Dernière modification: 4 février 2022

INF3995 : Projet de conception d'un système informatique
Hiver2022
Documentation du projet

Système aérien d'exploration

Documentation du projet répondant à l'appel d'offres no. H2022-INF3995 du département GIGL

Équipe No. 106

Alexis Foulon

Gaya Mehenni

Paul-André Bisson

Florence Cloutier

Félix-Antoine Constantin

Nicholas Legrand

Table des matières

1 Vue d'ensemble du projet

1.1 But du projet, porté et objectifs (Q4.1)

Au cours de ce projet nous avons comme tâche de développer un ensemble de logiciels permettant de gérer et d'interagir avec un essaim de drones. Ces drones auront pour but d'explorer une autre planète pour localiser des points d'intérêts, cartographier l'environnement et guider par la suite un robot plus massif vers ceux-ci. Les drones devront être en mesure d'explorer l'environnement de manière autonome et de communiquer avec une station au sol pour envoyer les données de cartographie et recevoir des commandes. Le projet final sera composé de 5 livrables : une interface utilisateur, un serveur, une base de données, le logiciel embarqué et une application simulant le comportement des drones.

L'interface utilisateur aura pour but de fournir à l'opérateur une vue d'ensemble du système. Il pourra visualiser la carte globale de l'environnement ainsi qu'une carte spécifique à chaque drone. L'interface utilisateur devra aussi permettre d'interagir avec les drones à l'aide de certaines commandes prédéfinies. Finalement, de l'information sur l'état des drones (distances lues par les capteurs, niveau de batterie, hauteur du drone, etc.) ainsi que des logs de déboguage seront aussi disponibles afin de permettre à l'utilisateur de s'assurer du bon fonctionnement du système.

Le serveur aura comme but de gérer les communication entre les différentes composantes du système. Il recevra les informations transmises par les drones et les enverra vers la base de données pour les stocker. Il utilisera ces informations pour produire une carte globale de l'environnement et une carte par drone selon les positions visitées par ceux-ci. Il aura aussi la responsabilité de relayer les commandes reçues de l'interface utilisateur aux drones. Finalement, il se chargera de transmettre les cartes ainsi que l'état des drones à l'interface utilisateur.

Le base de données aura pour but de stocker les informations relatives aux missions et aux logs. Elle recevra des requêtes du serveur pour stocker ou accéder aux données d'une mission selon les besoins de l'utilisateur.

Le logiciel embarqué permettra aux drones de se déplacer de manière indépendante pour explorer l'environnement. Il communiquera avec une station au sol pour transmettre les informations enregistrées par les différents capteurs. Le logiciel embarqué pourra recevoir et réagir aux commandes de la station au sol. Les drones pourront éviter les obstacles présents dans l'environnement. Finalement, les drones devront être en mesure de communiquer entre eux dans un réseau de pairs.

Le simulateur permettra d'émuler le comportement des drones physiques afin de tester le système et de se familiariser avec l'interface utilisateur. Le simulateur devra simuler l'ensemble des actions qui sont effectuées par le logiciel embarqué avec la plus grande fidélité possible.

1.2 Hypothèses et contraintes (Q3.1)

Pour la création et l'utilisation de notre système nous devons poser plusieurs hypothèses qui nous permettrons de simplifier grandement le système. Ces hypothèses sont :

Les drones et le serveur seront toujours connectés. Cela implique donc que lors des communication avec la Crazyradio, les drones seront toujours à une distance raisonnable de celle-ci pour recevoir le message et pour répondre. Ce qui implique aussi que les drones seront à une distance raisonnable l'un de l'autre pour pouvoir communiquer à l'aide d'un réseau de pairs.

L'algorithme de déplacement implémenté permet de couvrir la zone à explorer dans un délai convenable. Ceci veut donc dire que nous supposons que l'algorithme de «random walk» qui permet aux drones de se déplacer aléatoirement leur permettra de visiter l'ensemble de l'espace disponible avec un faible délai.

L'erreur produite par l'estimation de la position d'un drone est négligeable face aux distances parcourues. Puisque la position actuelle du drone n'est pas connue avec certitude, nous posons l'hypothèse que l'erreur produite par l'estimateur est minimale par rapport aux distances parcourues. Il est donc possible de négliger cette erreur et de poser qu'elle est nulle.

Le simulateur utilisé pour tester les fonctionnalités est représentatif de la réalité. Cette hypothèse implique que l'engin de physique est représentatif de la réalité et que le drone subit les mêmes forces qu'en simulation. De plus, ça nécessite que les fonctions utilisées en simulation pour contrôler les drones ont le même comportement que les fonctions utilisées dans le logiciel embarqué. Ça implique aussi que les capteurs de positions et de distances des drones en simulation ont la même précision que ceux présents sur les drones physiques.

L'ensemble du système peut opérer dans un réseau local. Puisque tous les artefacts produits peuvent opérer sur nos machines nous pouvons émettre l'hypothèse qu'une connexion internet n'est pas nécessaire au bon fonctionnement du système.

Un seul opérateur doit être en mesure de contrôler les drones à la fois. La visualisation des informations offertes par les drones n'est pas restreinte à un certain nombre d'utilisateurs, mais un seul d'entre eux doit être en mesure d'interagir avec les drones. Ceci permet donc d'avoir une cohérence au niveau des données envoyées aux drones.

L'opérateur sait comment utiliser le système selon ses besoins. Le système livré sera clair et simple à utiliser, l'hypothèse posée est que l'opérateur sait en quoi consiste l'application et comment celle-ci fonctionne. Il a déjà des connaissances des différentes commandes pouvant être envoyées aux drones et du résultat attendu.

L'opérateur doit appuyer sur le bouton démarrer lors d'une simulation. Puisque le simulateur est fourni par une entité externe, certaines contraintes doivent être respectées. L'une d'entre elle est le fait qu'il est nécessaire d'appuyer sur le bouton démarrer lors d'une simulation pour permettre à celle-ci de fonctionner.

Il existe toujours un chemin qui permet aux drones de parcourir l'ensemble de l'environnement. Puisque l'environnement à visiter n'est pas connu d'avance, une hypothèse est posée qu'il existe toujours un chemin que les drones peuvent suivre pour cartographier l'ensemble de l'environnement.

1.3 Biens livrables du projet (Q4.1)

Au cours de ce projet, trois dates distinctes sont prévues pour produire et livrer des artefacts. La première remise est celle du "Preliminary Design Review" et sera composée d'une conception initiale de la majorité des systèmes du projet. Ce livrable sera remis le 4 février et sera composé des artefacts énumérés ci-après. Une interface utilisateur permettant d'envoyer les commandes "identifier", "démarrer la mission" et "terminer la mission" au serveur. Un serveur permettant le relais des commandes reçues du client vers les drones ou la simulation. Un logiciel embarqué permettant de recevoir la commande "identifier" et d'allumer la DEL du drone physique associé. Un logiciel de simulation permettant de recevoir et de traiter la commande "démarrer la mission" et "terminer la

mission".

La seconde remise est celle du "Critical Design Review" et sera composée d'un système majoritairement fonctionnel. Ce livrable sera remis le 18 mars et sera composé des artefacts suivants : ne interface utilisateur permettant d'afficher l'ensemble des drones avec leurs informations associées, une carte globale de l'environnement exploré et une carte spécifique à chaque drone ainsi qu'un menu permettant d'envoyer des commandes aux drones pour leur indiquer l'état de la mission. Un serveur sera également implémenté, pouvant faire le pont entre les drones, l'interface utilisateur et la base de données. Ce serveur va donc pouvoir recevoir les informations des drones et les stocker dans la base de données, produire des cartes qui seront envoyées à l'interface utilisateur à partir des informations reçus et retransmettre les commandes de l'interface utilisateur vers les drones. De plus, une base de données sera mise en place, permettant de stocker et d'envoyer les logs des drones et les cartes. Un système embarqué sera également développé, permettant aux drones de se déplacer et d'explorer l'environnement, de recevoir les commandes du serveur pour connaître l'état de la mission et d'envoyer leurs informations au serveur afin de permettre à l'opérateur de connaître l'état des drones. Finalement, l'ensemble des requis intégrés dans le logiciel embarqué devront être fonctionnels en simulation pour nous permettre de faire des tests facilement.

La dernière remise est celle du "Readiness Review" et sera composée d'un système fonctionnel à 100%. Ce livrable sera publié le 19 avril et sera composé des artefacts suivants : une interface utilisateur composée de l'ensemble des fonctionnalités du dernier livrable, mais possédant en plus un éditeur de code permettant de modifier le code sur les drones et de les reprogrammer. L'interface utilisateur aura aussi comme fonctionnalité de définir la position initiale des drones et de restreindre le déplacement des drones dans une zone déterminée. Le serveur devra implémenter l'ensemble des fonctionnalités du dernier livrable en plus de pouvoir traiter les nouvelles commandes provenant de l'interface utilisateur et prendre en considération que les drones n'auront pas une position prédéfinie lors de la création de la carte. Le système embarqué sera lui aussi constitué de l'ensemble des fonctionnalités du dernier livrable, mais devra rester dans la zone définie par l'opérateur lors de ses déplacements, il devra pouvoir détecter un crash pour en avertir l'opérateur et il pourra revenir à son point de départ lorsque l'opérateur le demande. Finalement, la simulation implémentera l'ensemble des fonctionnalités du système embarqué. La base de données sera elle aussi présente dans ce livrable, mais n'aura pas eu de nouvelles modification par rapport au dernier livrable.

