POLYTECH
NICE-SOPHIA

Université Nice Sophia Antipolis

Rapport final du projet

Package delivery by drones

***V6** - resiliency, security, robustness*

Groupe F

Dina Abakkali - Sylvain Marsili - Thomas Martin - Clement Poueyto - Florian Striebel

Scope :

Notre application répond au besoin de livraison de colis par drone en respectant principalement les principes de sécurité, résilience et de robustesse du système.
Le système complet doit pouvoir communiquer avec les drones en vol du décollage jusqu'à leur retour de manière sécurisée afin d'éviter tout type d'attaque qui pourrait compromettre la livraison.
La planification de la livraison d'un colis et la gestion de la logistique sont des services externes.
C'est pourquoi notre partie du système doit être capable de communiquer et tracker l'ensemble des drones afin de les monitorer. Un drone est autonome, il est capable de se déplacer d'un point A à un point B avec un plan de vol. Le drone envoie ses informations régulièrement au serveur comme sa position.
Lorsque le drone s'apprête à livrer le colis, le client est notifié afin qu'il se prépare à réceptionner la commande.
Nous nous occupons que d'une partie du fonctionnement du drone, notamment de la communication. Nous considérons que le système du drone autonome n'est pas à développer par notre équipe.

Sécurité - robustesse - résilience

Nous considérons que le drone est interne au système car nous devons gérer la sécurité de la communication avec le serveur. Cette communication s'effectue directement avec le système en HTTP(s) en utilisant le réseau 4G/5G/LTE afin de garantir la sécurité tout en limitant les appels possibles entre les deux applications.

Pour la robustesse, les systèmes doivent vérifier les informations reçues avant de les traiter afin de s'assurer qu'elles ne soient pas un danger pour leur bon fonctionnement. L'émetteur d'une requête doit s'authentifier afin de vérifier son identité et ses droits.
De plus, dans le but d'éviter la surcharge de certains services, il peut être intéressant de mettre en place des tâches planifiées périodiquement.

Pour garantir la résilience du système, il faudra éviter les risques de Single Point Of Failure en séparant les services de manière indépendante avec l'utilisation d'un bus. Dans un premier temps, on optera pour une architecture micro-service. Si l'un des services tombe en panne alors cela n'impactera qu'une partie de l'ensemble des services. En effet, si drone status est toujours allumé alors les données des drones seront toujours actualisées malgré le redémarrage des autres services.

Contraintes

L'architecture micro-service n'est pas magique et nous sommes conscients que cela peut nous poser des problèmes d'intégration ou même de monitoring.

Notre application doit pouvoir gérer la communication avec de nombreux drones simultanément en subissant le moins de ralentissement possible. Nous avons envisagé l'utilisation d'un bus de messages pour améliorer la résilience du système. Le bus permettrait l'envoie de messages des drones (producteurs) qui seraient lu par les services *Drone Status, Fly Manager* et *Drone Controller* (consommateur), tout en conservant les données sur une certaine période si l'un des services venait à tomber. Cependant, nous préférons voir les limites de notre architecture actuelle avant d'implémenter cet outil.

