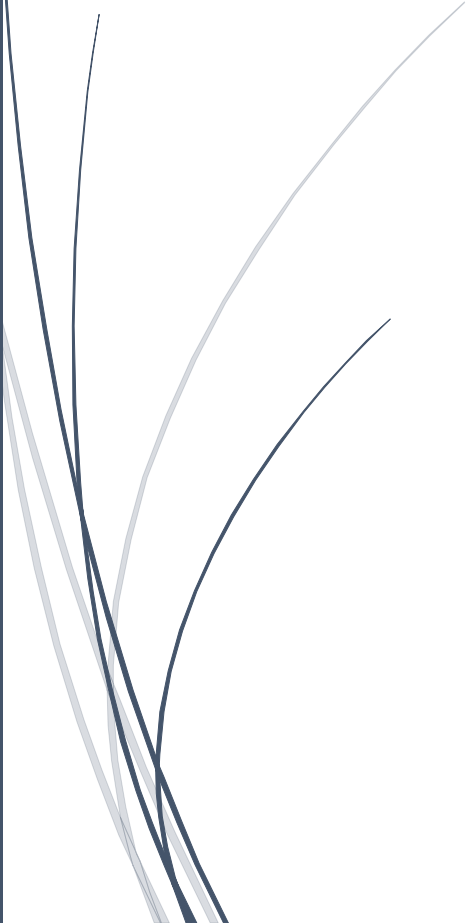
A thick dark blue vertical bar runs down the left side of the page. A blue arrow points to the right from this bar, containing the date.

11/11/2020

Rapport final d'architecture

Projet Blue Origin - SOA

Abstract line drawing consisting of several thin, curved lines in dark blue and light grey, originating from the bottom left and extending upwards and to the right.

Younes Abdennadher – Alexis Lefebvre –
Alexandre Longordo – Thomas Colomban
TEAM A

Table des matières

I) Vue fonctionnelle.....	1
A) Interprétations des user stories	1
1) Interactions des acteurs avec le système.....	1
2) Gestion des anomalies	2
a) Qu'est-ce qu'une anomalie ?	2
b) Comment sont-elles gérées ?	2
3) Gestion d'une flotte de fusées	3
a) Qu'est-ce qu'une mission ?.....	3
b) Indépendance de la fusée par rapport aux systèmes au sol.....	3
B) Scénarios	3
1) Progression d'une mission	3
2) Un problème survient sur deux fusées	4
II) Architecture actuelle.....	4
A) Diagramme d'architecture complet	4
B) Evolution de l'architecture	5
1) Interactions avec les utilisateurs.....	5
2) Evolution des services Mission et Telemetry	6
3) Interactions entre composants de la fusée.....	6
C) Rôles des services	6
1) Anomaly-Handler	6
2) Mission	6
3) Payload.....	6
4) Real-Time	6
5) Telemetry-Analyser	7
6) Telemetry-Writer	7
7) Booster	7
8) Missions-coordinator	7
9) Rocket	7
10) Telemetry-Listener.....	7
D) Vision d'ensemble.....	7
III) Interactions entre services.....	8
A) Diagrammes de séquences	8
1) Mission complète.....	8
2) Injection d'une anomalie dans la tête de la fusée	10
B) Justification des interactions entre services	11
1) Communication inter composants de la fusée	11

2)	Communication entre fusée et services au sol	12
3)	Communication événementielle au sol.....	12
IV)	Prise de recul.....	12
A)	Choix techniques	12
1)	Choix technologiques	12
	Node/TypeScript/Express	12
	Persistence	13
2)	Choix techniques	13
a)	Organisation des variables d'environnement.....	13
b)	Structure d'accès à la base de données.....	13
c)	Kafka & Factorisation	13
d)	Routes SOAP	14
B)	Perspectives d'améliorations.....	15
1)	Création événementielle d'une mission	15
2)	Découpage de TelemetryWriter.....	16
3)	Ajout d'un service de simulation.....	17
4)	Découpage du service mission	17
5)	Ajout d'une passerelle	17

I) Vue fonctionnelle

A) Interprétations des user stories

1) Interactions des acteurs avec le système

La figure 1 représente le diagramme de cas d'utilisation de notre système. Il permet de se rendre compte des interactions que les acteurs peuvent avoir avec notre système. Les User Stories qui n'apparaissent pas dans le diagramme sont des actions qui sont effectués automatiquement par le système.

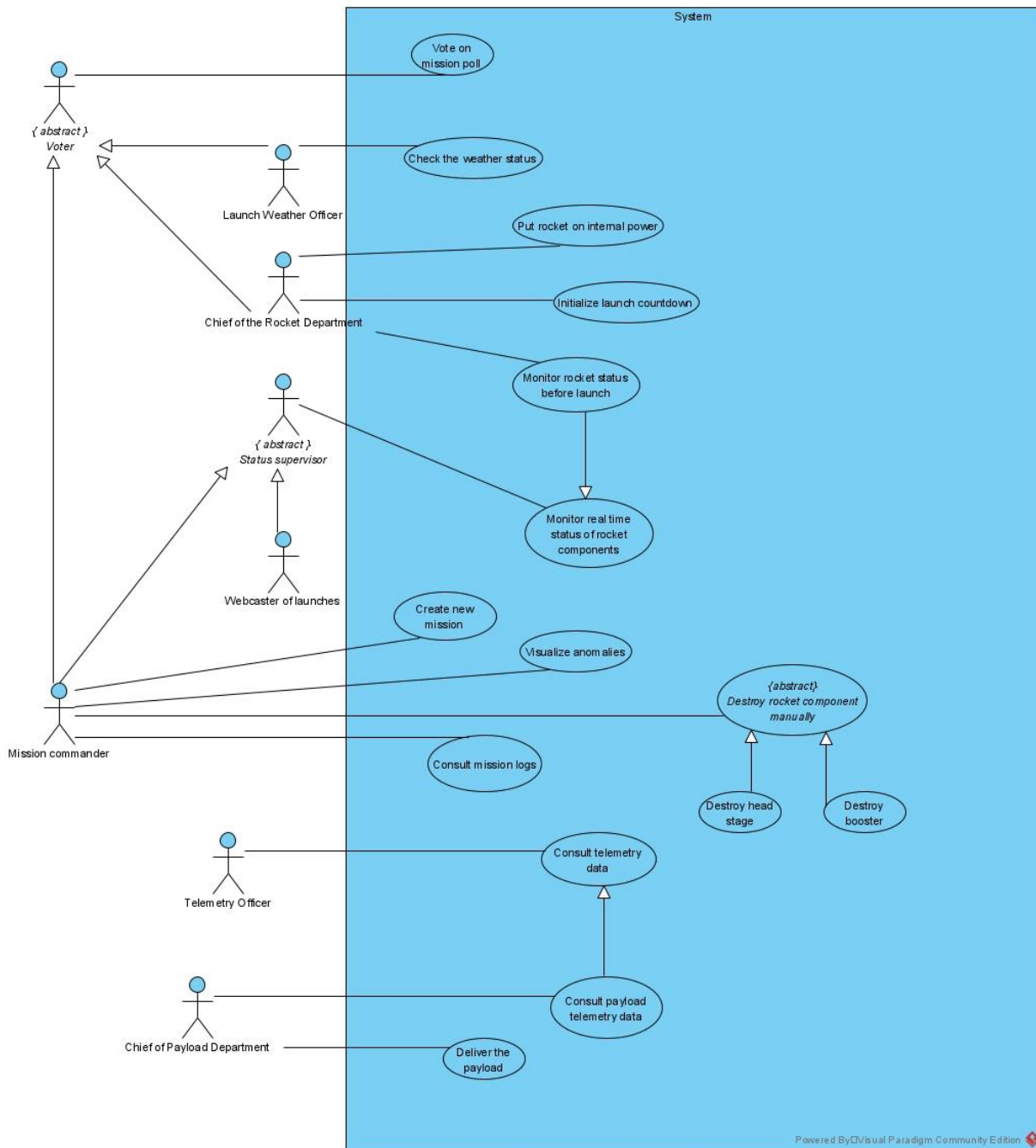


Figure 1 : Diagramme de cas d'utilisation

Plusieurs de nos choix par rapport aux actions non automatiques du système peuvent être visibles sur le diagramme. Par exemple, on peut voir que le commandant de la mission a la possibilité de détruire manuellement le propulseur ou la tête de la fusée. La destruction automatique du matériel est une User Story récente, et nous avons préféré laisser la possibilité de détruire manuellement la fusée en cas d'anomalies qui ne seraient pas identifiées automatiquement comme critiques par le système mais jugées importantes par les équipes au sol.

