le temps monotone s'arrête quand la machine est mise en veille. Fermez le couvercle d'un portable sur un serveur de réservations, rouvrez-le une heure après, et les réservations retenues « pour dix minutes » le sont encore. En clair, un délai n'est pas une date. Le prolongement du README qui conserve une échéance et la vérifie à chaque confirmation vous demande justement de choisir laquelle des deux horloges fait foi, et quelle règle s'applique à la frontière.

TROIS PIÈGES D'HORLOGE

Comparer des instants pris sur deux machines différentes ; mesurer une durée avec l'heure du système, qui peut sauter ; et supposer qu'une minuterie de dix minutes garantit un traitement dix minutes plus tard. Le délai borne l'envoi, pas le traitement ; le temps monotone est local ; et seule une échéance explicite, conservée dans l'état et comparée au moment de décider, donne une règle que l'on peut énoncer.

4 Un processus par séance

Le squelette vous fournit deux briques, `Registry` et `DynamicSupervisor`, et vous demande de les expliquer. Elles méritent d'être lues comme des patrons, car on les retrouve dans presque toutes les applications Elixir.

Un processus par entité, ici par séance, est le patron de base : il donne la linéarisabilité par séance, et le parallélisme entre séances, sans rien coordonner. Deux questions suivent aussitôt. Comment retrouver le processus d'une séance à partir de son nom ? Et qui le crée, puisque les séances apparaissent après le démarrage de l'application ? `Registry` répond à la première : c'est un annuaire, lui-même un ensemble de processus, qui associe une clé, `"cinema"`, à un PID, et qui retire l'entrée quand le processus meurt, grâce à un `monitor`. Le tuple `{:via, Registry, {Reservations.Registry, "cinema"}}` est un nom que `GenServer.call` sait résoudre : chaque appel consulte l'annuaire. `DynamicSupervisor` répond à la seconde : c'est un superviseur sans liste d'enfants prédéfinie, auquel on demande des démarrages au fil de l'eau, et qui applique à chacun la recette et la politique de redémarrage que vous connaissez. Le bout de code ci-dessous reconstruit ce couple avec deux `Agent`, hors de tout projet :

```

1 { :ok, _ } = Registry.start_link(keys: :unique, name: Demo.Registry)
2 { :ok, sup } = DynamicSupervisor.start_link(strategy: :one_for_one)
3 via = fn id -> { :via, Registry, { Demo.Registry, id } } end
4 for id <- ["cinema", "theatre"] do
5   spec = %{:id: id, start: { Agent, :start_link, [fn -> 0 end, [name:
6     via.(id)]]}}
7   { :ok, _pid } = DynamicSupervisor.start_child(sup, spec)
8 end
9 Agent.update(via.("cinema"), &(&1 + 1))
10 [{pid, nil}] = Registry.lookup(Demo.Registry, "cinema")
11 { Agent.get(via.("cinema"), & &1), Agent.get(via.("theatre"), & &1),
    is_pid(pid), DynamicSupervisor.count_children(sup).active }

```

VALEUR {1, 0, true, 2}

Deux séances, deux états indépendants, un annuaire qui retrouve le PID. Que se passe-t-il si l'on demande une seconde fois le démarrage de "cinema" ? L'annuaire refuse un nom déjà pris et le superviseur répond `{:error, {:already_started, pid}}` : la clé unique du registre est, elle aussi, une décision atomique, prise par le processus qui possède la table des noms.

Ce patron a une limite, que le projet vous fait observer en tuant un serveur : le superviseur relance le processus avec sa recette initiale, et l'annuaire enregistre le nouveau PID, mais l'état est perdu, et une confirmation précédente avec lui. La supervision restaure un *processus*, pas une *mémoire*. Pour qu'une réservation survive, il faut l'écrire ailleurs, et c'est là que les bases de données entrent en scène.

5 Ce que font les bases de données

Les bases de données relationnelles affrontent la même course depuis les années 1970, avec des milliers de clients et un état qui doit survivre à tout. Leurs réponses éclairent la vôtre.