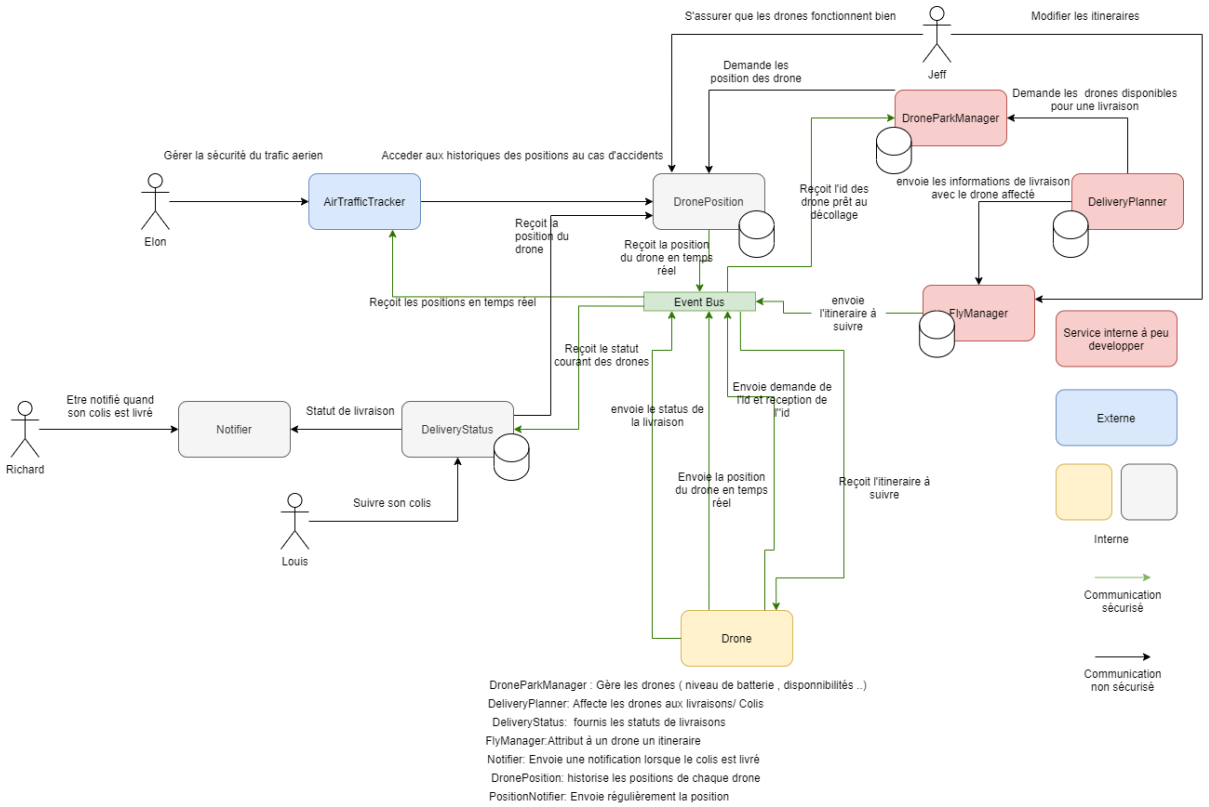
Risques

En cas de panne du service Drone Status, pour ne pas perdre les données lorsque les drones détectent que le serveur est tombé, ceux-ci stockent les données à envoyer (localisation, statut …). Mais si le drone rencontre lui aussi un problème, nous perdrons ses données et nous risquerons de le perdre totalement.

Personas :

- Elon, directeur la sécurité de l'aviation civile
- Nout, architecte logiciel
- Jeff, Responsable des livraisons par drone
- Louis, client, est livreur en auto entreprise, il a besoin d'être flexible sur la réception de ses colis
- Richard, client, aime et commande régulièrement des objets technologiques dernier cri

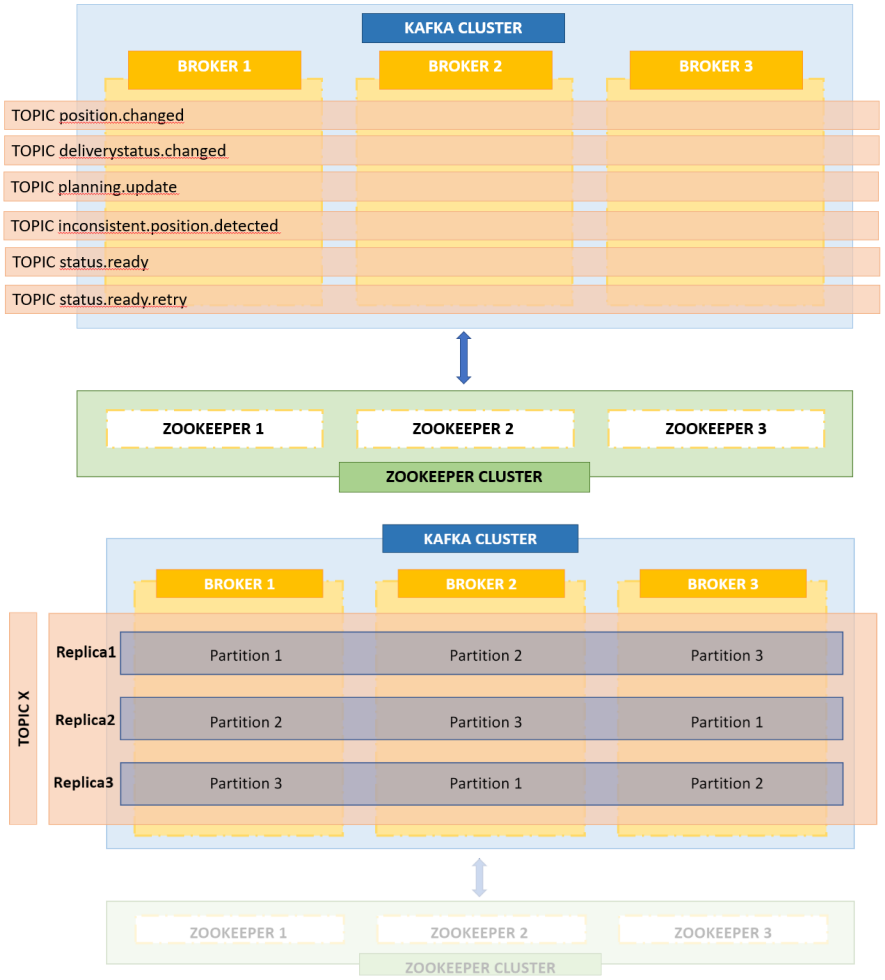
Schéma d’architecture



Explication choix d’architecture:

Nous avons réfléchi et nous avons vu en la sous partie du système de remontée de position des drones, un système de notification. Nous avons pensé que cette sous partie du système pouvait s'inspirer d'une Event Driven Architecture. Nous avons choisis d'utiliser un bus pour permettre d'une part la lecture en temps réel des données envoyés par les drones par les différents services qui ont besoin de ces informations le plus rapidement possible (pour le contrôle du trafic aérien et pour le calcul de l'itinéraire / la navigation du drone), et d'autre part pour centraliser l'abonnement à ces notifications dans un outil spécialisé et conçu pour cela. Nous pensons qu'avec ce bus, si par exemple notre service DroneStatus tombe, le contrôleur aérien peut toujours connaître la position des drones en temps réel sans aucun problème. De même, avec cette approche, le travail du drone est simplifié en ce sens où connaître l'état du serveur ne nous est pas utile, il n'attend aucune réponse, il envoie ses données dans le bus directement.

L'utilisation d'un bus avec la technologie Kafka, nous assure aussi une certaine résilience des données envoyées. En effet kafka vas garder en mémoire les messages envoyés de façon ordonnés. Ainsi si un service consommateur de messages tombe, une fois relevé il pourra récupérer les messages qu'il n'a pas pu consommer. Afin d'assurer le service que propose le bus, nous devions nous assurer que ce dernier soit assez robuste pour fournir les données même si une de ses instances venait à tomber. Nous avons donc multiplié les instances de bus. Ainsi avec 3 instances de bus, nous assurons le fonctionnement du bus même en cas de panne d'une de ses instances.



Chaque topic du bus représente un type de messages différents envoyés par nos producteurs et lus par nos consommateurs. Pour notre POC, nous n'avons pas trouvé nécessaire pour le moment de rajouter des ordres de priorités entre les topics. Toujours pour résister à une panne nous avons coupé nos topics en 3 partitions et répliqué chaque partitions entre nos différentes instances de bus pour que toutes aient accès aux aux messages envoyés sur ces topics.

Etant donné que la plupart de nos messages passe par le bus, nous avons sécurisé ce dernier afin que seuls les services autorisés y aient accès. De plus les messages sont sécurisés et cryptés. Ainsi il n'est pas possible par exemple de dérouter notre drone.

Pour faire cela nous avons une Authentication/Encryption de données TLS, régi par des certificats qui doivent être signé par notre Certificat d'Autorité. Seuls les services possédant un certificat signé par ce CA pourront se connecter et décrypter les données envoyées par le bus.
Nous avons rajouté une deuxième Authentification pour accéder à notre bus, une authentification SASL-PLAIN (mot de passe, nom d'utilisateur) afin d'assurer qu'aucun service ne se connecte sans ces identifiants. Notre bus contenant des informations essentielles, une double sécurité semblait approprié.
Ainsi pour se connecter à notre bus et pour lire ses données, il faut posséder un certificat TLS signé par notre CA, ainsi que les identifiant SASL que nous avons créés.

User stories :

1. **[41][M][US] Sécurité** - En tant que Richard et ayant l'habitude de commander des produits de valeur, je souhaite que la sécurité de la livraison me soit garantie afin de pouvoir effectuer mes achats sans crainte.

2. **[41][Technical] Sécurité** - En tant que Jeff je souhaite que la prise de contrôle des drones à distance soit empêchée aux personnes qui n'y sont pas autorisées, afin de garantir à mes clients la sécurité de leurs colis. (M)
   Critères d'acceptation:
   - Seul le serveur peut communiquer avec les drones.
   - Un utilisateur malveillant tente de communiquer avec le drone. Mais celui-ci n'étant pas reconnu comme le serveur ces paquets sont ignorés.

3. **[42][M][US] Résilience** - En tant que Elon, je souhaite pouvoir connaître à tout instant la position des drones de livraison afin de pouvoir gérer la sécurité du trafic aérien, et déterminer la cause d'accidents le cas échéant.

4. **[42][Technical] Résilience** - En tant que Nout, je souhaite que le serveur redémarre en toute transparence afin de pouvoir fournir des informations complètes aux autorités et aux partenaires.
   Critères d'acceptation:
   - Lorsque le serveur subit une panne une fois celui-ci redémarré il doit pouvoir reprendre les transactions là où elles se sont arrêtées.
   - Lors du redémarrage celui-ci ne doit pas retomber à cause de la charge des transactions à traiter

5. **[43][M][US]** - En tant que Louis, je souhaite pouvoir suivre la position du drone s'occupant de la livraison afin de m'organiser pour être disponible au moment de la réception du colis.
   Critères d'acceptation:
   - Lorsque le drone n'a pas encore décollé, le suivi est impossible. Un message d'information sera renvoyé.
   - Lorsque le colis est déjà livré, un message "colis livré le …" sera affiché.
   - Lorsque le drone est en cours de livraison, la position sera envoyée au client ainsi que le temps d'arrivée estimé.