Le chef du département des fusées a également la main sur les deux actions initiales permettant de lancer la fusée : il doit mettre lui-même la fusée sur alimentation interne puis lui-même initialiser le compte-à-rebours qui enclenche la série d'actions automatiques au sein de la fusée. Donner la main à l'utilisateur avant le lancement de la fusée lui permet d'attendre d'éventuelles informations liées à des protocoles externes au système.

2) Gestion des anomalies

a) Qu'est-ce qu'une anomalie ?

Nous avons défini une anomalie comme une perturbation au niveau de la fusée, que ce soit au niveau de son comportement ou au niveau de ses capteurs. Nous avons choisi de créer des anomalies qui concernent seulement les données télémétriques car ces erreurs nous ont semblé être les plus fréquentes que l'on rencontre lors d'un lancement de fusées. Tout passe donc par l'analyse des données télémétriques pour permettre de déterminer s'il y a un problème sur la fusée.

b) Comment sont-elles gérées ?

Nous avons mis en place 3 type d'anomalies différentes, avec différents niveaux de criticité :

- **Problème sur le capteur du niveau d'essence de la tête de la fusée** : Dans ce cas-là l'analyse des données télémétriques fait ressortir une valeur aberrante pour le capteur du niveau d'essence (valeur arbitrairement fixée à 0, une valeur qui ne devrait jamais être possible de la création de la fusée jusqu'à la livraison du *payload*). Cette anomalie a un niveau de criticité de 1, ce qui en fait une anomalie de niveau moyen qui ne nécessite pas l'abandon de la mission. En effet, il y a des actions possibles qui permettent de régler cette erreur telles que des diagnostics du capteur pour localiser le problème ou encore le redémarrage du capteur. Nous laissons donc le choix à l'utilisateur de déterminer si cette erreur doit entraîner l'arrêt de la mission.
- **Problème sur le capteur du niveau d'essence du propulseur** : Anomalie similaire à celle ci-dessus, elle ne concerne pas la tête de fusée mais le propulseur. Elle a aussi un niveau de criticité à 1.
- **Problème sur le capteur d'altitude de la fusée** : Cette anomalie se déclenche lorsqu'à l'analyse des données télémétriques, la donnée d'altitude baisse. Deux cas de figure peuvent correspondre à ce scénario : soit une erreur de capteur soit un problème très sérieux avec la fusée qui lui fait perdre de l'altitude. Nous avons donc choisi de lui attribuer une criticité de 2, ce qui va entraîner un abandon de la mission et la destruction automatique de la fusée. C'est une anomalie que nous avons jugé grave car elle peut signifier que la fusée est en train

de s'écraser et qu'elle peut potentiellement blesser des personnes au sol, nous la détruisons donc automatiquement.

Etant donné que le lancement de la fusée est une simulation nous n'obtenons pas d'anomalies avec le comportement naturel de la fusée qui est prédéfini du début à la fin. Pour cela nous injectons donc artificiellement des problèmes dans la fusée, par exemple en changeant les valeurs des capteurs ou encore en modifiant le comportement de la fusée (dans l'anomalie de niveau 2).

3) Gestion d'une flotte de fusées

a) *Qu'est-ce qu'une mission ?*

Une mission est caractérisée par un identifiant unique généré aléatoirement. Pour gérer une flotte de fusées nous gérons en fait plusieurs missions. Une mission possède son propre sondage Go/No-go et sa propre fusée.

b) *Indépendance de la fusée par rapport aux systèmes au sol*

La fusée est selon nous un système censé être complètement indépendant des systèmes qui sont au sol. En effet, il est important pour les systèmes de calcul au sein de la fusée de ne pas dépendre d'un système qui se trouve potentiellement à une distance très élevée et donc d'un délai de réponse qui peut être conséquent.

Nous considérons donc que la fusée possède son propre bus de données, un bus matériel qui permet de véhiculer l'information quasiment instantanément. Cela permet à tous les composants de la fusée de communiquer rapidement et de concentrer la logique de calcul de la trajectoire de la fusée au sein d'un système isolé et capable de faire communiquer ses services efficacement entre eux.

B) Scénarios

Afin de cerner précisément la portée de notre projet, nous avons défini deux scénarios couvrant toutes les User stories telles qu'elles ont été implémentées dans notre solution.

1) Progression d'une mission

- Une nouvelle mission va démarrer.
- Richard crée un nouveau sondage de début de mission pour que les chefs de service puissent donner leur accord avant le lancement de la fusée.
- Elon voit qu'un sondage a été créé, il regarde le statut actuel de la fusée et, s'il est bon, il donne son accord dans le sondage.
- Tory voit que c'est à son tour de voter sur le sondage, elle récupère l'état de la météo puis, s'il est bon, donne son accord dans le sondage.
- Une fois que Elon et Tory ont voté, c'est au tour de Richard de donner son accord sur le sondage pour que la mission puisse débuter.
- Elon peut maintenant mettre la fusée sur alimentation interne. Lorsque toutes les conditions de lancement sont réunies, il peut initialiser le compte-à-rebours de la fusée. A partir de ce moment-là, la fusée est complètement autonome.

Voici le comportement de la fusée en vol :

- Après 57 secondes, la fusée démarre le moteur principal du propulseur ;
- Après 3 secondes, la fusée décolle ;
- Une fois que la fusée atteint maxQ, elle arrête d'accélérer et ralentit ;
- Une fois que le propulseur a atteint un niveau de carburant défini, il arrête son moteur, et se détache de la fusée ;
- La tête de la fusée allume à son tour son moteur ;
- Le propulseur débute sa descente et passe par toutes ses phases d'atterrissage ;
- À la bonne altitude, la tête de la fusée se sépare de sa coiffe ;
- La tête arrête ensuite son moteur et est alors en orbite ;

Une fois que la fusée a atteint les données orbitales voulues, Gwynne peut envoyer un signal pour livrer la charge utile.

En parallèle, Marie peut visualiser en temps réel l'évolution des statuts de la fusée et du propulseur.

Jeff peut visualiser à tout moment les données télémétriques de la tête de la fusée et du propulseur stockées en base de données. Gwynne, elle peut voir les données télémétriques de la charge utile à tout moment.

2) Un problème survient sur deux fusées

- Richard peut lancer deux missions en parallèle.
- On considère que les deux fusées sont lancées. La fusée A s'est séparée de son propulseur et la fusée B a encore son propulseur attaché.
- Pendant le vol, la tête de la fusée A rencontre une anomalie identifiée comme non critique par le système. Richard la voit et effectue des vérifications auxiliaires en dehors du système. Après analyse, Richard décide de détruire manuellement la tête de la fusée. La tête de la fusée est alors détruite et le propulseur continue son atterrissage normalement.
- La fusée B rencontre une anomalie identifiée comme critique par le système ; la fusée est en train de chuter. La destruction du composant ayant un comportement anormal est alors automatique et la mission est abandonnée. Comme tous les modules de la fusée sont attachés, ils sont tous détruits.

