Blue Origin X

Team B

Masia Sylvain
Montoya Damien
Peres Richard
Rigaut François

2020 – 2021

Sommaire

Architecture

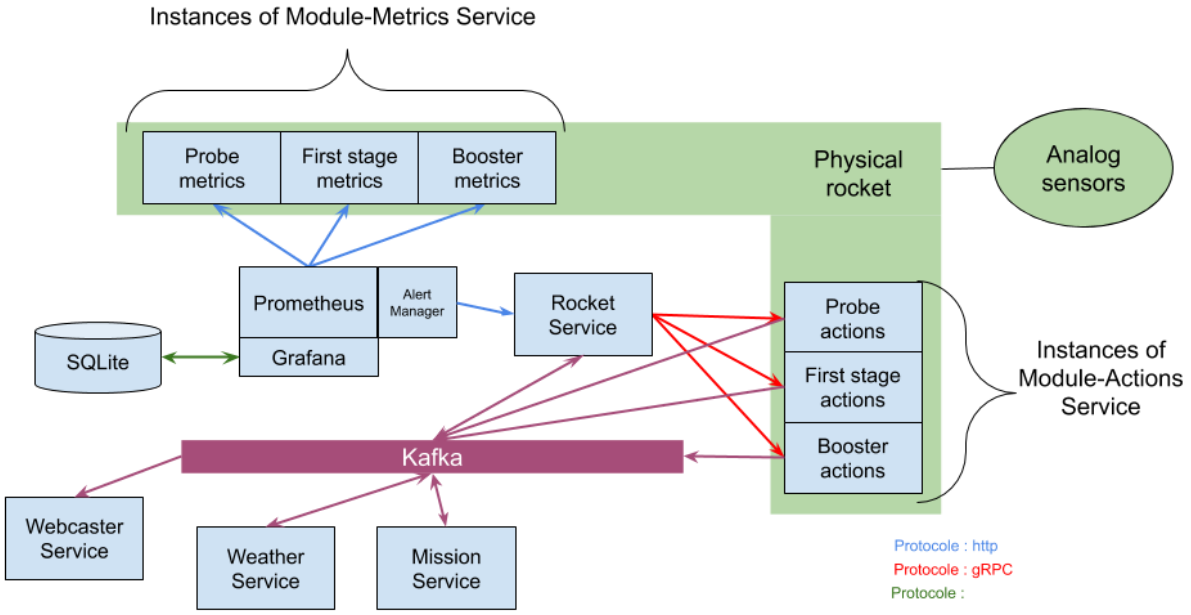

Figure 1 : architecture global du projet Blue Origin X (Voir en grand)

Workflow global

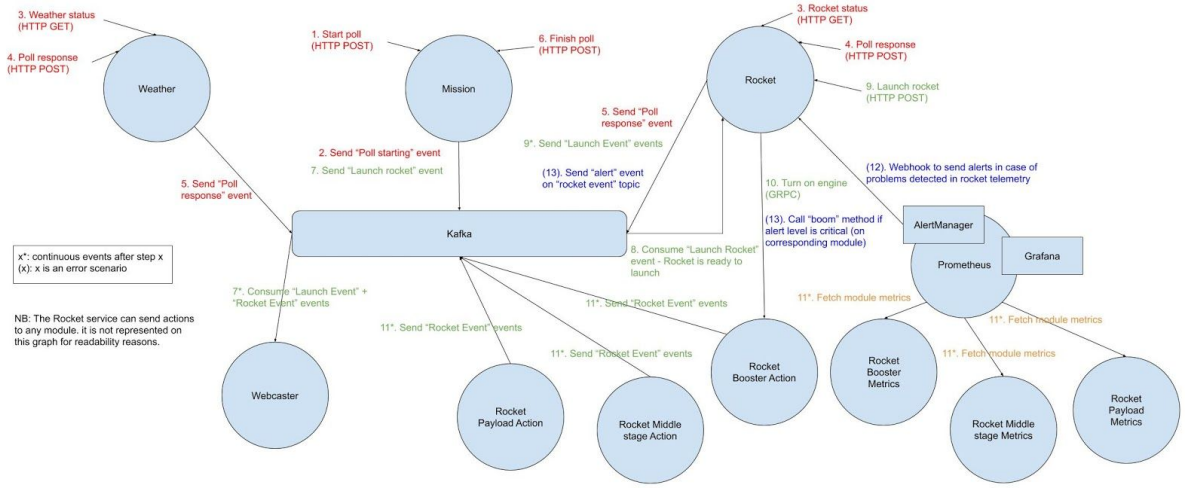

Figure 2 : Dataflow des scénarios du projet Blue Origin X (Voir en grand)

