POLYTECH NICE SOPHIA

UNIVERSITÉ CÔTE D'AZUR

Livraison de colis par drone

ARCHITECTURES LOGICIELLES

2021-2022

ANAGONOU Patrick
ANIGLO Jonas
FRANCIS Anas
ZABOURDINE Soulaiman

Table des matières

I. Introduction 2

II. Scénarios et cas d'utilisation 2

A. Définition des personas 2

B. Scénarios 3

III. Architecture du système 4

A. Technologies 5

B. Services 6

C. Changement de région 11

D. Résilience 12

E. Comparaison RabbitMQ et KAFKA 13

IV. Organisation de l'équipe 16

V. Perspectives 16

I. Introduction

Dans le cadre du cours architecture logicielle, on a été amené à implémenter l'architecture d'un système de gestion de livraison de colis par drone. Pour le cours d'architecture logicielles 2 nous avons eu à apporter changements pour supporter de nouveaux besoins. Notre application est utilisée par une société de livraison de paquets via des drones. On suppose que nos drones seront toujours joignables par le réseau.

Notre sujet était la variante V10bis : « battery management, charging dock networks with optimization (docking station are on the path of heavy drones for long-distance deliveries) ». Pour cette partie 2 nous devions supporter les nouveaux besoins :

- **On veut une résilience maximale sur le flightMonitor**
- **Gestion du passage d'une région à une autre**
- **Développement et comparatif complet entre versions Kafka et RabbitMQ vis-à-vis des problématiques fonctionnelles et non fonctionnelles**

Dans ce rapport nous allons mettre l'accent sur les différences par rapport au premier rapport.

II. Scénarios et cas d'utilisation

A. DEFINITION DES PERSONAS

- Etienne
  Rôle : client chez un vendeur tier
  Besoin : En tant que client ayant passé une commande sur le site NicAchat, **je souhaite** recevoir mon colis au jour et à l'heure indiquée sur le bon de commande.

- Alice
  Rôle Employée d'une boutique de vente de réfrigérateurs (objets lourds). Elle vend des réfrigérateurs et les fait livrer à son client via Joly Drone.
  Besoin : En tant qu'employée d'une boutique de vente **j'aimerais** pouvoir suivre le statut des livraisons **afin de** rassurer mes clients sur l'arrivée prochaine de leur colis. (V4)

- Mathieu
  Rôle : Employée chez Joly Drone chargée de la maintenance et de la réparation des drones / station de charge
  Besoin 1 : En tant qu'employé de la maintenance des stations de recharge **j'aimerais** être notifié quand une station de charge ne fonctionne pas **afin que** j'aille sur place pour la réparer. (V10).

Besoin 2 : **En tant qu**'employé de la maintenance des stations de recharge **j'aimerais** pouvoir notifier Poly Drone que la réparation d'un drone a été faite **afin que** Poly Drone puisse à nouveau utiliser ce dernier pour une livraison.

- Gabriel
  Rôle : Superviseur des drones / Pilote
  Besoin 1 : **En tant que** superviseur des drones **je veux** que mes drones aient des plans de vol optimisés **afin de** livrer leur colis à longue distance sans panne de batterie. (V1 & V5)
  Besoin 2 : **En tant que** pilote de flotte de drones, **j'aimerais** suivre l'itinéraire des différents drones **afin de** reprendre le contrôle manuel en cas d'imprévus sur le plan de vol pour continuer la livraison des colis aux destinations. (V1)

- Marion
  Rôle : Superviseur de drone
  Besoin : En tant que superviseur de drones voulant transporter une charge à des distances très lointaine (possible de changement de région), j'aimerais suivre tous les drones qui sont (entrent) dans ma région.

B. SCENARIOS

- **Scénario** : Etienne achète un ordinateur portable sur le site de NicAchat et choisit de se le faire livrer via notre entreprise Joly Drone. Le site NicAchat contacte Joly Drone avec les détails du produit, Joly Drone lui fournit une estimation de prix et une estimation de la date de livraison. Etienne paie le prix de la livraison par drone. Il reçoit son bon de commande par mail lui indiquant le jour et l'heure de la livraison. Après que le drone chargé de la livraison de Etienne est positionné pour faire la livraison, Gabriel directeur logistique chez NicAchat reçoit sur sa console de contrôle une notification lui affichant le plan que le drone va suivre pour effectuer la livraison.
  Gabriel peut donc valider le plan de vol et lancer la navigation automatique du drone, et Etienne recevra son colis une fois la navigation du drone terminé.

- **Scénario** : Un client vient contacter Alice pour lui demander où en est la livraison de son colis. Alice saisit le numéro de la commande qui lui a été donnée par le client, dans la console de notre application. Si la livraison n'est pas encore lancée un message disant *en cours de préparation* lui est affiché. Si la livraison est lancée le statut du drone lui est affiché.