II) Architecture actuelle

A) Diagramme d'architecture complet

En figure 2 se trouve le diagramme d'architecture actuel du projet.

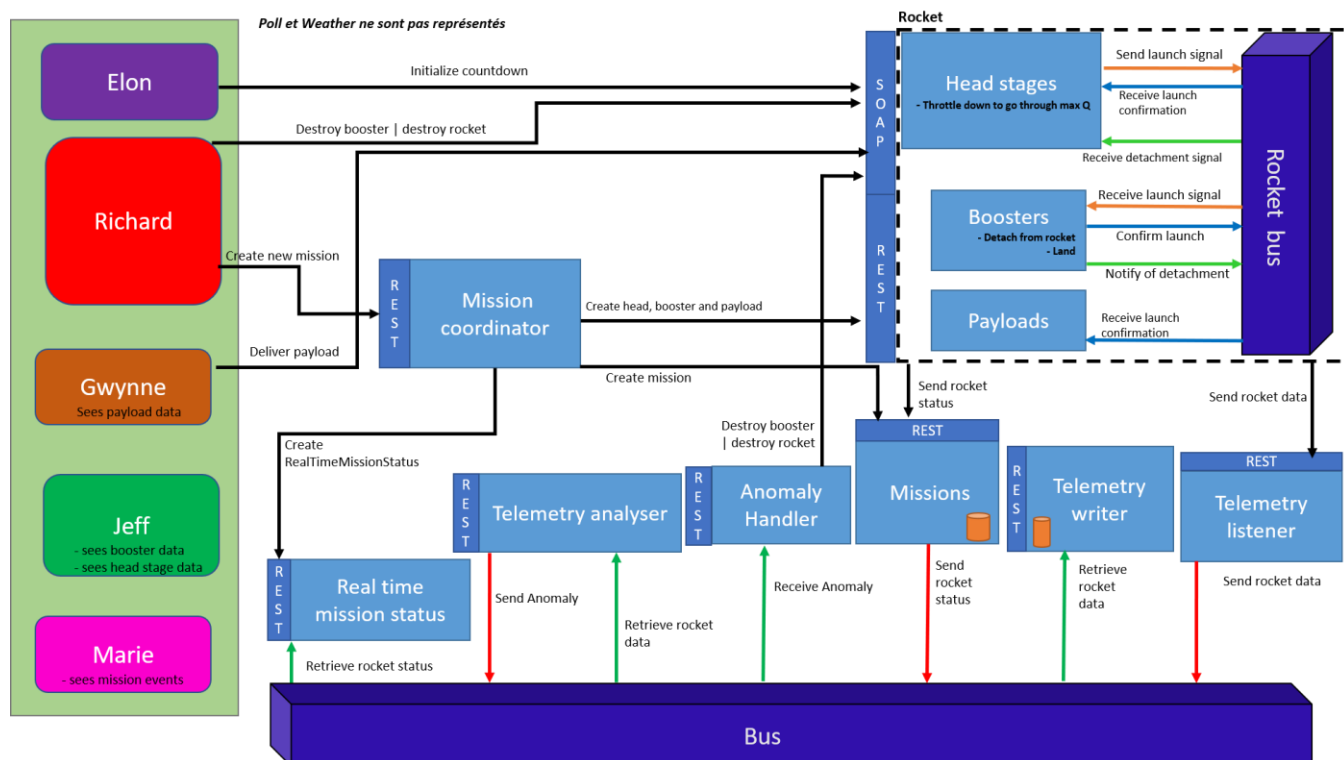


Figure 2 : Diagramme d'architecture actuel du projet

Sur ce diagramme, par soucis de clarté, nous n'avons pas représenté toutes les interactions qui ont lieu entre les composants de la fusée, mais seulement les interactions les plus importantes. Toutes les interactions sont visibles dans le diagramme de séquence en partie [III.A.1](#).

B) Evolution de l'architecture

1) Interactions avec les utilisateurs

Lors du premier rendu, les interactions des utilisateurs avec le MVP étaient nombreuses puisque nous privilégions l'implémentation de toutes les User Stories pour que tous les besoins clients soient satisfaits. Puis, au fur et à mesure, avec l'évolution des besoins, nous avons complexifié les interactions interservices afin d'automatiser le plus de comportement possible afin de ne pas faire dépendre certaines situations critiques du délai de transmission de l'information air-sol et de la réactivité humaine.

Notre diagramme d'architecture du MVP se trouve en figure annexe 1.

Nous avons notamment automatisé les différentes étapes de vol de la fusée comme la séparation du propulseur. La destruction du matériel a été partiellement automatisé afin de laisser la main à l'utilisateur dans certains cas (cf. partie [I.A.1](#)).

D'autres interactions que l'on aurait pu retirer sont restées dans le système comme la livraison de la charge qui reste du ressort du chef du département de la charge utile puisqu'une fois que la tête de la fusée est en orbite, la charge peut être livrée n'importe quand et l'utilisateur doit pouvoir choisir à quel moment il veut livrer la charge selon les spécifications de son client.

L'initialisation du protocole de lancement de la fusée pourrait également être rendu entièrement automatique avec des vérifications non manuelles des différents statuts pour le sondage de la mission et le lancement automatique de la fusée selon l'heure spécifiée. Cependant nous avons décidé de laisser les utilisateurs gérer les étapes de pré-lancement puisque de très nombreux paramètres, qui ne sont pas forcément du ressort de notre système, entrent en jeu avant une mission spatiale. Une intervention des experts du métier est nécessaire selon nous.

2) Evolution des services Mission et Telemetry

Le diagramme d'architecture du MVP (figure annexe 1), montre un service *TelemetryService* responsable de la récupération des données télémétriques de la fusée et de leur envoi au client lorsque celui-ci demande à les voir. Ce service possède désormais une couche de persistance puisque ces données sont des données que l'utilisateur souhaite stocker. De plus, nous avons décidé de le diviser afin de ne pas y concentrer trop de responsabilité. Nous avons donc actuellement un service *TelemetryListener* et un service *TelemetryWriter* afin de séparer la responsabilité d'écriture en base de données et de récupération de données de la fusée.

En ce qui concerne le service mission, il possède désormais également une couche de persistance afin que les statuts des missions soient stockés pour que l'utilisateur puisse aussi les récupérer plus tard. Enfin, auparavant, le service mission gérait le sondage de mission, il n'a plus cette responsabilité aujourd'hui.

3) Interactions entre composants de la fusée

Le diagramme en **figure annexe 1** montre que les services de la fusée interagissent entre eux via des routes SOAP. Nous avons décidé de transformer toute la communication au sein de la fusée en communication événementielle. Nous en parlons plus en détail en partie [III.B.1](#).

C) Rôles des services

Dans cette partie, nous explicitons le rôle des services dont le comportement mérite des explications supplémentaires.

1) Anomaly-Handler

Ce service récupère les anomalies qui ont été découvertes, gère la criticité et détruit la fusée et le propulseur si nécessaire.

2) Mission

Le service *Mission* a pour rôle de gérer la mission et son déroulement, c'est lui qui interagit avec la fusée en vol. Il récupère les statuts du propulseur, de la tête de la fusée au sol et de la charge utile. Il annonce aux autres services le statut de la fusée. Il est l'intermédiaire entre le sol et la fusée en vol pour le statut.

3) Payload

Le service *Payload* a pour mission de gérer la charge utile, son statut, et sa mise en orbite.