Endpoints et objets métiers

Weather service

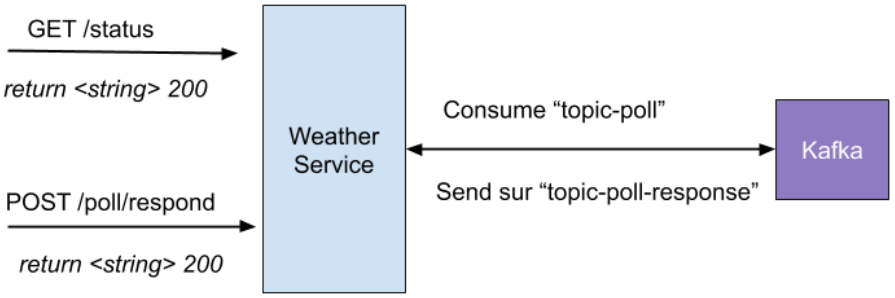

Figure 3 : Endpoints et bus du service Weather

Mission service

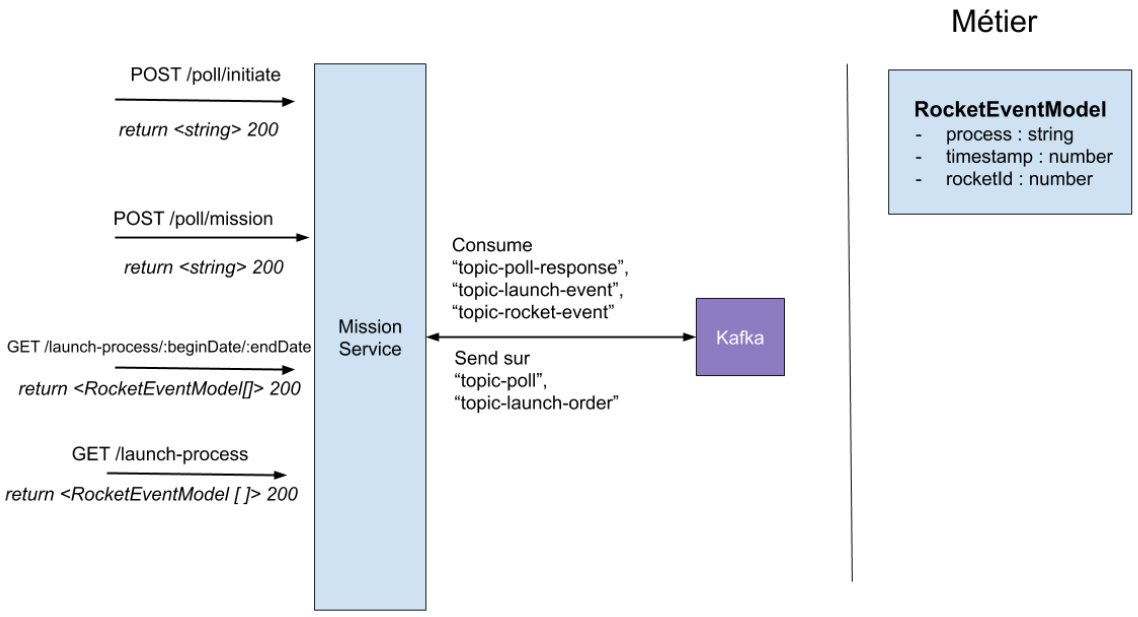

Figure 4 : Endpoints, bus et objets du service Mission

Rocket service

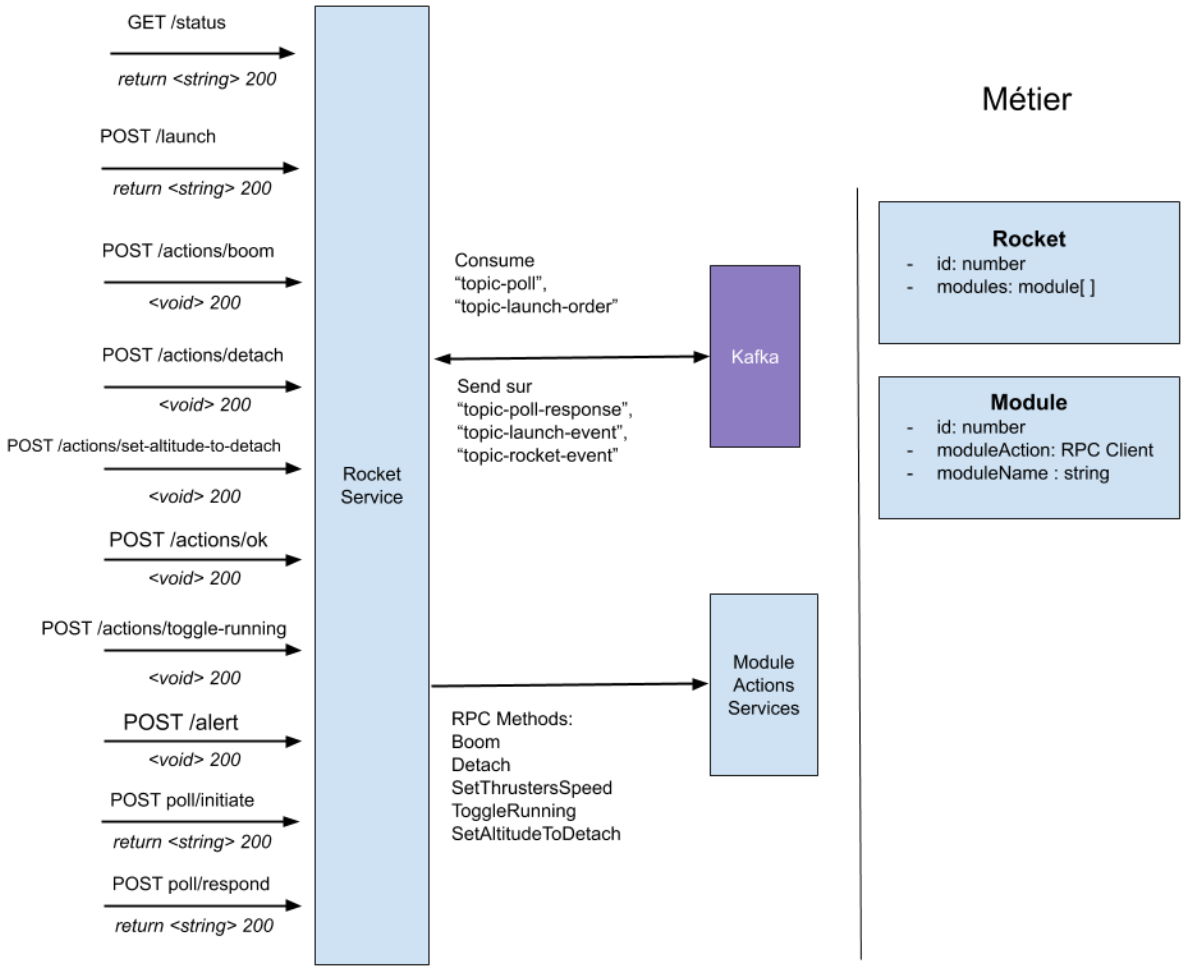

Figure 5 : Endpoints, bus et objets du service Rocket

Webcaster service

Figure 6 : Bus du service Webcaster

Module-metrics services

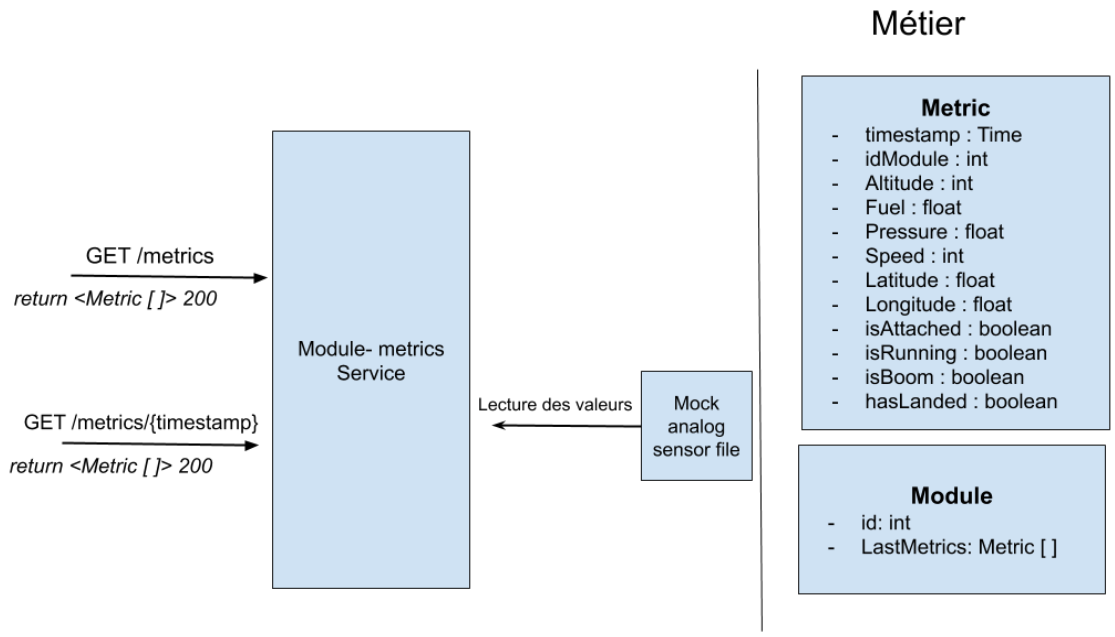

Figure 7 : Endpoints et objets des services Module-metrics

Module-actions services

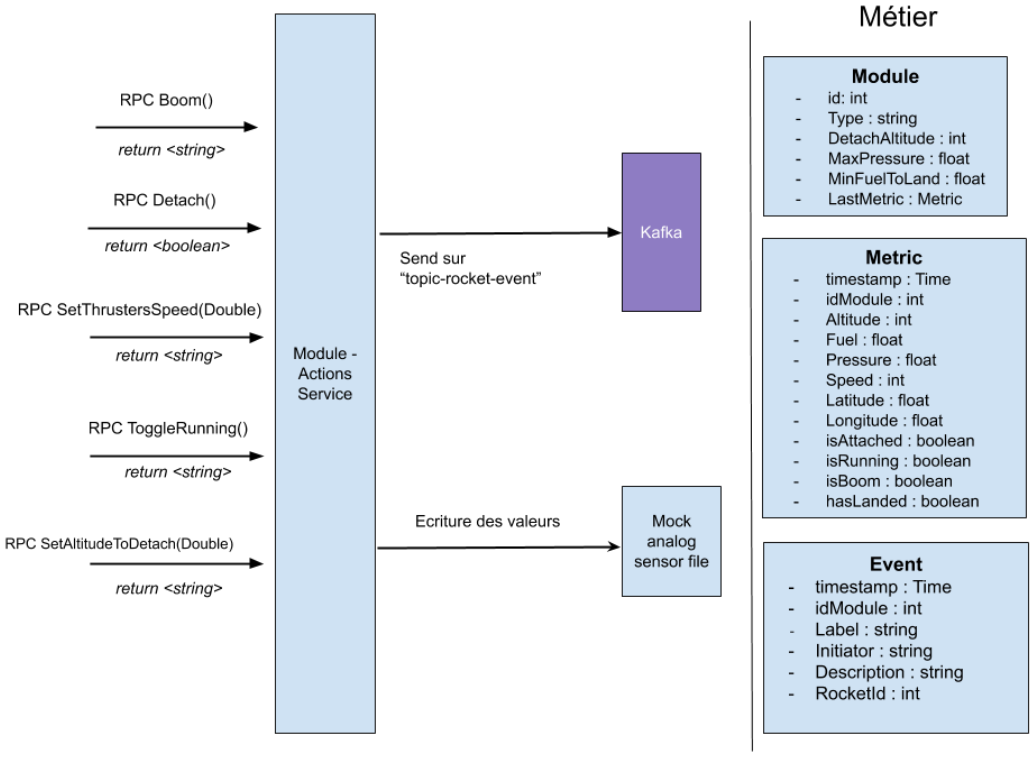

Figure 8 : Endpoints et objets des services Module-actions

Développement des user stories

User story	Acteur (s)	Implémentation
1 - check the weather status	Launch Weather Officer	Une route http GET sur le service weather permet à l'acteur d'obtenir la météo.
2 - monitor the status of the rocket	Chief Rocket Department	Une route http GET sur le service rocket permet à l'acteur d'obtenir le statut de la fusée.
3 - perform a go/no go poll	Mission Commander (MC) Launch Weather Officer (WO) Chief Rocket Department (CR)	Le MC envoi une requête POST sur le service mission pour démarrer le poll. Ce service envoi l'événement de début de poll sur le bus Kafka. Les acteurs WO et CR envoient ensuite leur réponse (true pour prêt sinon false) via une requête POST sur leur service respectif, weather ou rocket. Le service transmet alors la réponse sur le bus Kafka. Le MC peut alors envoyer une requête POST sur le service mission pour indiquer si tout est bon ou non. Le message de la requête est ainsi transféré sur le bus Kafka pour que le CR puisse récupérer l'ordre de lancement.
4 - send the launch order	Chief Rocket Department	Une requête POST sur le service rocket permet à l'acteur d'envoyer l'ordre de lancement. Cela va également initialiser la séquence de lancement et envoyer les messages sur le bus Kafka.
5 - receive, store, consult the telemetry of rocket	Telemetry Officer	Chaque métrique de chaque module est stockée en local dans le module. Elles sont générées de manière semi aléatoire et sont exposés via un "puit" sur le service Module Metrics. Prometheus nous permet de récupérer à interval régulier ces données en requetant sur ces puits de modules. Grâce à l'API de prometheus, une simple requête GET permet de consulter les données. L'acteur dispose également d'une interface graphique via Grafana pour consulter les données de manière plus efficace.
