POLYTECH NICE SOPHIA | UNIVERSITÉ CÔTE D'AZUR

Rapport de projet

Service-oriented Architecture

L. Gaillard, M. Chevalier, P. Collet

Groupe J
Paul-Marie DJEKINNOU
Paul KOFFI
Florian AINADOU
Djotiham NABAGOU

PLAN

Vue d'ensemble

Le premier livrable de ce projet se composait déjà de microservices échangeant entre eux via différents protocoles de communication tels RPC, REST, Socket. Nous avions basé notre architecture sur certains choix forts qui parfois étaient dus à une mauvaise interprétation de certaines user stories. Nous avons donc revu notre architecture initiale pour ce livrable en gardant la plupart de nos services déjà en place et en revoyant les protocoles de communication entre ces derniers et les nouveaux.

Architecture du projet

I. Hypothèses et reconsidération des choix

1. Séparation des composants terrestres (en bleu sur l'architecture finale) et des composants à distance (en rouge)

Pour cette itération, nous avons décidé de faire la différence entre les composants terrestres et à distance. En effet, nous avions fait un choix d'interprétation plutôt fort qui nous faisait considérer la fusée comme un système de simulation. Ce choix nous a fait faire abstraction du comportement réel d'une fusée typiquement pour ce qui est de la séparation réelle des deux étages de la fusée.
En effet, une simulation semblait bien plus simple car nous nous comportions comme si elle était au sol. Ainsi dans notre ancien système de simulation, nous étions parti sur l'hypothèse où le secondStage de la fusée se substitue à notre satellite et donc ces 2 composants représentaient un seul élément. On décidait donc de la séparation de cet élément du firstStage à partir d'un moment donné T. Nous avons remédié à ceci dans notre nouvelle version où la rocket est constituée de 2 parties (firstStage & secondStage) et sera affectée ensuite à la mise en orbite d'un satellite. Ainsi dans notre nouveau système, les 3 composants (firstStage , secondStage & payload) ont bien leurs propres données télémétriques différentes et notre système de simulation laisse place à un vrai lancement (bien amélioré par l'intégration des messages KAFKA) avec les 3 composants bien définis.

2. Le lancement de la fusée

Pour le lancement de la fusée, l'application se comporte toujours de la même façon mais les échanges en REST se trouvent remplacés par des messages KAFKA. Cela rend le processus un peu plus facile à monitorer.

3. Les étapes de lancement de la fusée

Pour effectuer le lancement de la fusée, la version précédente faisait un polling permanent sur un service "**launcher**" afin de spécifier ces étapes. Le système semble d'autant plus simple à monitorer avec un système événementiel car chaque séquence du lancement d'une fusée est un événement qui sera traité par les services **consummer** intéressés.

4. Le système d'anomalie

Pour le dernier lot de User Stories, une des fonctionnalité était de donner à Richard la possibilité de détruire une fusée en cas de détection d'anomalie. A cela, nous avons ajouté un tout autre service permettant de détecter automatiquement ces anomalies. Ce service supervise la trajectoire de la fusée puis en cas d'anomalie envoie un message qui lance la destruction de la fusée.

Une autre façon de concevoir ce système aurait été de l'entrevoir comme un dispositif placé sur la fusée qui s'auto-détruirait avec la fusée en cas d'anomalie. Ce système nous semblant plutôt coûteux, nous avons choisi la première option.

Le comportement avancé que doit avoir le trigger Anomalies :

- Avant la séparation du first et du secondStage , recevoir les données télémétriques du firstStage au fur et à mesure qu'elles sont transmises par le firstStage, faire la comparaison avec les données télémétriques de la table anomaly. S'il y'a un incident ===> détruire le firstStage, le secondStage ainsi que le payload
- Après la séparation, recevoir les données télémétriques du secondStage au fur et à mesure qu'elles sont transmises par le secondStage, faire la comparaison avec les données télémétriques de la table anomaly. S'il y'a un incident ===> détruire le secondStage ainsi que le payload seulement car le firstStage après séparation n'a pas besoin d'être détruit à moins qu'une anomalie ne soit détectée quand le firstStage revient sur terre.
- Après la séparation du secondStage et du payload, recevoir les données télémétriques au fur et à mesure qu'elles sont transmises par le payload, faire la comparaison avec les données télémétriques de la table anomaly. S'il y'a un incident ===> détruire seulement le payload car il est le seul concerné.
- Après la séparation du first et du secondStage, recevoir les données télémétriques du firstStage au fur et à mesure qu'elles sont transmises par le firstStage lorsqu'il revient donc sur terre, faire la comparaison avec les données télémétriques de la table anomaly. S'il y'a un incident ===> détruire seulement le firstStage.

Le comportement que nous avons implémenté pour le trigger Anomalies :

- Dès qu'il remarque une anomalie, le firstStage , le secondStage et le payload sont détruits.

Nous avons fait ce choix car le projet n'était pas orienté sur la qualité de nos algorithmes mais sur l'architecture proposée ainsi que la bonne intégration des protocoles de communication entre nos différents services.

5. La spécificité de l'eventCollector

L'eventCollector est le seul composant qui écoute une multitude de topics (presque tout les topics). En effet, ce composant est celui qui est chargé de retracer l'historique des différents événements relatifs au système. Nous avons donc trouvé pertinent de lui permettre d'écouter tous les topics.

Nous avons donc eu le choix de procéder de deux manières différentes :

- Implémenter deux 'eventCollector' : l'un qui se charge d'envoyer à Marie tous les logs du système et l'autre qui se charge de les enregistrer dans la BD.
- Implémenter un seul 'eventCollector' qui se charge d'envoyer à Marie tous les logs du système et fait appel à un autre service REST pour l'enregistrement des logs dans la BD.

Nous avons choisi la seconde option compte tenu du nombre important de messages à traiter par un eventCollector afin de faire les deux actions (envoi à Marie et sauvegarde dans la BD) en une seule fois pour chaque message reçu.

Nous avons préféré déléguer cette tâche à un service en particulier plutôt que de permettre à tous les autres services de faire appel au service REST pour enregistrer leurs logs.