4) Real-Time

Le service *Real-Time* récupère les événements de changement de statut de la fusée et stocke le dernier statut. C'est à lui qu'on demande pour savoir quel est le statut actuel de la fusée.

5) Telemetry-Analyser

Le service *Telemetry-analyser* analyse les données télémétriques et cherche si une donnée est anormale. Si une donnée n'est pas normale, il crée une anomalie et la transmet.

6) Telemetry-Writer

Le service *Telemetry-Writer* récupère les données télémétriques et les insère dans la base de données. Actuellement, il est capable de fournir les données sur demande.

7) Booster

Le service *Booster* gère le propulseur, le décollage de la fusée, toute la première phase de vol (avant le détachement)

8) Missions-coordinator

Le service *Missions-coordinator* est responsable de créer une mission, une fusée et tout ce qui la compose (comme la tête de fusée, le propulseur et la charge utile), un sondage, un service temps réel. Il vérifie également que tout est bien en ordre pour démarrer la mission.

9) Rocket

Le service *Rocket* (ou plutôt tête de la fusée) gère la tête de fusée, sa poursuite après le détachement du propulseur jusqu'à la mise en orbite de la charge utile.

10) Telemetry-Listener

Le service *TelemetryListener* récupère les données télémétriques de la fusée actuellement en vol et envoie ce qu'il a reçu aux services qui en ont besoin. Il est l'intermédiaire entre le sol et la fusée pour les données télémétrique.

D) Vision d'ensemble

Les services sont séparés en deux catégories, la première catégorie est celle des services au sol et la seconde celle des services sur la fusée.

Comme dit en partie [I.A.3.b](#), nous avons décidé que les services sur la fusée communiquent sur un bus événementiel qui modélise un bus physique en réalité. Les services concernés par cette communication sont les trois étages de la fusée : *Payload*, *Head-stage*, *Booster*. Les échanges entre les composants de la fusée sont événementiels (exemple : Détachage du *booster*), un bus est donc une solution adaptée.

Les services au sol correspondent à tous les autres services, ils communiquent également de manière événementielle durant l'exécution de la mission. En effet la mission est coordonnée par des événements et les services au sol sont à l'écoute de ces événements pour ensuite effectuer des actions.

La création d'une mission passe par le service *mission-coordinator* qui va communiquer avec les services de la fusée et avec le service de la mission et du temps réel pour les avertir d'une nouvelle mission.

La communication entre la fusée et le sol (une fois la fusée démarrée) se fait via deux services au sol, *TelemetryListener* récupérant les données télémétriques et *mission* les statuts de mission. Ainsi la

fusée envoie les informations au sol à travers ces deux services qui transmettent ensuite ces données aux autres services via le bus évènementiel.

La télémétrie est séparée en deux services, un premier service est chargé de recevoir les données envoyées par la fusée et un autre en charge de stocker en base de données les informations reçues. Nous avons séparé ces services car les données télémétriques arrivent à très grande fréquence (potentiellement plusieurs fois par secondes). Recevoir les données et les stocker en base dans le même service chargerait ce service. L'interaction entre ces services est décrite dans la partie [III.B](#).

Les anomalies télémétriques sont détectées par un le service *TelemetryAnalyser*, si une anomalie est détectée un évènement est envoyé dans le bus. Le service *AnomalyHandler* est à l'écoute des évènements de type anomalie et prend des décisions en fonction de la gravité de l'anomalie. Ces deux services sont séparés pour permettre une prise de décision efficace (car on n'a pas d'analyse faite en même temps que la prise de décision dans le même service). Cela permet également un découplage entre une anomalie et une anomalie télémétrique, en effet plusieurs types d'anomalies peuvent éventuellement être gérées grâce à cette répartition.

L'affichage du statut de la mission en temps réel est géré par *RealTime* qui expose une api REST pour permettre aux clients de voir les statuts que *RealTime* lit dans le bus.

III) Interactions entre services

A) Diagrammes de séquences

1) Mission complète

La mission complète a été séparé en 2 diagrammes pour des raisons de lisibilités, cette mission se déroule comme prévu et qu'aucune anomalie n'a été rencontrée. C'est pour cela que le service anomalie n'est pas représenté. Nous ne représentons pas non plus l'ajout dans la base de données par *TelemetryWriter*. Représenter ce service apporte peu d'information et complexifie le diagramme.

D'autre part, le service *RealTime* n'est pas représenté aussi pour des raisons de compréhension. Il faut donc rajouter dans ce diagramme le fait qu'après avoir reçu le statut, Mission envoie au bus Kafka le statut de la mission puis *RealTime* le lit depuis le bus évènementiel.

Dans ces diagrammes de séquences, nous représentons essentiellement les interactions entre services et nous ne nous concentrons pas sur le comportement interne des services.

Le diagramme de séquence en figure 3 illustre la création de la mission jusqu'à la mise en alimentation interne. Le diagramme en figure 4 représente le déroulement de la mission à partir de la mise en alimentation.