6 - stage the rocket mid-flight	Chief Rocket Department (CR)	Pour séparer la fusée nous avons mis en place 2 moyens d'y parvenir, un moyen automatique et un manuel. Premièrement, le service action situé sur le module booster dispose d'une valeur minimum de fuel. Une fois cette valeur atteinte, le booster va automatiquement se détacher du reste de la fusée puis atterrir. Le second moyen est de le faire manuellement. En effet, le CR peut envoyer une requête POST sur le service rocket en précisant l'identifiant de la fusée et l'identifiant du module. Le service va alors utiliser une connexion gRPC pour contacter le module et ainsi envoyer l'ordre de se détacher. Quand le module est détaché, un message est alors envoyé sur le bus Kafka.

7 - deliver the payload	Chief Payload Department	Comme pour la story précédent, nous avons mis en place deux moyen de délivrer le payload. Un moyen manuel et un moyen automatique qui fonctionnent tous deux comme dans la story précédent. Un message est également envoyé au bus Kafka une fois le payload délivré.
8 - destruction of the flight hardware	Mission Commander	Une route POST est présente sur le service rocket et permet de détruire n'importe quel module. Le service va alors utiliser la connexion gRPC pour envoyer l'ordre de destruction au module. Avant de se détruire, le module va se détacher pour ne pas faire exploser toute la fusée si un seul module est défaillant. Une fois la destruction enclenchée, le service action va signaler au service metrics du module que le module va exploser afin qu'il s'arrête d'émettre un signal avant cela. Cela permet aussi d'avoir les 2 modules qui "explosent" en en contactant qu'un seul et sans connexion particulière entre ces 2 là (pour les besoin de la démo, dans la vraie vie nous pensons que l'explosion suffit pour arrêter d'émettre).
9 - booster to land	Chief Executive Officer	Quand le booster atteint la valeur minimum du fuel pour atterrir, il va automatiquement déclencher la procédure pour arrêter. Des messages sont alors envoyés sur le bus Kafka pour décrire la procédure.
10 - receive, store, consult the telemetry of the first stage	Telemetry Officer	Comme pour le booster, les métriques fonctionnent de la même façon.
11 - receive, store, consult the telemetry of the payload	Chief Payload Department	Comme pour le booster et le premier étage, les métriques fonctionnent de la même façon.
12 - go through Max Q	Chief Rocket Department	Pour gérer le cas Max Q nous avons également deux méthodes mise en place. Une méthode automatiquement permettra en fonction de la pression du module de réduire automatiquement sa vitesse pour réduire la pression exercée sur le module. Une méthode manuelle permet via une requête POST sur le service rocket, d'éteindre ou ralentir les moteurs ce qui aura pour effet de réduire la vitesse et donc la pression.
13 - launch procedure events sequence	Mission Commande	Comme dit précédemment, lors du lancement de la fusée les logs de procédure sont envoyés sur le bus Kafka. Le service mission s'abonne à ce topic pour récupérer toutes les informations mais également les persister en base de données.