De plus, il sera nécessaire de produire et remettre à chaque semaine un rapport d'avancement des travaux. Celui-ci sera utilisé pour évaluer l'avancement du projet et permettre à l'agence de faire un suivi du développement. Il sera constitué du travail fait lors de la dernière semaine et du travail prévu pour la semaine suivante. Un document PDF sera aussi produit, détaillant les différents tests d'intégrations qui doivent être effectués sur le système pour s'assurer de son bon fonctionnement.

2 Organisation du projet

2.1 Structure d'organisation (Q6.1)

La structure organisationelle de l'équipe est simple. L'équipe ne possède pas de chef central qui décide chaque élément de la conception de notre système. Toutes les décisions qui affectent l'entièreté du projet sont discutées en équipe. Nous avons commencé par discuter de la technologie que nous allions utilisés pour chaque composante de l'application. Pour faire cela nous avons procédé par un vote après que tous les membres aient proposés leur idées.

Les membres de l'équipe se sont également spécialisés par rapport aux différentes composantes du projet, mais nous allons tous contribuer également aux autres composantes. Paul-André s'est spécialisé dans le développement et la conception de l'interface utilisateur. Florence s'est spécialisée dans la création de la station au sol qui communique avec les drones. Alexis, Gaya et Félix-Antoine se sont spécialisés dans les composantes embarquées du système. Nicholas quant à lui s'est concentré sur la base de données et les technologies de déploiement.

Les tâches ont été réparties équitablement entre les membres de l'équipe et d'une manière permettant que chaque coéquipier puisse travailler sur diverses composantes. Cette répartition des tâches entraînera que chaque membre de l'équipe possédera un bonne compréhension de l'ensemble du projet. Les décisions affectant plusieurs tâches sont prises par l'ensemble des membres qui travaillent sur les tâches en question. Cette méthode de travail assure que les composantes liées fonctionnent toutes ensemble et elle évite les problèmes découlant de la mauvaise communication au sein de l'équipe. Cette structure de projet a également été choisie, car l'équipe a jugé que consulter tous les membres de l'équipe pour chaque décision à prendre serait une grande perte de temps.

L'équipe se rencontre trois fois par semaine et Alexis s'occupe de prendre le compte rendu de ces réunions. Paul-André s'occupe de recueillir les points importants de la semaine qu'il faut aborder pendant les rencontres.

2.2 Entente contractuelle (Q11.1)

Le contrat lié à ce projet est de type livraison clé en main. Effectivement, ce type de contrat est à privilégier dans le cas de projets dont tous les requis sont connus d'avance. De plus, du point de vue de l'Agence Spatiale de Polytechnique, il est plus avantageux de connaître le prix total du projet, ce qui est le cas dans un contrat du type livraison clé en main. Ce prix peut être estimé à l'aide du cahier de charge et d'une estimation du temps de développement nécessaire pour satisfaire chaque requis. Cela permet à l'Agence de s'assurer d'avoir un livrable à la fin du projet. Du côté des développeurs, cela leur permet d'avoir une somme d'argent minimale une fois le projet terminé.

En ce qui concerne la négociation des requis, il ne devrait pas y avoir de problèmes, car l'équipe envisage de tous les compléter. Au niveau fiscal, Les travaux sont également à bas risque pour l'Agence puisque le développement des technologies implémentées n'engendreront pas l'achat de matériel supplémentaire. Il ne devrait donc pas avoir de dépenses additionnelles.

Puisqu'il n'y a aucun profit généré par le projet, un contrat clé en main est préférable à un contrat de partage des économies. Toutes ces raisons permettent de conclure qu'un contrat de livraison clé en main à prix ferme serait plus bénéfique et c'est pourquoi ce type d'entente contractuelle a été choisi.

3 Description de la solution

3.1 Architecture logicielle générale (Q4.5)

Tout d'abord, on peut observer plusieurs requis qui impliquent des choix de conception par rapport à l'architecture du projet. Ces derniers ont été regroupés dans le tableau suivant :

Requis	Description
R.F.2	L'essaim de drones doit répondre aux commandes suivantes, disponibles sur l'interface utilisateur : "Lancer la mission" (décollage et début de la mission) et "Terminer la mission" (atterrissage immédiat)
R.F.3	Pour chaque drone, l'interface utilisateur doit montrer l'état des drones (attente, en mission, etc.), mises à jour avec une fréquence minimale de 1 Hz.
R.F.8	La station au sol doit collecter les données des capteurs de distance des drones et générer des cartes de l'environnement.
R.F.9	Lors d'une mission, la position d'un drone dans la carte doit être affichée en continu.
R.F.10	L'interface utilisateur pour l'opérateur doit être disponible comme service Web et visualisable sur plusieurs types d'appareils (PC, tablette, téléphone) via réseau.
R.F.11	La station au sol doit intégrer les données de tous les drones et créer une seule carte globale de l'environnement exploré.
R.F.13	Le système doit pouvoir détecter un crash de drone. Dans ce cas, l'interface utilisateur doit montrer l'état "Crashed" pour ce drone.
R.F.14	L'interface utilisateur doit permettre la mise à jour du logiciel de contrôle sur les drones.
R.F.16	Un éditeur de code pouvant modifier le code des contrôleurs de drones (en totalité ou en partie) doit être disponible dans l'interface utilisateur pour modifier le comportement des drones avant ou entre les missions.
R.F.17	Une base de données doit être présente sur la station au sol et enregistrer au minimum les attributs suivants pour chaque mission : date et heure de la mission, temps de vol, nombre de drones, physique/simulation et distance totale parcourue par les drones
R.F.18	Les cartes générées lors d'une mission doivent être enregistrées sur la station au sol.
R.F.19	Les drones doivent pouvoir s'entendre sur la couleur d'une ou plusieurs de leur DELs
R.C.1	L'opérateur du système doit pouvoir vérifier que le système fonctionne correctement et avoir les informations nécessaires pour résoudre les problèmes.
R.C.5	Le logiciel complet de la station au sol doit pouvoir être lancé avec une seule commande sur un terminal Linux (ex. docker-compose).

Tableau 1 : Tableau des requis en lien avec l'architecture du logiciel

Les requis **R.F.2**, **R.F.2**, **R.F.13**, **R.C.1** impliquent que l'architecture doit permettre une communication bidirectionnelle entre les drones et l'interface utilisateur. En effet, l'interface utilisateur doit permettre d'envoyer des commandes aux drones ("Lancer la mission", "Terminer la mission", ...) et les drones doivent envoyer leurs positions, leur log de débogage et les lectures de leurs capteurs de distances. De plus, puisque la communication doit se faire en temps réel (1Hz minimum), il est préférable d'utiliser des WebSockets pour une communication efficace. Il serait aussi possible de faire de la scrutation à chaque seconde pour satisfaire le requis. On peut observer les différents schémas de communication possibles entre la station au sol et l'interface utilisateur dans la figure 1 suivante.

Puis, au niveau de la station au sol, les requis **R.F.8**, **R.F.9**, **R.F.11**, **R.F.14** et **R.F.16** indiquent que l'architecture choisie doit permettre une connexion entre la station au sol et les drones. Effectivement, puisque la station au sol doit collecter toutes les données des capteurs de distance des drones et les intégrer pour former une carte individuelle par drone ainsi qu'une carte globale, il est nécessaire pour la station au sol d'avoir une connexion unique avec chaque drone (et non un *broadcast*). Cette connexion doit être bidirectionnelle, car la station au sol doit, en plus de recevoir l'information des drones, relayer des commandes aux drones et aussi permettre de les reprogrammer (en fonction

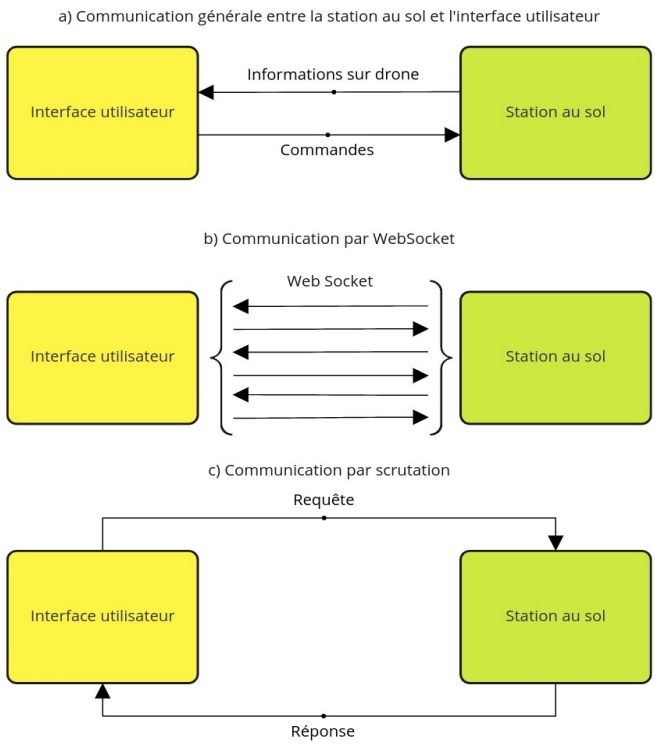

Figure 1 : Communication entre la station au sol et l'interface utilisateur

d'un code écrit sur l'interface utilisateur). Cette communication se fera à l'aide de la Crazyradio qui permettra d'envoyer des données aux drones et de les programmer à distance avec l'API fournie par la compagnie Bitcraze. Le tout est présenté dans la figure suivante : 2.