6. **[43][Technical] Robustesse** - En tant que Nout, je souhaite que le serveur tente de se reconnecter aux drones en cas de perte de connexion afin de pouvoir continuer le tracking
   critère d'acceptation côté drone:
   -Lorsque la connexion avec le serveur est perdu le droit doit stocker ses données de position
   - Et il tente toutes les 5 secondes de se reconnecter au serveur jusqu'à y arriver

7. **[43][Technical] Robustesse** - En tant que Nout je souhaite que le système résiste aux paquets corrompus ou manquant afin qu'il ne tombe pas en cas d'erreur
   Critères d'acceptation :
   - Les paquets qui transitent dans l'application possèdent un checksum. Le receveur de son de son côté calculer cette valeur pour la comparer avec celle du paquet reçu.
   - Lors de la réception des données si la sommes de contrôles calculée est différente de celles dans le paquet alors le renvoie des données est demandé à l'envoyeur.

8. **[43][M][US]** - En tant que Jeff, je souhaite connaître l'état de tous les drones afin de m'assurer du bon fonctionnement du système.
   Critères d'acceptation côté serveur:
   -On possède une carte de la qualité du réseau mobile à laquelle on y associe le temps de réponse accepté en fonction de la qualité dans la zone où le drone se trouve.
   -si le drone est en zone blanche le serveur accepte que celui-ci ne répondent pas pendant 5 min.
   -si le drone est dans un zone bien couverte alors le drone doit être capable de répondre en 30 sec

-Si le drone ne répond pas au-delà de ce délai accepté, les autorités sont prévenues.

9. **[44][C][US]** - En tant que Richard, je souhaite être notifié lorsque mon colis est arrivé afin de le récupérer
   Critères d'acceptation:
   - Lorsque le colis est livré le message "votre colis à été livré le … à ..h " sera envoyé

10. **[44][S][Optionnelle] [US] - Sécurité** - En tant que Jeff je souhaite détecter la perte de contrôle d'un drone afin de garantir la livraison du colis

11. **[44][S][Optionnelle] [US] Sécurité** - En tant que Richard je souhaite que ma livraison soit décalée en cas de mauvaise météo afin protéger mon colis
12. **[44][Optionnelle] [US] - Sécurité** - En tant que Jeff je souhaite changer l'itinéraire d'un drone ou lui demander de retourner à la station afin d'avoir la main sur son comportement en cas de problème

Spike:

1. Recherche sur la sécurisation des paquets avec https
2. Recherche sur event bus

Scénario 1 (Poc début novembre):

1. Le système de livraison par drone prend en charge une livraison lors d'un pic d'utilisation du réseau
2. Le colis est pris en charge par un drône qui reçoit son itinéraire, le client peut constater le changement de statut de sa livraison
3. Le drone envoie sa position au serveur en temps réel, permettant au client de localiser son colis quand il le souhaite
4. Le drone étant ancien, aléatoirement il envoie des positions erronés, ces positions ne sont pas stockés et les erreurs sont remontées
5. En raison de la charge, le service drone-position tombe
6. Le service est relancé en toute transparence
7. Le client est livré et le drone retourne à la base
8. Un accident s'est produit durant la période d'indisponibilité du service drone-position
9. Le contrôle aérien, dans son enquête, regarde la position du drône durant cette période et en détermine qu'il est hors de cause

Scénario 2 (Poc début novembre) :