5.1 Transactions et verrous

Une *transaction* regroupe plusieurs opérations en une unité qui réussit ou échoue en bloc. Theo Härder et Andreas Reuter ont résumé en 1983 ses garanties par l'acronyme ACID : atomique, cohérente, isolée, durable. L'*isolation* est la version base de données de notre atomicité : deux transactions concurrentes ne doivent pas se voir à moitié. La manière la plus directe de l'obtenir pour une réservation est un

verrou *pessimiste* : on demande la ligne de la place en annonçant qu'on va la modifier, et tout autre client qui la demande de la même façon attend que la transaction se termine.

```
1 BEGIN;
2 SELECT state FROM seats WHERE id = 'A1' FOR UPDATE;
3 -- la ligne est verrouillée jusqu'au COMMIT : l'autre client attend
  ici
4 UPDATE seats SET state = 'held', token = 't1' WHERE id = 'A1';
5 COMMIT;
```

Le `FOR UPDATE` transforme le « vérifier puis agir » en « verrouiller, vérifier, agir, déverrouiller ». Le second client, libéré après le `COMMIT`, relit la ligne et la trouve retenue. C'est votre callback, avec la base dans le rôle du propriétaire unique, et la file des clients bloqués dans le rôle de la boîte aux lettres.

5.2 Le verrouillage optimiste

H. T. Kung et John Robinson ont proposé en 1981 une autre stratégie, dite *optimiste* : ne rien verrouiller, laisser chacun travailler sur sa lecture, et vérifier au moment d'écrire que personne n'est passé entre-temps. La forme la plus répandue est un *numéro de version* sur chaque ligne : on lit la place avec sa version, 7 par ex., et l'on écrit en exigeant que la version soit encore 7.

```
1 UPDATE seats SET state = 'held', token = 't1', version = 8
2 WHERE id = 'A1' AND version = 7;
3 -- 1 ligne modifiée : gagné. 0 ligne : quelqu'un est passé avant, on
  relit.
```

Un seul `UPDATE` atomique décide : la base compare et écrit sans que rien ne s'intercale, exactement comme votre serveur. Le perdant n'attend pas, il apprend qu'il a perdu et recommence. L'optimisme paie quand les conflits sont rares, ce qui est le cas de la plupart des places d'une salle ; il coûte quand tout le monde veut la même, où le verrou pessimiste fait attendre plutôt que recommencer. En Elixir, `Ecto` propose les deux : une requête peut porter `lock: "FOR UPDATE"`, et `Ecto.Changeset.optimistic_lock/3` ajoute la comparaison de version à une mise à jour, en levant `Ecto.StaleEntryError` si elle échoue. `Repo.transaction/1` enveloppe le tout, et `Ecto.Multi` compose plusieurs écritures, ce qui répond à « deux places, les deux ou aucune » quand elles vivent dans la même base.

Les niveaux d'isolation de SQL complètent le tableau : avec le niveau `SERIALIZABLE`, la base promet la sérialisabilité de Papadimitriou pour l'ensemble de la transaction, et c'est elle qui détecte les conflits et annule un des participants. PostgreSQL

l'implémente depuis sa version 9.1, en 2011, par une technique dite d'isolation sérialisable par instantanés, qui est, elle aussi, optimiste.

EN PRATIQUE

Si vous deviez brancher votre projet sur une base, la décision atomique quitterait le callback pour rejoindre la transaction, et le GenServer deviendrait surtout un lieu pour la minuterie et les abonnés. Ce déplacement est fréquent : l'important n'est pas *où* se trouve le propriétaire unique, mais qu'il n'y en ait qu'un.

6 Un seul chef, ou un consensus

Votre serveur de séance est un *chef unique* : une décision, un processus. C'est simple et linéarisable, et cela a deux faiblesses que le projet vous fait toucher du doigt. Si le processus meurt, l'état meurt avec lui ; et tant qu'il est injoignable, personne ne réserve. Les systèmes répartis, où l'état doit survivre à la perte d'une machine, affrontent ces deux faiblesses à la fois, et le prix à payer est considérable.

6.1 Deux généraux

Commençons par une impossibilité. Deux généraux campent de part et d'autre d'une vallée tenue par l'ennemi et veulent attaquer ensemble, à la même heure ; ils ne communiquent que par des messagers qui peuvent être capturés. Le premier envoie « attaquons à l'aube ». A-t-il été reçu ? Le second renvoie un accusé ; mais cet accusé peut se perdre, et le second ne sait pas si le premier l'a reçu ; un accusé de l'accusé pose la même question, sans fin. Aucun protocole en un nombre fini de messages ne permet aux deux de savoir avec certitude qu'ils sont d'accord. Le résultat a été démontré en 1975 par Akkoyunlu, Ekanadham et Huber, et Jim Gray lui a donné en 1978 le nom qu'il porte. Pour vous, la conséquence est celle du chapitre 10 : un client dont l'appel expire ne sait pas si sa réservation a été prise, et le serveur ne sait pas si sa réponse est arrivée. Ni l'un ni l'autre ne peut conclure. La seule issue est de rendre la répétition inoffensive, par un jeton ou une clé d'idempotence, et c'est pourquoi ces outils reviennent dans chaque section.