14 - store, consult the mission logs	Mission Commander	Comme expliqué dans les précédentes user stories, chaque action et événement se produisant lors de l'ordre d'envoi et ce jusqu'à la livraison du payload / l'atterrissage du booster,

		chaque log est envoyé sur le bus Kafka. Nous avons mis en place deux topics, l'un pour la procédure de lancement (rocket status, Startup(T-00:01:00)...) et un autre pour tous les événements se produisant lorsque la fusée décolle (max q, atterrissage, détachement…). Ainsi, le service mission s'abonne à ces deux topics et peut récupérer toutes les informations et ainsi les persister en base de données.
15 - be aware of the launch procedure events	Webcaster	Comme les événements sont envoyés au bus Kafka, notre service webcaster s'est tout simplement abonné au bus pour recevoir toutes les informations. Il les rend ensuite disponible via un endpoint http GET.
16 - handle multiple launches	Chief Executive Officer	Pour mettre en place cette user story, nous n'avons pas en beaucoup de changements à apporter. Cela est notamment dû à notre architecture globale et comment chaque composant communique avec les autres (fort découplage). En effet, nous déclarons chaque rocket (avec un identifiant) dans le docker-compose avec tous ses modules et leur identifiant. Grâce à cela, le service rocket pourra ainsi lire une variable d'environnement contenant des tuples (idRocket, idModule, nomModule). Grâce à cela, il sera facile de déterminer quelle fusée existe ainsi que les modules associés. Nous avons ainsi modifié le poll en spécifiant un identifiant de rocket et toutes les actions peuvent être lancées depuis le service rocket en précisant un identifiant de rocket et de module.
17 - trigger anomalies of rocket	Mission Commander	Étant donné que nous utilisions déjà Prometheus pour les metrics du module, nous avons choisi d'utiliser Prometheus AlertManager afin de déclencher des alertes en fonction des données qu'il reçoit des métriques. Il permet de les stocker et de configurer un webhook lorsqu'une alerte est détectée. Ainsi, n'importe quel opérateur ou service (présent ou futur) peut "s'abonner" au webhook pour être alerté.
18 - aborted mission due to detection of anomalies	Mission Commander	Grâce à la configuration du webhook dans la configuration de AlertManager et à la création d'un endpoint POST /alert sur le service rocket, nous avons pu recevoir les anomalies et déclencher la destruction des parties de la fusée concernées par une anomalie de type critique. (Le niveau de criticité d'une alerte étant également déjà implémenté dans alertManager)

