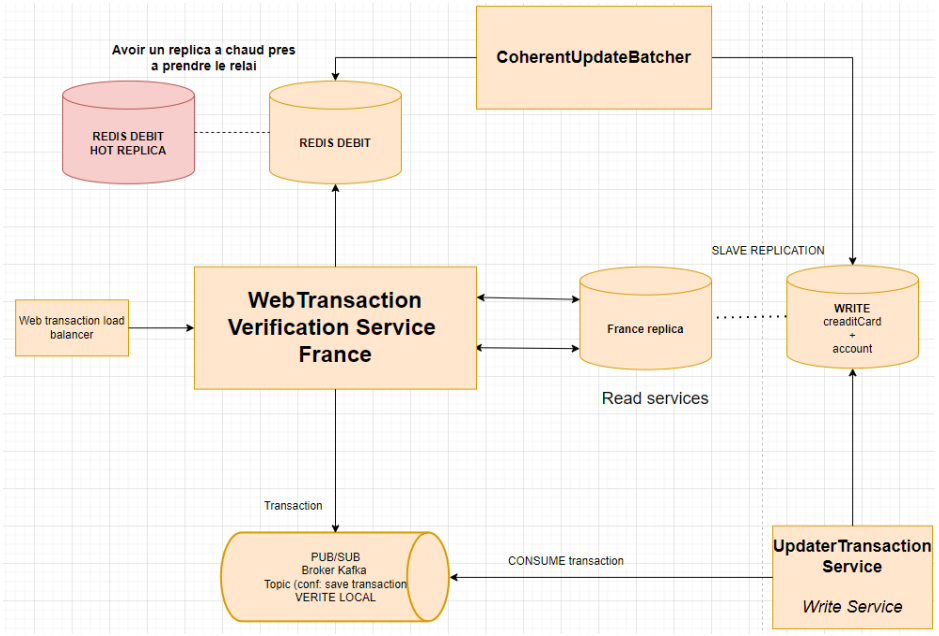
Rapport de suivi d'architecture
AL Néo-Bank - Evolution

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.

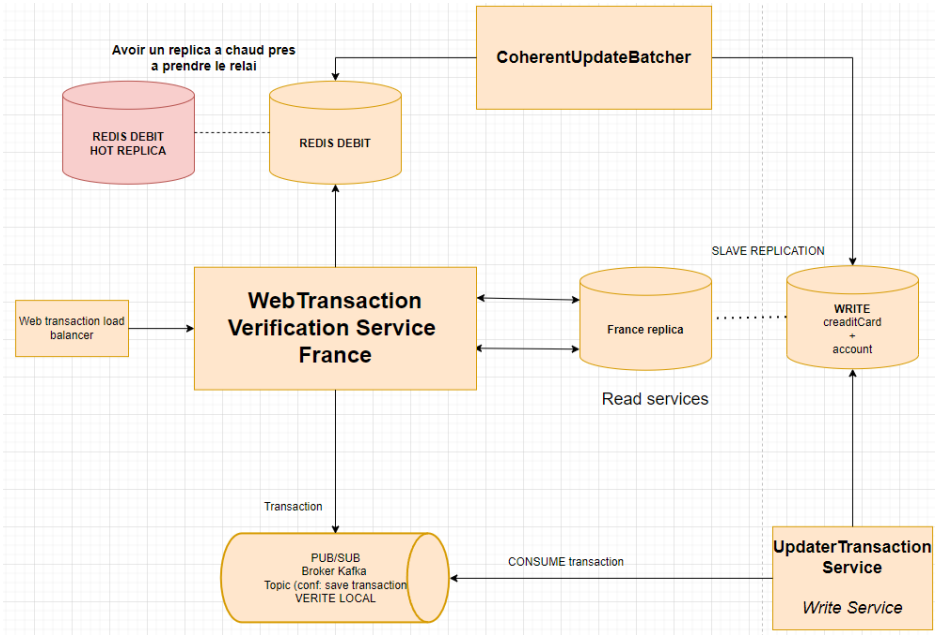
Cependant, si une ferme devient indisponible pendant un certain temps, il faudra tout de même que ce soit transparent pour nos utilisateurs. De ce fait, nous avons pensé mettre en place un mécanisme de fail over inter région, et ce à commencer dès le début de la conception du système dans une région 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.