- **Scénario** : Avant de faire une nouvelle escale pour recharger la batterie du drone X61 à la station de recharge C3, une notification a été envoyé au centre lui informant que toutes les bornes de cette station sont tombées en panne, le centre va se charger de notifier les drones qui sont concernés par ce message (dont l'itinéraire inclut de passer par la station C3). Le centre va ensuite notifier Gabriel afin qu'il prenne le contrôle manuel sur X61, en attendant le recalcul d'un nouvel itinéraire optimisé qui va être communiquer à X61 lui permettant de continuer sa navigation automatique.

- **Scénario** : Un drone vient d'arriver à la station de recharge B2 pour se recharger. Il se branche sur une borne de cette station de recharge. Il se rend compte qu'il n'y pas de courant. Le drone envoie une notification de type, *no electric power in dock* ainsi Mathieu sera notifié directement. Il pourra en ce moment se rendre sur la station de recharge et faire la maintenance de la borne. En parallèle le centre est aussi notifié pour prendre les mesures nécessaires pour assurer le succès de la livraison (par ex : envoie d'un nouveau drone).

Scenario introduit en AL2

- **Scénario** : Un drone transportant un colis de Nice vers Paris vient de passer dans la région Ile-de-France, Gabriel qui est le superviseur de la région sud de la France le voit disparaître de son écran et Marion qui est le superviseur de la partie Nord de la France (incluant l'île de France le voit apparaître sur son écran de surveillance).

III. Architecture du système

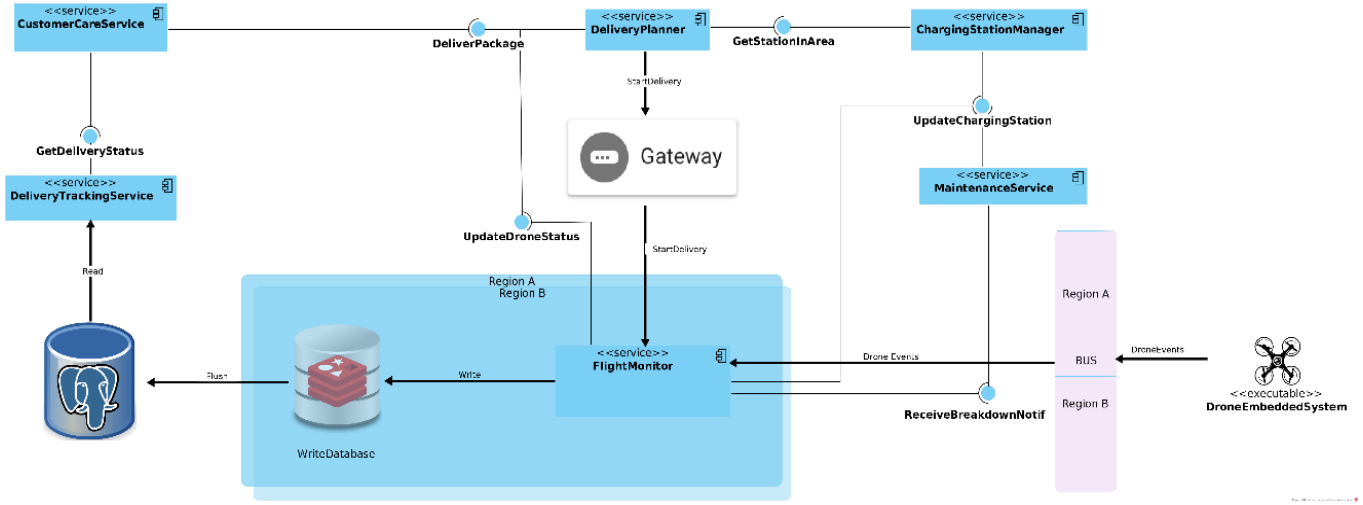

Figure 1: Architecture de l'application

Notre architecture a été changée afin de pouvoir supporter le changement de région. En effet l'objectif est de subdiviser l'espace géographique en plusieurs régions. Ce changement impacte le service FlightMonitor étant donné qu'il est responsable de suivre les drones et de sauvegarder leurs progressions. Etant donné que les drones pourront désormais traverser des régions on veut éviter à un service ou à un client de devoir contacter toutes les régions avant de pouvoir trouver le statut d'une livraison. Pour cela on réduit la responsabilité du FlightMonitor en dégageant un nouveau service DeliveryTrackingService. Ensuite, nous plaçons une Gateway qui servira de point d'entrée unique pour tous les flightMonitors des régions. Cette Gateway redirige une demande

de livraison en fonction de son point d'origine. Nous avons également fait une séparation logique du bus de message en fonction des régions.