6. La destruction de la fusée

Avant la séparation du firstStage et du secondStage de la fusée, l'ordre de destruction de la fusée lancé par Richard impactera les 3 parties de la fusée (firstStage , secondStage et satellite tous détruits). Par contre si après la séparation du firstStage et du secondStage, Richard émet son ordre de destruction, seul le firstStage est détruit. Nous avons fait ce choix car nous considérons qu'après la séparation du first et du secondStage, le système de détection des anomalies prend la suite des évènements en main et se chargera donc de détruire le composant adéquat dans lequel survient l'incident automatiquement.

7. Le supplierService (bonus)

Comme User Stories pour le dernier livrable, nous avons choisi un système de ravitaillement de fusées en essence. Ce service se chargera de sa gestion.

8. La répartition des topics KAFKA

KAFKA nous permet de gérer les événements en les envoyant sur des topics. Il aurait été possible d'envoyer tous les messages sur le même topic. Nous avons néanmoins décidé de répartir les communications sur des topics spécifiques. Nous avons ainsi:

- ➔ Le topic *launcherTopic* qui sera utilisé pour envoyer les événements relatifs au bloc de la fusée
- ➔ Le topic *pollsystemTopic* qui sera utilisé pour la requête de lancement de la fusée
- ➔ le topic *rocketTopic* qui permet à la première partie de la fusée de donner des informations relatives à son évolution
- ➔ le topic *rocketTopicS* qui permet à la deuxième partie de la fusée de donner des informations relatives à son évolution
- ➔ le topic *payloadTopic* qui permet au chargement de donner des informations relatives à son évolution
- ➔ le topic *eventCollectorTopic* qui permet au système d'analyse des évènements de les diffuser
- ➔ le topic *supplierTopic* qui permet le partage des informations relatives à un nouveau service de gestion de livraison d'essence sur un satellite

II. Evolution d'Architecture

1. DELIVERY-FIRST : Restructuration

Nous avions axé notre compréhension du fonctionnement de la fusée sur une simulation en le justifiant par le fait qu'on ne travaillait pas sur une fusée de la vraie vie. Ces choix forts nous ont

fait négliger les contraintes physiques et interactions humaines auxquelles les fusées sont soumises pour leur décollage.

L'introduction d'un nouveau protocole de communication entre les services (Kafka) a fortement contribué à l'amélioration de notre architecture.

Les dashboards respectivement de Jeff et Gwynne qui étaient des clients socket à cause des données télémétriques en continu, deviennent donc des 'consumers' kafka qui après récupération d'un message les affichent.
Parallèlement, les serveurs de données télémétriques qui se chargent de récupérer les données télémétriques auprès des services pour les envoyer aux dashboards, deviennent quant à eux des 'producers'.

Nous voulions aussi un système temps réel et automatique qui collecte les réponses des votes de Tory et Elon quand Richard le leur demande puis envoie le signal de lancement à la fusée si la réponse est favorable. Nous avions donc mis en place un système REST permettant de faire continuellement des requêtes 'GET' pour savoir quand la fusée doit décoller et quand elle doit effectuer ses différentes séparations. Cela marchait mais nécessitait de demander à chaque fois au serveur et d'attendre en retour les réponses. Ce système à la demande avec des requêtes incessantes a été remplacé par kafka qui nous permet d'envoyer au bon moment et à tous les services cibles, le message souhaité sans avoir à requêter continuellement le serveur.

Suite à l'introduction de kafka, nous avons également migré trois de nos services REST initialement implémentés en Node JS vers Python. Les 'consumers' en Node JS prenant un laps de temps à démarrer contrairement à ceux de Python, nous avons donc préféré changer la techno de ces derniers pour éviter toute perte de message pouvant entraîner des erreurs.

Nous avons donc revu notre compréhension de certaines user stories en se basant sur ces critères précités. Ceci nous a mené à la version suivante de l'architecture.

2. Scope 4 (Ajout des US du 12/10)

Suite au premier livrable, nous nous sommes concentrés sur la restructuration du projet et l'intégration de kafka dans notre schéma d'architecture. Nous avons donc laissé les 3 sets de user stories de cette semaine en stand by pour finir la restructuration avant d'implémenter les nouvelles fonctionnalités. Nous avons préféré ensuite nous concentrer sur l'élaboration d'une architecture solide avec intégration de kafka là où cela nous semblait le plus pertinent. Cette nouvelle architecture serait présentée donc à la séance suivante.

3. Scope 5 (Ajout des US du 19/10)

Une fois la restructuration effectuée et notre nouvelle architecture validée, nous avons ajouté les fonctionnalités de ce sprint et du sprint précédent. Il faut noter que l'intégration de kafka a considérablement facilité l'écriture du code ainsi que la réalisation des fonctionnalités demandées. La nouvelle architecture nous a permis d'avoir une vue d'ensemble du déroulement de la mission grâce à un nouveau service, l'EventCollector qui se charge de récupérer toutes les informations sur le déroulement de la mission auprès des autres services. Cela se traduit par la

récupération des messages sur tous les topics de notre système comme illustré sur la figure de l'architecture finale. En faisant un zoom sur l'event collector on peut le constater :