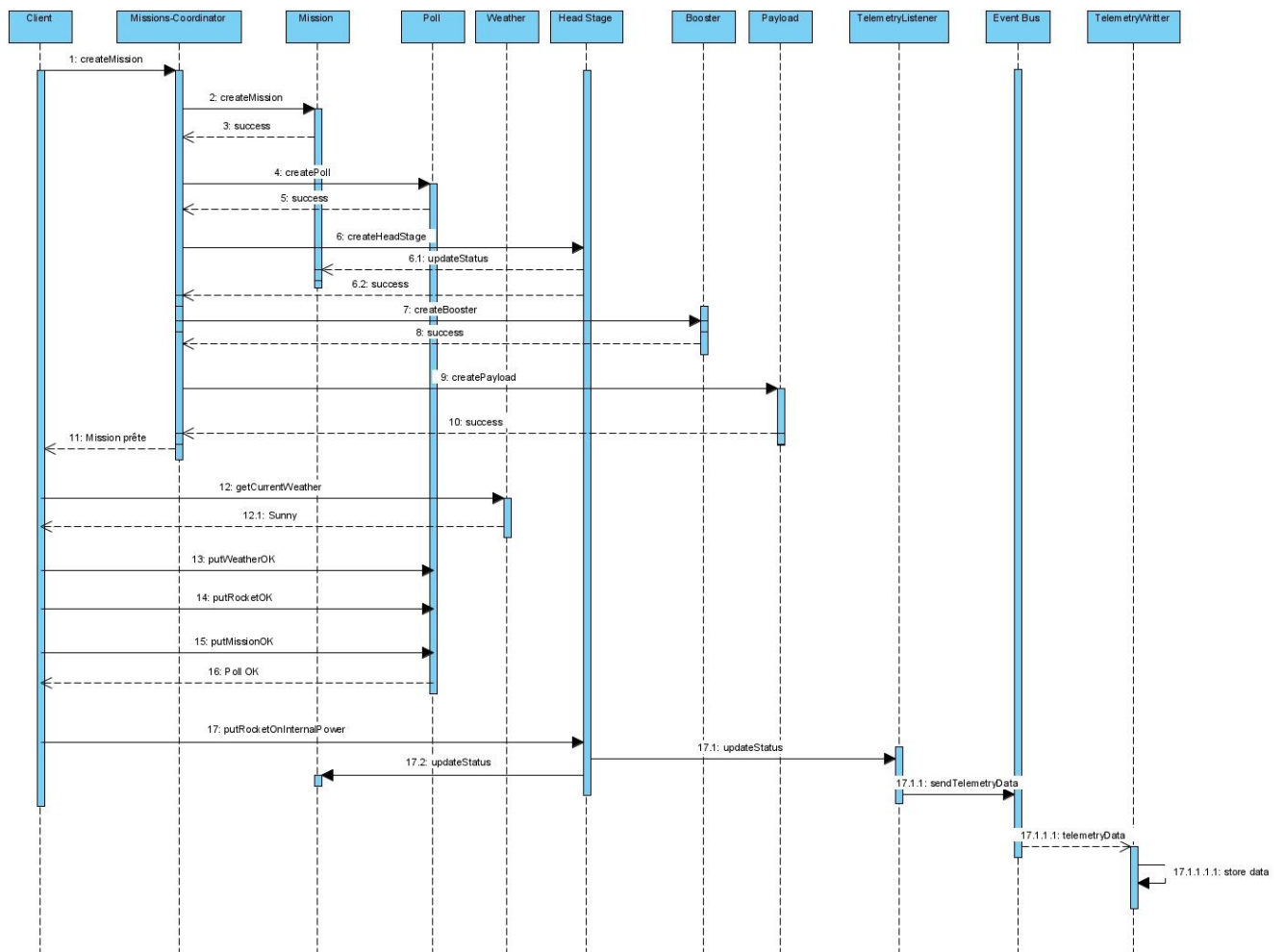
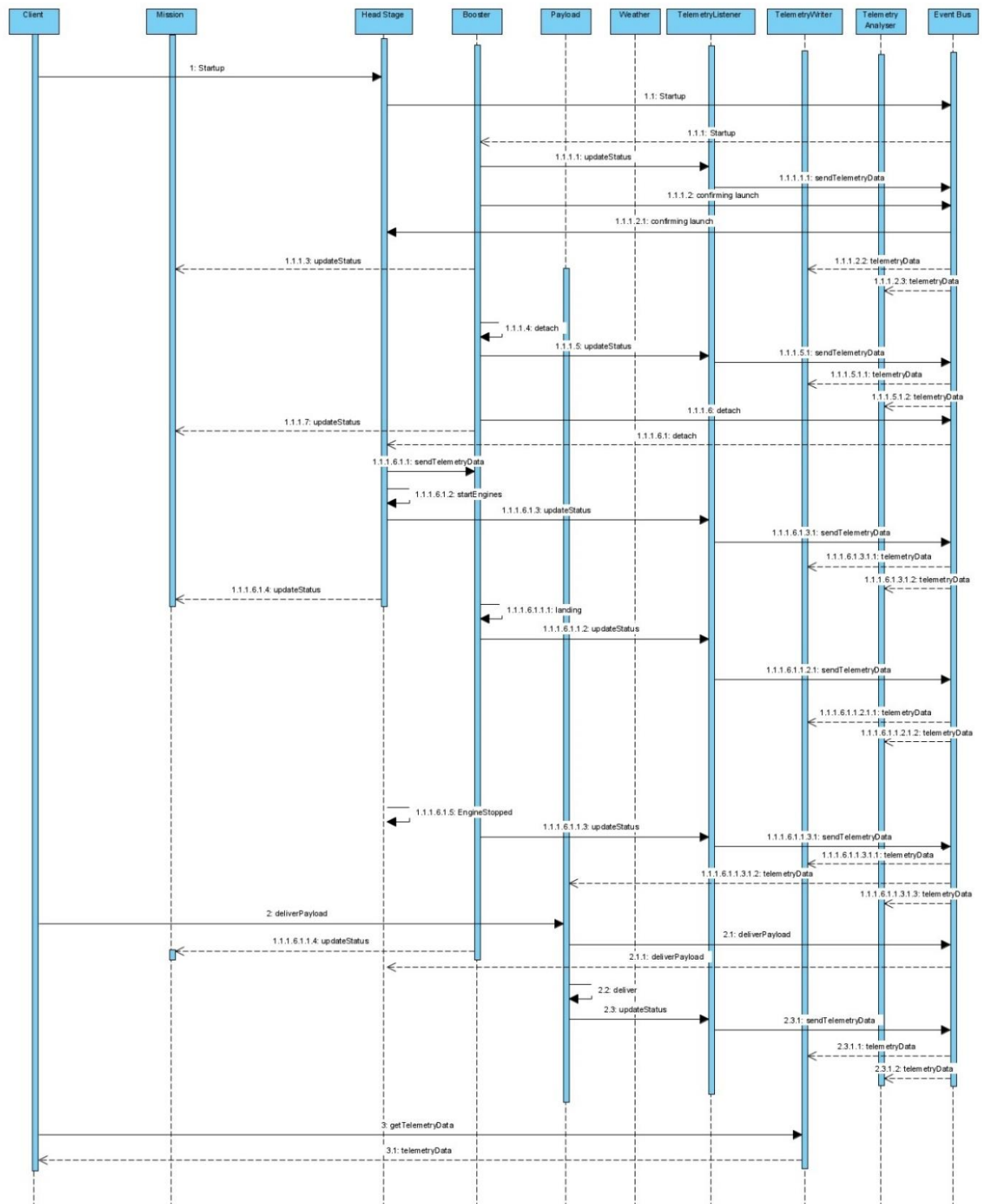


Figure 3 : Diagramme de séquence de la création d'une mission et de l'initialisation du lancement de la fusée



