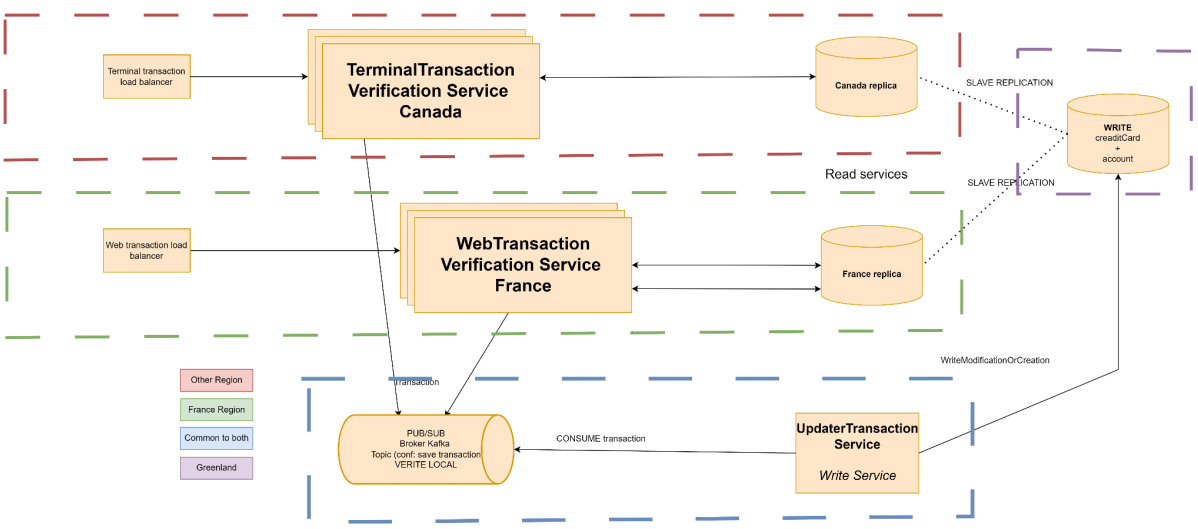
Rapport de suivi d'architecture
AL Néo-Bank - Evolution

Final Rapport

Architecture Construction :

Objectif Mis en perspective par le sujet



Lors du module d'architecture Construction, nous avions désigné une architecture du système de paiement de la néo-bank qui avait pour objectif :

- **D'être hautement disponible** : En effet, le système doit être disponible à nos clients un maximum de temps, sur le plan conceptuel. Nous n'avons pas pris en compte la possible tolérance aux pannes infrastructurelles (SLA).
- **D'être Scalable :** Notre système devait pouvoir prendre en charge un grand nombre de requêtes de paiement en même temps.
- **D'être résilient** : Dans une perspective de haute disponibilité; il fallait que notre système soit capable de revenir à un état stable en cas de faute.

Ainsi nous avons finalement monté une architecture qui permette de réaliser des transactions dans le cadre d'un paiement, tout en restant fidèle aux attentes du sujet.

Problèmes liés au scaling ⇒ Solutions Apportées

Deux problèmes principaux liées à la montée en charge devaient être pris en charge en priorité, à savoir :

Le fait qu'en cas de montée en charge, notre service Spring renvoyait un grand nombre d'erreurs mémoire. En effet, chaque flux de paiement créait de la ressource, et l'appel au service ayant lieu plus rapidement que le passage du garbage collector en mémoire, il n'y avait plus de place.
⇒ Afin de prendre en charge ce problème, nous avions mis en place un scaling horizontal.

Le fait qu'un grand nombre de connexions (lecture/écriture) sur une même base de données provoquait un grand nombre de timeouts, dû aux limites de connexions simultanées à la base de données PostGres ⇒ C'est pour cela que nous avons mis en place un système Master/Slave.

Evolution

Dans le contexte de l'évolution de notre système, certains point ont été relevés, et devaient être retravaillés, qui sont :

- **Algorithme réaliste de vérification des CBs**

  En effet, dans le contexte de notre démonstration, nous avons réduit la complexité finale de la vérification d'une carte bancaire afin de la simplifier.
  L'objectif de cette évolution est de rallonger le temps de vérification dans le cas d'un paiement, ce qui impacte forcément notre design de scalabilité qui sera alors à revoir.

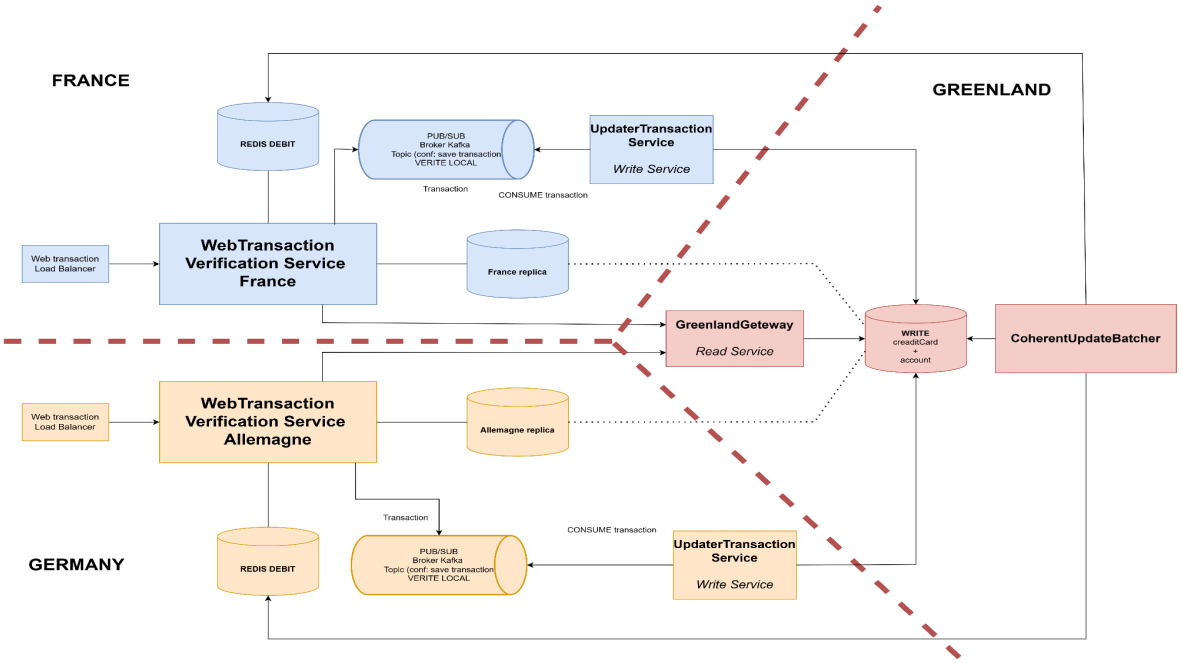
- **Lever l'hypothèse de temporisation de entre transactions**

  Lever l'hypothèse implique qu'un client peut enchaîner plusieurs transactions sur un court moment. En effet, cela n'était pas pris en charge par notre système, car un enchaînement de paiements pour un même client avait de fortes chances de provoquer des erreurs de cohérence.