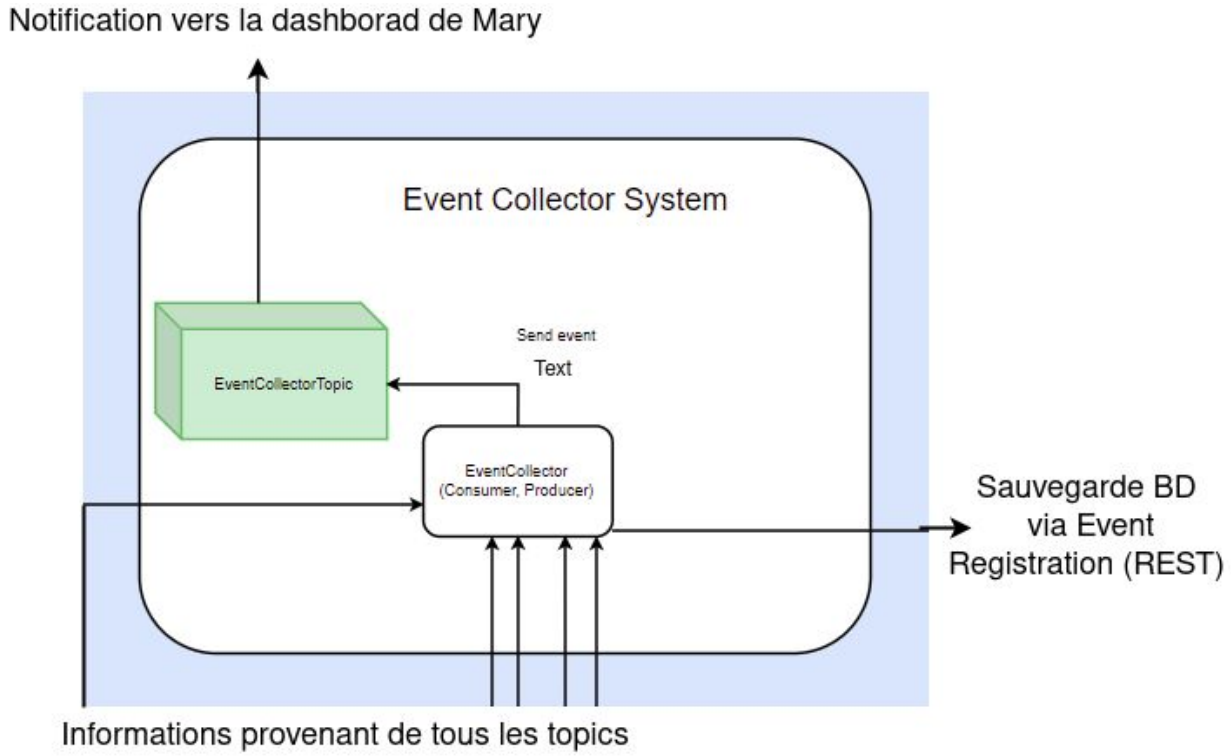

Figure 1 : Zoom sur l'EventCollector (tiré du schéma d'architecture final)

À chaque étape de la mission, l'event collector reçoit en temps réel les logs émis par les autres services et qui transitent via le bus d'événement. Ces informations sont par la même occasion stockées dans la base de données pour être consultables au besoin et sont également diffusées à Marie en temps réel.
Aucun des services de ce fait, n'enregistre directement les logs qui lui sont associés dans la base de données. Ils transitent tous par l'event collector qui renvoie à son tour les données à l'event registration service (REST) qui se charge de les insérer en BD.

Nous avons associé la dashboard de Peter à celle de Jeff et affichons dans celle-ci les données télémétriques du firstStage & du second Stage ainsi que toutes les notifications concernant ces 2 parties de la fusée.

4. Scope 6 (Ajout des US du 26/10) - DELIVERY FINAL

Pendant ce scope, tout en peaufinant le reste de nos services, le vrai développement consistait à l'élaboration du système d'anomalies qui se devait de détruire automatiquement la fusée en cas d'incident. Nous avons créé dans notre BD une nouvelle table pour les anomalies où des données télémétriques prévues concernant le firstStage, le secondStage et le payload étaient récupérables en utilisant comme clé le satellite dont la mise en orbite s'effectuait. Pour que tout

se passe bien, il suffisait alors d'utiliser les mêmes données que celles utilisées par les composants sinon pour provoquer une anomalie de changer l'une d'entre elles.

Nous avons choisi comme nouveau lot de US :

- En tant que Richard, je veux avoir la possibilité de ravitailler un satellite mis en orbite avec du carburant.
- En tant que Richard je veux connaître la quantité de carburant restant pour un satellite en orbite pour décider d'un ordre de ravitaillement si cela est nécessaire.
- En tant que Victor, responsable du 'Supplier Service', je veux pouvoir ajouter les nouvelles fusées de ravitaillement construites dans le système
- En tant que Victor, responsable du 'Supplier Service', je veux pouvoir suivre les événements relatifs au lancement d'une mission de ravitaillement

Ces nouvelles US nous ont donc conduit à implémenter les nouveaux services suivants :

- satelliteService (REST) : qui permet la récupération de la quantité restante de carburants d'un satellite mis en orbite ainsi que la mise à jour de cette quantité en cas de ravitaillement.
- supplierService (REST) : qui permet l'enregistrement d'une nouvelle capsule de ravitaillement, la recherche d'une capsule disponible ainsi que la MAJ du carburant de la capsule elle-même.
- fuelSupplier (Consumer&Producer) : qui reçoit l'ordre de ravitaillement d'un satellite, vérifie en s'adressant au deliveryService si ce satellite a été bien mis en orbite, s'adresse ensuite au supplierService pour vérifier s'il y'a un supplier disponible pour la mission. Il effectue ensuite l'opération de ravitaillement tout en notifiant Victor sur le déroulement de la mission du début à la fin. Il se charge également de faire les MAJ des carburants du satellite et de la capsule grâce au satelliteService et au supplierService.

L'eventCollector est également abonné au topic de la mission de ravitaillement et donc Mary voit également le déroulement des missions de ravitaillement. Les logs de cette mission sont également enrégistrés dans la BD.

III. Objets métiers

1. Les objets stockés en base de données

Pour stocker nos données, nous utilisons une base de données NoSQL en l'occurrence MongoDB. Les principales données stockées sont les suivantes:

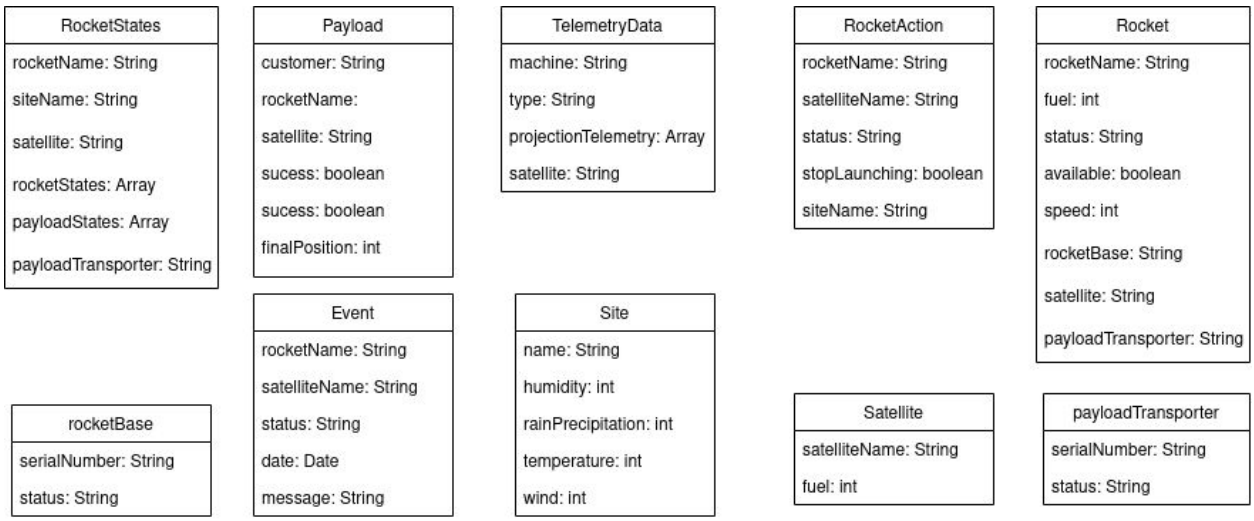

Figure 2 : Objets métiers du système

Outre les objets métiers, nous avions besoin de modéliser afin de pouvoir assurer les différents scénarios. L'objet Event nous permet de stocker les événements.
L'objet **RocketAction** avait été ajouté afin de rendre possible des actions sur la fusée distante. On faisait à la version juste avant l'intégration de KAFKA un polling continu sur des objets de ce type.
Avec l'intégration de KAFKA, nous pensions ne plus avoir besoin de cet objet. Car, en effet, KAFKA nous permet justement de gérer ces événements. Néanmoins, nous avons été obligés de garder cet objet car un problème est survenu.
En effet, afin de gérer la fonctionnalité de destruction de la fusée, la solution qui nous paraissait évidente était un système de variable globale qui nous permettrait d'arrêter le déroulement du processus de lancement de la fusée.
Pour ce faire, nous avions utilisé dans le code du **rocketFirstStage**, deux "consumers KAFKA" dont un qui réagissait à l'évènement de lancement de la fusée et le second qui écoutait le topic de destruction. Malencontreusement, pour des raisons que nous ignorons, cette solution ne fonctionnait pas à certains moments. Nous avons donc dû revenir à la version précédente de la destruction et donc maintenir l'objet **RocketAction** au travers du LauncherService**. (Voir Annexes Figure 7)**

2. Les objets échangés

Avant l'intégration de Kafka, nous n'avions pas réellement d'objets métiers échangés mais plus des paramètres (id d'objets, nom de site, nom de satellite, etc...). En effet, notre architecture était essentiellement composée de composants RPC et REST. Tout se faisait donc au travers de routes REST ou d'appel à des requêtes sur des composants RPC.
Avec l'intégration de Kafka, nous avons mis en place un contrat fort: une structure de données échangée au travers des messages Kafka. Cette structure nous permet entre autres de faire des contrôles sur les messages échangés sur les différents topics. Nous échangeons les objets généralement au format JSON.

Par exemple, le message de lancement de la fusée se présente de cette façon:

```
{       'action'       : "launch",
        'siteName'     :  site,
        'rocketName':   rocket          }
```

De façon générale, la plupart sinon tous les messages envoyés au travers de Kafka contiennent le champ **'action'**. C'est le champ clé qui nous permet de savoir comment le système doit se comporter. Les champs **'siteName'** et **'rocketName'** sont aussi fréquents car la plupart de nos messages comportant le nom du site de lancement ainsi que celui de la fusée qui est lancée. Nous conservons ces deux informations car elles nous sont utiles tout au long du processus de lancement de la fusée. On retrouve dans certains contrats des champs comme ("satelliteName", "position", "message" etc...)

IV. Scénarios

Les différents scénarios couverts par notre application sont décrits au travers de diagrammes de séquences. Dans les divers diagrammes, notez que la plupart des interactions se font au travers de topics kafka que nous n'avons pas modélisés. Dans le cas où l'échange serait fait par **requêtes REST**, nous préciserons la méthode REST utilisée afin de faire la différence.

Le scénario de lancement de la fusée a été séparé en deux parties:

- la partie du Poll lancé par Richard pour lancer la fusée **(Voir Annexes Figure 5)**
- l'interaction entre les différentes parties de la fusée et la station terrestre **(Voir Annexes Figure 6)**

Pour ce qui est de la destruction, deux scénarios ont été représentés:

- La destruction effectuée par Richard depuis sa CLI **(Voir Annexes Figure 7)**
- La destruction réalisée par TriggerAnomalie en cas d'anomalie détectée **(Voir Annexes Figure 8)**

Enfin, un diagramme permet de décrire un scénario de lancement d'une mission de ravitaillement. **(Voir Annexes Figure 9)**

Rétrospective

I. Limites du système actuel

Dans ce projet, bien que cela ait été prévu, nous n'avons pas eu le temps de séparer les données de lancement de l'application de nos données de test. Ce qui pose un problème typiquement lorsqu'on essaye de lancer les scénarios alors que les tests sont en train de tourner sur Travis. Ce qui peut provoquer une pollution des données de l'application en exécution simultanée. Cependant, si ces lancements ne sont pas faits simultanément, nous avons mis en place des scripts qui nettoient automatiquement la base de données.

Notre système ou du moins une partie de notre système ne supportera pas le lancement de fusées en simultanée. Etant donné que notre architecture nous permettait déjà le lancement de plusieurs fusées à la suite lorsque la US a été introduite et que cette US ne prenait pas en compte le caractère simultanée du lancement, nous avons décidé volontairement de ne pas transiter l'architecture vers une qui permettrait cela et qui serait plus complexe. N'apportant pas réellement une Plus-value au projet, nous avons donc préféré mettre l'accent sur d'autres points plus importants.

II. Idées d'amélioration

La première piste d'amélioration serait de provoquer la destruction grâce à un event kafka comme ce qui était prévu ce qui nous permettrait de supprimer les appels GET pour savoir si une destruction a été enclenchée. D'après nos premières recherches, il semblerait que le problème d'avoir 2 consumers dans le même fichier est propre à notre techno (python).

Une autre piste d'amélioration consisterait à faire une MAJ de notre architecture pour permettre le lancement en simultanée. On pourrait tirer partie des partitions en kafka et un peu de multi-threading devrait faire l'affaire.