2) Injection d'une anomalie dans la tête de la fusée

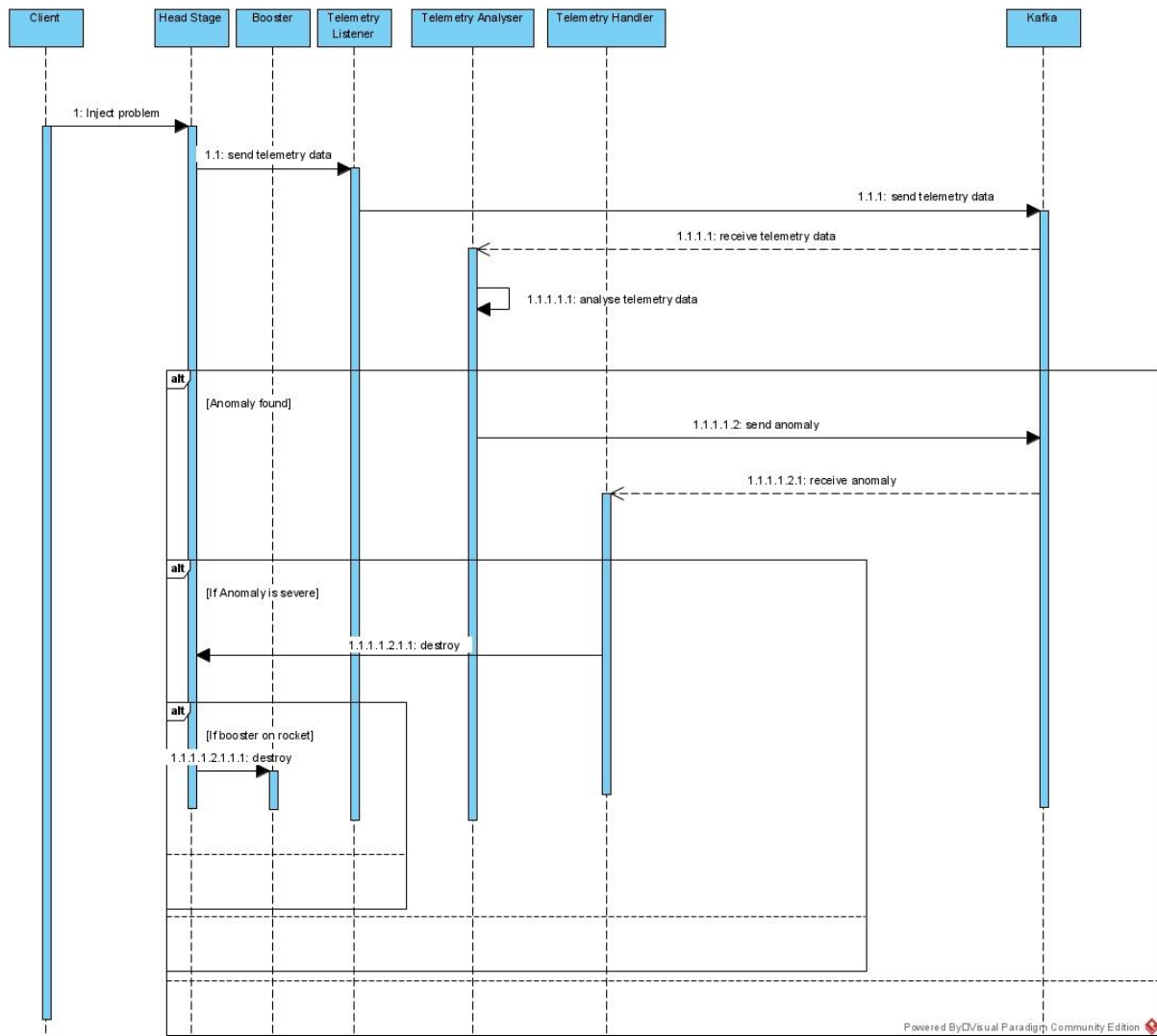


Figure 5 : Diagramme de séquence d'une injection puis traitement d'une anomalie

B) Justification des interactions entre services

Comme nous l'avons vu, les services du système ont différents modes d'interaction. Ils passent soit par le bus événementiel, soit communiquent via API REST ou SOAP.

1) Communication inter composants de la fusée

En partie [1.1.3.b](#), nous avons vu que les services qui constituent la fusée sont censés communiquer entre eux via un bus événementiel qui leur est propre. En pratique, ces services utilisent le même bus événementiel. Nous simulons le fait que le bus sur lequel communiquent les composants de la fusée soit différent en utilisant un topic de discussion particulier : `rocket-<id-mission>-<composant-de-destination>`. Les composants de la fusée, lorsqu'ils sont attachés, communiquent donc en temps réel via un bus que l'on considère être un bus matériel faisant transiter rapidement les informations au sein de la fusée.

2) Communication entre fusée et services au sol

Il n'est pas question de faire transiter une information venant de la fusée vers un service au sol via un bus de communication car cela n'est pas réaliste. La fusée envoie donc des données télémétriques au service *TelemetryListener* via une API REST et des données concernant son statut au service Mission via une API REST également.

En ce qui concerne la communication depuis le sol vers la fusée, on utilise une communication SOAP pour les commandes dont la criticité est très élevée comme la destruction de l'un des composants ou encore l'initialisation du lancement de la fusée car ces actions s'inscrivent dans le cadre d'une mission déjà enclenchée et on veut s'assurer que les actions manuelles ne compromettent pas le déroulement de la mission. SOAP permet d'avoir un contrat fort entre le composant appelant et le service, ce contrat permet notamment de s'assurer que le service appelant envoie les bons paramètres à l'appel des routes exposées par la fusée et qu'ils ne peuvent pas appeler une méthode tant qu'elle ne correspond pas exactement au prototype exposé par les services de la fusée.

Les services de la fusée exposent également certaines routes REST. Ces routes REST correspondent à des commandes qui n'ont pas besoin d'un contrat très fort et concernent la création des composants de la fusée par *MissionCoordinator*. Si ces requêtes échouent, la mission n'ayant pas commencé, le problème n'est pas critique.

3) Communication événementielle au sol

La communication entre services au sol (à savoir *RealTimeMissionStatus*, *TelemetryAnalyser*, *AnomalyHandler*, *Missions*, *TelemetryWriter* et *TelemetryListener*) se fait presque entièrement via un bus de communication. Cela permet à ces services de communiquer rapidement entre eux. *TelemetryListener* et *Mission* peuvent ainsi recevoir des informations de la fusée (c'est là leur unique rôle), et les envoyer dans le bus. Les autres services au sol n'ont plus qu'à récupérer les informations dont ils ont besoin en temps réel dans le bus.

Ce bus au sol permet notamment de ne pas perdre des données si le service *Mission* ou encore *TelemetryWriter* écrivent en base de données et ne peuvent pas traiter directement les informations qui leurs sont destinées.

En partie [IV.B.1](#), nous expliquerons ce qui devrait être fait idéalement en matière de communication entre services au sol. Il faudrait tendre idéalement vers une communication entièrement événementielle entre services au sol.

IV) Prise de recul

A) Choix techniques

1) Choix technologiques

Node/TypeScript/Express

Nous avons choisi de construire nos services en utilisant Node qui est une solution rapide pour créer un service. Cela nous a effectivement permis de construire facilement des services basiques tel que *Weather* qui expose uniquement une API REST avec Express nous facilitant l'écriture des routes. Cela

nous a également permis de développer des services plus complexes comme *Mission* qui communique avec le bus événementiel, expose une API REST et communique avec une base de données. Nous avons également ajouté une couche de TypeScript pour avoir une compréhension claire du code et éviter les confusions sur les types de données.

Persistence

Nous avons choisi de mettre en place de la persistance pour les données télémétriques. Un besoin clair a été énoncé par le client : stocker des données télémétriques et pouvoir les récupérer n'importe quand.

De plus, ajouter de la persistance a permis de décharger la télémétrie et de ne pas garder tout en mémoire. D'ailleurs, puisqu'un grand nombre de données télémétriques sont susceptibles d'être enregistrés au cours du vol, nous avons un fort besoin de mise à l'échelle.

Nous avons donc choisi une base de données NoSQL de type document et ce choix est pertinent par sa simplicité de mise en place et en particulier sa capacité de mise à l'échelle puisque nous traitons beaucoup de données télémétriques en temps réel.

Finalement, ce choix a été bénéfique pour le passage à N missions, puisque les changements ont été beaucoup plus rapides à effectuer car la base de données Mongo permet de facilement ajouter des identifiants à des données télémétriques avec très peu de modifications à effectuer.

2) Choix techniques

a) Organisation des variables d'environnement

Nous avons dans chacun de nos services des variables d'environnement comme les ports d'émission, les informations de connexion à la base de données ou encore la connexion à Kafka.

Ces variables d'environnement ont été très mal gérées par l'équipe, car nous avons dans un premier temps des valeurs par défaut dans le code, qui a rendu le développement fastidieux car nous avons régulièrement des problèmes de port.

La solution aurait été d'utiliser un *.env* commun à tous les projets pour éviter les problèmes de port systématiques chaque fois qu'un service est ajouté.

b) Structure d'accès à la base de données

Chacune de nos entités héritent d'une classe qui gère l'accès à la base de données avec tous les accès CRUD. Cela nous a permis de gagner du temps sur la suite du projet puisque pour persister des données, il suffit juste d'hériter de cette classe et toute la connexion est gérée.

c) Kafka & Factorisation

Bien que nos services tournent de manière indépendante les uns des autres, certains éléments auraient pu être créés dans un module externe, comme *Kafka* ou *Mongo* et être importés dans notre projet pour éviter la duplication. Nous aurions utilisé le système de modules de NPM et ajouté le code commun comme dépendance à notre micro-service.

Dans chaque service communiquant avec le bus évènementiel deux classes sont présentes : un *producer* permettant de publier dans le bus et un *consumer* permettant d'écouter dans le bus. Etant donné que la majorité de nos services communiquent avec le bus évènementiel nous aurions pu factoriser cela dans un module qui propose le code d'un *producer* et d'un *consumer*.

Comme mentionné précédemment une unique classe permet de gérer les accès en base de données, nous aurions pu mettre cette classe dans un module externe afin d'éviter la duplication dans les services.