- **Failover sur une région connexe**

  En effet, nous avons déployé notre solution entière dans chaque région à chaque fois, mais en cas de failure sur toute la région, pour une raison x. Il fallait que notre système soit disponible. D'où le failOver.

Architecture Evolution



Voici notre architecture finale dans le cadre du développement des évolutions.

En plus des évolutions, nous avons accordé une grande importance à allier la cohérence des données et la disponibilité. Nous avons donc percuté de plein fouet les limites exposées par le théorème CAP.

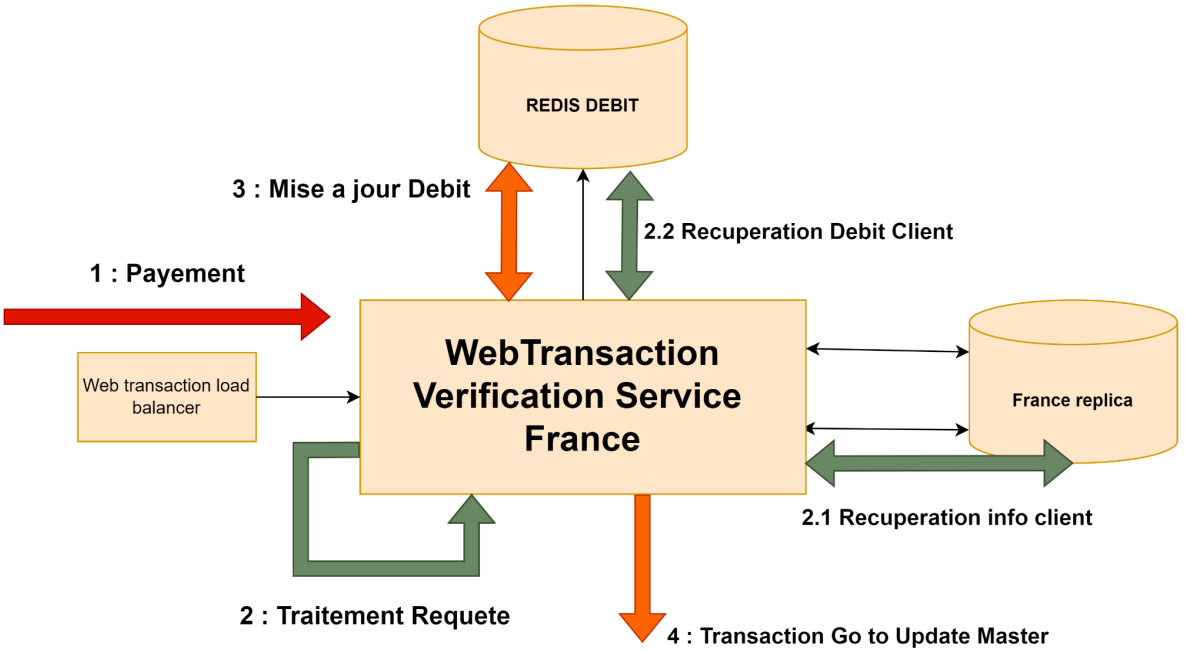
En effet, il était très difficile de trouver la bonne solution, celle qui répond à tous nos problèmes, car chaque nouvelle solution apportée, dans le cadre de la cohérence ou de la disponibilité, provoquait strictement un problème respectivement dans le cadre de la disponibilité/résilience ou dans la cohérence.

Nous avons aussi déduit que la seule solution qui permet de forcer répondre à la problématique des deux cas et en effet un scaling vertical des machines portatives de la solution. En effet, dans le cadre de banque “réelle”, le matériel “hardware” joue un grand rôle dans la haute disponibilité du système et la cohérence des données.

Cependant, nous avons pour chaque type de problème imposé par les évolutions, nous avons implémenté des solutions nous permettant de nous rapprocher au maximum des attentes.

Solution Evolution

Multitude de requêtes instantanées :



Notre système de transaction devrait désormais :

- Permettre à un grand nombre de clients de réaliser un paiement en même temps.
- Assurer que les paiements soient réalisés rapidement.
- Garantir que le client réussisse son paiement dans un temps imparti (*c'est à dire que si le client a 3 secondes, et que la première tentative échoue au bout d'une seconde, alors nous avons toujours 2 secondes pour effectuer une nouvelle tentative*), et que l'erreur de paiement lui soit explicitée.
- Permettre à un même client de faire plusieurs transactions d'affilée.

Afin que tout cela soit pris en compte, nous avons changé notre manière de faire. En effet, maintenant dans le cadre d'une transaction, nous avons ajouté un concept qui est le "Débit'. Le débit permet de stocker l'argent ayant été dépensé par un client. Dans le cadre de la vérification de la possibilité d'effectuer un achat, nous faisons la différence entre le compte en banque et le débit, et regardons si le client, après achat, passe dans le négatif ou non.

Nous maintenons le débit dans une base de données Redis. Nous propageons aussi la valeur du nouveau débit vers notre base de données client au Groenland, étant notre source de vérité.

Le fait de ne plus modifier directement le compte mais de passer par un débit, et de modifier le comptes plus tard sur un timing de faible utilisation, (*via un batch process qui fait la mise à jour des comptes clients, passant les débits à 0, et le nouveau solde à l'ancien moins le débit*), permet au client de pouvoir enchaîner les transactions de paiement. Cela casse notre hypothèse des 5 minutes.