Annexes

- ## Architecture du livrable 1

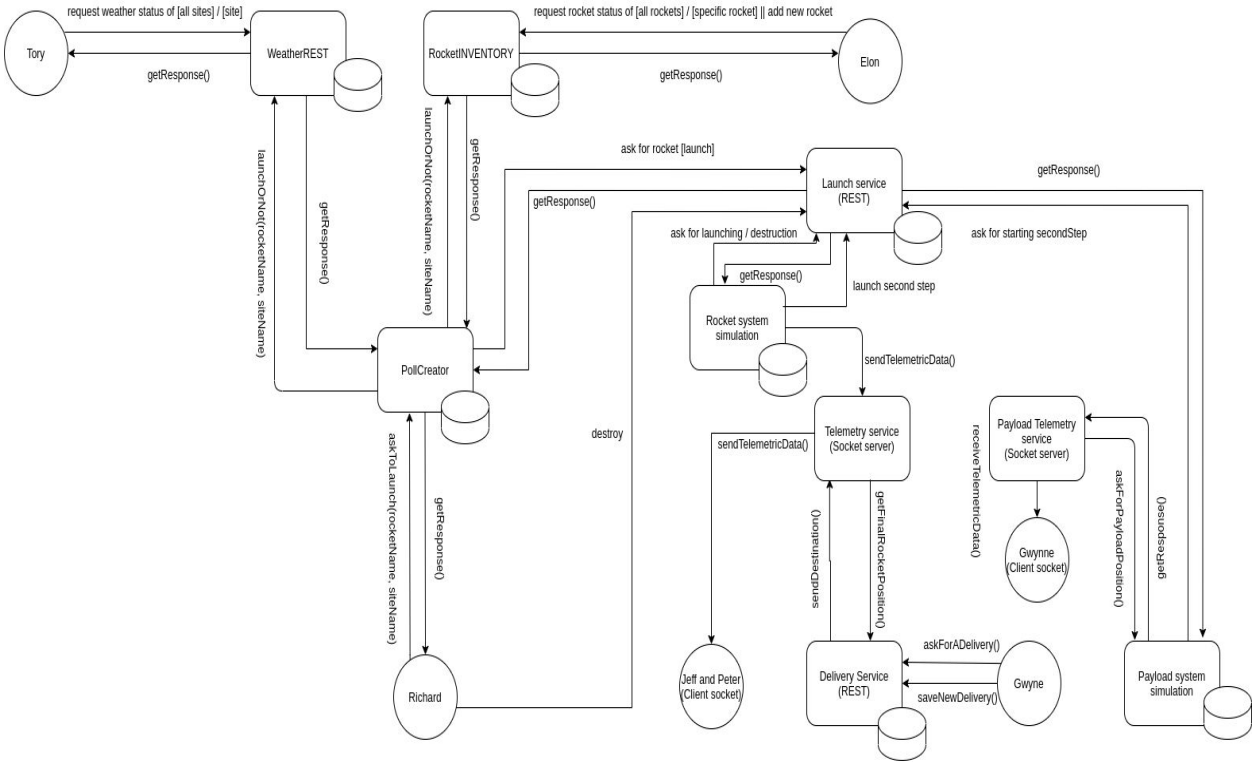

Figure 3: Architecture du delivery-first

(PS : Voir en taille réelle sur https://github.com/pns-si5-soa/box-20-21-team-f/blob/master/docs/archi_scope_3.png)