Possédant plusieurs instances de FligthMonitor, la problématique d'avoir accès aux informations d'une livraison donnée s'est très vite posée. En effet dans l'ancienne implémentation dans laquelle nous n'avions qu'un seul service de FlightMonitor, toute la responsabilité de sauvegarde d'historiques et de données temps réels des livraisons lui était confié. En dégagent un nouveau service DeliveryTrackingService nous confions à ce dernier cette responsabilité. Ce qui revenait à avoir dans le service DeliveryTrackingService une base de données devant contenir toutes les données de livraison. Seul problème est que le nouveau service et toutes les instances de Flightmonitor devront se partager la même base donnée, les flightMonitor majoritairement en écriture et le nouveau service en lecture. Sachant que nous sommes partis sur le gestionnaire de base de données relationnel PostgresSQL, nous nous retrouveront très vite avec des temps de latences considérables qui seront causée par des locks d'écritures. Pour pallier ce problème nous sommes inspirés du pattern CQRS pour mettre en place un système de synchronisation de lecture écriture. Nous avons utilisé comme base de données d'écriture REDIS, ce choix a été fait pour la capacité de cette base de données en mémoire à être très rapide. Nous avons alors gardé la base de données de lecture en PostgreSQL, car elle est persistante et pourra donc garder les informations de livraisons. Pour la synchronisation qui devrait consister à ce que la base de données PostgreSQL ait toutes données de REDIS nous avons mis en place un scheduler qui se lance toutes les 15 minutes. La méthode appelée essaye d'obtenir un Lock sur la base de données, ce système de lock a été implémenté avec REDIS, et la mise en place d'un lock permet de s'assurer qu'on aura un seul écrivain ainsi donc amortir une éventuelle montée en charge. Une fois le lock obtenu la méthode retire les données de REDIS pour les insérer dans la base de données PostgresSQL.

Pour les messages signalant qu'une livraison est finie le `flightMonitor` tente d'obtenir un lock d'écriture pour insérer cette donnée dans la base de données de lecture. Le but étant de rendre disponible l'information de livraison complète le plus vite possible.

A. TECHNOLOGIES

Spring Boot - Java 17 : Nous avons un typage fort qui permet à l'équipe de bien voir les objets qu'on fait transiter entre nos services. Spring Boot offre de nombreuses dépendances comme data-repository permet d'exposer automatiquement des routes REST pour nos entités et Actuator pour monitorer l'état de nos services.

PostgreSQL : nous avons choisi une base de données relationnelles parce qu'elle offre la possibilité de faire des transactions pour l'allocation d'un drone pour une livraison (afin de ne pas allouer un même drone pour deux livraisons en concurrence). Mais dans l'implémentation nous n'avons pas utilisé les transactions.

Python => Java : Nous avions utilisé Python pour programmer le comportement de nos drones. Nous avons choisi cela pour la rapidité de développement et parce que la logique du drone n'était

pas trop complexe. Pour cette deuxième partie nous sommes passés à Java parce que la logique commençait à devenir importante, utiliser du Java nous permet aussi d'avoir une pile technologique moins diverse, du code et des types de données partageables avec nos services et du code plus maintenable sur du long terme.

RabbitMQ : Nous avons utilisé RabbitMQ pour faire de la communication entre notre application et les drones. Cette solution a été choisie parce qu'elle nous évite de gérer l'adresse IP des drones ou des WebSocketSession pour maintenir la connexion entre les drones et l'application.

B. SERVICES

1) Gateway

Dans l'architecture que nous avons proposé précédemment le `Gateway` est implémenté utilisant Spring Cloud Gateway.

Ce service a comme responsabilité d'orienter la requête envoyé par le `deliveryPlanner` vers le service `flightMonitor` de la région où la livraison va démarrer. Etant donnée que on instanciera plusieurs service `flightMonitor` pour chaque région.

Le Gateway contribue à la résilience du `flightMonitor`. C'est-à-dire que si une (ou toutes) les services `flightMonitor` tombent en pannent, le Gateway fait des retry afin d'attendre que ces services redémarrent avant de forwarder la requête.

Donc le Gateway contient une méthode CustomRouteLocator :

```
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder
builder) throws Exception
```

Cette méthode définit la logique pour orienter la requête vers le bon service.
Si la région n'existe pas, ou bien si la position de démarrage de la livraison n'est définie à l'intérieur d'aucune des régions, alors une exception sera levée.

2) DeliveryTrackingService

Ce service permet de rendre la recherche de l'état d'une livraison indépendante des régions. Elle expose une route reste pour récupérer l'état d'une livraison à partir de son numéro.

```
@GetMapping("/delivery/tracking/{trackingNumber}")
public ResponseEntity<DeliveryStatus>
getDeliveryStatus(@PathVariable("trackingNumber") String
trackingNumber)
```

3) CustomerCare

Le service CustomerCareService est chargé de traiter les requêtes concernant l'utilisateur. Elle expose 2 routes essentielles en REST. Ces routes font appelles chacune aux méthodes EstimateCost, ConfirmDelivery et CheckDeliveryStatus.