Utilisation de Redis

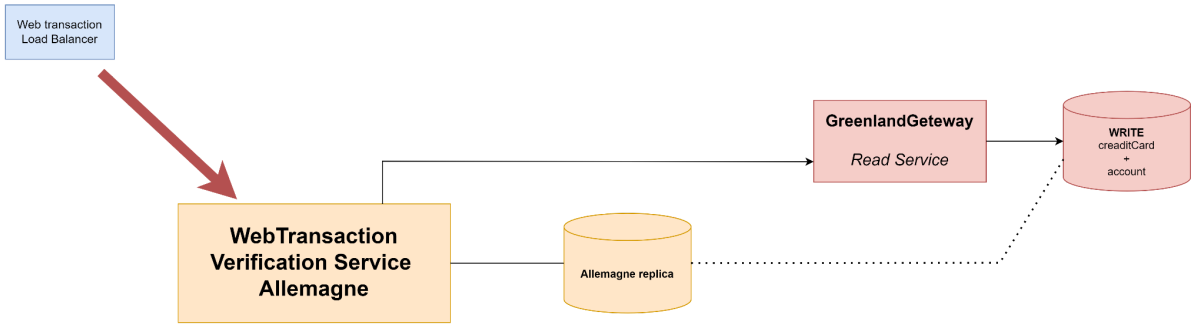
Nous avons choisi d'utiliser REDIS car cette base de données agit en mémoire sur la RAM, et nous permet de lier fortement nos instances de vérification de transaction à REDIS, permettant des interactions rapides avec celle-ci.

En effet, nous effectuons des lectures et écriture sur REDIS, dans ce context, les autres base de donnée sont plus lentes dans l'accès aux données, et surtout posent des limites de connexions simultanées, en effet, l'élasticité de scaling horizontal de nos instances pourrait provoquer des erreurs si connectées à d'autres types de base de données (majoritairement SQL) en lecture et écriture.

Le Drawback principale et que, en cas démarrage d'une nouvelle instance de manière dynamique par exemple (Difficile à tester avec docker-compose), celle-ci prend du temps à se connecter à Redis.

Aussi, le second Drawback est qu'en cas de panne de redis, notre service devient temporairement inutilisable. C'est pourquoi nous avons mis en place un hot replica. Mais le principal souci et que, de manière générale, on préfère avec docker-compose lancer un cluster redis sentinel, qui gère la redistribution des requêtes vers Redis. Du point de vue des instances, celles communiquant avec Redis, même si une base de données du cluster tombe, l'autre sera utilisée de manière transparente, sentinel gérant la relation master-master. Mais pour revenir au problème de sentinel, avec docker-compose, une fois lancé, il n'est pas possible de manière dynamique de lier une nouvelle instance. Et l'accès aux données est nettement plus long.

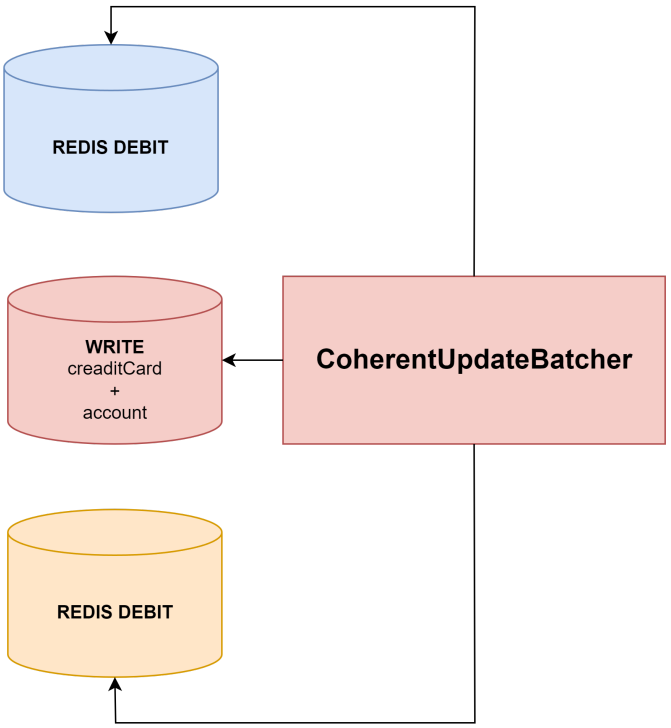
FailOver



Dans le cas où le load balancer de la région France, par exemple, ne trouve plus d'instance vers qui envoyer une requête, alors celui-ci va renvoyer la requête vers une instance mais d'une autre région voisine, à savoir ici l'Allemagne. Mais alors, comment cela pourrait se faire si les informations du client sont propres à la région France ?

Nous avons mis en place au niveau du Groenland, un gateway qui permet aux instances allemandes relais, de récupérer les informations clients depuis le Groenland. En effet, nous centralisons l'ensemble des informations de nos clients au niveau du Groenland.

Batch Process : mise à jour cohérente



En effet, suite à l'ajout du débit, il nous fallait avoir un processus nous permettant de remettre les compteurs à zéro pour retrouver un état stable.

Le CoherentUpdateBatcher, de manière automatique dans la nuit, à un moment de très faible utilisation, effectue la remise à zéro des débits.

Cependant, la mise en œuvre de ce processus a un coût en termes de disponibilité. En effet, cela permet de retomber sur un état plus stable de notre système, mais pendant que le service de mise à jour cohérente fonctionne, un client ne peut techniquement pas faire de paiement, sinon il y a un fort risque d'incohérence.

En effet, l'écriture du nouveau débit ayant lieu de manière asynchrone dans le master, nous pourrions avoir des cas où un client a un débit à 0 sur redis et à 10 euros, par exemple, sur master.

La première solution a été de mettre en place un “circuit breaker” qui effectue les recalls seul sur une plage de 3 secondes, en cas d’erreur liée au fait que le client soit en cours de batch process. Cela impacte la disponibilité de notre système.