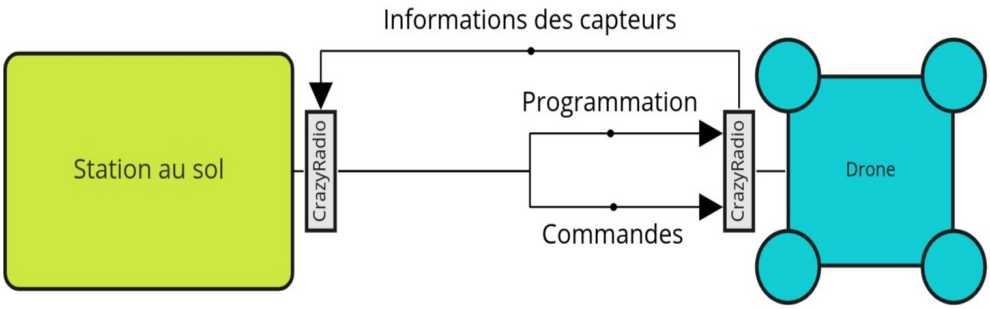

Figure 2 : Communication entre la station au sol et un drone

En ce qui concerne la communication des drones, le requis **R.F.19** impose une contrainte de communication entre chacun des drones pour permettre aux drones de s'entendre sur leur couleur. Cela se fera à l'aide d'une communication sans fil *Peer to Peer (P2P)*. Comme indiqué dans la figure 3, il y aura une connexion P2P entre tous les drones.

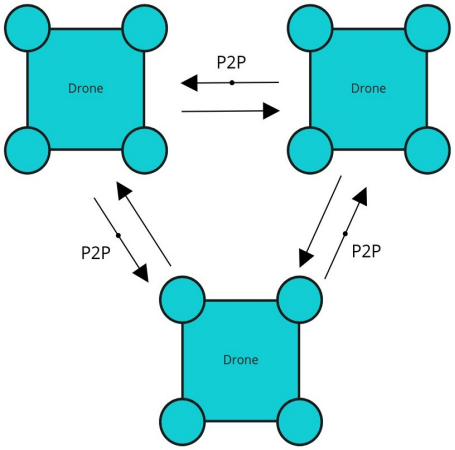

Figure 3 : Communication entre plusieurs drones

Aussi, le requis **R.F.10** implique que l'architecture doit pouvoir supporter plusieurs clients qui se connectent au serveur en même temps. De plus, l'interface utilisateur doit être portable sur plusieurs appareils (PC, tablettes, téléphone). Ainsi, l'utilisation de cadriciels externes comme Angular, React et Vue sont conseillés, car cela facilite la création d'une application Web qui réagit bien sur tous les types d'appareils.

Au niveau des données prises par les drones, les requis **R.F.17** et **R.F.18** énoncent la présence d'une base de données qui doit enregistrer les missions des drones. Ainsi, l'architecture finale du logiciel doit inclure un système de stockage permettant aux données d'être conservées lorsque le système est éteint. Cet objectif peut être atteint à l'aide d'une base de données locale sur la station au sol ou à l'aide d'une base de données externe. Ainsi, il faut ajouter une communication entre la station au sol et une base de données quelconque.

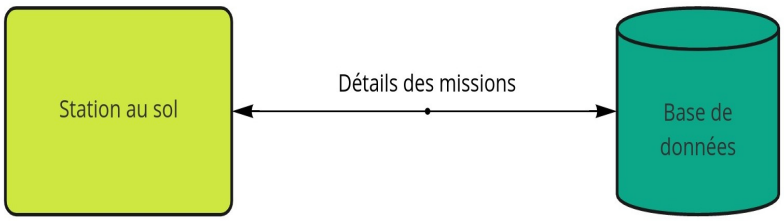

Figure 4 : Communication entre la station au sol et la base de données

Finalement, le requis **R.C.5** indique que le système doit fonctionner pour un nombre arbitraire de drones entre 2 et 10. Combiné au requis **R.F.10**, on peut constater que l'architecture du système doit être évolutive à deux niveaux : les clients et les drones. En effet, cette dernière doit supporter autant de clients que possible et jusqu'à un maximum de 10 drones.

Finalement, l'architecture globale du système, incluant le flot des données, est présentée dans la figure 5.

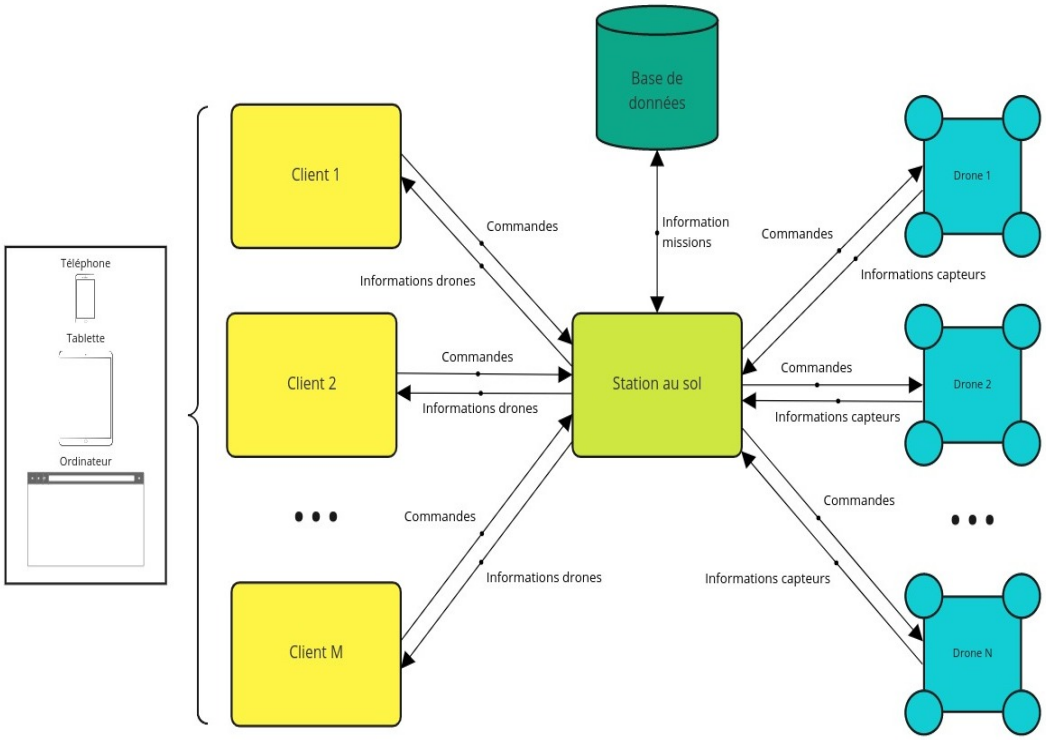

Figure 5 : Architecture globale du système

3.2 Station au sol (Q4.5)

Requis	Description
R.M.2	Le seul moyen de communication entre la station au sol et les drones physiques doit être une Bitcrazy Crazyradio PA connectée à la station au sol.
R.M.4	La station au sol doit être un laptop ou PC.
R.L.3	La station au sol doit envoyer des commandes de haut niveau (Lancer la mission, Mise-à-jour, etc.) et les informations associées.
R.F.8	La station au sol doit collecter les données des capteurs de distance des drones et générer des cartes de l'environnement. Il doit y avoir une carte individuelle pour chaque drone.
R.F.11	La station au sol doit intégrer les données de tous les drones et créer une seule carte globale de l'environnement exploré.
R.F.12	La position et l'orientation initiale respective des drones (relative ou absolue selon les besoins du système) dans l'environnement (physique ou simulé) peuvent être déterminées automatiquement par la station au sol.
R.F.17	Une base de données doit être présente sur la station au sol et enregistrer au minimum les attributs suivants pour chaque mission : date et heure de la mission, temps de vol, nombre de drones, physique/simulation et distance totale parcourue par les drones.
R.F.18	Les cartes générées lors d'une mission doivent être enregistrée sur la station au sol.
R.C.1	Les logs de débogages doivent être disponibles en continu lors de chaque mission. Ces logs comprennent toutes les informations de débogage nécessaires au développement, dont au minimum : les lectures des senseurs (distance et position) de chaque drone (à minimum de 1 Hz) ainsi que les commandes envoyées par la station au sol. Ces logs doivent être sauvegardés sur la station au sol à la fin de chaque mission.
R.C.2	Le logiciel complet de la station au sol doit pouvoir être lancé avec une seule commande sur un terminal Linux (ex. docker-compose).
R.C.5	Le système doit être conçu pour fonctionner avec un nombre arbitraire de drones, entre 2 et 10 drones. La station au sol et l'interface utilisateur doivent s'adapter automatiquement et se connecter à tous les drones disponibles pour la mission, sans que l'utilisateur n'est à en spécifier le nombre.