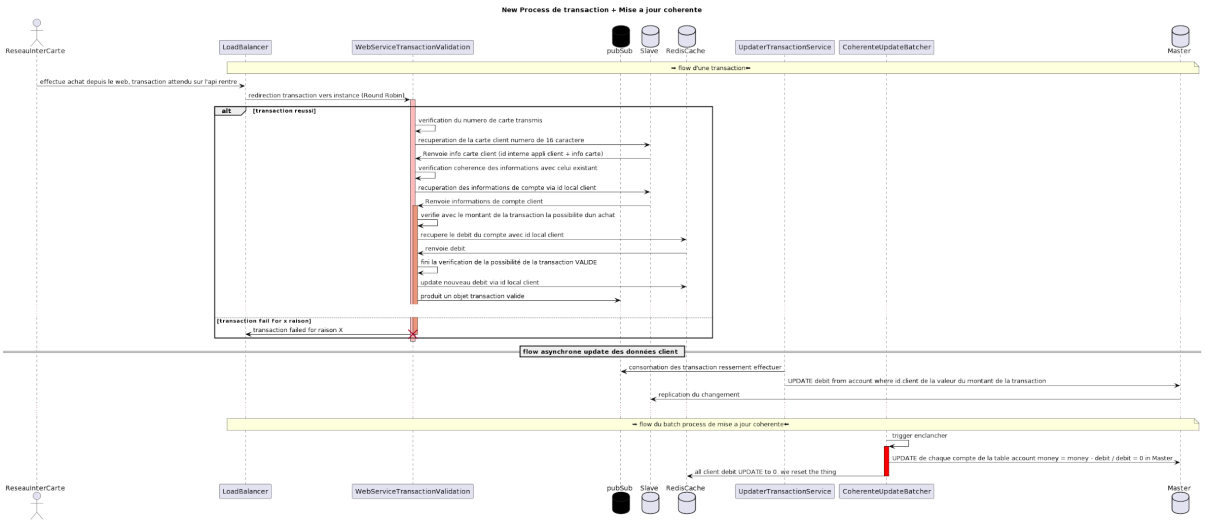
Cependant, notre problème reste tout de même lié à la réalisation d’une démonstration de notre PoC. En effet, afin de faire un déploiement, nous lançons nos conteneurs via un docker-compose. Il nous faudrait une autre solution qui soit plus simple que Kubernetes, et nous avons choisie de nous pencher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

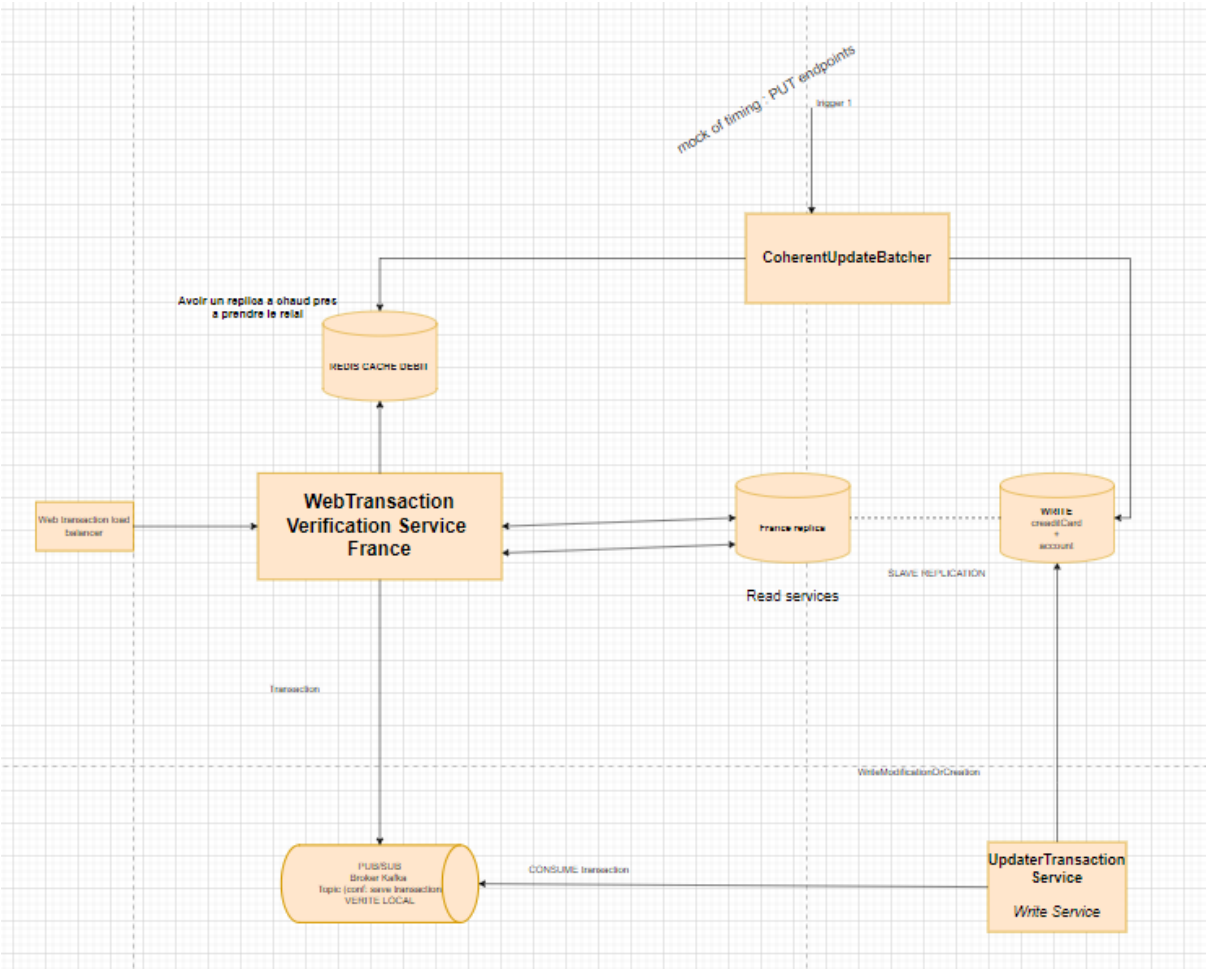
      Etant donnée que nous somme dans un cas ou nous somme limité sur le plan matériel, la première possibilité se prête plus au jeu.
    - Solution 2 : Cette solution présente des avantages au niveau de la disponibilité car peut permettre de nous ramener à un cas où l'on ne bloquerait qu’un certain nombre 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)