Une seconde solution ayant été choisie et implémentée, est de modifier la structure de données que l’on stocke dans REDIS, passant d’une clé valeur sur un simple integer, vers maintenant une liste FIFO, et de modifier légèrement la logique du service, mais sans changer le business de celui-ci. En effet, le FIFO nous permet de retirer un nombre prédéfini de débit de l’historique des débits du client, ainsi même si pendant le processus, le client fait un paiement, dans la structure on passe à 5 débits, mais le batch étant fixé à 4 au début du processus, ne retire que 4 débits parmis les plus anciens. Cela nous permet de garder le

débit associé au paiement ayant eu lieu après que le batch process ait été enclenché pour le client.

Faiblesses de l’architecture

Problème de scalabilité lié au FailOver :

Tout d’abord, le premier problème, qui est le plus évident, est que si un grand nombre d'utilisateurs allemands et français effectuent des paiements au même moment, tous en partant du principe que nous sommes en état de failOver sur la région france, les instances allemandes devront traiter un bien plus grand nombre de requêtes, ce qui n’est pas le soucis en soi, mais elles devront fortement sollicité le gateway situé au Groenland et cela pose deux grand soucis:

- En effet, le gateway vers le Groenland ne fait que lire sur master, mais notre service de mise à jour des transactions lui effectue une écriture afin de mettre à jour la base de données master sur toutes les transactions qui ont eu lieu. Cela crée une très forte augmentation du nombre de connexions vers la base de données master, ce qui résulte sur un certain nombre de timers dans le cadre du paiement failOver.

  Ce n’est pas le cas pour les paiements internes aux régions, car kafka permet de temporiser le traitement de la requête de mise à jour.

- Ensuite, tout simplement, si le Gateway reçoit 5000 requêtes d’un coup, il ne pourra traiter qu’une faible partie. En effet, REST ne permet pas d'attendre que la requête se réalise pour passer à la suivante. Ainsi, à l'image de notre service de mise à jour de master, nous devrons passer par kafka.

Problème de cohérence suite au failOver :

Un client ayant effectué un paiement failOver, ne mettra pas à jour son débit dans le REDIS de sa région. Ainsi, une fois les instances de la région de nouveau fonctionnelles, il y aura un déphasage entre la valeur du débit dans le REDIS et celle dans le Master.

Actuellement, nous acceptons que le client puisse passer dans le rouge avec une transaction de différence, car son passage dans le négatif sera de petite valeur. Mais cela pose quand même un très grand problème de cohérence.

Nous avons envisagé des solutions :

- Au redémarrage de la région, il faut avoir un processus qui permet de remettre au même niveau les informations dans Redis et Master. Deux choix s'offrent à nous :
    - Ne pas maintenir la valeur du débit, et donc utiliser le batch process afin de remettre les comptes à 0.
    - Maintenir la valeur du débit, et donc implémenter une pipeline kafka qui stockera l'ensemble des événements de paiement failOver sur la région, et qui sera utilisée pour rejouer les événements de paiement simplement dans un contexte de remplissage de REDIS pour qu'il reviennent au même niveau que Master.

Dans les deux cas, nous gagnons fortement en cohérence de notre système, mais perdons fortement en disponibilité, étant donné que les processus envisagés prennent un certain temps.

Au niveau du choix, nous avons à peser la simplicité contre la disponibilité. En effet, la première solution est déjà implémentée, il ne faut que d'effectuer l'appel du service de batch au moment de la relance de la région.

La seconde solution, à implémenter, implique une plus grande utilisation de ressources, étant donné qu'elle nécessite kafka, donc zookeeper au groenland. Avec un nouveau service dans toutes les régions permettant de réaliser le business. Mais, étant donné qu'il n'y a qu'une seule interaction avec Redis, le processus est plus rapide.

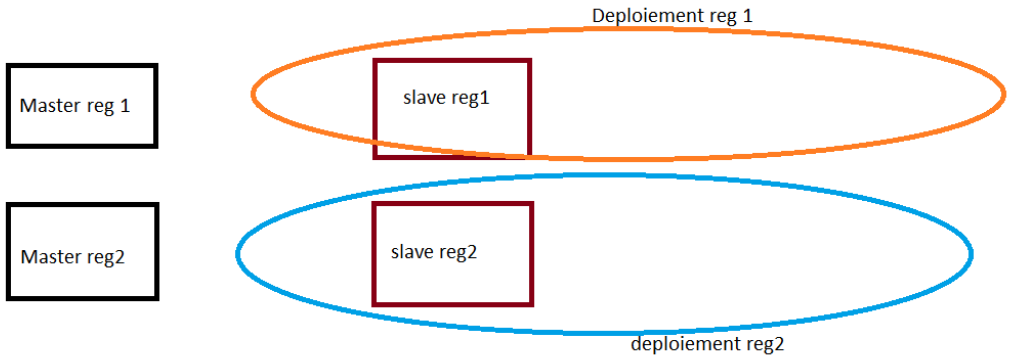
Nous avons finalement décidé de garder la première solution, car le batch process est déjà en place, et nous avons déjà envisagé comment l'optimiser. En effet, le gain de quelques secondes ne justifie pas une perte de temps et de ressources.

Week 50

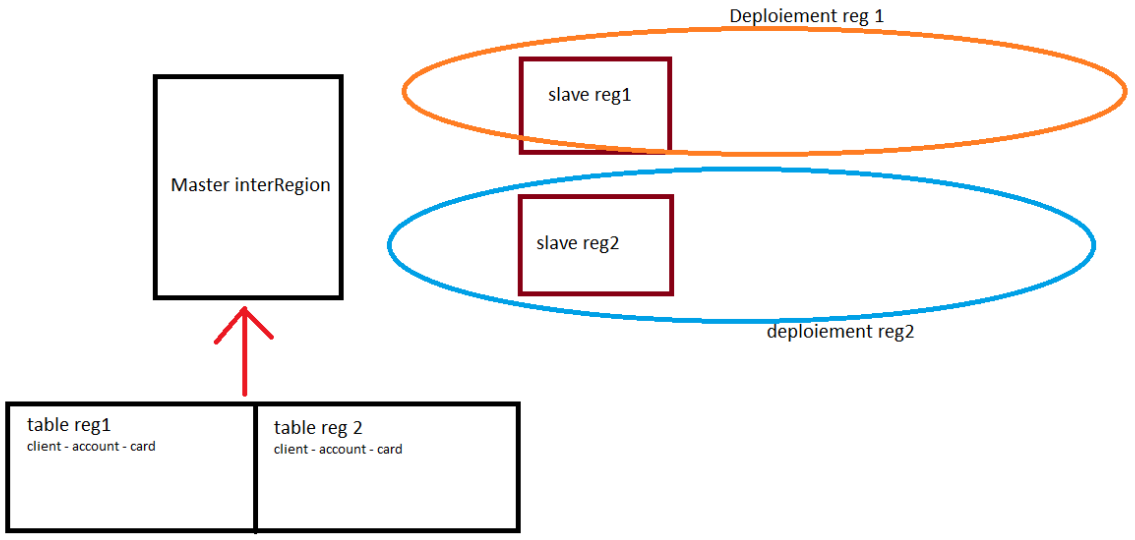
Base de donnée au groenland :