Tableau 2 : Tableau des requis en lien avec la station au sol

La station au sol héberge le serveur responsable de faire le lien entre l'application web, la communication avec les drones, la communication avec la simulation et la cartographie de l'environnement.

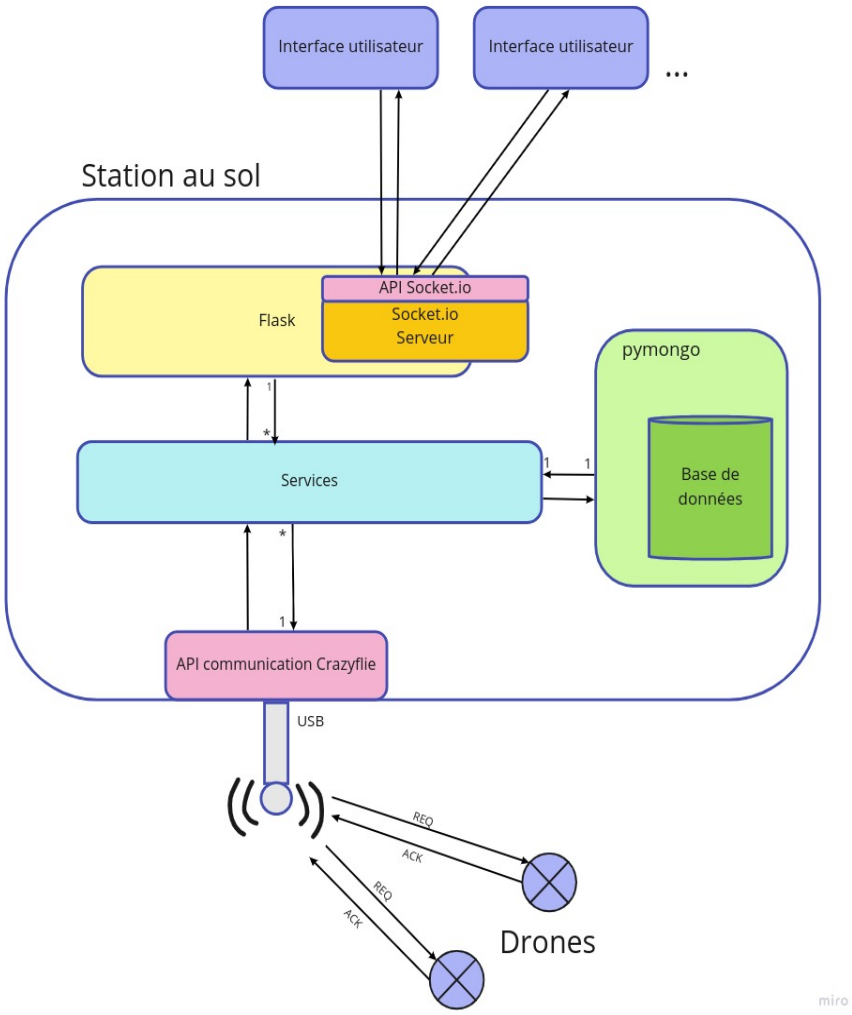

Figure 6 : Architecture station au sol

Pour ce qui est du langage utilisé dans le serveur, l'équipe a hésité entre les langages TypeScript et Python. TypeScript est intéressant pour le côté serveur du fait qu'utilisé conjointement avec Node.js et Express.js, ce regroupement de technologies permet le développement simple et rapide d'applications. De plus, tous les membres de l'équipe étaient déjà familiers avec TypeScript. Cependant, il semble plus avantageux d'utiliser Python pour ce projet puisque plusieurs algorithmes de création et de gestion de cartes sont déjà implémentés en Python et l'API de communication avec les Crazyflies est en Python. De plus, Python est un langage reconnu pour sa facilité d'apprentissage. [3] Le choix s'est donc arrêté sur le langage Python, principalement pour sa simplicité d'usage, pour les algorithmes de cartographie dont il dispose déjà en exemple ainsi que pour l'API de communication avec les Crazyflies.

Après s'être arrêté sur Python comme choix de langage pour le développement du serveur, il s'ensuit le choix du cadriciel à utiliser pour la conception de ce dernier. Parmi les différents choix de cadriciel pour le langage Python se trouvent : Django, FastAPI et Flask.

Quoique le cadriciel Django soit réputé pour son efficacité de développement et son usage répandu, un désavantage apparant est le fait que Django est en train de devenir obsolète.[2] Quant à FastAPI, ce dernier est un cadriciel reconnu pour sa rapidité puisqu'il supporte du code concurrent et asynchrone.[5] Contrairement à FastAPI, Flask permet seulement le traitement séquentiel des requêtes. FastAPI étant un cadriciel plus récent que Flask, il ne possède pas autant de documentation et d'exemples. Puisque l'affichage des cartes doit se faire en temps réel sur le client, avoir l'option d'utiliser des WebSockets en est une souhaitable. C'est pour cette raison que le choix du cadriciel pour le serveur s'est arrêté sur Flask, du fait que ce dernier permet l'utilisation facile de WebSockets.

Suite au choix du langage et du cadriciel pour le serveur, découle les choix des différentes méthodes de communication dont disposera la station au sol, ainsi que l'architecture des services offerts sur celle-ci.

Lors du choix de la communication entre le serveur et l'interface utilisateur, il est important de considérer que la communication doit se faire en temps réel. En effet, l'interface utilisateur doit pouvoir voir la position actuelle des drones à une fréquence minimale de 1Hz selon le requis **R.C.1**. C'est pour cette raison que le module *Socket.io* a été choisi pour la communication entre le serveur et le client puisqu'il fournit une implémentation de WebSockets en Python. Les WebSockets ont été privilégiés face à une approche par scrutation pour leur efficacité et vitesse de communication plus élevées (*polling*). Cette communication est détaillée d'avantage dans la section 3.1.

La librairie *Flask-SocketIO* a été choisie pour initier la communication entre le serveur et l'application client. Cet outil permet la création d'un centre de contrôle SocketIO pour l'application serveur utilisant le cadriciel Flask. *Flask-SocketIO* supporte une communication bidirectionnelle à faible latence entre les clients et notre serveur.

La communication entre un Crazyflie et la station au sol est un protocole avec plusieurs couches.

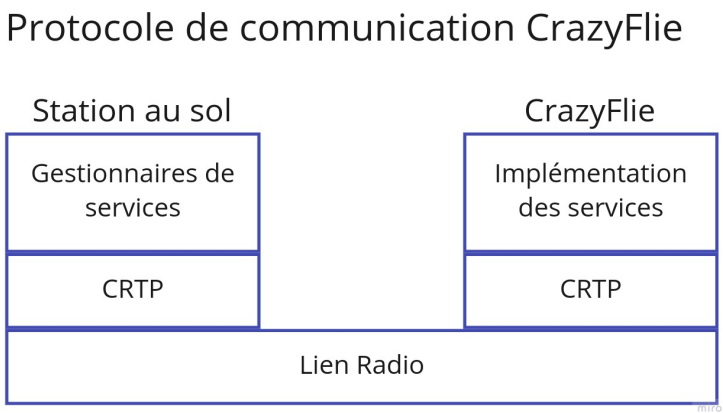

Figure 7 : Protocole de communication Crazyradio et Crazyflie

La communication entre la station au sol et les drones se fait par l'intermédiaire de la Crazyradio. Celle-ci utilise le protocole ESB pour envoyer des paquets et recevoir l'accusé de réception. Les ser-

vices mis en évidence sur le diagramme ci-dessus sont des fonctionnalités de haut niveau permettant d'aller chercher des données du Crazyflie à des intervalles réguliers. CRTP est un protocole qui encapsule les commandes des différents services. Finalement, le paquet est envoyé sur une chaîne radio, laquelle est identifiable par une adresse (sur 5 octets) et quand le paquet est reçu par le destinataire, ce dernier renvoit un accusé de réception destiné à la même adresse de la chaîne radio ayant effectuée l'envoie. Les deux communications peuvent contenir des données.[**Comm-Crazyflie**]

La communication entre la station au sol et le Crazyflie est bidirectionnelle. La station au sol est celle qui initie la conversation via la Crazyradio et les Crazyflies sont constamment à l'écoute pour un paquet. Un désavantage de ce type de communication est que le Crazyflie ne peut initier de communication. C'est donc pour cette raison que la communication se fait par scrutation. Les paquets seront donc envoyés à intervalles réguliers.