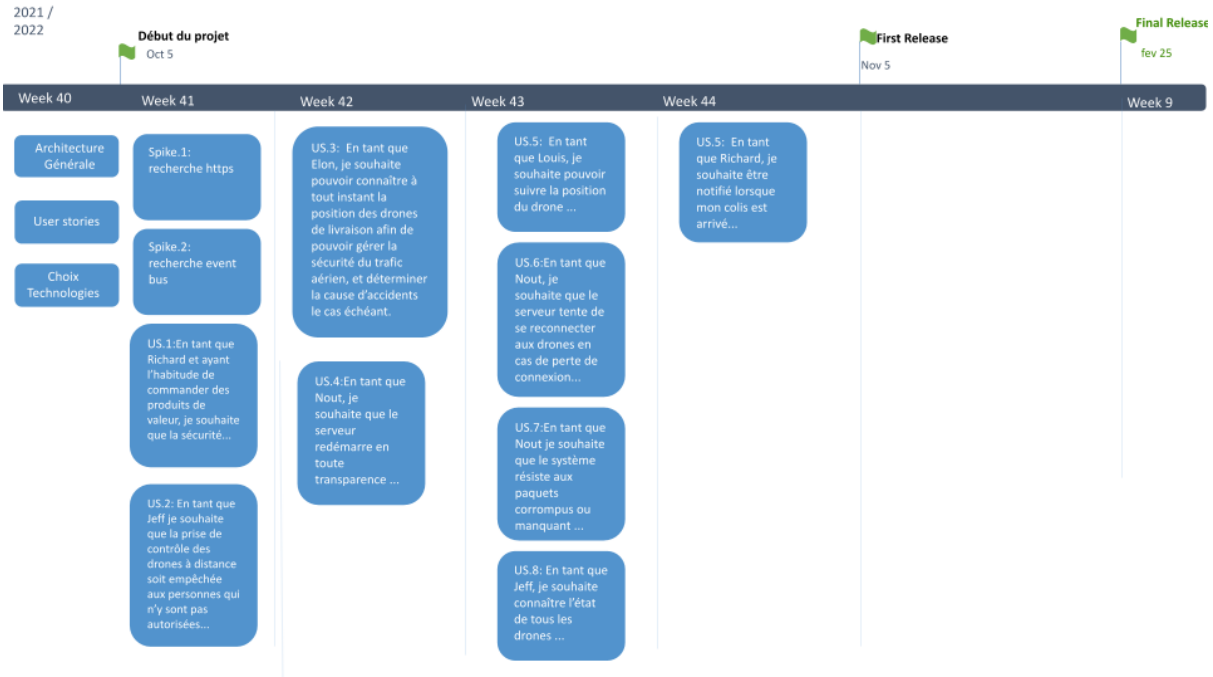
1. La sécurité de communication du drone n'est pas activé
2. Un utilisateur malveillant envoie des paquets frauduleux en se faisant passer pour le serveur et change l'itinéraire du drone
3. La sécurité de communication du drone est activé

4. Un utilisateur malveillant tente alors d'envoyer des paquets frauduleux en se faisant passer pour le serveur. Mais le drone détecte que ceux-ci ne proviennent pas du serveur et ne les prends pas en compte

Scénario 3 (optionnel) :

1. Un utilisateur passe une commande lors d'un pic d'utilisation de l'application
2. La commande est livrée et le drone notifie le serveur
3. Le service de notification est saturé et tombe
4. Après le redémarrage, les notifications n'ont pas été perdues et sont envoyées

RoadMap



Perspectives futures :

- Ajouter un patron circuit breaker : qui est un modèle de conception utilisé pour détecter les pannes et encapsuler la logique.
  Nous l'aurions implanté du côté du drone lorsqu'il ne peut plus se connecter au bus afin qu'il réessaie plusieurs fois de renvoyer sa position dans le bus. Durant ce temps, il aurait dû sauvegarder ses positions le temps de la reconnexion.

- Utiliser des jetons JWT côté utilisateurs finaux afin d'authentifier l'utilisateur qui se connecte au serveur et de garantir son rôle le client final. Ainsi on pourrait garantir que seuls les personnes autorisées peuvent accéder aux positions du drones comme le contrôleur aérien.

- Pour le moment on détecte les mauvaises positions mais pour aller plus loin cela pourrait prendre la forme d'un évènement créé par done-position lorsqu'une position reçue est incohérente et d'un service qui écoutera ces signalements, et qui a son tour pourrait émettre un évènement qui indiquera qu'un drone ne fonctionne plus correctement.

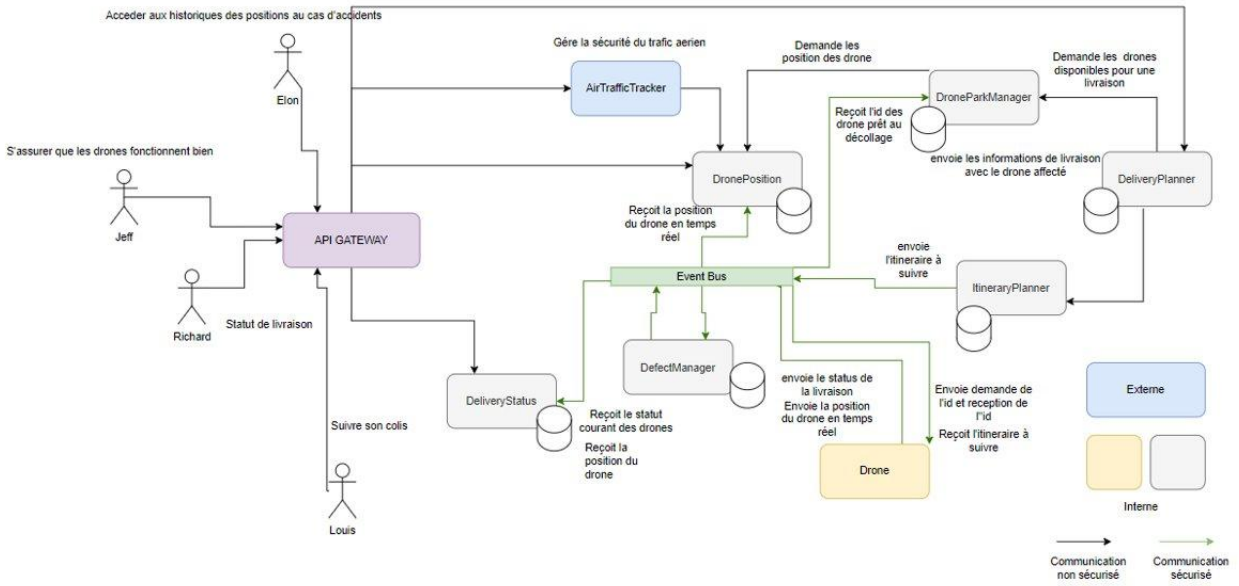
Partie 2

Évolution du scope :