- **Architecture du livrable final**

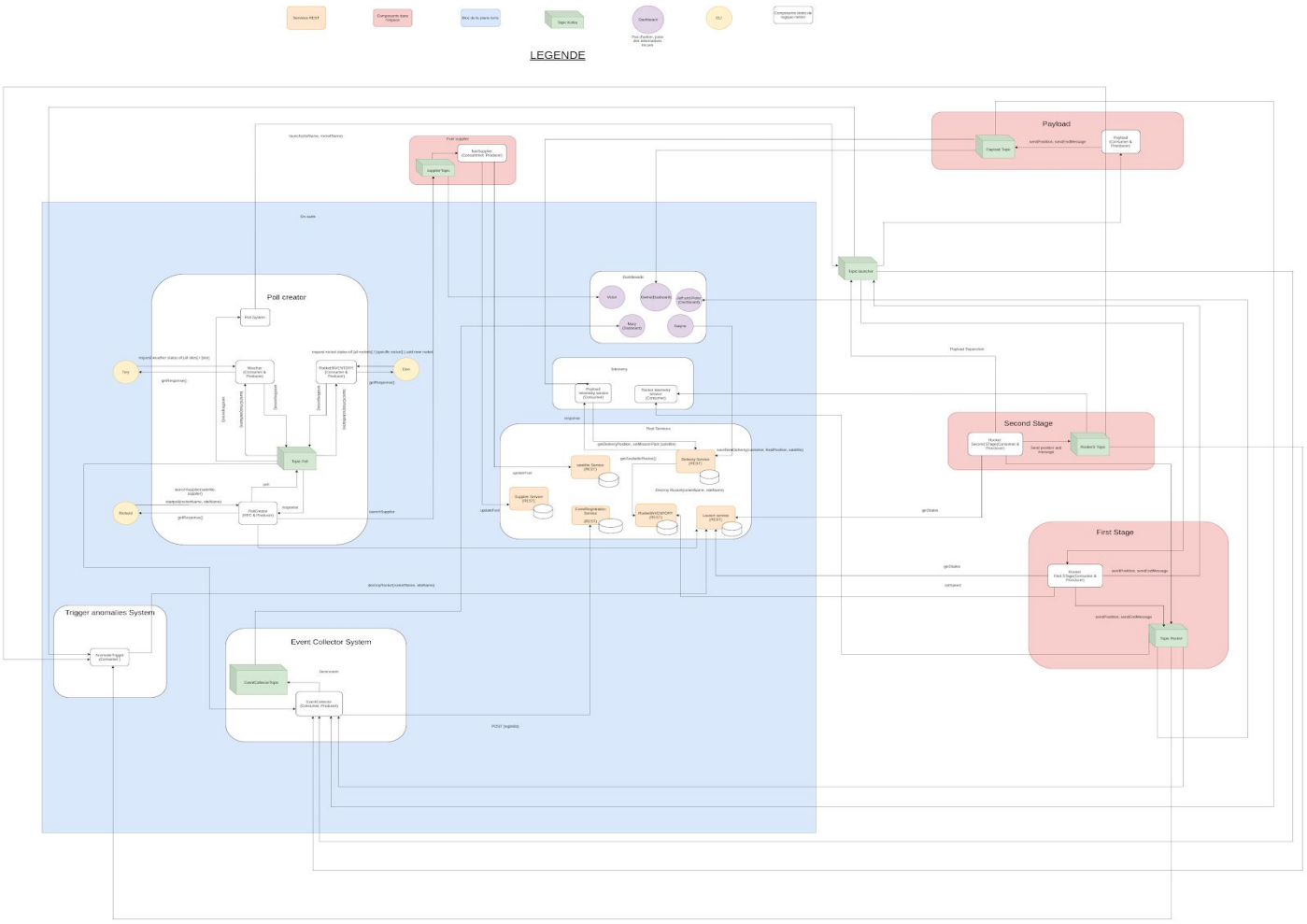

Figure 4 : Architecture du livrable final

(PS : Voir en taille réelle sur https://github.com/pns-si5-soa/box-20-21-team-f/blob/master/docs/final_architecture.png)

- **Diagramme de séquence du lancement de la fusée**

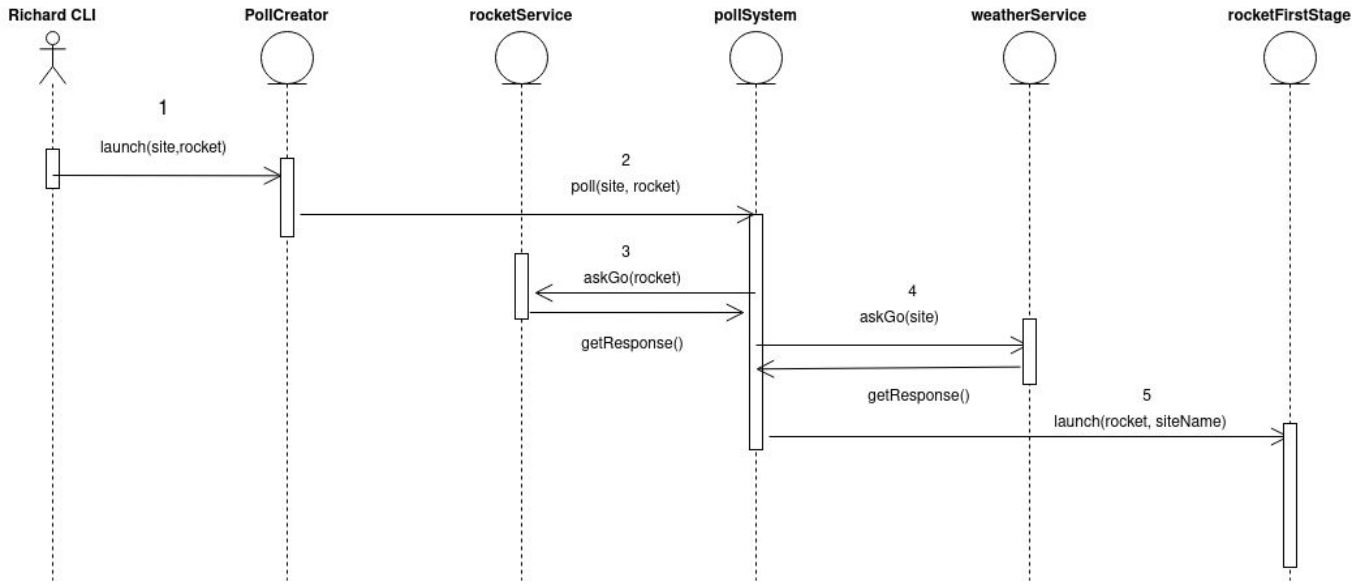

Figure 5: Lancement de la fusée

(PS : Voir en taille réelle sur https://github.com/pns-si5-soa/box-20-21-team-f/blob/master/docs/launchPollSystem.png)

Description :(Richard lance l'ordre de lancement depuis sa CLI, le pollCreator publie sur les topic du weather et du rocket consumer qui se chargent de vérifier les statuts du site et de la rocket en question. Ces 2 consumers répondent ensuite sur le topic du pollSystem qui après réception des 2 messages démarre le lancement si il a les 2 GO.

- **Diagramme de séquence décrivant la communication entre les parties de la fusée et la terre après le lancement**