Nous avons décidé, dans le cadre de la réalisation du fail over de revoir notre manière de déployer plusieurs régions différentes. Et notamment comment nous allons organiser les données de nos clients.

Avoir plusieurs Master au niveau du groenland:



Avoir une unique Database Master au niveau du groenland :



Etude des solutions apportées

Solution des Master séparés

Dans chacun des cas, les données client seront séparés en fonction de leur région,
Dans le premier cas, l'on aurait une base de données par région nouvellement déployées. Dans lequel les informations des clients seraient .

L'avantage est que le coût des bases des données serait moindre, car un scaling horizontal est effectué.

Cependant, **l'inconvénient** est que, au niveau de la cohérence d'une donnée qui apparaît sur les deux bases de données qui ne sont pas liées d'une manière ou d'une autre poserait des soucis, en effet un client qui apparaît dans deux régions pourrait poser un souci.
Aussi, Il faudra donc implémenter une couche de logique au niveau du groenland pour monitorer ce "pseudo-cluster".

Solution d'un unique master

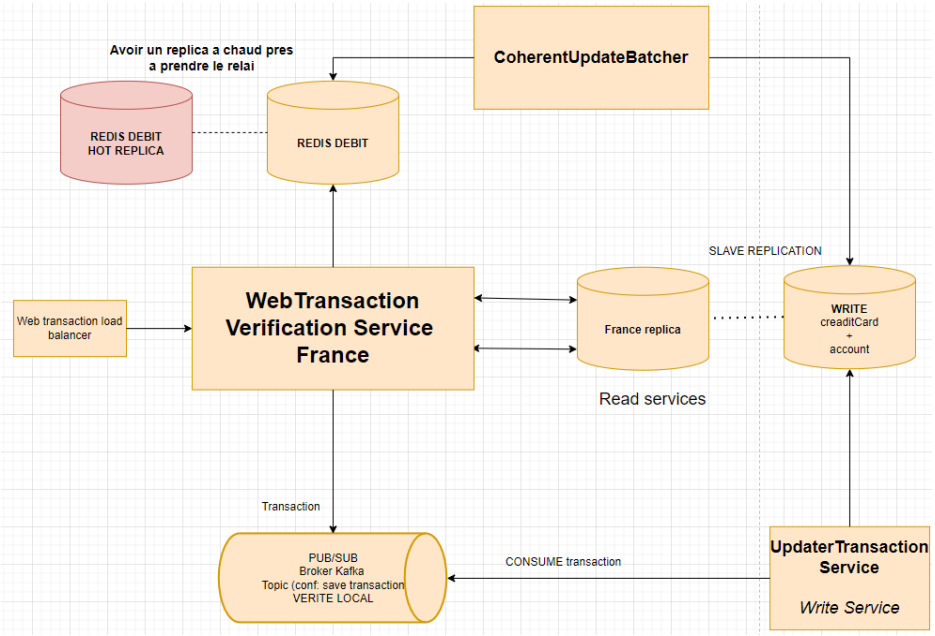
Dans cette solution, l'avantage est que les données client pourront être récupérées par une région, même si le client n'appartient pas à la région en question.

L'étude est effectuée dans l'adr numero 006.

Au final, la solution se prétend au mieux au problème est la seconde solution, en effet Dans un cas réel, un cluster de base de données, représenté comme une unique grosse base de donnée, sera mise en place, et monitoré par des administrateur de base de données.

Week 49:

State of the services architectures :



Pendant le semaine, une grande partie du temps a été investi dans l'implémentation et la recherche de solution liée à la cohérence des données et à la vitesse d'une transaction. Dans l'ADR_005, nous avons discuté deux possibles solutions qui marqueraient la fin de cette recherche d'optimisation infinie.

Inter-Regional failOver

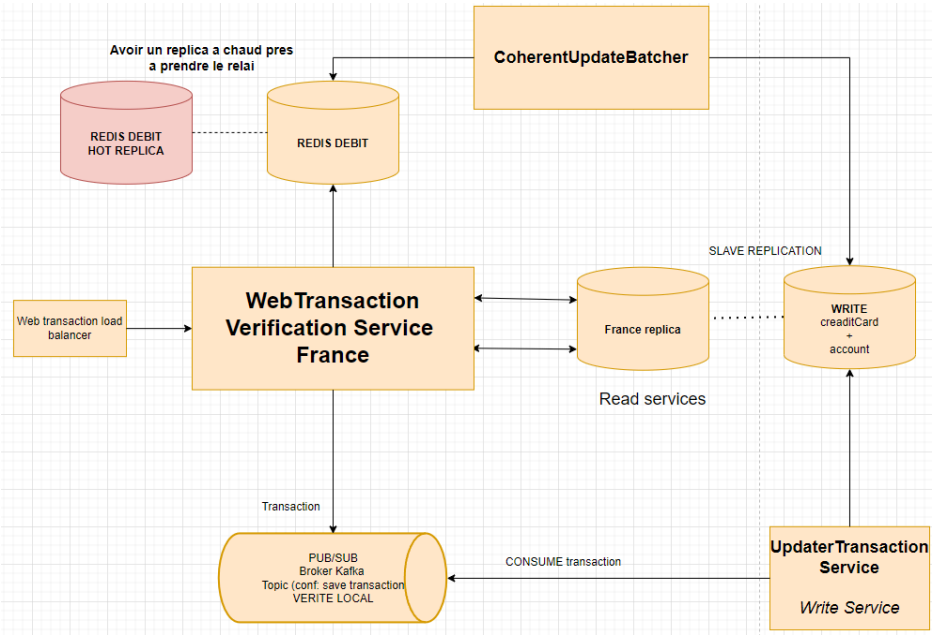
Le troisième scénario qui doit être mis en place et au niveau des déploiements, avoir un mécanisme de failOver inter-régions. En effet, dans notre stratégie de déploiement, nous avons décidé de mettre en place un déploiement par région, qui nous permettrait de desservir un nombre de clients par région, afin que les services soient à leur proximité.