6.2 Paxos et Raft

Comment garder un propriétaire unique quand une machine peut tomber ? En le répliquant : plusieurs machines tiennent une copie de l'état et s'accordent, décision après décision, sur l'ordre des opérations, de sorte que l'ensemble se comporte comme un seul serveur. Cet accord s'appelle le *consensus*. Leslie Lamport l'a résolu dans un article écrit en 1989, déguisé en récit archéologique sur le parlement d'une île grecque imaginaire, que les relecteurs n'ont pas compris et qui n'a été publié qu'en 1998 : c'est *Paxos*. L'algorithme tolère la panne d'une minorité de machines et garantit que deux machines ne décideront jamais deux valeurs différentes pour la même place dans l'ordre. En 2014, Diego Ongaro et John Ousterhout ont publié *Raft*, conçu pour être compréhensible : une machine est élue *chef* pour un mandat, elle seule reçoit les demandes des clients, et elle réplique un *journal* d'opérations sur une majorité de machines avant d'appliquer chacune. Si le chef meurt, un nouveau est élu, et le journal garantit qu'il connaît tout ce qui a été décidé. Raft est au cœur d'etcd, le magasin de configuration de Kubernetes. Vous reconnaissez votre serveur de séance dans le chef de Raft ; ce qui s'ajoute, c'est la majorité à convaincre avant chaque réponse, et l'élection quand il disparaît.

6.3 Le théorème CAP, prudemment

En 2000, Eric Brewer a conjecturé qu'un service réparti ne peut pas offrir à la fois la cohérence forte, la disponibilité de chaque demande et la tolérance aux coupures du réseau ; Seth Gilbert et Nancy Lynch l'ont démontré en 2002, en prenant pour « cohérence » précisément la linéarisabilité. Le résultat est plus étroit que son slogan « deux sur trois ». Il dit que *pendant* une coupure qui sépare les machines en deux groupes, chaque groupe doit choisir : répondre quand même, au risque de vendre la même place deux fois, comme les répliques du compteur d'AP8 ; ou refuser, pour préserver l'exclusivité. Hors coupure, le compromis réel se joue entre latence et cohérence, et Brewer lui-même a rappelé en 2012 combien le slogan avait été mal compris. Pour une billetterie, la réponse est presque toujours la même : on préfère refuser pendant quelques secondes que vendre deux fois, et les systèmes de consensus sont faits pour cela.

POUR EN PARLER

Un chef unique est linéarisable mais fragile ; le consensus réplique le chef sans renoncer à l'exclusivité, au prix d'une majorité à convaincre. Les deux généraux interdisent la certitude après un timeout ; les jetons et les clés d'idempotence rendent la répétition inoffensive. CAP parle du comportement pendant une coupure, pas d'un choix permanent.

7 Au-delà du projet

7.1 Billetterie et surréservation

Les grandes ventes de billets ressemblent à votre test à cent clients, avec un million de clients et une seule machine qui ne suffit plus. Les billetteries y ajoutent une *file d'attente virtuelle* : les clients sont admis par lots vers la page de réservation, ce qui borne le nombre de demandes concurrentes, une contre-pression appliquée aux humains. La réservation elle-même ressemble à la vôtre : une place est retenue pour quelques minutes, le temps de payer, puis libérée si le paiement n'arrive pas. L'état « retenue » et sa minuterie sont une pratique universelle, et la reconnaissance des expirations périmées un bug classique des premières versions.

Les compagnies aériennes font autre chose, qu'il faut distinguer : elles vendent volontairement plus de billets qu'il n'y a de sièges. Ce n'est pas une course perdue, c'est un calcul. Une part mesurée des passagers ne se présente pas, et un siège vide au décollage est perdu pour toujours ; la *surréservation* parie sur ce taux pour remplir l'avion, en acceptant de payer les rares passagers refusés à l'embarquement. La recherche opérationnelle l'étudie depuis les années 1950, Marvin Rothstein en a fait une synthèse en 1985 dans la revue *Operations Research*, et le règlement européen 261/2004 fixe les compensations dues en cas de refus d'embarquement. L'invariant de votre projet, « au plus un propriétaire par place », y est donc remplacé par « au plus n passagers pour m sièges », avec $n > m$ choisi par un modèle économique. La leçon est que l'invariant n'est pas une évidence technique : c'est une décision de l'entreprise, que le code doit ensuite tenir sans faille.