- EstimateCost : Estimation d'un coup de livraison de colis par un client.

```
@PostMapping("/customers/package/cost")
public CostEstimation estimateCost(@RequestBody @Valid paquet)
```

Il s'agit de la phase initiale avec laquelle le client intergit avec notre système. Il renseigne les informations à propos de son collis, le composant customerCare recupère l'objet puis fait appel au composant DeliveryCostEstimator pour calculer le prix puis le retourne.

- ConfirmDeliveryOrder : Confirmer une livraison.

```
@PostMapping("/customers/package/confirm/{id}")
public DeliveryConfirmationResult
confirmDelivery(@PathVariable("id") Package paquet, @RequestBody
@Valid CreditCard creditCard)
```

À cette phase le client confirme la livraison de sa commande. En effet à cette phase il fournit son numero de carte bancaire, le customerCare va faire appel au composant **PaymentService** qui va à son tour faire appel au service externe de banque. Il s'agit d'un simple serveur en python qui retourne ok avec le status 200 pour toutes les requêtes sur la route « / ». Après cela le CustomerCare va faire appel à la méthode DeliverPackage qui va appeler une route fournit par le service **DeliveryPlanner,** cette méthode retourne une réponse contenant le warehouse à partir duquel le drone débutera la livrasison et le trackingNumber(Numéro de suivi). Une fois ce résultat récupéré, leCustomerCare retourne une reponse contenant le résultat du payement auprès de la banque, le trackingNumber et le warehouse de livraison.

4) DeliveryPlanner

Le service DeliveryPlanner est chargé de traiter les requêtes concernant la plannification de livraison. Elle expose 3 routes essentielles en REST. Ces routes font appelles chacunes aux méthodes deliverPackage, getNewFightPlan de DroneAllocator et updateDroneStatus de DroneAvailibilityManager. Toutes les interfaces consommées par ce service sont en REST.

- DeliverPackage : livraison d'un packet, cette interface est utilisée par le service CustomerCareService après le paiement d'une commande.

```
@PostMapping("/deliver")
public DeliverPackageResponse deliverPackage(@RequestBody
Package deliverPackage)
```

Il prend en paramètre le packet à livrer, puis fait appel dans un premier temps au composant **DroneAvailabilityManager** pour recupérer tous les warehouses. Ensuite il passe ces warehouses et la destination du paquet au composant **PathFinder**. Ce dernier va faire appel à l'interface **GetStationsInArea** servit par le service **ChargingStationManager**. Ce service met à disposition une route qui sera appellée par la méthode getStationInArea, qui permet de demander les stations se trouvant entre une origine et une destination au **ChargingStationManager,** les stations recupérées sont ensuite triées par rapport à leur distance à l'origine**.** Le pathFinder recuperera les stations entre chaque warehouse et la destination du colis. Une fois recupérées il va ensuite utiliser notre algorithme[1] pour trouver un chemin entre une des stations recupérées et la destination. Le meilleur trajet en termes de distances minimum ainsi que le drone l'effectuant seront retenus et retournés au composant **DroneAllocator.** L'état du drone est alors changé en état de livraison**.** Ce composant fait alors appel à l'interface StartDelivery servit par le service **FlightMonitor.** Il met à disposition une route que va appéler la méthode startDelivery de l'interface StartDelivery pour demander la livraison par drone. Finalement le **DroneAllocator** retourne le trackingNumer de la commande ainsi que le warehouse d'où le drone decollera pour effectuer la livraison.

- GetNewFlightPlan : recupération d'un nouveau plan de vol par le service **FlightMonitor.**

```
@PostMapping("/drones/{droneName}/newFlightPlan/{droneCurrentCapacity}")
public FlightPlan getNewFlightPlan(@PathVariable(value = "droneName") String
droneName,@PathVariable(value = "droneCurrentCapacity") Integer
droneCurrentCapacity,@RequestBody OriginDestination originDestination)
```

Il prend en paramètre le nom du drone, la capacité de vol maximale que ce dernier peut parcourir *(on demande généralement un nouveau plan de vol quand un drone n'a pas réussi à se charger, du coup il aura déjà parcouru une distance donnée et il ne lui restera qu'une distance qu'il pourra parcourir),* ainsi que les coordonnées d'où il se trouve actuellement. Tout comme deliverPackage il va faire appel au composant **PathFinder** pour trouver le meilleur chemin, la seule différence est que cette fois ci le **PathFinder** devra prendre en compte l'énergie du drone pour atteindre la prochaine station. Cet après quoi la suite de l'algorithme reste inchangé car le drone se serait rechargé sur la