Multiples services seront offerts et résideront sur le serveur de la station au sol. Ces services permettront de traiter et gérer les différentes requêtes et réponses entre le client et les Crazyflies ainsi que de générer les cartes de l'environnement.

Le gestionnaire de cartes est le service dans lequel sera implémenté l'algorithme de cartographie. Ce dernier sera appelé par les gestionnaires de communication à chaque réception de nouvelles données envoyées par les Crazyflies.

Après avoir généré la nouvelle carte à jour, ces données seront transmises et emmagasinées dans la mémoire locale. Cette mémoire locale, dont le but est d'imiter le rôle d'une RAM, permettra d'accéder rapidement aux données les plus récentes transmises par les drones et à l'état actualisé des cartes. Pour éviter d'engorger cette mémoire locale dont la capacité est plus faible que la base de données, les données seront transférées sur celle-ci après un certain intervalle de temps. La mémoire locale du serveur sera alors vide et prête à recevoir des nouvelles données des drones. Ce choix de conception permettra de limiter la communication entre le serveur et la base de données du fait que ça évite d'envoyer des données après chaque réception. Par exemple, lorsque les drones envoient les données recueillies par leurs senseurs, ces nouvelles données seront sauvegardées sur le serveur et pourront être facilement accessibles pour les clients. Nous posons ainsi l'hypothèse que les clients demanderont plus souvent les données les plus récentes que les données d'anciennes mission afin de justifier ce choix de conception. De plus, puisque l'affichage des cartes de la mission en cours se fait en temps réel, il sera plus rapide d'obtenir les données directement de la mémoire du serveur afin de permettre un fréquence minimale de 1Hz.

Concernant la gestion des données sur le serveur, ce service est ajouté suivant les principes du patron médiateur. Ce dernier aura comme responsabilité de rediriger les données aux bons services. Son but est de diminuer les dépendances entre chaque service. Lorsqu'il reçoit des données du gestionnaire de communication Crazyflie, il les redirige selon leur traitement spécifique. Par exemple, si ce sont des données pertinentes à la cartographie de l'environnement, le gestionnaire de données redirige ces données vers le générateur de cartes, etc. De plus, ce gestionnaire de données s'occupe de gérer l'intervalle de temps dans lequel les données sont enregistrées sur le serveur. Pour éviter d'avoir une quantité trop élevée de données enregistrées sur le serveur, après un certain intervalle de temps, le gestionnaire de données va vider la mémoire sur le serveur et transférer son contenu

sur la base de données.

De plus, un gestionnaire de communication entre la station au sol et le Crazyflie sera présent sur la station au sol. Ce service s'occupera d'envoyer les requêtes aux Crazyflies à une fréquence minimale de 1Hz pour déterminer le statut de ces derniers. Il se chargera ensuite de recevoir les accusés de réception et d'acheminer les données reçues au gestionnaire de données. Le gestionnaire de communication utilisera l'API *cflib* de *bitcraze*, permettant la communication avec les Crazyflies.

Les requis **R.F.17** et **R.F.18** indiquent qu'une base de données doit enregistrer l'information sur les missions et les cartes générées. Pour répondre à ces requis, le choix de la base de données s'est arrêté sur MongoDB. Les deux grands types de bases de données sont les bases de données relationnels et les bases de données non-relationnelles.[4] Le choix d'une base de données non-relationnelle est favorable du fait que cela permet que le projet soit extensible et qu'il est possible de modifier les paramètres enregistrés sur la base de données rapidement. Les bases de données non-relationnelles sont plus extensibles puisque chaque objet présent dans la base de donnée est unique et séparé, ce qui n'est pas le cas dans une base de données relationnelle. Si une base de données relationnelle est utilisée, il est possible qu'un imprévu force à enregistrer de l'information supplémentaire. Il faudra alors modifier l'architecture au complet, ce qui peut s'avérer très long. Il est souvent complexe de concevoir une architecture de données dans une base de données relationnelle, car il faut minimiser le couplage et la redondance des données. Les données sauvegardées seront simplement celles des anciennes missions. Elles ne sont pas liées entre elles et ne bénéficierons alors pas des caractéristiques d'une base de données relationnelle.[1]

En ce qui concerne les bases de données relationnelles, il existe d'innombrables sous-types de celles-ci. Nous n'avons pas pris une base de données basée sur des graphes tel que Neo4j, car elle présente les mêmes problèmes que les bases de données relationnelles. La base de données dont les objets sont enregistrés comme des documents est le choix le plus approprié dans le contexte de ce projet, car les objets enregistrés seront tous uniques et ne possèdent pas de liens entre eux.

Finalement, pour suivre à la lettre le requis **R.F.17**, une base de données locale sera utilisée. Cette décision écarte donc MongoDB Atlas des choix, une base de données hébergée sur le nuage. De plus, DockerHub possède déjà plusieurs conteneurs Docker de bases de données MongoDB. Il existe également plusieurs librairies Python pour communiquer avec une base de données MongoDB. La figure 8 montre un exemple d'outil de communication possible.

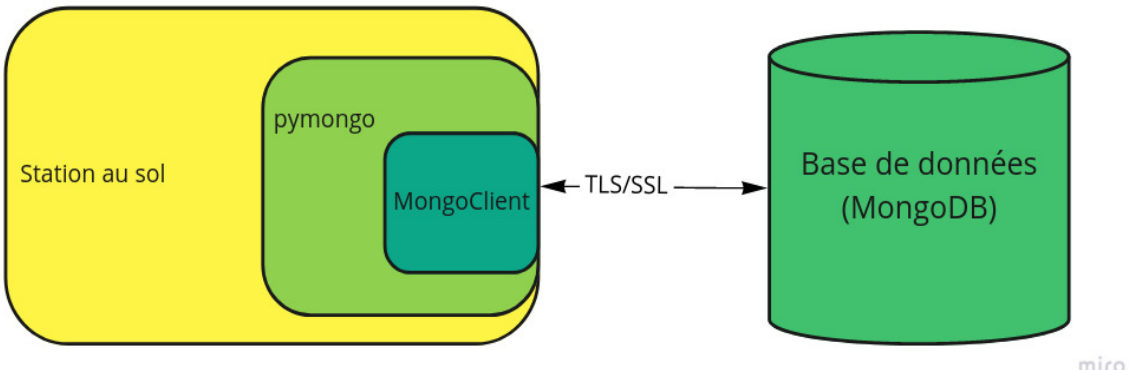

Figure 8 : Communication entre la station au sol et la base de données

3.3 Logiciel embarqué (Q4.5)

Requis	Description
R.M.1	Le prototype doit être implémenté avec deux drones Bitcraze Crazyflie 2.
R.M.2	Communication avec la station au sol au travers de la BitCrazyradio PA
R.M.3	Seul le ranging deck et l'optical flow deck doivent être utilisés
R.L.1	Les drones doivent être programmés en utilisant l'API de BitCraze
R.L.3	Le controle des drones doit se faire à bord de ceux-ci sur le code embarqué
R.F.1	Chaque drone physique doit répondre à la commande "identifier"
R.F.2	L'essaim doit répondre aux commandes : "Lancer la mission" et "Terminer la mission"
R.F.4	Les drones doivent explorer l'environnement de façon autonome
R.F.5	Les drones doivent éviter les obstacles détectés par leurs capteurs de distance
R.F.6	Le retour à la base doit rapprocher les drones de leur position de départ (< 1m)
R.F.7	Les drones doivent revenir à la base automatiquement lorsque le niveau de batterie < 30%
R.F.8	Les drones doivent envoyer les informations de leur capteur à la station au sol.
R.F.14	Le drone doit pouvoir être mis à jour lors de la réception d'un paquet binaire
R.F.19	Les drones doivent pouvoir s'entendre sur la couleur d'une DEL selon la distance avec leur point de départ (P2P)
R.C.1	Le drone doit pouvoir envoyer des informations de déboguage à la station au sol

Tableau 3 : Tableau des requis en lien avec le logiciel embarqué

Pour le développement du code embarqué, une API est fournie par la compagnie *Bitcraze* permettant d'intégrer le logiciel facilement sur les drones. Puisque La partie de l'interface embarquée a été programmée en C, cela implique qu'un langage compatible doit être utilisé lors du développement de ce sous-système pour faciliter la création d'un exécutable. Cete contrainte restreint donc les choix de langages à C, C++ ou BUZZ. Après analyse, le langage C++ a été choisie, car malgré les modifications nécessaires à l'API fournie par la compagnie pour supporter la compilation de ce langage, il permet une plus grande flexibilité que le langage C et possède un plus grand support que BUZZ.