7.2 Les machines à états

Votre place a trois états et quelques transitions, et vous avez sans doute dessiné son automate avant d'écrire `seats`. Ce réflexe a un nom et des outils. Une *machine à états* décrit un objet par ses états possibles et par les événements qui le font passer de l'un à l'autre ; tout événement non prévu dans l'état courant est ignoré ou refusé, ce qui est exactement le sort de l'expiration périmée. OTP fournit depuis 2016, avec sa version 19, le comportement `gen_statem`, un cousin de `GenServer` dont les callbacks sont indexés par l'état courant : on écrit « dans l'état retenu, sur l'événement confirmer » plutôt qu'un `case` dans un `handle_call`. Les systèmes de billetterie et de paiement réels ajoutent des états que votre prototype n'a pas, « expirée » et « annulée » notamment, non pour la logique mais pour la trace : savoir qu'une place a été retenue puis libérée est une information que l'on veut garder, pour le service client

comme pour l'audit. La transition d'un état à l'autre est alors une ligne écrite dans une base, et la décision atomique devient une transaction. Vous avez, en quelques centaines de lignes, construit le cœur de cette machine ; le reste est de la persistance, de l'authentification et des protocoles de reprise, et ce n'est pas peu.

8 Pour aller plus loin

Linearizability: a correctness condition for concurrent objects, Maurice Herlihy et Jeannette Wing, ACM TOPLAS (1990).

<https://cs.brown.edu/~mph/HerlihyW90/p463-herlihy.pdf>

La définition, les exemples de files et la preuve que la linéarisabilité est locale. Les sections 1 à 3 suffisent, et la file d'attente de l'introduction est un cousin de votre place.

On optimistic methods for concurrency control, H. T. Kung et John Robinson, ACM TODS (1981).

<https://doi.org/10.1145/319566.319567>

L'origine du verrouillage optimiste et du numéro de version : lire, travailler, valider au moment d'écrire.

Paxos made simple, Leslie Lamport (2001).

<https://lamport.azurewebsites.net/pubs/paxos-simple.pdf>

Quatorze pages sans aucune formule, écrites après que personne n'eut compris le parlement de Paxos ; le résumé commence par « The Paxos algorithm, when presented in plain English, is very simple ».

In search of an understandable consensus algorithm, Diego Ongaro et John Ousterhout, USENIX ATC (2014).

<https://raft.github.io/raft.pdf>

Raft, avec son chef élu et son journal répliqué ; le site raft.github.io propose une animation qui vaut une heure de lecture.

Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, Seth Gilbert et Nancy Lynch, ACM SIGACT News (2002).

<https://doi.org/10.1145/564585.564601>

La démonstration de CAP, courte, et surtout ses définitions : vous y verrez que la « cohérence » du théorème est la linéarisabilité de ce fascicule.

Time and time correction in Erlang, documentation d'Erlang/OTP.

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

Les deux temps de la machine virtuelle, ce que fait la VM quand l'horloge saute, et l'avertissement sur la mise en veille.

Explicit locking, documentation de PostgreSQL.

<https://www.postgresql.org/docs/current/explicit-locking.html>

Ce que `FOR UPDATE` verrouille au juste, et les autres verrous de ligne ; à lire avec le chapitre voisin sur les niveaux d'isolation.

Designing Data-Intensive Applications, Martin Kleppmann, O'Reilly (2017).

<https://dataintensive.net/>

Le livre à lire après ce cours ; les chapitres 7 (transactions) et 9 (cohérence et consensus) reprennent tout ce fascicule avec les systèmes réels.

9 Une question pour la route

Un client veut une place au cinéma *et* une au théâtre, les deux ou aucune, et les deux séances sont deux serveurs. Vous pourriez demander à chacun de retenir sa place, puis confirmer les deux ; mais si votre client meurt entre les deux confirmations, ou si le théâtre a répondu « indisponible » après que le cinéma a déjà confirmé, dans quel état sont les deux séances ? Les bases de données réparties répondent par la *validation en deux phases*, décrite par Jim Gray en 1978 : un coordinateur demande à chacun « êtes-vous prêt ? », puis annonce la décision à tous. Cherchez ce qui se passe si c'est le coordinateur qui meurt entre les deux phases, et combien de temps les places restent bloquées. Vous retrouverez les deux généraux, et vous comprendrez pourquoi tant de systèmes préfèrent une place par séance et un client qui réessaie.

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