[1] Notre algorithme consiste dans un premier temps à trouver un chemin de manière naïve entre deux points passant par des stations. A chaque fois que la distance maximale que peut parcourir le drone lui permet peut atteindre une prochaine station dans l'ordre, il l'ajoute comme point d'arrêt. Une fois la destination atteinte. L'algorithme essaye d'optimiser le plan de vol en éliminant les arrêt inutiles, si le drone peut quitter une station pour celle qui la suit celle sur laquelle elle devait s'arrêter, elle ignorerait la station intermédiaire qui est inutile. De cette manière elle peut même faire un vol directe quittant l'origine sans arrêt pour la destination. Le point faible de cet algorithme est qu'il peut arrêter de rechercher un plan de vol lorsque la première itération naïve ne marche pas, parce qu'il a essayé de rejoindre une station trop loin avant d'accéder à une qui est plus proche. Ce point sera amélioré dans une prochaine version.

première station qu'elle atteindra. Le **PathFinder** retourne alors le nouveau chemin au **DroneAllocator** qui le retourne ensuite.

- UpdateDroneStatus : Mettre à jour l'état d'un drone.

```
@GetMapping("/drones/{name}/status/{droneStatus}")
public ResponseEntity<Drone> updateDroneStatus(@PathVariable("name")
String droneName, @PathVariable("droneStatus") DroneStatus droneStatus)

@PostMapping("/drones/{droneName}/delivering")
public Drone updateDroneCurrentPosition(@PathVariable(value = "droneName") String
droneName, @RequestBody Coordinates currentPosition)
```

Il permet de mettre à jour soit le *statut* d'un drone soit son état. Il sera utilisé par le **FlightMonitor** pour mettre à jour la position courante du drone lorsqu'il change de station et d'état lorsqu'il est cassé par exemple. La **MaintenanceService** l'utilisera aussi pour changer l'état d'un drone lorsqu'il est réparé.

5) ChargingStationManager

Le service ChargingStationManager est chargé de traiter les requêtes concernant la gestion des stations. Elle expose 2 routes essentielles en REST. Ces routes font appelles chacunes aux méthodes getStationInArea et updateChargingStation.

- GetStationInArea : recupère toutes les stations se trouvant entre deux points.

```
@PostMapping("/j/stations")
public ChargingStationList fetchStationsInArea(@RequestBody Area area)
```

Area contient un point A et un point B. **StationAvailability** va récupérer la liste des stations puis pour chaque station vérifier si elle se trouve entre le cercle passant par A et B et si cette station a au moins un terminal AVAILABLE. La liste de stations récupérées est ensuite retournée.

- updateChargingStation : mettre à jour l'état d'une charging station

```
@GetMapping("/j/stations/{name}/sentForRepair")
public ChargingStation sentForRepair(@PathVariable("name") String name)

@GetMapping("/j/stations/{name}/repair")
public ChargingStation repairStation(@PathVariable("name") String name)
```

La méthode *sentForRepair* permet de mettre à jour l'état d'une station pour inaccessible, il met alors tous les terminaux d'une ChargingStation à OUT_OF_SERVICE; elle sera utilisée par le **FlightMonitor** pour mettre à jour une station qui est tombé en panne. La méthode *repairStation* permet de mettre jour l'état d'une station comme étant

accessible. Elle fait le contraire de *sentForRepair*, c'est-à-dire qu'elle met les terminaux en AVAILABLE.

6) FlightMonitor

Le service FlightMonitor est chargé de traiter tout ce qui concerne la naviguation d'un drone. Elle expose 2 routes en REST. Ces routes font appelles chacunes aux méthodes startDelivery et getDeliveryStatus. Il utilise aussi une communication à base d'évenement avec le drone, il discute au travers 4 canaux : begin delivery, updateStatus, updateFlightPlan, sendMayday et deliveryFinished.

- StartDelivery : démarre le trajet d'un drone.

```
@PostMapping("/flight/start")
public String launchDrone(@RequestBody DeliveryDTO deliveryDTO)
```

  DeliveryDTO ici contient le plan de vol, le drone l'effectuant le numéro de livraison et l'état de livraison. Cette méthode sauvegardera la commande puis enverra un évènement dans le bus sur le canal *(drone.nom_du_drone.launch)* pour lancer *nom_du_drone*.

- updateStatus : reçoit du bus les mises à jour depuis le drone.

```
@Bean("listenerAdapterForUpdateLocation")
MessageListenerAdapter listenerAdapter(UpdatePositionMonitorBean
positionUpdateMonitorBean){
    return new MessageListenerAdapter(positionUpdateMonitorBean);
}
```

  PositionUpdateMonitorBean est le composant qui traite les mises à jour reçues par le drone. Il fait appel à UpdateDroneStatus pour mettre à jour la position du drone puis met à jour en local les stations déjà parcourues par un drone.