En analysant les différents requis fournis, il est possible d'extraire plusieurs requis reliés au logiciel embarqué. Par exemple, les requis **RM.2**, **RF.19** et **RF.20** impliquent de pouvoir communiquer avec la station au sol au travers la Crazyradio et avec les autres drones à l'aide d'une communication dans un réseau de pairs. Une classe de gestion des communications a donc été ajoutée pour faciliter l'implémentation de ces communications. En effet, celle-ci aura pour but de simplifier la création et la réception de paquet et de rendre les communications simples et efficaces. Elle se chargera donc de réceptionner les paquets envoyés par la radio, de retransmettre ceux-ci au bon module pour qu'ils soient traités et de renvoyer une réponse à la radio pour lui indiquer si les paquets ont été traités avec succès ou non.

De plus, selon les requis **RF.1** et **RF.2**, le drone doit être en mesure de recevoir et de traiter des commandes de la station au sol. Ainsi une classe chargée de recevoir les commandes et d'effectuer le traitement associées à celles-ci devra être implémentée. Cette classe aura pour but de centraliser le code associé aux différentes commandes possibles et de s'assurer que le traitement de celles-ci se fasse de manière uniforme. Elle va donc recevoir les paquets récupérés par le module de communication et effectuer par la suite les actions associées aux commandes en utilisant les bons modules.

Puis, les requis **RL.3**, **RF.4**, **RF.5**, **RF.7** et **RF.20** indiquent que les drones doivent être en mesure

de se déplacer de manière autonome selon les paramètres définis par un opérateur. L'algorithme "random walk" a donc été choisi pour permettre aux drones d'explorer le périmètre voulu de manière convenable. Une classe sera responsable de gérer la navigation du drone. Cettre dernière prendra en entrée les valeurs des capteurs et les commandes de la station au sol et produira la direction finale du drone. Cet algorithme s'assurera donc que les drones n'entrent pas en collision et restent dans le périmètre défini par l'opérateur avant le début de la mission.

Les requis **RM.3**, **RF.5**, **RF.7**, **RF.8** et **RF.19** impliquent que les drones doivent pouvoir utiliser de nombreux capteurs pour pouvoir interagir avec le monde extérieur. L'utilisation de ces capteurs sera donc regroupée dans une classe permettant de s'assurer que l'ensemble du code associé à ceux-ci soit regroupé aux mêmes endroits et accessible de manière uniforme. La classe se chargera par la suite d'envoyer les valeurs lues vers le système de communication pour que la station au sol ait accès aux données et vers le système de navigation pour que le robot puisse se déplacer tout en évitant les obstacles et en restant dans la zone définie par l'opérateur.

Finalement, pour faciliter les tests, un contrôleur universel sera implémenté. Ce dernier permettra de réutiliser le même code que les drones en simulation. Il va donc définir une interface donnant une liste de fonctions permettant d'interagir avec le drone. L'implémentation de ces fonctions changera par la suite en fonction d'un paramètre de compilation qui permettra de déterminer si la version utilisant la simulation sera utilisée ou pas. Le code regroupant les deux environnements (simulation et réel) ne sera donc pas écrit deux fois.

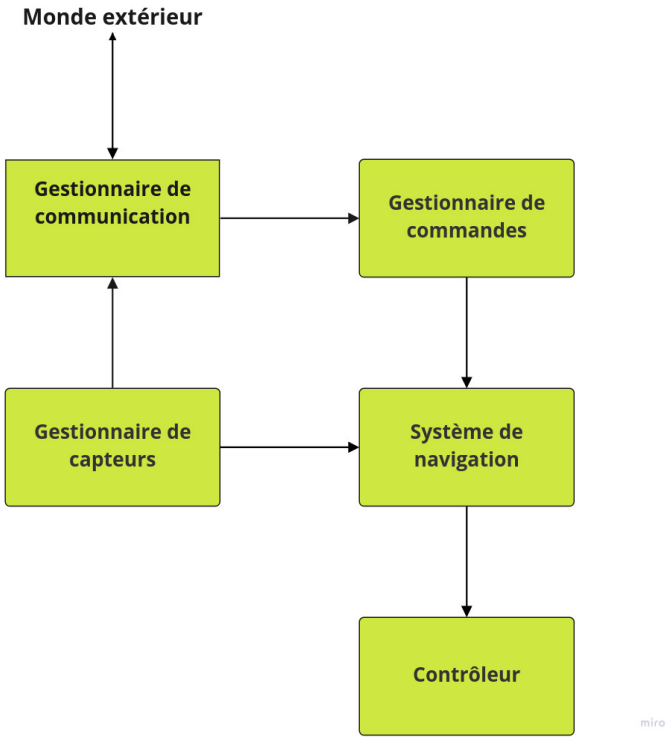

Figure 9 : Architecture de l'interaction des différentes classes du code embarqué

3.4 Simulation (Q4.5)

Les requis en lien avec la simulation sont présentés dans le tableau ci-dessous :

Requis	Description
R.L.2	Interface utilisateur identique à celle pour les vrais drones
R.L.4	Simulation dans un conteneur avec Docker
R.F.7	Simulation du niveau de batterie des drones
R.F.11	Récupérer la position initiale des drones en simulation
R.F.14	Permettre la mise a jour du code embarqué des drones dans la simulation
R.F.17	Transmettre les données de la simulation : date et heure de la mission, temps de vol, nombre de drones, physique/simulation et distance totale parcourue par les drones, etc...
R.C.1	Capture et transmission de logs de débogage
R.C.2	Lancement de la simulation/serveur/UI avec une seule commande (Docker-compose)
R.C.3	Environnement virtuel de simulation générer aléatoirement (voir fichier de configuration du simulateur)
R.C.5	Simulation avec de 2 à 10 drones.
R.Q.2	Procédure de test ou test unitaire pour la simulation

Tableau 4 : Tableau des requis en lien avec la simulation

Une composante importante du projet est la simulation de l'environnement et des drones avec le programme ARGOS3. En effet, l'utilisation d'une simulation est primordiale pour le projet, car elle permettra de tester le code embarqué rapidement tout en évitant des bris sur les drones physiques. Celle-ci pourra aussi fournir des données précises de position et de distance pour chacun des drones, permettant à l'équipe de mieux peaufiner les algorithmes d'exploration et de création de cartes. Enfin, avec son interface graphique et la possibilité de visualiser la simulation étape par étape, il sera plus facile d'observer et de déboguer d'éventuels problèmes dans notre algorithme.

Ainsi, l'outil de simulation devra donc permettre plusieurs tâches afin que celui-ci nous soit le plus utile possible durant le projet. Tout d'abord, il est nécessaire que le code embarqué, c'est-à-dire la logique permettant le déplacement et la collecte de données sur chaque drone, soit le même pour la simulation que pour les drones physiques. Cette transparence entre la simulation et les drones physiques en ce qui concerne le code embarqué permettra un développement plus facile et plus rapide. En effet, en évitant ainsi une duplication non nécessaire du code, les chances d'obtenir un comportement différent en simulation et dans le monde réel diminueront grandement. De plus, centraliser ainsi le code embarqué permettra de faciliter l'implémentation du requis **R.F.14**, puisqu'il ne faudra faire des modifications qu'à un seul endroit pour affecter le comportement des drones en simulation et dans le monde réel.

La transparence entre la simulation et les drones physiques entraine plusieurs complications lors du développement du code embarqué. Premièrement, les différentes méthodes permettant le déplacement, l'acquisition de données et le contrôle des drones dans la simulation sont différentes de l'API fourni par Bitcraze pour le contrôle des drones physique. Ensuite, la communication entre les drones de la simulation et la station au sol ne passe pas par la radio Crazyflie comme c'est le cas avec les drones physiques. Il faudra donc établir une communication entre les drones de la simulation et la station au sol afin d'obtenir de l'information des drones telle que le demande les requis **R.L.11**, **R.L.17** et **R.C.1**

Afin de pallier le premier problème, c'est-à-dire la différence entre l'API des drones physiques et l'API de la simulation, des interfaces (classe abstraite) seront définies au niveau du code embarqué qui aura ensuite des implémentations différentes pour la simulation ou pour les drones physiques. De cette manière, lors de la programmation au niveau du code embarqué, il sera possible d'utiliser les méthodes fournies par ces interfaces afin de contrôler le drone sans se soucier si celui-ci sera un drone physique ou un drone simulé. Les figures suivantes illustrent cette architecture pour le module de gestion des capteurs ainsi que le module de contrôle du drone.