Systèmes au sol

- HTTP - interaction avec les utilisateurs: opérations simples (CRUD), branchement facile avec tout type d'interface ou de système de contrôle. Pas de REST à proprement parler car pas ressenti le besoin d'avoir des données formalisées de manière stricte.
- Bus Kafka pour le poll go / no go: On n'a pas besoin de recevoir de réponse immédiatement et tout doit être asynchrone, ce qui justifie l'utilisation du bus. De plus, cela permet de découpler un maximum et rendre le projet modulable facilement : par exemple ajouter un service dans le poll (service de la Nasa qui doit donner son autorisation de décollage), cela pourrait se faire très facilement, puisque le service de mission ne fait qu'envoyer un événement de démarrage de poll. Il suffirait de brancher le nouveau service sur le bus kafka, et de dire à mission d'attendre un service de plus pour que la poll soit effectuée correctement.
- Sur le bus, des événements concernant la fusée sont également envoyés afin qu'ils puissent être récupérés pour stockage ou simplement pour de la visibilité en temps direct si besoin est.
- Service de télémétrie accessible avec un dashboard Grafana pour la lisibilité des métriques
- Enfin tous les composants et services au sol sont tous inter-connecté via un middleware Traefik. Il permet de faire du reverse proxy, très utile pour les endpoints des services car il permet de s'abstraire de la partie URL+PORT du networking pour ne garder que la gestion des paths obligatoires lorsque l'on fait des services http. L'autre avantage de Traefik et qu'il est directement "connecté" au daemon docker et permet donc, en plus de la détection de nouveaux services et leurs configuration dans le docker compose uniquement, de faire du Loadbalancing sur les services s'ils sont dupliqué. Par exemple, si le weather service reçois trop de trafic qu'il y ne peut plus gérer, on peut scaler le service via docker et Traefik répartira la charge sur les X versions répliqués.

Systèmes dans l'espace

Séparation de la fusée en modules - permet d'avoir le contrôle sur chaque module individuellement (ex: on fait exploser un module à la fois) et idem pour les métriques (l'évolution de la télémétrie de chaque module permet de faire la corrélation avec les événements, ex: détachement du premier étage).
Deux services par module:

- module-actions: permet de recevoir les ordres manuels depuis le sol (ex: boom)
- module-metrics: service qui rend les données de télémétrie accessibles

Cette séparation nous permet d'avoir un meilleur contrôle sur la simulation et les actions et de séparer les responsabilités de manière efficace. De plus, toute la logique de prise de décision automatique (ex: Max Q) peut être implémentée dans module-actions. Enfin, on a un découplage fort qui permet de faire en sorte que, si un des services plante, l'autre reste intact (si module-actions plante, on a la télémétrie de la fusée pour voir les données, et si module-metrics plante, on peut toujours avoir le contrôle sur la fusée).

La simulation de l'évolution des metrics est faite dans module-actions en écrivant dans un fichier qui est lu par module-metrics pour simuler la récupération d'informations depuis des capteurs physiques (pression, température, vitesse…). Cela permet plusieurs choses:

+ La facilité d'avoir des metrics réalistes et communes entre les deux services (un seul fichier commun permet une cohérence
+ Puisque les données dépendent des actions prises par module-actions (automatiquement ou manuellement) et que c'est lui qui écrit les données dans le fichier de simulation, il est facile de faire correspondre les données aux actions effectuées.
+ Module-metrics n'a qu'à lire dans ce fichier qui est mis à jour régulièrement pour récupérer les données
+ Module-actions n'a qu'à écrire dans ce fichier les télémétries (la prise de décision autonome s'effectue en gardant en cache les dernières données écrites dans le fichier, donc pas besoin de lire dedans).

Le point négatif de cette implémentation est la représentation de la fusée elle-même:

- On a besoin d'un fichier de simulation de capteur par module, donc on doit faire correspondre les données simulées pour qu'elles soient réalistes.
- Pas de communication inter-modules: Tout doit être paramétré au préalable pour que cela fonctionne. Pour cela, on définit des altitudes auxquelles les modules doivent se détacher ou démarrer pour que le module suivant prenne le relais. Étant dans une optique d'automatisation complète du processus de décollage, cela ne nous pose pas trop de problème, mais peut rendre plus difficile le contrôle et la vision sur une fusée entière.

Communication sol-espace

Actions à distance

Pour prendre des actions à distance sur les modules, on utilise du gRPC avec Protocol Buffer entre le service rocket et chaque module-actions. L'utilisation d'une méthode RPC est tout à fait justifiée dans le contexte d'appel d'actions à distance (Remote Procedure Call) sur la fusée.

+ support offert par Protocol Buffer, qui met l'accent sur l'évolution du schéma de données par rapport à du SOAP ou du XML-RPC (on peut ainsi plus facilement ajouter des actions dans la fusée).
+ Par rapport à des requêtes HTTP, on peut définir des méthodes d'appel plus strictes, et savoir exactement quel type de retour on aura, ainsi que la liste des actions disponibles.
+ Protocol Buffer permet une grande facilité d'implémentation en différents langages entre le client et le serveur, ce qui est le cas dans notre architecture, en fournissant directement les outils de génération de code (il suffit d'utiliser un script avec la commande `protoc` et les bonnes options pour générer le code dans n'importe quel langage).