Cependant, si une ferme devient indisponible pendant un certain temps, il faudra tout de même que ce soit transparent pour nos utilisateurs. De ce fait, nous avons pensé mettre en place un mécanisme de fail over inter région, et ce à commencer dès le début de la conception du système dans une région unique. En effet, nous proposons tous les changements effectués sur un client vers le Groenland. Afin de "centraliser" les données des clients, il en va de même pour les débits des clients.

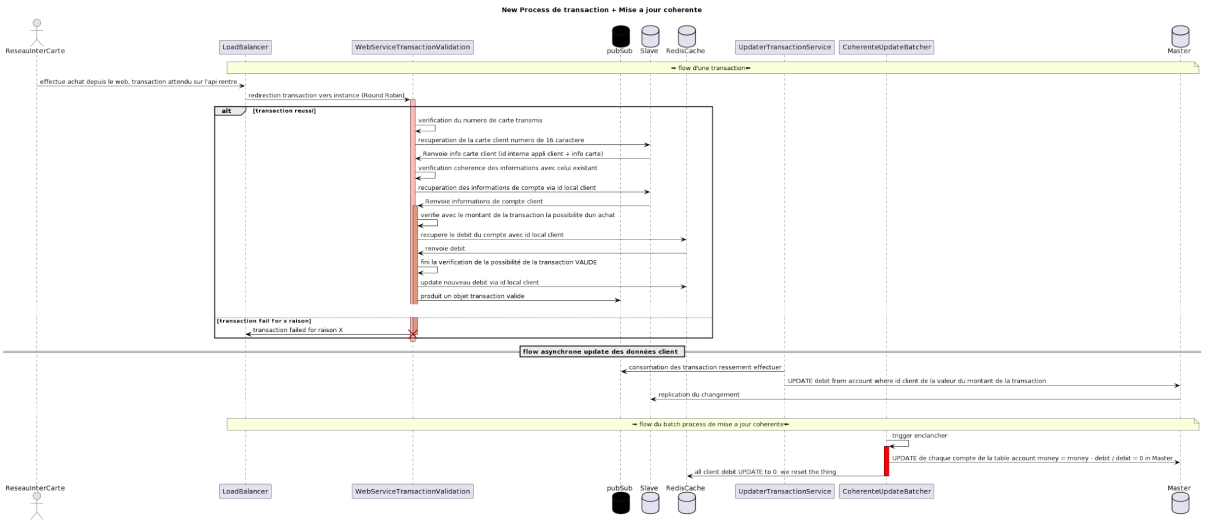
Cependant, notre problème reste tout de même lié à la réalisation d'une démonstration de notre PoC. En effet, afin de faire un déploiement, nous lançons nos conteneurs via un docker-compose. Il nous faudrait une autre solution qui soit plus simple que Kubernetes, et nous avons choisie de nous pencher pour la suite sur docker-swarm.

Week 48 :

State of the services architectures :



Scénario de test de charge sur la nouvelle version



Nous avons fini d'implémenter le scénario suivant. Le plus important était de mettre en place et de tester le flux de base d'une transaction. En effet c'est celle-ci qui s'inscrit dans un contexte de stress dû à une grande charge de requête à un moment donné.

Nous avons aussi mis en place le service de batch processing sur les transaction qui effectue notre "mise à jour cohérente" afin que le système retrouve son état initial, avant toute transaction, quand tous les débit sont a 0.

Coherent UpdateBatch service :

Le service :

Ce service a été mis en place en TypeScript. Pour l'instant, dans son implémentation actuel, il traite toute la base de données Redis puis la Base de données master.

La mise en place que nous avons effectué permet de résoudre le problème que l'on avait lié à notre cohérence éventuel nous forçant à mettre en place l'hypothèse des 5 min. Mais pour revenir à un état initial du système pour remettre les compteurs à zéro. Et pouvant dans un cas réel d'utilisation nous permettre de transmettre les transactions effectuées dans une journée vers le réseau interbancaire.

Drawback :

Le service pendant son temps d'exécution présente des failles dans la cohérence des données si en parallèle des transactions sont effectuées, en effet durant le processus de mise à jour cohérente, les données entre Redis et le Slave replica issue du Groenland ne sont plus du tout synchrone.

Possibles solutions :

Afin de résoudre le problème choisi, un choix est à faire entre la cohérence des données et la disponibilité. Le système que l'on tente de concevoir, à savoir une banque, nous force la main. Ainsi plusieurs solution peuvent être mis en place afin de maintenir une forte cohérence des données, avec en plus une autre Disponibilité :

- Cohérence des données :
    - Solution 1 : Verrouiller complètement le système pendant la mise à jour cohérente. Et ainsi placer celle-ci a une heure de trafic au plus bas en fonction de la région. Ainsi, pendant cette période de mise à jour de plusieurs minutes seulement, l'on est assuré qu'aucune transaction ne peut se faire, mais effectivement cela peut poser des soucis de disponibilité.
    - Solution 2 (ADR_003) : Verrouiller le système individuellement pour chaque client pendant une durée très courte. En effet, la mise à jour se fera toujours a un timing de faible trafic, mais cette fois-ci se fera client par client, et ainsi pour chaque client pour qui la mise à jour serait en train d'être faite, un verrouillage de la possibilité de faire une transaction sera mis en place. Ainsi la mise à jour n'impactent pas tout les client de la banque mais seulement ceux pour qui au moment t de leur maj essaye de faire une transaction
- Disponibilité de l'action d'achat sur le Web :
    - Solution 1 : Cette solution est très contraignante sur le plan de la disponibilité de notre banque auprès des utilisateurs, en effet, pendant plusieurs minutes il ne pourrait pas faire de transaction à un moment x de la journée. La solution serait alors d'accélérer le processus de mise a jours, via l'utilisation de thread exécutant la même tâche de manière parallèle.