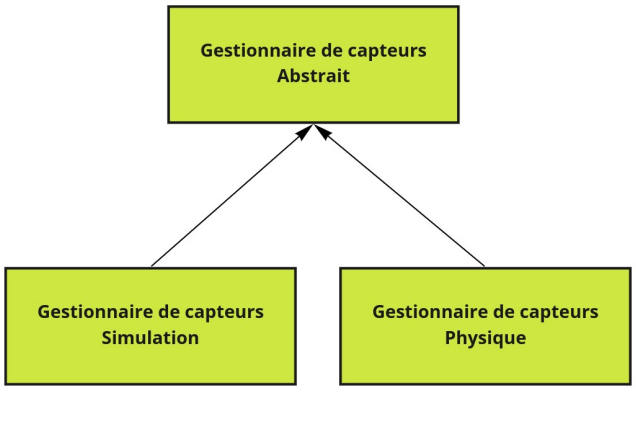

Figure 10 : Abstraction du module de gestion des capteurs

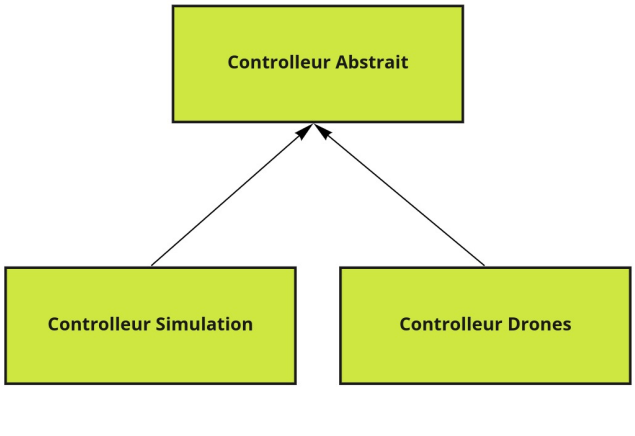

Figure 11 : Abstraction du module de contrôle

Ensuite, afin de résoudre le second problème, c'est-à-dire la communication entre la station au sol et la simulation, nous allons utiliser une communication par socket UNIX. Ce type de communication permet à deux processus situés sur un même ordinateur de communiquer entre eux à l'aide d'un fichier situé sur celui-ci. Dans notre cas nous utiliserons donc un UNIX socket afin de permettre au processus de la simulation de communiquer avec le processus de la station au sol. La figure suivante présente un diagramme simplifié de la communication entre deux processus au travers d'un socket UNIX.

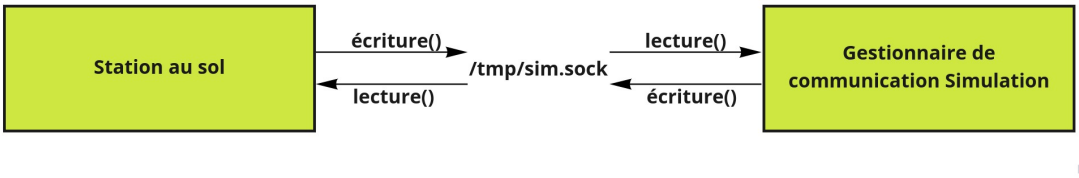

Figure 12 : Communication interprocessus avec un socket UNIX

Le choix d'utilisation d'un socket au lieu d'un simple *pipe* est dû à la flexibilité additionnelle que permettent ceux-ci. En effet, un socket permet une communication bidirectionnelle, ce qui est nécessaire dans notre cas afin de pouvoir envoyer des commandes aux drones et recevoir de l'information de ceux-ci. De plus, un socket UNIX permet la communication entre deux processus, ce qui pourrait être pertinent dans l'éventualité où chaque drone de la simulation pourrait être représenté par un processus distinct. Nous aurions aussi pu utiliser un WebSocket pour la communication. Cependant, puisque les deux processus se trouvent sur la même machine, utiliser la pile TCP pour une communication réseau ajoute de la complexité et de la latence inutile.

Afin d'utiliser les sockets UNIX avec la simulation et la station au sol, nous allons avoir recours à deux librairies qui implémentent déjà plusieurs fonctionnalités des sockets. Ceci devrait grandement simplifier la tâche de développement en ce qui concerne la communication entre la station au sol et la simulation. Du côté de la simulation, nous utiliserons les sockets UNIX de la librairie BOOST pour C++ et le module socket pour Python.

Encore une fois, afin de permettre un code embarqué découplé de la simulation ou des drones physiques, nous allons utiliser une interface (classe abstraite) pour le gestionnaire de communication et laisser les implémentations spécifiques à l'utilisation de la Crazyflie radio ou des sockets UNIX à des sous-classes. L'architecture du module de gestion des communications peut être observée dans la figure 13 ci-dessous.

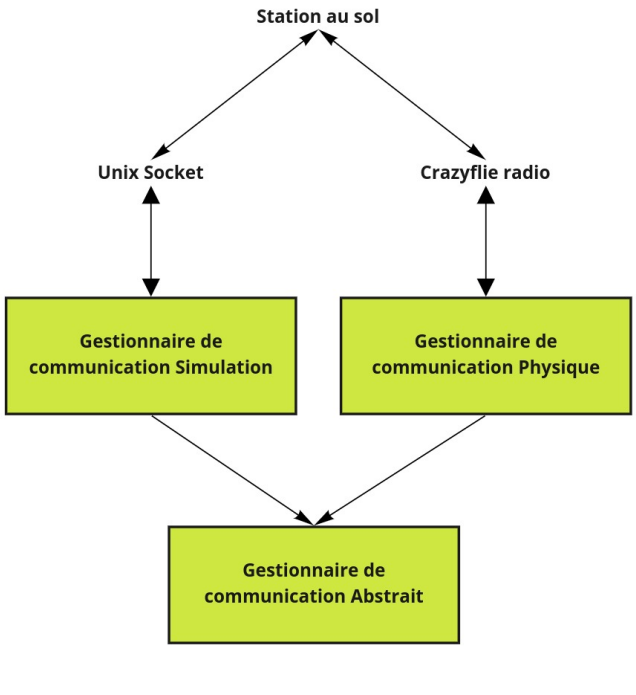

Figure 13 : Abstraction du module de communication

Ainsi, lorsque l'on assemble toutes les abstractions nécessaires au bon fonctionnement du code embarqué pour les drones physique et pour la simulation. Nous obtenons l'architecture générale de notre logique embarquée telle que présentée dans la figure 14.

Un autre aspect important de la simulation est notre capacité à faire varier les différents paramètres de celle-ci. En effet, il est important de pouvoir faire varier la taille de l'arène ainsi que les obstacles qui s'y trouvent comme le mentionne le requis **R.C.3**. De plus, plusieurs drones (2 à 10) doivent pouvoir être supportés. Il serait donc pertinent d'aussi pouvoir faire varier le nombre de drones dans la simulation. Enfin, il doit être possible de spécifier une position initiale pour les drones dans la simulation selon le requis **R.F.11**. Il sera possible de réaliser ces requis en utilisant une simple routine sur la station au sol qui sera chargée de lire le fichier de configuration de la simulation (.argos) et de modifier ce dernier selon les informations obtenues par l'interface utilisateur.

Enfin, il est nécessaire selon le requis **R.L.4** que l'entièreté de notre application soit contenue à l'intérieur d'un conteneur Docker. Puisque l'application de simulation possède une interface graphique, il faut modifier la configuration de l'environnement afin de permettre le transfert de l'affichage X11 entre la machine hôte (notre ordinateur) et le client (le conteneur docker). Ces modifications permettront à la simulation de s'exécuter sur n'importe quelle machine Linux grâce à l'utilisation des conteneurs Docker.

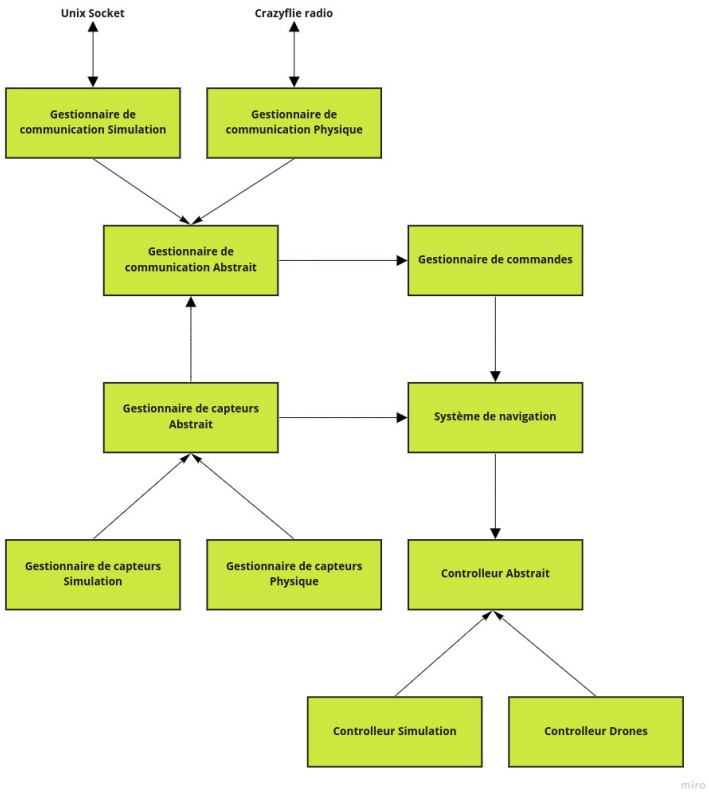

Figure 14 : Architecture générale du code embarqué

3.5 Interface Utilisateur (Q4.6)

Une des décisions prise pour la gestion de l'interface utilisateur est d'utiliser le principe de l'architecture en couches quant à la communication avec les autres services [6, p. 57]. En d'autres mots, l'interface devra aller chercher et communiquer elle-même des informations sur le serveur, sans que le serveur initie lui-même la communication.