Télémétrie

Dans une première version de notre architecture, les télémétries étaient récupérées via des WebSockets (les modules de télémétrie les envoyaient, on était donc en mode “push”).
Cependant, par la suite, nous avons choisi d’utiliser Prometheus (en mode “pull”), pour plusieurs raisons:

+ Séparation des responsabilités: Prometheus va sonder (en http) chaque module et récupérer les données tout seul
+ Simplicité d’implémentation et de configuration
+ Stockage automatique sous forme de données en séries temporelles
+ Ouverture sur d’autres outils: Grafana pour la représentation, et alertmanager pour les alertes
+ Passage à l’échelle très facile, puisque les modules de télémétrie n’ont pas à connaître le serveur Prometheus

- Pas du “temps réel” à proprement parler (Prometheus requête les données de télémétrie toutes les X secondes)
- Grosse responsabilité sur le système prometheus (mais scalabilité horizontale possible)

Grafana:

+ simplicité de son implémentation avec Prometheus
+ vision claire offerte sur les données de la fusée et leur évolution dans le temps
+ Grafana stocke aussi les informations avec SQLite3, ce qui permet de garder un historique en plus de Prometheus

Alertmanager:

+ simple à brancher avec prometheus (il fait partie de la suite “prom”) pour ajouter de la logique d’analyse au sol des données
+ lancer des alertes dans des cas précis. Par exemple, il est très simple de définir des alertes si des valeurs sont trop hautes pendant un temps donné (par exemple si la fusée va trop vite pendant 30 secondes).
+ Intégration avec Grafana (dashboards existent pour afficher logs d’alertmanager)
+ Possibilité d’associer des webhooks (ou envoie de mail, notifs slack) pour catch les alertes et les traiter

Figure 9 : Dashboard Grafana pendant l'atterrissage du booster

Événements de décollage

Pour récupérer les logs des événements de décollage (pour que le webcaster puisse les visualiser), on a choisi de relier les modules au bus Kafka présent dans notre architecture. Cela permet facilement de les récupérer et de faire en sorte que n'importe qui puisse les écouter.

Initialement, nous avions voulu utiliser un système de logging déjà existant. Nous avons donc essayé d'implémenter loki, système qui se présente comme un "prometheus, mais pour des logs", et qui, en plus, est facilement intégrable avec Grafana. Il se trouve que loki n'était pas implémentable dans notre projet de par son fonctionnement. Il nécessitait l'ajout d'un nouveau service et de modifier des points qui marchaient déjà sans problème. Tester de l'implémenter nous a fait perdre du temps, nous nous sommes donc tournées vers une solution plus simple, celle de relier les modules au bus.

Cela ne pose pas vraiment de problème direct dans notre implémentation, cependant des systèmes plus efficaces existent pour effectuer ces opérations. L'intérêt d'avoir un système au sol qui se charge uniquement de sonder les modules serait de pouvoir y ajouter de la logique, tout comme dans prometheus, avant de les redistribuer sur le bus Kafka. Un point négatif cependant serait de créer un goulot d'étranglement en termes de responsabilité (il faudrait donc que le système puisse passer à l'échelle).
De plus, Kafka, même s'il peut faire transiter beaucoup de données en même temps, possède un temps de latence assez important.

Des solutions comme Graylog ou la suite ELK existent pour répondre à ce problème avec un système intermédiaire.

Perspectives d'amélioration

User stories additionnelles

Nous avons réfléchi à des user stories supplémentaires par rapport au sujet :

- Ajout dynamique de modules de fusée : Actuellement, enregistrer un module dans notre système demande de modifier et redémarrer le service rocket, modifier les targets de Prometheus, et ajouter un dashboard dans Grafana. Nous avons pensé à faire un service additionnel, rocket-telemetry, qui permettrait d'écouter sur le bus kafka des événements sur un topic "add-module". Cela permettrait deux choses : Dans le service rocket (qui écouterait ce topic), on pourrait initialiser des clients gRPC dynamiquement pour effectuer des actions sur les modules, et dans le service rocket-telemetry, on pourrait également écouter cet événement afin de faire les modifications nécessaires dans Prometheus et Grafana (en modifiant leurs fichiers .yml de configuration). Dans Grafana, il faudrait également, au lieu d'avoir un dashboard par fusée, implémenter un sélecteur pour voir les données de la fusée que l'on veut en fonction de son identifiant. Avec cette méthode, chaque module de fusée doit être enregistré individuellement. Pour simuler tout cela, un script bash qui ajoute une fusée en créant une instance de chaque module (via un docker-compose) pourrait faire l'affaire, en plus d'ajouter une logique dans le code de module-actions pour envoyer le premier message sur le bus Kafka.

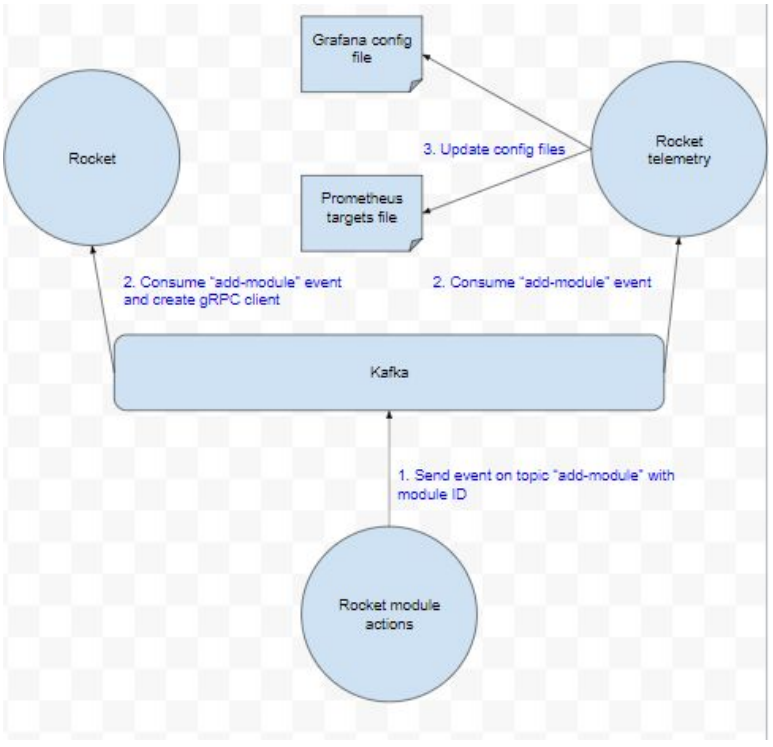

Figure 10 : Flot de données de l'ajout dynamique de modules

Améliorations techniques

La thématique du sujet et son scope étant très vaste, concevoir une architecture répondant à tous les besoins ne peut être faite en 2 mois, surtout quand il n'y a pas une solution "juste" mais une multitude de solutions possibles. Aussi bien que notre architecture puisse être, nous essayons de garder un maximum de recul sur ce qui a déjà été fait et d'anticiper les évolutions technologiques et fonctionnelles.