Concernant les logs, nous avons factorisé le système de *logging* en le plaçant dans un module externe, le simple import de ce module dans un service permet d'activer le *logging* dans le service correspondant.

Actuellement Kafka utilise uniquement un seul *broker*, ce qui fait que le *broker* Kafka doit toujours tourner quelques soit les circonstances pour que l'application fonctionne. Nous avons fait ce choix car pour le moment le projet n'a pas vocation à être déployé à grande échelle. Par la suite, il faut donc ajouter plusieurs *brokers* ainsi que plusieurs *partitions* pour éviter le SPOF.

d) Routes SOAP

Nous avons mis en place des requêtes SOAP dès les premières semaines, cependant après avoir utilisé cette technologie nous nous sommes rendu compte qu'il ne s'agissait pas de la meilleure option pour plusieurs raisons :

- La documentation est très peu présente
- Pas de générateurs de WSDL en Nodejs
- Besoin de générer des wsdl différent pour docker qui n'a pas les même hosts qu'en local.

De plus l'ajout de Kafka a retiré la plupart des routes SOAP et il existe aussi des contrats forts en REST où l'on vérifie précisément le contrat du body.

B) Perspectives d'améliorations

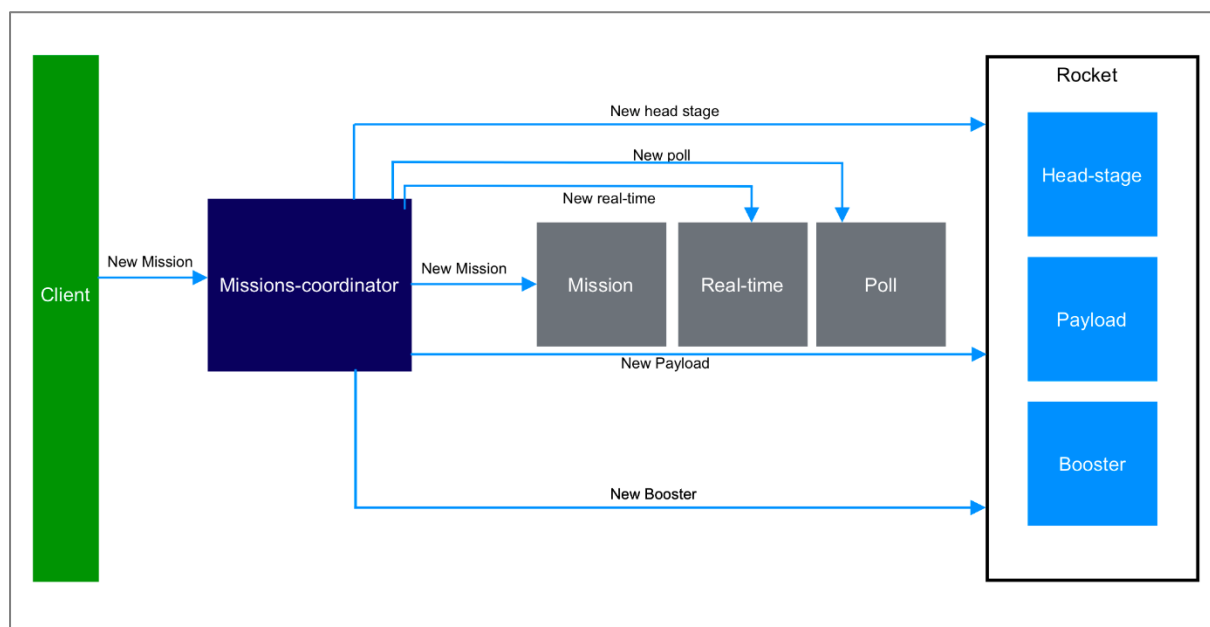
1) Création événementielle d'une mission

Du fait que nous avons un système gérant une seule et unique mission, lorsque nous créons une mission, le service MissionCoordinator avertit tous les autres services par des requêtes REST afin de créer une tête de fusée, un propulseur, une charge utile et un sondage associé à une nouvelle mission.

Lorsque nous n'avons qu'une mission à gérer, la fusée était déjà prête sans que l'on n'ait rien à faire. Pour initialiser une mission, nous pouvions appeler uniquement le sondage.

Le but pour l'avenir est donc de transformer ces appels en événements dans le bus. Lorsqu'une mission se crée, tous les services concernés seraient au courant. Cela donnerait sens à notre *Missions Coordinator* qui attendrait que tout le monde ait répondu et que tout le monde soit prêt.

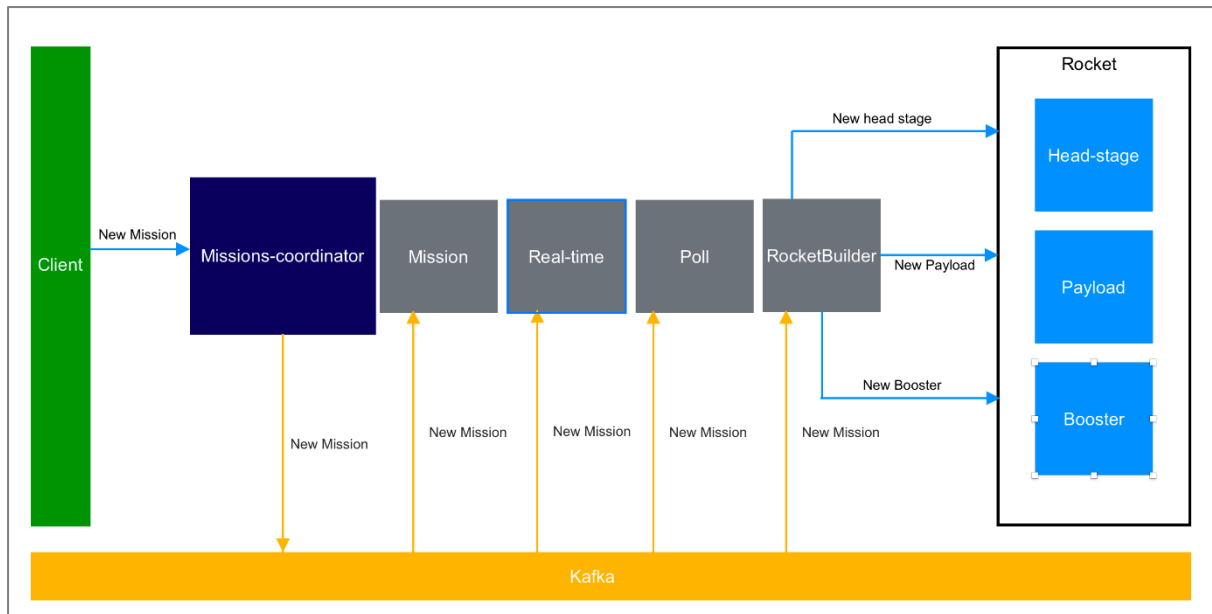
La figure 6 représente notre architecture actuelle pour la création d'une mission.



Légende :
requête REST →

Figure 6: Architecture de création de mission actuelle

La figure 7 est l'architecture de création de mission vers laquelle nous devrions tendre.



Légende :

requête REST →

requête Kafka →

Figure 7: Évolution de la création d'une mission avec une architecture évènementielle

2) Découpage de TelemetryWriter

L'objectif est de découper le service *TelemetryWriter* en deux micro-services, actuellement *TelemetryWriter* peut être appelé par des requêtes REST pour récupérer les données télémétriques. L'objectif ici c'est de décharger *TelemetryWriter* qui doit écrire énormément de données en base de données en créer un service *TelemetryReader* qui aurait pour rôle de récupérer les données télémétriques de la base de données et les fournir via une API REST. L'avantage de cette implémentation, *TelemetryWriter* n'a plus qu'une fonction et s'il tombe, on peut toujours récupérer des données anciennes et avoir la gestion des anomalies sur les données.