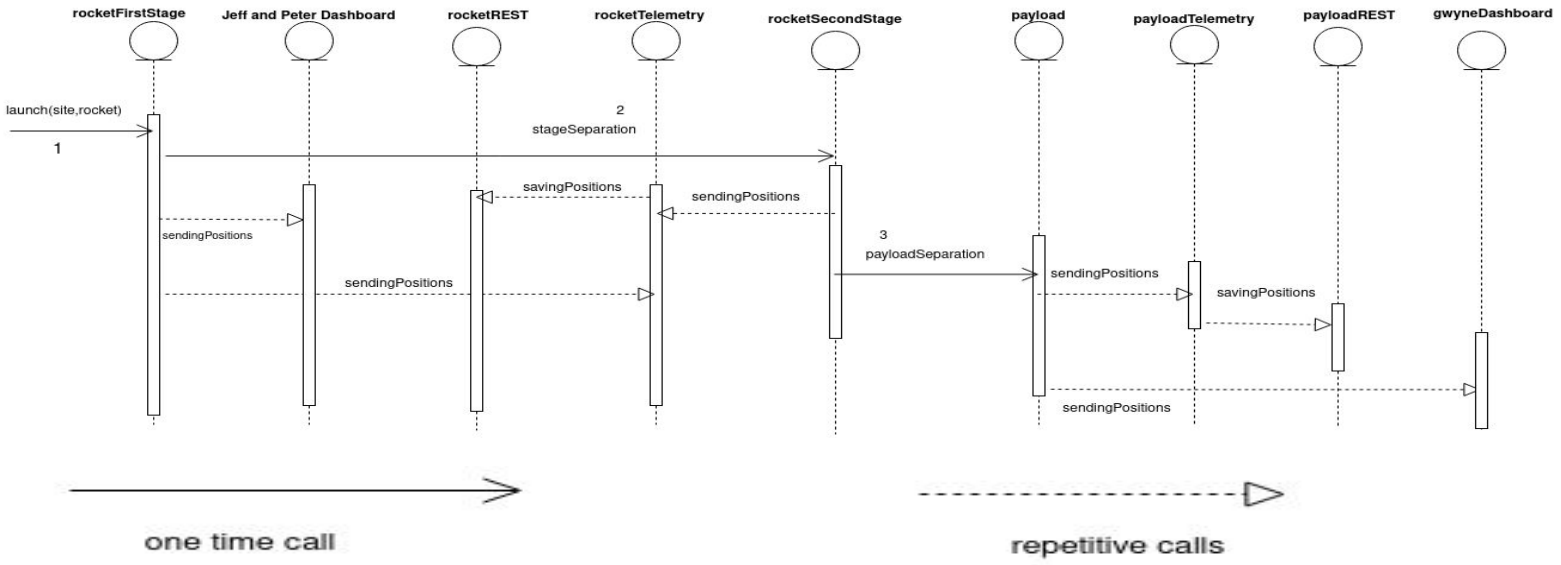

Figure 6 : Communication entre les parties de la fusée et la terre après le lancement

(PS : Voir en taille réelle sur https://github.com/pns-si5-soa/box-20-21-team-f/blob/master/docs/communication%20entre%20les%20parties%20de%20la%20fus%C3%A9e%20et%20la%20terre%20au%20lancement.png)

Description : le firstStage reçoit "launch" sur le launcherTopic, il envoie en continue ses données telémetriques au rocketTelemetryServer et aux dashboard. Il envoie également les étapes de lancement ainsi que les données télémétriques à l'eventCollector qui n'est pas représenté sur le schéma. Il envoyera au moment adéquat l'ordre de séparation au secondStage qui fera les mêmes actions. Le second stage enverra quant à lui l'ordre de séparation au payload. Le rocketTelemetry ainsi que le payload Telemetry se charge d'enregistrer les données télémétriques au fur et à mesure dans la BD par l'intermédiaire des serveurs REST. (**non représenté : Le payloadTelemetry se chargera aussi de vérifier à la fin si les positions finales concordent en s'adressant au deliveryService, si tel est le cas il changera donc le statut de la mission à "success"**).

- **Diagramme de séquence du scénario de destruction de la fusée par Richerd**

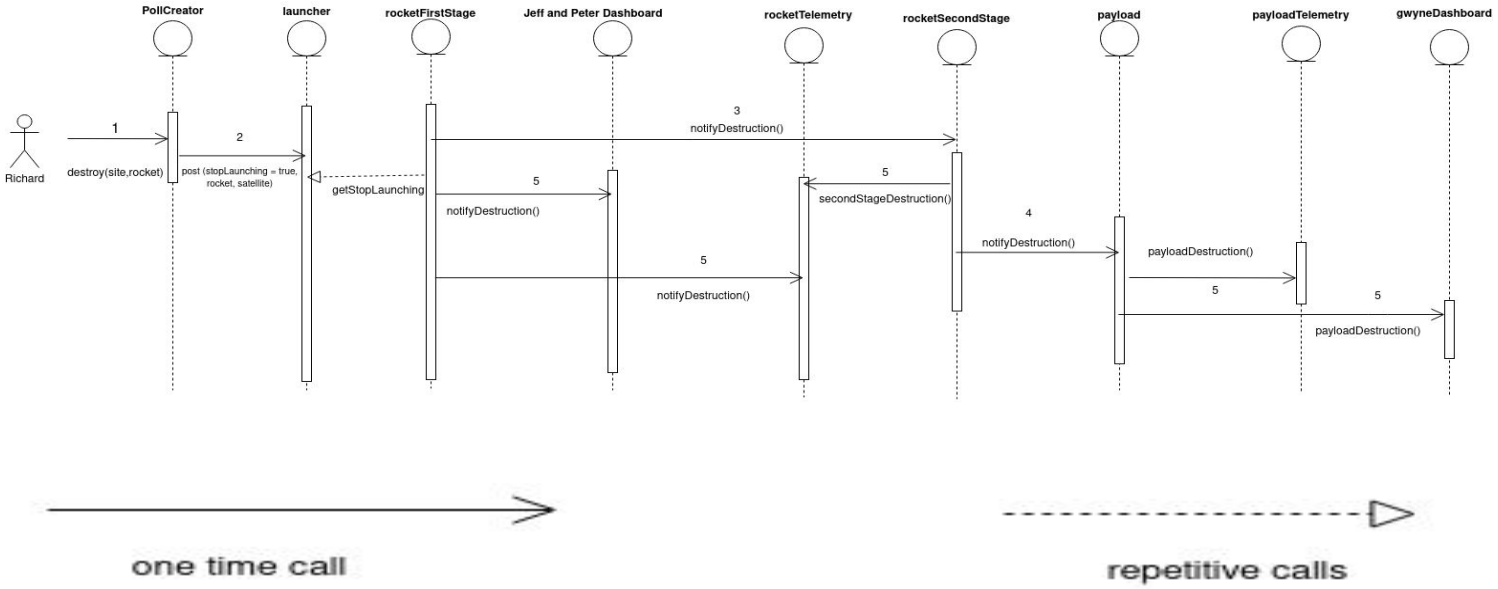

Figure 7 : Scénario de destruction de la fusée (Par Richard)

(PS : Voir en taille réelle *sur https://github.com/pns-si5-soa/box-20-21-team-f/blob/master/docs/richardDestruction.png)*

Description : Quand Richard décide de la destruction de la fusée, le statut de destruction passe à True dans la BD pour cette mission. Le firstStage à chaque étape vérifie la valeur de ce statut. Un effet de chaîne s'ensuit.