Nous avons isolé un certain nombre de pistes d'améliorations techniques en partant de notre architecture actuelle. Elles concernent principalement la partie critique du sujet à savoir la gestion des rockets/modules en vol (métriques, actions, etc.).
Voici donc la liste non-exhaustive des pistes d'améliorations trouvées :

Service Analogique (Module)

Dans notre implémentation actuelle d'un module, c'est Module-Actions qui se charge de "simuler" la partie analogique du module remontant les informations comme la température, vitesse, etc. Cela fait sens du fait que les actions appelées sur module-actions impactent ces mêmes métriques. Le problème principal découlant de cette approche est qu'il est nécessaire de maintenir une synchronisation entre metrics et actions (pour le développement uniquement). Dans la réalité, la partie analogique fait la "synchronisation" entre les 2 parties. Cependant pour le développement, nous avons dû mocker la liaison entre les 2 services, nous avons choisi de la faire via un volume partagé car cela peut être facilement retiré dans une version "prod" du produit (comparé à une connexion HTTP entre les 2 par exemple).

Une amélioration possible de ce système serait de connecter actions et metrics à un même service "utilitaire" qui servirait d'intermédiaire pour l'accès aux données analogiques, qu'elles proviennent de capteurs, d'un volume ou de la RAM (il serait donc stateful). Cela permettrait également de réglementer l'accès read et write à ces données.
L'inconvénient d'ajouter un tel service (ou classe utilitaire par exemple) est que cela ajoute une dépendance pour ces 2 services ce qui, en plus du développement de ce service, rajoute du travail de maintenabilité et donc de sûreté du module.

Refonte de la récupération de Logs (Module)

Pour rester dans la problématique des modules, nous avons également pensé à améliorer notre récupération des logs des modules en suivant le même principe d'un crawl sol-air. Nous l'avons déjà évoqué dans la partie "Système dans l'espace", nous avons commencé à implémenter Loki à notre solution afin de parvenir à ce but.

Il faut savoir que dans notre implémentation actuelle de la récupération des logs, et donc des events, c'est le module qui publie un message sur le bus Kafka directement. Ce n'est clairement pas la meilleure solution que non seulement mélanger la seule occurrence des

flux d'events dans un bus commun mais également de connecter ce même bus avec un module dans l'espace (et toutes les contraintes que cela implique).

Une façon de régler ce problème serait donc d'utiliser Loki, un outil qui permet d'exposer et de scrapper des logs à l'image de Prometheus. Il est même directement implémenté dans Grafana (via un plugin), ce qui permettrait de renforcer le dashboard Grafana pour qu'il se rapproche un peu plus d'un véritable outil de monitoring. Seulement cela nécessite, pour être exécuté correctement, de créer 2 nouveaux services : un qui exposerait ces logs et un autre qui les scrapperait et les enverrait à Grafana (et sur le bus Kafka également). Cela rajoute donc une certaine complexité et une légère refonte de notre système de logs/events actuel, ce qui explique pourquoi nous ne l'avons pas implémenté dans le temps imparti.
Enfin, il est bon de noter que Loki est un outil, comme Traefik, beaucoup plus accès Kubernetes que Docker ou simplement "from scratch".

Amélioration de la récupération des métriques, voir logs améliorer (Module et Monitoring)

Toujours dans l'optique de la communication sol-air pour le scraping de métriques (et de logs potentiellement), il est bon de réfléchir et prendre du recul sur l'implémentation actuelle que nous avons faite pour répondre à ce(s) besoin(s). Nous pensons que pour un MVP et une version de développement, ce système est correct car il répond efficacement aux besoins énoncés. Cependant, dans le cadre d'un déploiement réel nous pensons que cela n'est pas suffisant voir nécessiterait un refactor.

En effet, dans des conditions réelles, la criticité fait son entrée parmi les besoins prioritaires. Notamment au niveau de la synchronisation temps réel et aux contextes de réseau/latence entre une multitude de modules et la(les) station(s) au sol. Même si Prometheus permet une récupération très rapide et efficace des métriques, ce qui a priori solutionne en partie ces nouveaux besoins de criticités, non seulement le "très rapide" ne signifie pas (forcément) "temps réel" mais en plus sur une flotte de modules (comme avec Starlink par exemple), pas sûr qu'il tienne la charge à la même vitesse.

Pour palier à ce problème, nous avons trouvé 3 améliorations possibles :

- Rendre (et faire) scalable Prometheus

Cela est la solution la "plus facile" à mettre en place mais ne répond qu'au problème de charge

- Faire 1 Prometheus par fusée (voir module)

Un peu similaire à la précédente, cette approche a par contre pour avantage de pouvoir spécifier précisément le rôle de chaque instance et ainsi pousser les capacités de Prometheus au maximum en ne requêtant qu'une seule et même target

- Ou bien changer d'approche et créer un système de socket ou pub/sub "maison"

C'est selon nous la solution la plus efficace que ce soit pour le problème de nombre ou de connexion temps réel dans des conditions particulières. Cependant c'est également la plus lourde à concevoir

Rendre les autres services scalables (Sol)

Une autre piste d'amélioration générale cette fois serait de tester et ajuster les services au sol pour les rendre scalables. À l'heure actuelle, tous nos services au sol sont "stateless" mais il peut être nécessaire de tester cela en s'assurant qu'ils sont de fait (ou également) scalables.

Cela peut s'avérer très utile mais aussi très positif pour le projet dans son ensemble car faire des services stateless et scalables à volonté résout en très grande partie le problème de mise à l'échelle de certains composants. D'autant que cela n'est pas très dur à faire vu l'état actuel de l'architecture et des services.