- Mayday : reçoit un message mayday d'un drone.

```
@Bean("listenerAdapterForMayDay")
MessageListenerAdapter listenerAdapter(MayDayMonitorBean
mayDayMonitorBean) {
    return new MessageListenerAdapter(mayDayMonitorBean);
}
```

  MayDayMonitorBean est le composant qui traite les *Mayday* reçus de la part des drones. L'évènement de *MayDay* est produit lorsqu'un drone arrive à une station n'arrive pas se charger. Dans un premier temps, Ce composant met à jour la station en question en mettant ses terminaux hors service via l'interface **updateChargingStation**. Il demande ensuite un nouveau plan de vol à **DeliveryPlanner** via l'interface GetNewFlightPlan, une fois le nouveau plan de vol reçu il le communique au drone via le canal updateFlightPlan. Après cela il recupère un nouveau plan de vol pour toutes les livraisons en cour passant par la station inacessible. Et pour finir il demande à la **Maintenance** de reparer la station endomagée via le ReceiveBreakdownNotif.

- DeliveryFinished : cet évènement est traité par le service par le **DeliveryDroneMonitor.**

```
@Bean("listenerAdapterForSpecialEvent")
MessageListenerAdapter listenerAdapter(RealtimeMonitorWS
monitorWS) {
    return new MessageListenerAdapter(monitorWS);
}
```

Il met à jour la livraison en changeant son état en *livré*, puis change l'état du drone en AVAILABLE via l'interface UpdateDroneStatus.

7) Maintenance

Le service Maintenance est chargé de traiter tout ce qui concerne la maintenance d'un drone et d'une station. Elle expose 2 routes en REST. Ces routes font appel chacunes aux méthodes receivebreakdownNotif et reparationDroneOrStation.

- ReceivebreakdownNotif : il permet de demander la reparation d'un drone ou d'une station.

```
@PostMapping("/j/station")
public String repairStation(@RequestBody String chargingStationName)

@PostMapping("/j/drone")
public Drone repairDrone(@RequestBody Drone drone)
```

- ReparationDroneOrStation : il permet de mettre à jour l'état d'un drone ou d'une station comme étant réparé, et pour cela il passe respectivement par UpdateChargingStation et UpdateDroneStatus

```
@PostMapping("/j/station/repair")
public ChargingStation
updateReceivedChargingStationStatusFromRepairMan(@RequestBody StationToRepairDTO
stationToRepairDTO)

@PostMapping("/j/drone/repair")
public Drone updateReceivedStatusFromRepairMan(@RequestBody DroneToRepairDTO
droneToRepairDTO)
```

C. CHANGEMENT DE REGION

Lorsqu'un drone envoie un message sur le bus de message celui-ci est consommé par un flightMonitor de la région correspondante. Le message est stocké dans une base de données Redis qui sert donc de cache dans chaque région. Ce cache se synchronise fréquemment avec une base de données PostgresSQL. Lorsque le drone évolue les informations qu'il envoie ont un champ clock (qui sert d'horloge logique). En effet au début de la livraison sa valeur est à 0 puis elle est incrémentée à chaque message envoyé. Du coté des services nous stockons tous les messages

envoyés sans réécrire les messages existants. Ainsi pour retrouver le dernier état d'une livraison nous trions les messages envoyés suivant ce champ `clock`. Nous utilisons le champ `clock` pour éviter l'utilisation de l'horloge qui peut être soumis à des problèmes comme le changement dérèglement des horloges des services et des drones ou leur non-synchronisation en particulier ou les différences de fuseau horaire.

D. RESILIENCE

Notre second objectif était de placer de la résilience sur le service flightMonitor. Pour cela nous avons ciblé notamment les communications entre le flightMonitor et les autres composants.

1) DeliveryPlanner et FlightMonitor

Nous avons mis en œuvre l'idempotence sur la route `startDelivery` c'est-à-dire que si par erreur on reçoit deux fois le même message de lancement de livraison on ne le traite qu'une seule fois.

Nous avons également renforcé la validation des messages reçus du `DeliveryPlanner` et nous retournons des statuts Bad Request lorsque ces requêtes sont invalides pour éviter des exceptions qui peuvent faire crasher l'ensemble du service au cours du traitement.

Enfin nous avons mis en place des mécanismes pour rejouer un certain nombre de fois les requêtes entrantes et sortantes du `FlightMonitor` s'ils n'ont pas pu atteindre leurs destinations.

Les appels du `DeliveryPlanner passant par un Spring Cloud Gateway`, les 'retry' ont été implémentés dans le gateway. En ce qui concerne les appels sortant du `FlightMonitor`, les 'retry' ont été implémentés avec Resilience4J.

2) Drone et FlightMonitor

Pour la communication entre le `FlightMonitor` et les drones nous avons distingué différents types de communication par importance : REGULAR, SPECIAL, MAYDAY.