        - Première possibilité : 2 thread chacun sur la mise à jour d'une Base de données : temps quasiment divisé par deux .
        - La seconde possibilité : x thread pour les x clients de notre banque, avec pour chaque threads un accès au deux base de données : Temps diviser par y nombre de CPU présent sur le serveur.

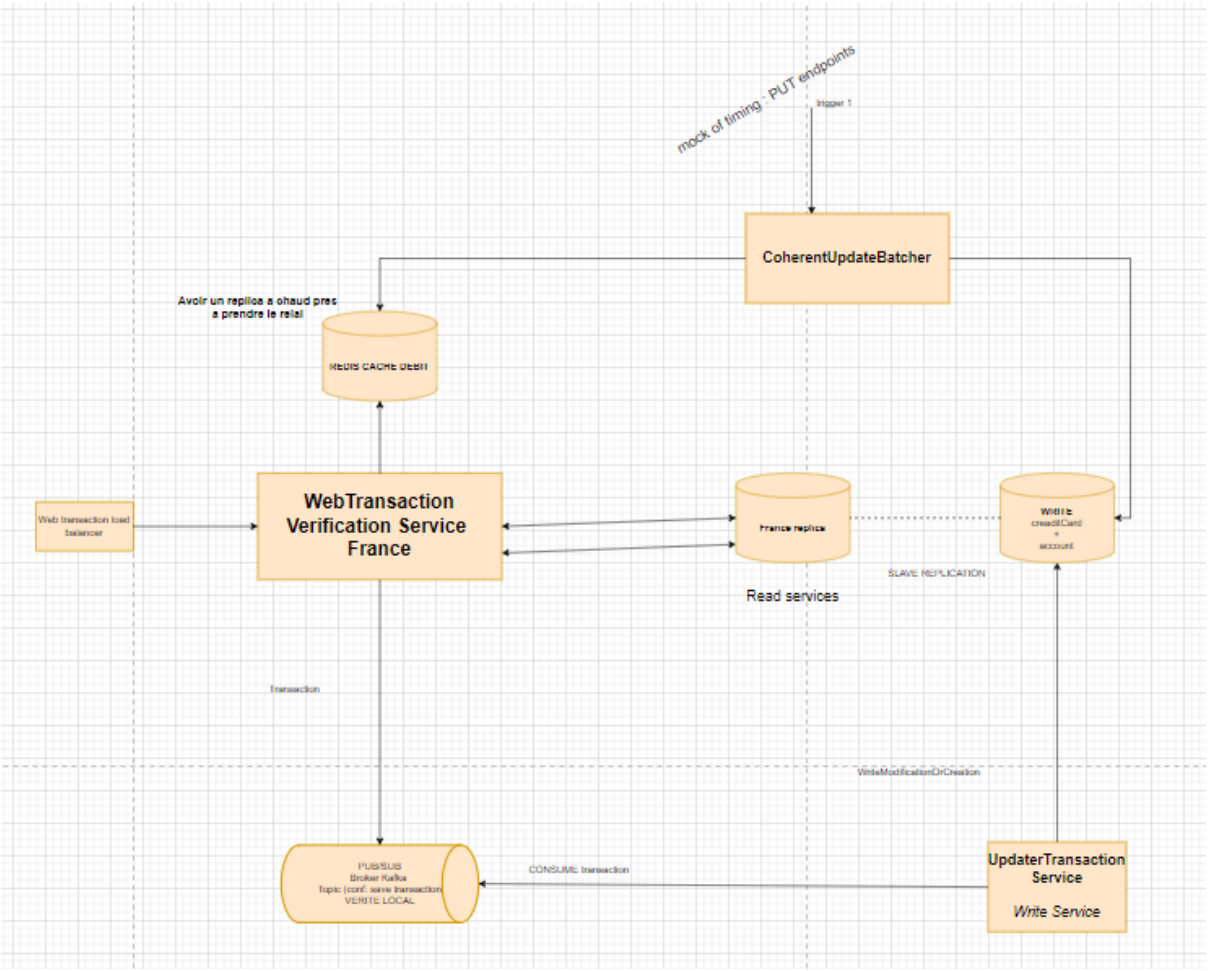
      Etant donnée que nous sommes dans un cas ou nous sommes limité sur le plan
      matériel, la première possibilité se prête plus au jeu.
    - Solution 2 : Cette solution présente des avantages au niveau de la disponibilité car peut permettre de nous ramener à un cas où l'on ne bloquerait qu'un certain nombre de clients pendant un instant. Cela permet du point de vue d'un client d'être bloquer d'une transaction pendant un instant plus court que s' il aurait dû attendre l'entièreté de la mise à jour. Ainsi la mise à jour continue, mais ne bloque pas tout le système.
      Cependant, cela a un coût, en effet il faudra mettre en place une logique supplémentaire de vérification d'un statut verrouillé ou non du client, ce qui ralentit drastiquement la mise à jour d'un client, car en effet ce statut serait à stocker quelque part.

Resilience Redis :

Redis joue un rôle important dans notre nouveau système, c'est pourquoi la perte de cette base de données pourrait couter tres chere, et est très dangereuse. A l'image de n'importe quelle base de donnée, l'on a donc décidé de mettre en place un Hot Replica de cette base de données. Ainsi en cas de problème, l'on pourrait instantanément changer vers le réplica chaud, le temps de relancer l'ancien master se relancer pour redevenir le replica chaud. (ADR_004)

Week 47 :

State of the services architecture :



Dans l'objectif de la réalisation des points évoqué dans le cadre de l'évolution qui sont :

- Lever l'hypothèse de temporisation entre transactions pour le web (et à terme pour les TPE).
- Algorithme réaliste de vérification des CBs.

Nous avons revu notre architecture et notre manière de peupler la base de données.

Levé de l'hypothèse :

L'hypothèse avait pour principal objectif de nous acheter du temps afin de permettre de procéder au flow de l'update des comptes clients après un achat, que nous avons décidé de rendre asynchrone par soucis de temps.