Le seul point délicat dans cette opération est qu'il est assez compliqué (mais possible) de faire scaler un bus Kafka. Car sans cela, scaler les autres services sans scaler le bus Kafka (à terme du moins) ne s'avérera pas très utile étant donné le SPOF d'une architecture de services avec communication événementielle par bus.

Conversion en microservices (Sol)

Une piste d'amélioration, toujours globale notamment aux services au sol, qui succéderait parfaitement à la précédente serait d'aller plus loin dans l'approche service en proposant une architecture à base de micro-services !

En plus de rentrer dans le thème de la matière ;) cela aurait pour avantage d'améliorer la qualité globale du produit. Mise à l'échelle facile, forte résilience, découplage fort, modularité, robustesse : le passage en micro-services est gage de tout cela **à condition** que cela soit fait correctement. Bien-sûr, il ne faut pas oublier que le micro-service n'est pas l'outil magique ou la solution parfaite (#GoldenHammer). Elle a aussi ses inconvénients, à savoir principalement : la difficulté (et le coût) de maintenabilité/opérabilité, et bien entendu, la difficulté à mettre en place

Pour ce qui est de la mise en place, l'avantage que nous avons est que nos services sont déjà très proches de micro-services.
Parmi les points restants pour faire basculer cette architecture en architecture micro-services, on aurait, par exemple, la roadmap suivante :

- Les autres points d'améliorations (scalabilité, scraping des métriques et logs améliorés, etc.)
- Améliorer la résilience des services (notamment pour leurs dépendances externes à MongoDB ou Kafka principalement)
- Découper certains services de manière plus fine (notamment pour la partie monitoring, rocket et mission)

- Et pour une version optimale : une abstraction matérielle et réseau encore plus forte (qu'avec Docker et Traefik) via l'utilisation de side-cars, de l'émulation ou encore en "Kubernetisant" toute l'architecture (du sol)

Kubernetes

Comment conclure ce rapport sans parler de Kubernetes !

Avant toute chose, il est bien sûr évident que déployer une architecture pareille pour un POC / MVP n'est pas (très) utile. Par contre, avec l'évolution des besoins induits par une mise en production de la solution, un outil comme Kubernetes prend tout son sens et peut apporter plusieurs solutions à certains problèmes.
C'est notamment une "suite logique" à la mise en place d'une architecture micro-services car il permet d'amplifier les avantages du micro-service tout en palliant certains de ses défauts. C'est également un bon moyen d'intégrer l'user story additionnelle d'ajout de fusées / modules dynamiquement ("à la volée").

Pour résumer, les avantages / inconvénients sont :

+ Scalabilité facilitée (avec ReplicaSet)
+ Résilience accrue (via liveness/readiness probes ou tout simplement par le ReplicaSet)
+ Monitoring des instances et gestion ressources plus facile (via Prometheus et Grafana notamment + il est possible de configurer chaque pod avec un certain set de ressources CPU et GPU prédéfinis)
+ Déploiement facile (pas de réseau, config, matériel, etc.)
+ Extensibilité accrue (grâce aux pods "multi-containers" ou à l'aide d'opérateurs kubernetes par exemple)
+ Migration cloud public facile (hors config de sécurité et réseaux spécifiques au fournisseur)
+ Mise en production des mises à jour plus faciles (RollingUpdates lors d'up de version d'image Docker) et contrôlée (canary et blue/green release)
+ Et enfin, c'est "plus" compatible avec Loki et facilite l'ajout de fusée dynamique (à l'aide d'un YML de déploiement + ConfigMaps)

- Demande beaucoup de compétences pour être mis en place efficacement
- Ne peut se faire qu'avec des services résilients
- Pas adapté pour l'espace car le stateful difficile (obligation de passer par des StatefulSets)
- Coûte cher à implémenter (mais c'est le coût de la qualité ;))

Nous avons déjà commencé à implémenter les ressources nécessaires au déploiement de l'architecture. Nous avons pour l'instant créé les YAML (pods et services) correspondant aux services sol (hors Prometheus et Grafana). Pour rendre ces services accessibles, nous avons également créé un Ingress Traefik permettant d'exposer les endpoints des services sur le port 80 (comme sur l'implémentation sur Docker).

Nous n'avons dû modifier aucune logique au sein de nos services pour ajouter Kubernetes. Nous avons uniquement dû modifier le nom du container kafka (de kafka en kafka-service), afin de pouvoir faire fonctionner le pod, pour que les autres services puissent communiquer avec le bus (kafka étant un nom réservé qui ne permettait pas de faire fonctionner nos pods). C'est une modification mineure, mais bel et bien la seule qui ait eu un impact sur le reste de notre projet.

De plus, nous avons ajouté de la résilience au niveau de kafka, zookeeper, et dans les services au sol, par le biais de liveness et readiness probes dans les YAML des pods. Dans un projet orienté micro-services, la résilience serait ajoutée notamment dans les services eux-mêmes. Par exemple, actuellement, si kafka ou la base de données n'est pas trouvée, les services plantent. Il faut donc les redémarrer (via les probes) afin de pallier ce manque de résilience. Les liveness et readiness probes ont été implémentées dans cette optique mais resteront tout de même utiles une fois l'implémentation complète.

Enfin, pour compléter l'implémentation actuelle des ressources, il faudrait finir d'implémenter les deploys / pods pour Prometheus et Grafana puis configurer les modules (via des StatefulsSets et endpoints Traefik spécifiques). Une fois cela fait, le déploiement Kubernetes sera à jour avec l'architecture basique actuelle et il sera donc possible de lancer le run.sh sans soucis.

Pour tester l'implémentation, il suffit de lancer le prepare-kubernetes.sh puis le run-kubernetes.sh qui va loader tous les Pods / Services / Ingress présents dans le dossier kubernetes situé à la racine du projet. Attention cependant, en l'état il n'est pas possible d'utiliser pleinement l'application, uniquement les services au sol. Mais vous pourrez quand même visualiser la création des ressources et le chargement "intelligent" et résilient des Pods (en attendant que ZooKeeper charge, Kafka va redémarrer ainsi que les autres services dépendants de Kafka).