Les nouvelles spécifications de notre système de livraison de drone:

- résister à un grand nombre de drones différents ainsi qu'à une attaque DDoS.
- un système robuste de détection de services défaillants et de redémarrage automatisé.
- On veut un démonstrateur qui applique les patterns de résilience sur nos services et la gateway, des monkeys de démonstration.

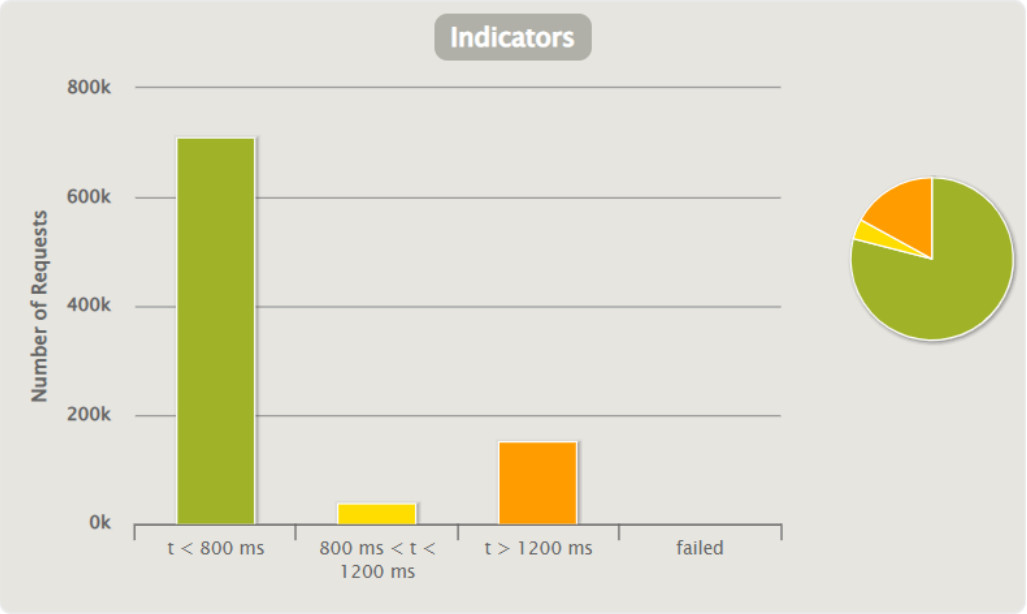
Évolution de l'architecture :



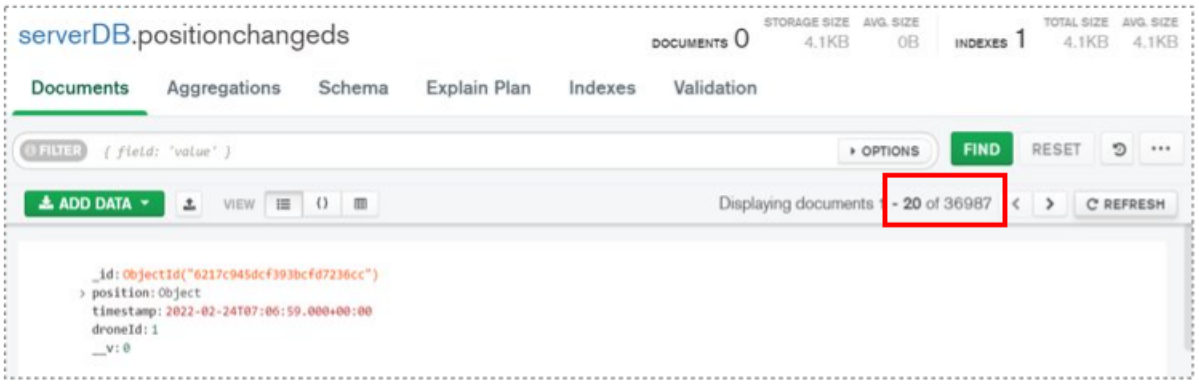
Dans cette nouvelle architecture, nous avons ajouté une API Gateway qui correspond au point d'entrée de notre application. En effet, dans la première partie du cours nous avions sécurisé la communication entre le bus kafka et nos drones cependant dans le cadre d'un déploiement nos services étaient complètement exposés. L'ajout de l'API Gateway pouvait être l'origine d'un nouveau Single Point Of Failure, c'est pourquoi nous avons déployé de nouveaux containers avec Docker Swarm.