- **Diagramme de séquence du scénario de destruction de la fusée par TriggerAnomalie**

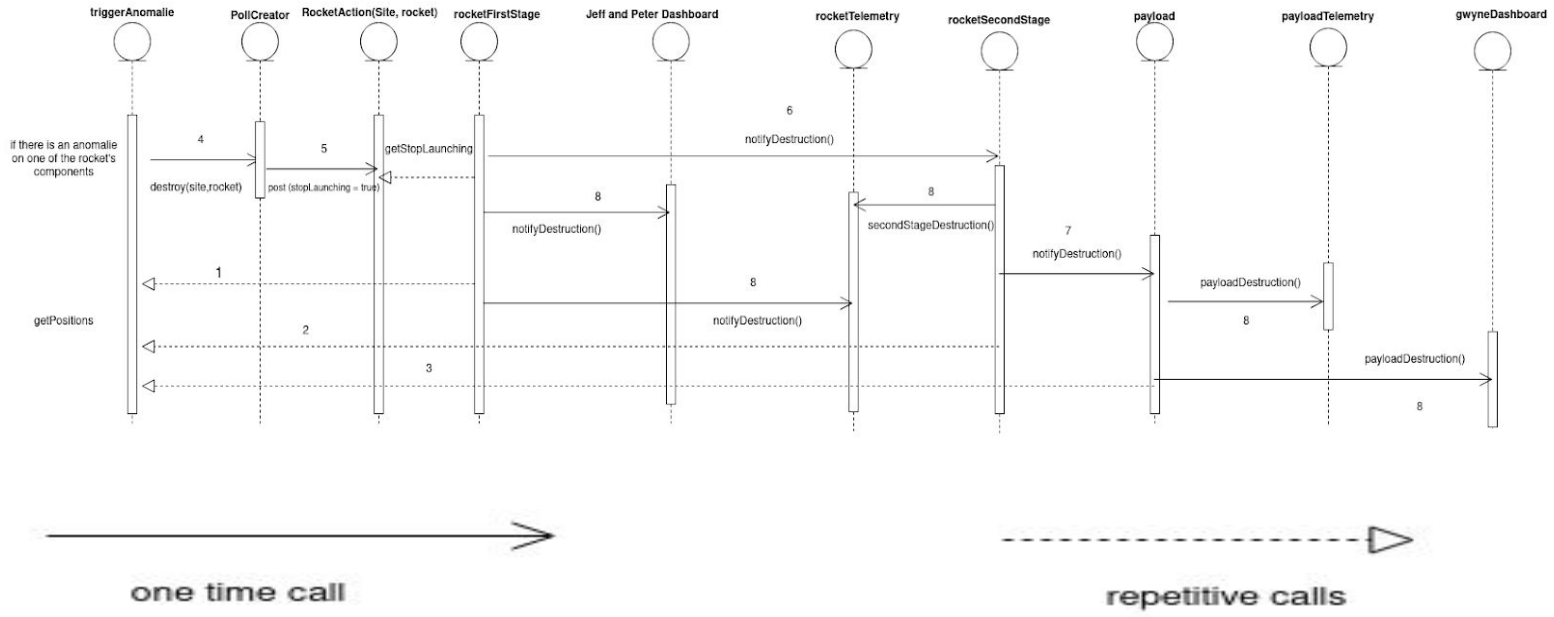

Figure 8 : Scénario de destruction (Par TriggerAnomalie)

(PS : Voir en taille réelle *sur https://github.com/pns-si5-soa/box-20-21-team-f/blob/master/docs/triggerAnomalie.png)*

Description : Le triggerAnomalies à chaque nouveau “launch” récupère les valeurs prévisionnelles télémétriques de la table anomaly. Il reçoit toutes les valeurs télémétriques et pour chacune vérifie si elles sont égales à celle prévue. Si non il modifie donc la variable de destruction de cette mission en passant par le launcherService.

- **Diagramme de séquence du scénario de ravitaillement du satellite**

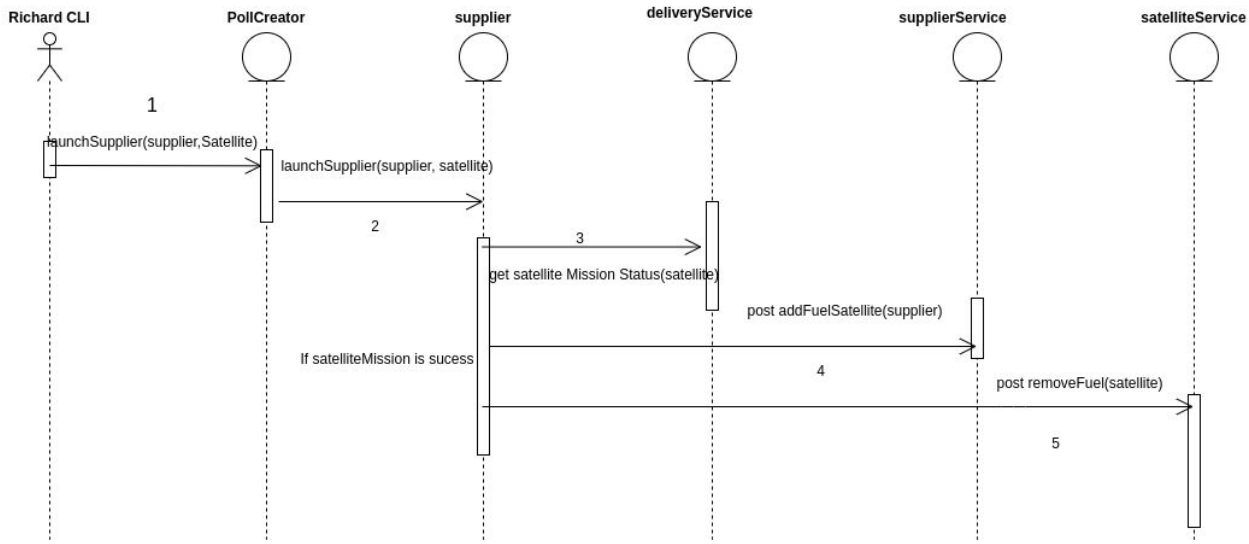

Figure 9: Scénario de ravitaillement de satellite

(PS : Voir en taille réelle *sur https://github.com/pns-si5-soa/box-20-21-team-f/blob/master/docs/supplier.png)*

Description : Quand Richard lance l'ordre de ravitaillement de la fusée en indiquant le supplier, le supplier "consumer & producer" se charge d'abord dans un premier temps de vérifier la disponibilité du supplier ainsi que la vérification du succès de la mise en orbite du satellite. Si tel est le cas, il procède ensuite au ravitaillement tout en mettant à jour les carburants du satellite et de la capsule.