En effet nos objectif primordiaux été :

- Réaliser une transaction de manière rapide en minimisant le nombre d'interactions avec une base de données sql pendant la période de validation de la transaction.
- Pouvoir traiter plusieurs demandes, au nombre de 2500 en même temps.
- Être fortement disponible, en effet, sans transaction pas d'achat.

Afin de résoudre le problème de cohérence des données, l'hypothèse a été formulée. Nous avons réfléchi à une solution pour résoudre le problème.
En effet, au lieu de directement mettre à jour le compte d'un client après un achat pour retomber sur un montant dans le compte qui est de nouveau cohérent pour le client, nous allons à l'issue d'une transaction envoyer avec la transaction valide un montant, donc rien ne change, mais l'update service va lui faire monter un la valeur du débit du compte client.
Le débit et l'argent consommée par le client.

Les instances de validation de transaction pourront elles dans une BD redis ajouter le débit associé à un client et venir le récupérer pour faire la vérification de la capacité du client à pouvoir effectuer une transaction. En comparant le montant de la transaction avec la somme du solde du compte client (récupérer depuis le slave en lecture) et du débit client (récupérer depuis redis). (Voir ADR_001)

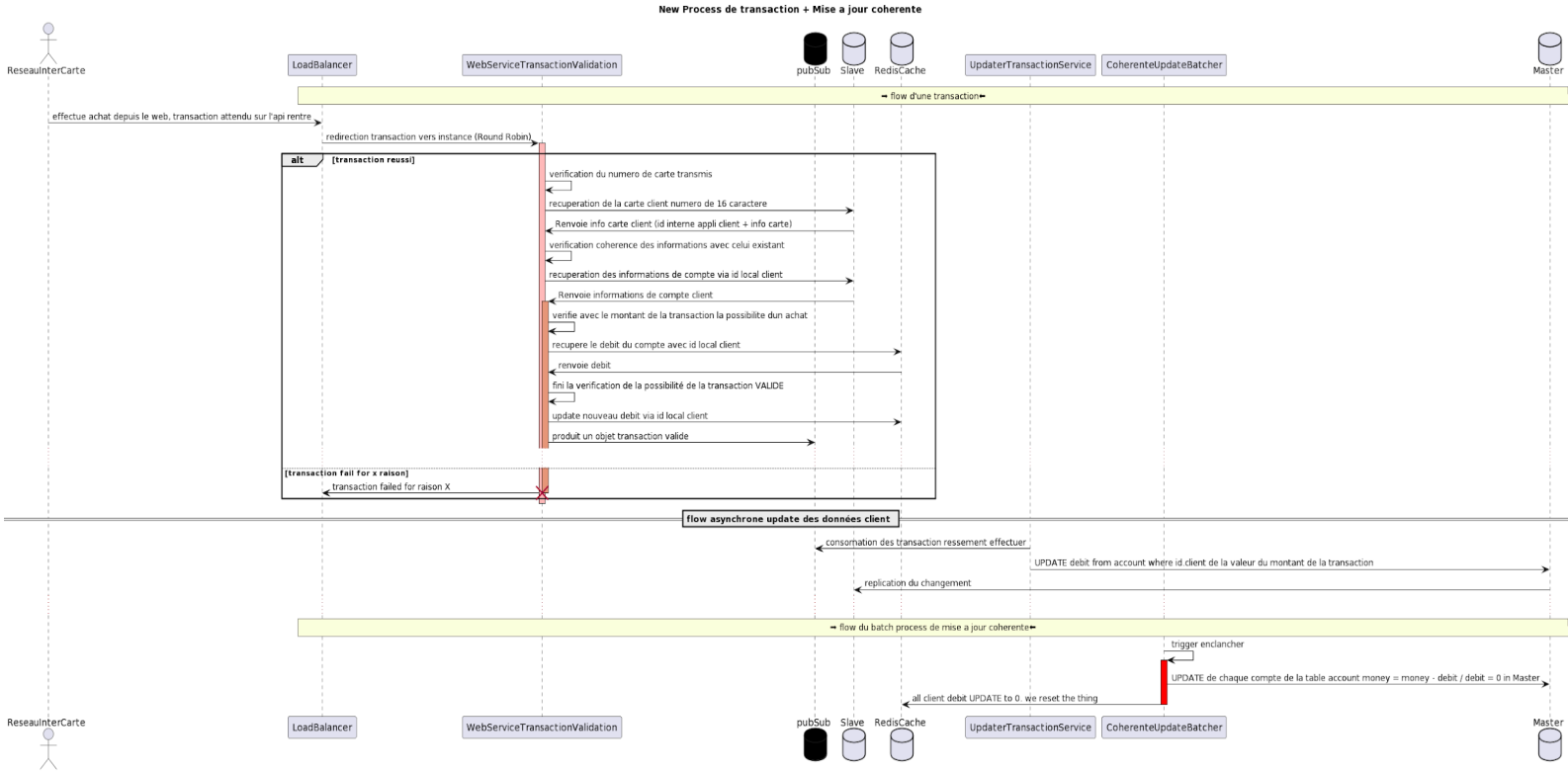
Programmer à un moment T, un service de batch va venir effectuer une "mise à jour cohérente", durant ce laps de temps, le service va venir update les débits à 0 sur Master et Redis, et va mettre à jour les comptes des clients.

REDIS

Redis sera utilisé dans notre cas comme un DB clé-valeur opérant sur la ram, donc étant nettement plus rapide dans la récupération d'un integer. N'impactant ainsi que très peu le temps et les performances.

On pourrait penser qu'utiliser redis comme un cache avec spring pourrait constituer un avantage, mais le caching sous spring implique que la valeur de la clé demandée sera renvoyé sans entrer dans la méthode, or dans notre cas, nous devons toujours a l'issue d'une transaction de nouveau changer la valeur du débit en lui ajoutant le montant de la transaction. Ainsi mettre en cache une valeur que l'on modifie souvent constitue un anti-pattern au cache.

Diagramme de séquence d'une transaction



Remplissage de la DB

Étant liée à notre second problème, le remplissage de la DB pour lancer nos tests nous a obligé à réduire la complexité de nos vérifications. Nous avons changé la manière de faire et maintenant utilisons un CSV pour remplir la DB avec 2500 compte et carte cohérente.