Résistance à la charge de l'architecture:

Pour tester la résistance à la charge de notre architecture nous avons utilisé gatling. L'objectif a été que nos scénarios gatling soit au plus proche du fonctionnement en situation réel de l'envoie de position par les drones. Nous avons donc fait le choix d'avoir un nombre constant de requêtes durant tout le test sur une période de temps assez longue pour vérifier que celui-ci ne tient pas la charge que sur une très courte période de temps.



Pour 10 000 requêtes/secondes pendant 90 secondes, ce qui représente plus de 10 fois ce que nous estimons comme la charge normale pour le serveur. Nous nous rendons compte que celui-ci tient bien la charge. Mais ce que kafka ne nous montre pas, c'est que toutes les positions, elles, à la fin des 90 secondes, n'ont pas eu le temps d'être stockées.



Seulement 36 000 environs des 900 000 positions sont stockées. On pourrait se dire que ce n'est pas grave si les données mettent du temps à être stockés tant qu'elles le sont mais dans certains cas cela peut poser problème notamment si un agent de la DGAC a besoin d'accéder aux informations d'un crash de drone il pourrait devoir attendre plusieurs jours avant de pouvoir étudier les raisons du crash. Nous avons donc cherché d'où venait le problème en vérifiant que celui-ci ne venait pas du service DronePosition qui n'arrive pas à traiter assez rapidement les données reçues. Lors de ses différents tests nous nous sommes rendu compte qu'en enlevant le traitement des positions le nombre de positions enregistré diminué. Nous avons donc pensé que cela venait d'un problème de lock de la base de données et avons tenté de résoudre ce problème en créant un cluster, n'ayant pas eu les résultats attendus nous avons abandonné le cluster MongoDB. Pour mitiger le risque de problèmes dû au stockage de données dans un premier temps nous pouvons limiter à 1000 drones au maximum qui effectue des livraisons en même temps dans ce cas le serveur arrive à stocker assez vite les positions pour que le retard ne soit pas perceptible.

Traitement des positions erronées

Dans l'optique d'améliorer la robustesse de notre système, nous avions en tête de poursuivre le travail déjà effectué pour la détection d'envoi de positions erronées pour le traitement de ces dernières.

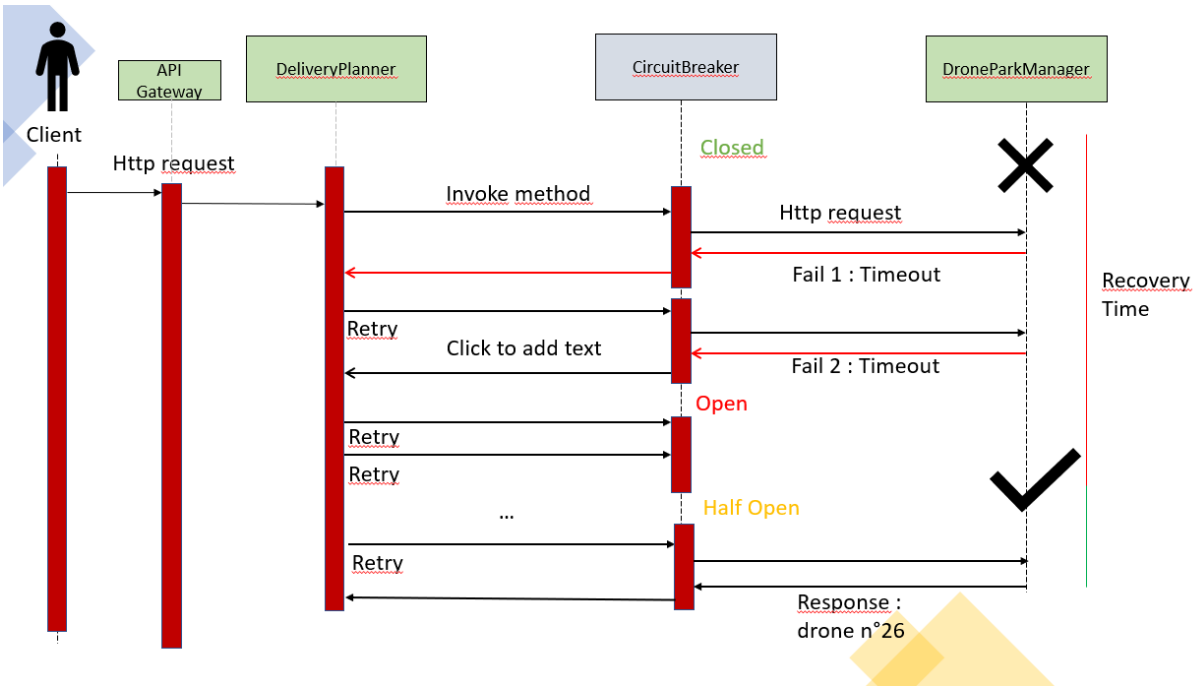
En effet, lors de la première partie du projet, nous avions implémenté un système de détection des positions erronées qui intervenait juste avant l'archivage des positions. Si la position à stocker est trop éloignée de la dernière position connue, alors un message est émis sur le bus indiquant qu'une mauvaise position a été envoyée par un drone.