La figure 8 représente l'architecture de télémétrie vers laquelle nous voudrions tendre.

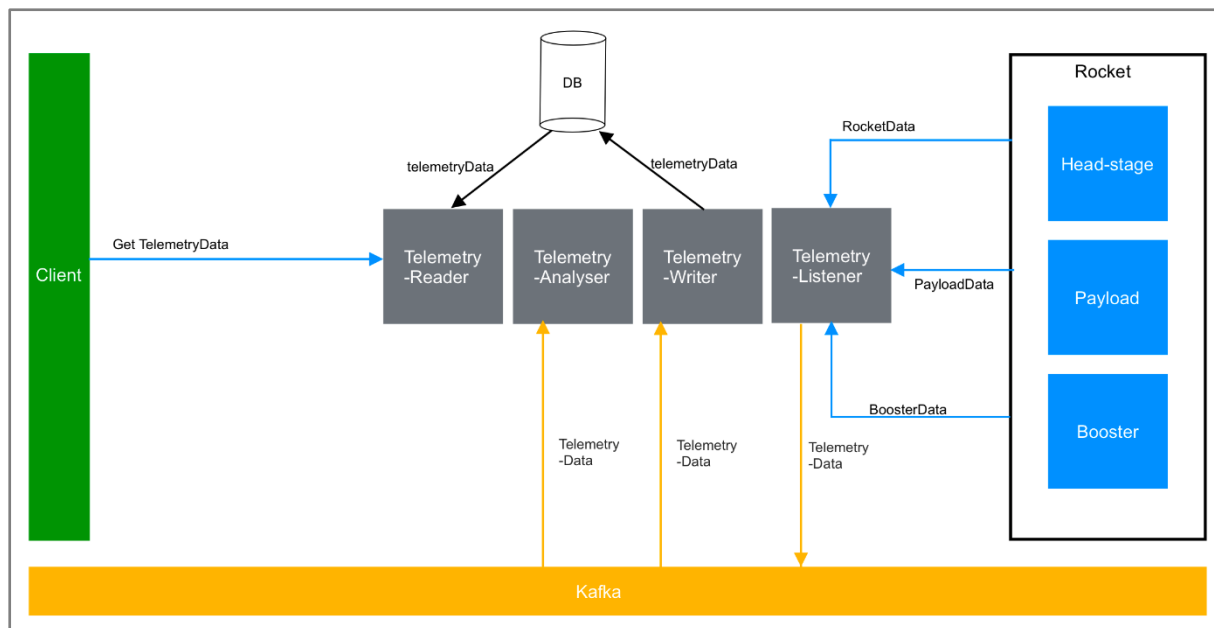


Figure 8: Architecture découpée pour la télémétrie

3) Ajout d'un service de simulation

Notre service booster ne connaît pas son altitude tant qu'il n'a pas été détaché de la fusée. En effet la fusée décolle et transmet son altitude au booster au moment du détachement pour lui permettre de connaître son altitude. En réalité le booster possède des capteurs pour connaître son altitude. Une solution serait d'ajouter un service de « simulation » qui permettrait de simuler une altitude et l'envoyer au booster.

4) Découpage du service mission

La capacité de mission à résister à la charge est une limite de notre système. En effet *Mission* reçoit de *HeadStage* des données et stocke en base de données les statuts de la mission. Ce service pourrait être découpé afin d'optimiser les flux.

5) Ajout d'une passerelle

Actuellement, pour communiquer avec la fusée, les clients communiquent directement avec les services de la fusée (pour lancer la fusée, détruire les modules ou livrer la charge). Pour coller mieux à la réalité il faudrait créer un service passerelle par lequel les requêtes vers la fusée passeraient et qui permettrait de faire sortir les requêtes vers la fusée d'un seul endroit.

ANNEXES

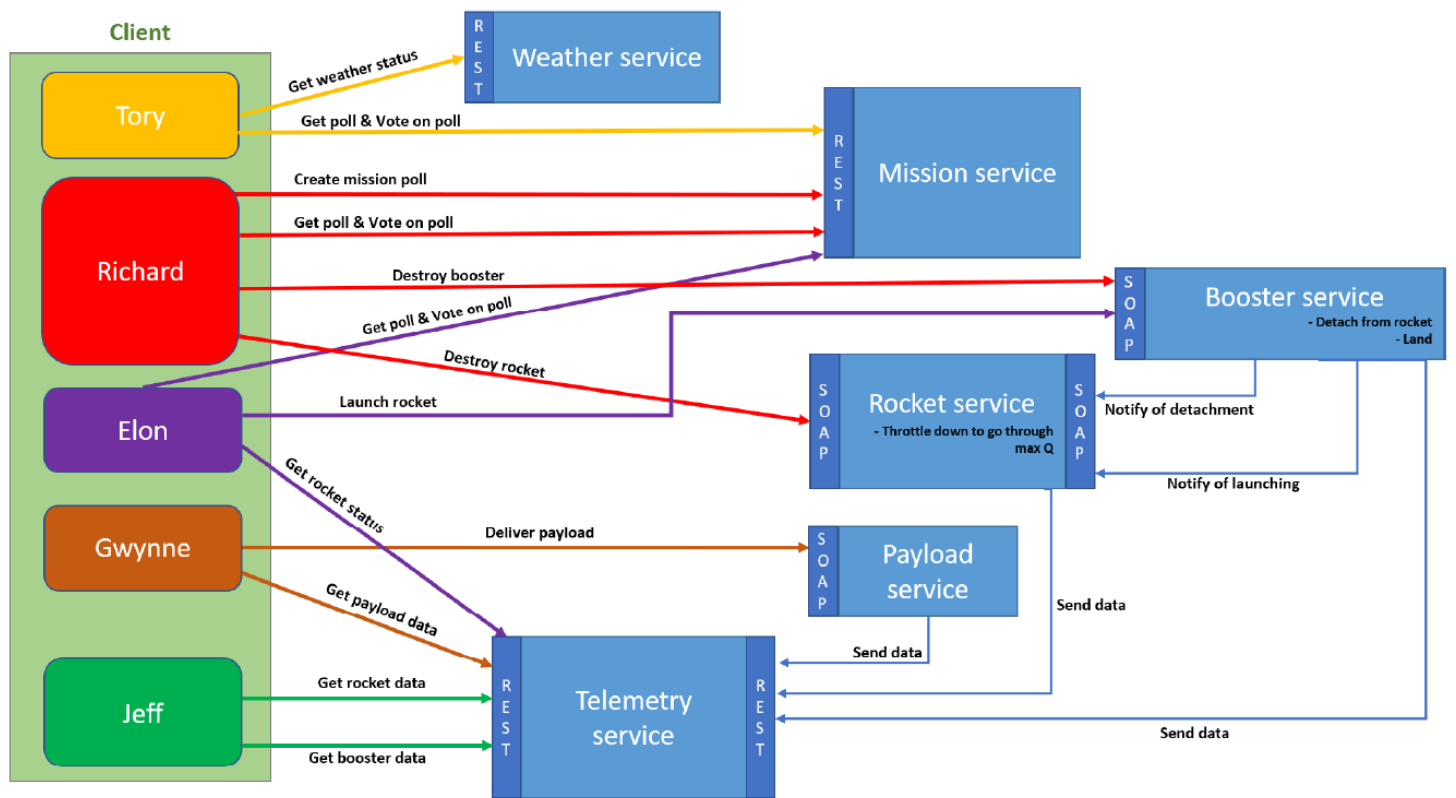


Figure annexe 1 : Diagramme de séquence du premier rendu