Nous avons utilisé le système d'accusé de réception manuel de `RabbitMQ` pour nous assurer que nous sauvegardons les messages importants dans la base de données avant de confirmer à `RabbitMQ` qu'il peut les enlever de sa file de message. Ainsi si le service tombe entre après qu'il est reçu le message mais qu'il n'a pas pu le sauvegarder alors le message reste toujours dans `RabbitMQ` qui le renvoie. Nous avons laissé le mécanisme par défaut sur les messages de type regular qui nous permettent de suivre la position du drone parce qu'il est acceptable de perdre un certain nombre de ces messages.

3) Scénarios de panne

Nous avons étudié ce qui se passait dans notre architecture si certains éléments autour du FlightMonitor tombait en panne.

a) FlightMonitor

Nous pouvons déployer plusieurs instances de FlightMonitor destiné à une région donnée ainsi en cas de panne d'une région on peut toujours enregistrer les informations que les drones envoient. Pour réduire les risques liés à une telle panne il faut donc un orchestrateur de conteneur capable de redémarrer très vite les instances qui sont tombées.

b) Gateway

Si la Gateway est injoignable nous pouvons toujours gérer les drones en vols et accepter de nouvelles commandes de livraisons, en revanche nous ne pourrons pas démarrer de nouvelles livraisons.

c) Bus de message

Si le bus de message tombe en panne, nous ne pourrons plus suivre les drones en vols mais toutes les autres activités peuvent continuer. Pour réduire les risques liés à cette panne il faudrait avoir des clusters du bus de message déployés sur plusieurs régions.

E. COMPARAISON RABBITMQ ET KAFKA

Dans notre solution d'AL1 nous avions utilisé `RabbitMQ` pour communiquer entre le `FlightMonitor` et les drones. Nous avions fait ce choix parce que le setup de RabbitMQ était le plus rapide. Afin de voir si nous avions fait le bon choix nous avons donc voulu comparer RabbitMQ avec une alternative comme Kafka.

Kafka est basé sur les concepts de Topic et de partitions et de consumer group. Les topics sont un premier niveau de séparation des messages. Les messages sont ensuite répartis dans des partitions en fonction d'un critère que le développeur peut définir. Plus on a de partions plus on peut consommer des messages en même temps.

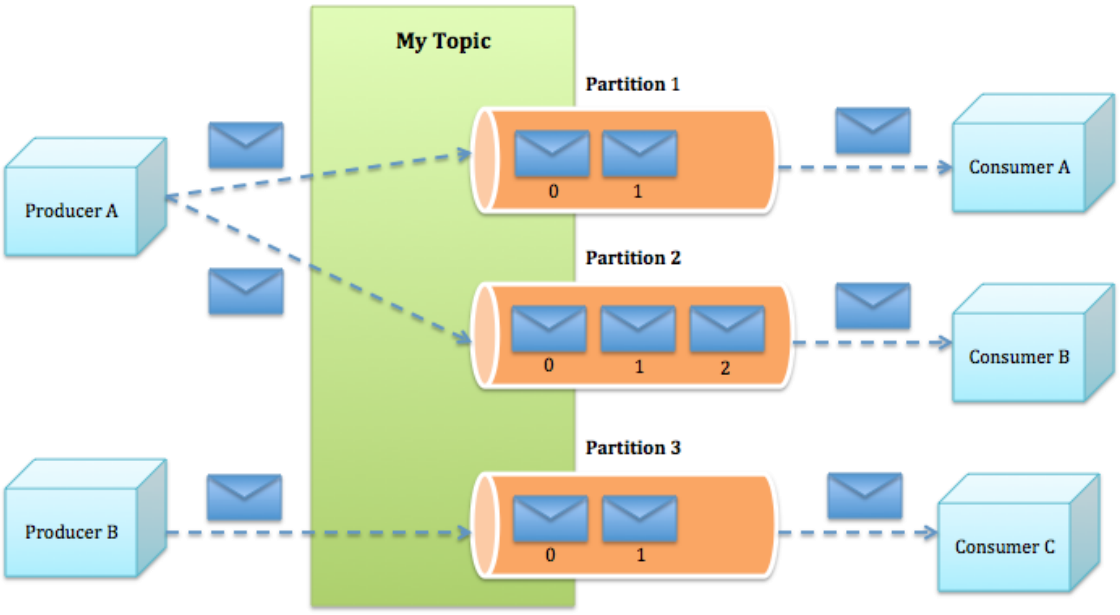

Figure 2 Architecture of Kafka

RabbitMQ est un distributeur de message plus complexe. Les messages sont envoyés sur des exchanges qui peuvent être de différents types : direct, topic, fanout, headers (qui n'est pas sur le schéma). Un exchange est ensuite associé à une file de message. Cette association peut être faite par un routing Key dans le cas des direct et topic exchanges.