Nous sommes donc parti de ce message et nous avons rajouté un service, DefectManager qui va archiver les mauvaises positions détectées pour en garder trace, mais il va également déterminer si un drone est défectueux.

Cela se fait à l'aide d'une petite machine à états, si un drone envoie une position erronée, il est mis dans un état de warning. S'il renvoie une position erronée dans un laps de temps très court, on va lui demander de redémarrer. Si après redémarrage le drône continue de renvoyer des positions erronées, il va être marqué comme défectueux et un message sera alors émis sur le bus pour informer que ce drône ne doit plus servir pour d'autres livraisons.

Résilience des services

Afin de fiabiliser nos services, nous avons implémenté le pattern circuit breaker sur le service *DeliveryPlanner*. Ce service dépend du service *DroneParkManager* et *ItineraryPlanner* pour planifier la livraison d'une commande et l'envoie des informations à un drone. Le circuit breaker permet de contrôler la fiabilité des requêtes entre les différents services afin d'offrir une grande tolérance à la latence et à l'échec.
Comme on ne sait pas s'il s'agit d'une erreur temporaire ou non, on a implémenté en plus, le pattern retry qui permet en cas d'échec, de reproduire la requête un certain nombre de fois. Les requêtes sont répétées avec un délai qui s'incrémente (x4 pour la démonstration, x10 initialement). Enfin, le circuit breaker utilise la pattern timeout lorsque la latence est trop élevée.



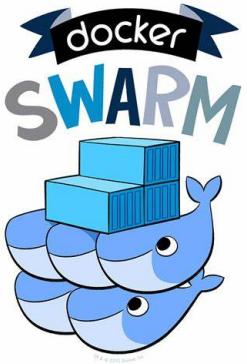
Sécurité du serveur:

Pour détecter les attaques DDOS on a mis en place Snort , un outil de détection d'intrusion.

On a d'abord commencé par définir le nombre de drones qu'on manipule, on s'est basé sur des statistiques de livraison de colis pour décider de se limiter à 1000 drones et on a fait le calcul pour déterminer le nombre de segments TCP que notre serveur devra recevoir dans un laps de temps.
Chaque drone envoie une requête par seconde et en moyenne, on reçoit 5 segments TCP par requête.
Pour 1000 drones et en 10 secondes , on devra recevoir 50000 segments TCP.
Et vu qu'un ensemble de drones peut perdre la connexion , on a ajouté une marge de 10000 segments TCP, et donc lors de la reconnexion même si on reçoit toutes les données en même temps, les alertes Snort ne seront pas déclenchées.
Et on a fini par utiliser une règle qui envoie des alertes si on reçoit plus de 60000 segments tcp dans une période de 10s.

Et pour mitiger les attaques DDOS, on pourra bloquer les adresses ip source identifiées par snort comme étant à l'origine de l'attaque, et on pourra également filtrer nos sources selon leur position géographique, en utilisant des équipements spécifiques pour cela, vu qu'on est opérationnel que dans une ville.

Déploiement



Nous avons décidé de déployer notre système sur un cluster Docker Swarm. Swarm est un choix judicieux au vu du temps que nous avions. En effet, créer un Cluster Swarm est assez simple, la transition entre un déploiement Compose et Swarm est rapide.
Swarm propose également des features intéressantes pour notre projet. Il nous permet d'atteindre les objectifs de robustesse, résilience et charge grâce aux points suivants:

- Swarm nous a permis de créer des réplicas de services assurant résilience et charge.
- Il propose également un load-balancer associé. Ceci permet de répartir la charge entre les réplicas, et de redistribuer les requêtes en cas de crashs d'un service.
- Swarm permet de détecter les crash de service via des healthcheck et de recréer un container. Actuellement nous n'avons pas de healthcheck, Swarm

détecte les crashs bruts de container lorsque le service associé se coupe totalement).
- Swarm est facilement déployable entre plusieurs Node et donc le cluster est facilement extensible.

Un des points négatifs de Swarm, c'est qu'il ne permet pas de faire de l'auto-scaling (comme pourrait le faire Kubernetes). Mais partant du postulat que nous pouvons savoir à l'avance le nombre de drones dans la compagnie, ce critère est mis de côté.

Perspectives Futures:

- Implémentation d'un système de Health Checks qui permet de demander à un service sur un serveur précis, s'il est en mesure ou non d'effectuer les tâches avec succès.
- Implémentation d'un solution pour permettre de stocker plus de positions de drones toutes les secondes