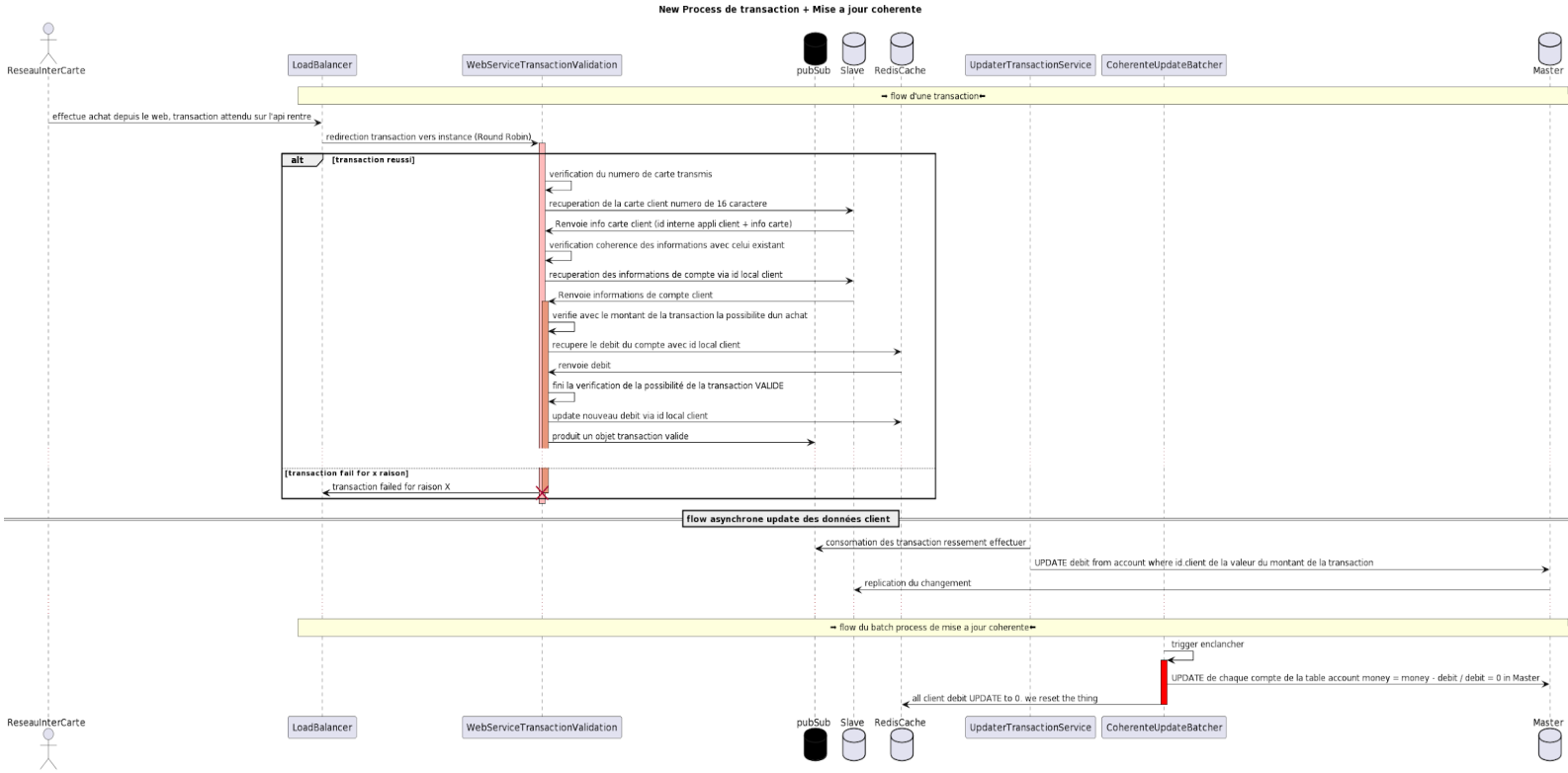
Programmer à un moment T, un service de batch va venir effectuer une "mise à jour cohérente", durant ce laps de temps, le service va venir update les débits à 0 sur Master et Redis, et va mettre à jour les comptes des clients.

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.