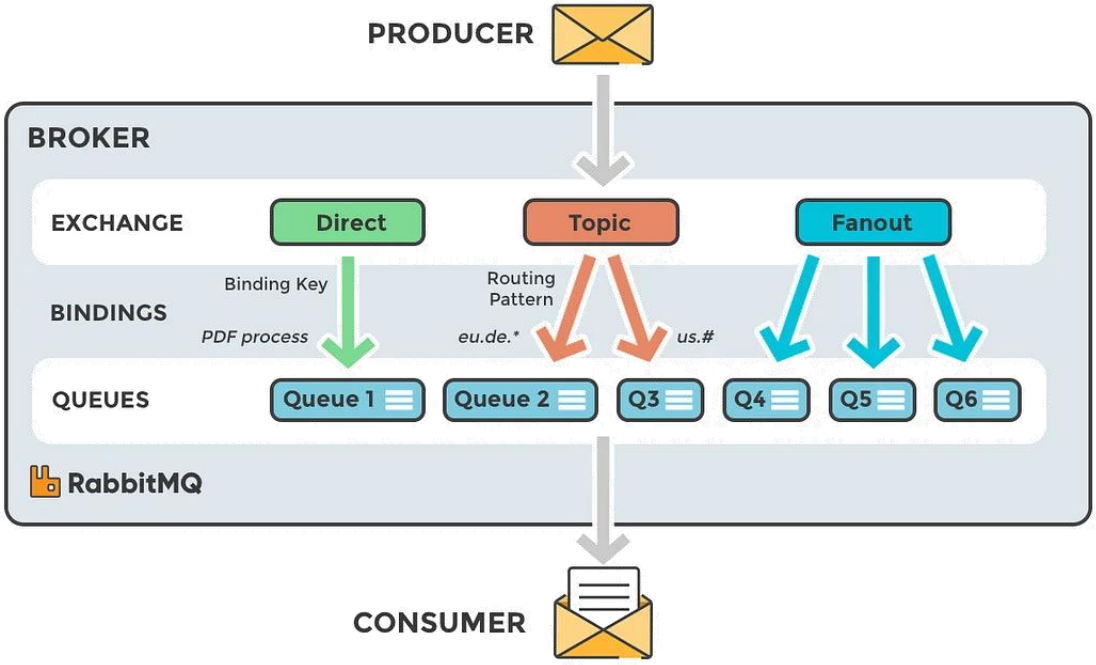

Figure 3 Architecture of RabbitMQ

Nous avons comparé l'utilisation de `RabbitMQ` et de `Kafka` sur les points de vue fonctionnels et non-fonctionnels. Pour ce faire nous avons remplacé sur une branche git toutes les utilisations de `RabbitMQ` par `Kafka`. Cela nous a permis de tirer des conclusions que nous résumons dans les tableaux suivants.

	Management & debug	Setup	Dependencies	Protocols
RabbitMQ	Official Web UI for management and debug	Minimum configuration for starting up	Standalone, no additional dependencies	Many underlying protocols supported
Kafka	No official UI but many third party's tools	More configuration parameters to start	Need Zookeeper to run	Binary protocol over TCP

Table 1: Comparaison non-fonctionnelle de Kafka et RabbitMQ

	Guarantee Message Delivery	Routing rules for grouping and separation of events	Reply / react to an event	Scalability
RabbitMQ	With manual consumer acknowledgments, messages are kept until consumed	Many ways to configure the routing of a message	The message is pushed to the process	Just add or remove consumers on the fly
Kafka	Message replay and acknowledgments	Only topics and consumer groups	The process must pull the message, so the work is shifted to the program	Can add partitions, can't remove them at runtime

Table 2 : Comparaison fonctionnelle de Kafka et RabbitMQ

Ainsi notre solution de bus de message avec RabbitMQ reste pour nous la solution idéale.

IV. Organisation de l'équipe

Afin de couvrir le développement de nos fonctionnalités nous avons marqué chaque fonctionnalité comme Spike que nous avons subdivisé et réparti à chaque membre de l'équipe.

ANAGONOU Patrick : Gestion changement de régions + Comparatif Kafka et AMQP.

ANIGLO Jonas : Synchronisation de REDIS avec PostgreSQL + implémentation des visualisations du chemin des drones pour la démo.

FRANCIS Anas : Implémentation de la Gateway + maintien des tests d'intégration continue sur Github Actions.

ZABOURDINE Soulaiman : Implémentation de la résilience sur les services.

V. Perspectives

Notre application est un POC qui permet de démontrer notre sujet de livraison de paquets sur de longues distances avec des drones. Afin de l'améliorer on pourrait :

- Supporter des coordonnées plus standards et des adresses : actuellement nous utilisons une simple grille avec des positions x, y. Il faudrait donc ajouter le traitement d'adresse (numéro de rue, code postal,) et des coordonnées GPS (latitude, longitude, altitude).
- Améliorer l'algorithme de création de plan de vol
- Améliorer la résilience sur tous les services
- Replacer la fonctionnalité de MAYDAY