Pour implémenter l'interface utilisateur, l'équipe a décidé d'utiliser Vue.js et Typescript. Vue.js a été choisi car ce cadriciel permet le développement d'applications web qui sont plus légères, ce qui correspond au projet actuel. Angular était initialement un choix de l'équipe à cause de notre familiarité avec ce dernier. Cependant, étant donné que ce cadriciel s'applique pour le développement d'applications plus complexes et alors implique de mettre plus d'efforts dans la conception de l'interface utilisateur, il a été décidé d'explorer d'autres options. À partir de ce point, il fallait choisir entre Vue.js et React, deux cadriciels avec lesquels l'équipe n'était pas très familière. Une des raisons principales pourquoi React n'a pas été choisi est que sa courbe d'apprentissage peut être assez escarpée comparativement à Vue.js. Finalement, afin d'assurer une qualité élevée du code pour l'interface utilisateur, Typescript a été choisi. En effet, puisque ce langage impose de déclarer explicitement les divers attributs utilisés, il sera alors possible de bénéficier d'une meilleure compréhension de la logique implémentée. De plus, puisque l'application doit fonctionner sur divers appareils selon le requis **R.F.10**, une attention particulière sera apportée quant à la réalisation des fonctionnalités de navigation. La liste des requis associés à cette architecture peut être retrouvée dans le tableau 5.

Requis	Description
R.L.2	L'interface utilisateur doit être la même lorsque la station au sol est connectée à la simulation ou aux drones physiques. À l'exception des fonctionnalités spécifiques à un des deux systèmes.
R.F.1	Chaque drone physique doit individuellement répondre à la commande "Identifier" disponible dans l'interface utilisateur
R.F.2	L'essaim de drones doit répondre aux commandes suivantes, disponibles sur l'interface utilisateur : — "Lancer la mission" : décollage et début de la mission — "Terminer la mission" : atterrissage immédiat.
R.F.3	Pour chaque drone, l'interface utilisateur doit montrer l'état des drones (attente, en mission, etc.), mises à jour avec une fréquence minimale de 1 Hz.
R.F.6	Une commande de "Retour à la base" doit être disponible sur l'interface utilisateur.
R.F.7	Le niveau de batterie de chaque drone doit être affiché sur l'interface utilisateur.
R.F.8	Il doit y avoir une carte individuelle pour chaque drone. Ces cartes doivent être affichées en continu lors de la mission (une ou plusieurs à la fois) dans l'interface utilisateur.
R.F.9	Lors d'une mission, la position d'un drone dans la carte doit être affichée en continu.
R.F.10	L'interface utilisateur pour l'opérateur doit être disponible comme service Web et visualisable sur plusieurs types d'appareils (PC, tablette, téléphone) via réseau.
R.F.11	La station au sol doit intégrer les données de tous les drones et créer une seule carte globale de l'environnement exploré.
R.F.12	La position et l'orientation initiale respective des drones (relative ou absolue selon les besoins du système) dans l'environnement (physique ou simulé) doit pouvoir être spécifiés par l'opérateur dans l'interface utilisateur avant le début de la mission.
R.F.13	L'interface utilisateur doit montrer l'état "Crashed" pour ce drone.
R.F.14	L'interface utilisateur doit permettre la mise à jour du logiciel de contrôle sur les drones.
R.F.16	Un éditeur de code pouvant modifier le code des contrôleurs de drones (en totalité ou en partie) doit être disponible dans l'interface utilisateur pour modifier le comportement des drones avant ou entre les missions.
R.F.17	Une base de données doit être présente. L'interface utilisateur doit en afficher les informations à l'écran pour consultation.
R.F.18	Il doit être possible à partir de l'interface utilisateur d'ouvrir une carte générée dans une mission précédente pour l'inspecter de nouveau.
R.F.20	Dans l'interface utilisateur, l'opérateur doit pouvoir spécifier une zone de sécurité (geofence) de forme rectangulaire dans la carte d'une aire minimale de 4 m2.
R.C.1	L'opérateur du système doit pouvoir vérifier que le système fonctionne correctement et avoir les informations nécessaires pour résoudre les problèmes. À cet effet, des logs de débogages doivent être disponibles en continu lors de chaque mission.
R.C.4	L'interface utilisateur doit être facile d'utilisation et lisible.

Tableau 5 : Tableau des requis en lien avec l'architecture de l'interface utilisateur

Avant de passer à une description plus détaillée du fonctionnement interne de l'interface utilisateur, il est important d'illustrer son fonctionnement général qui est illustré à la figure 15.

Lorsque l'utilisateur se connecte à l'application web, il va se retrouver avec quatre choix concernant la gestion des missions. En effet, un des premiers choix que l'utilisateur pourra faire sera d'observer ou de démarrer une mission. Les autres choix qui sont proposés à l'utilisateur n'ont pas de

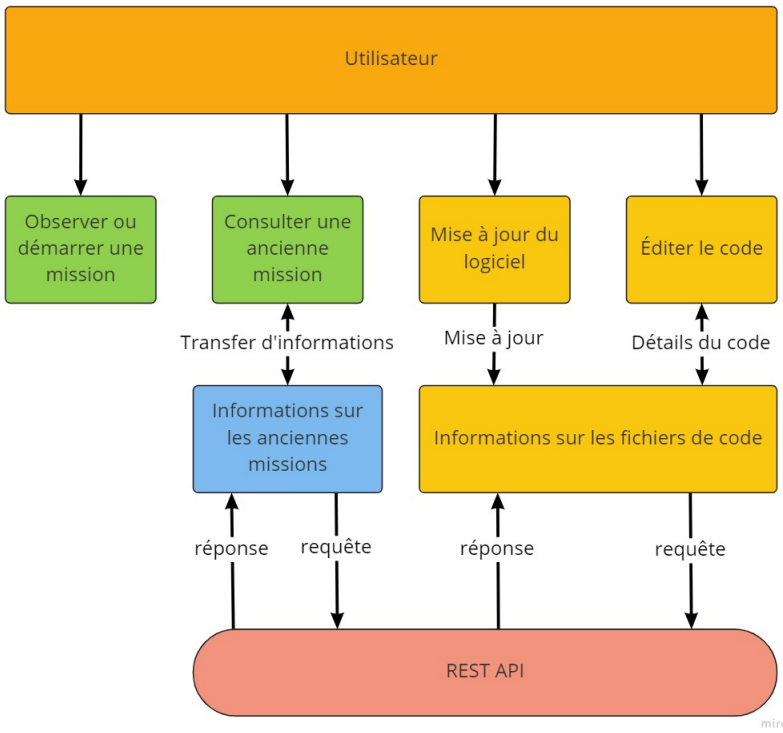

Figure 15 : Options principales de l'interface utilisateur

liens directs avec la mission courante et sont plutôt reliés à des requis spécifiques. Conformément aux requis **R.F.17** et **R.F.18**, l'utilisateur aura l'option de consulter les diverses informations quant aux missions précédentes à partir de la base de données et d'afficher les cartes correspondantes à chaque mission. Les informations pourront être triées selon chaque attribut. Afin d'obtenir toutes les informations nécessaires à la sélection des anciennes missions, une API REST sera utilisée. En effet, puisqu'il s'agit ici de consulter l'historique des missions, il n'est pas nécessaire d'entretenir une communication constante avec le serveur. La même logique est appliquée en ce qui concerne les requis **R.F.14** et **R.F.16**. L'option de mise à jour du logiciel donnera l'option à l'utilisateur de mettre à jour le logiciel de contrôle alors que l'option d'édition de code se résumera à récupérer et afficher les informations sur le code, pour que l'utilisateur puisse le modifier et envoyer par la suite ses modifications. Il faut noter que l'utilisateur sera également en présence d'une option lui permettant d'identifier s'il est en simulation ou connecté aux drones physiques, permettant d'activer ou de désactiver différentes fonctionnalités spécifiques dans l'entièreté de l'application, selon le requis **R.L.2**.

En observant de plus près les différentes options associées à la gestion et au démarrage d'une mission, on peut identifier plusieurs requis. La figure 16 présente un résumé de la logique des différentes fonctionnalités qui seront implémentées.

Tout d'abord, il est important de définir une séparation entre les commandes spécifiques à chaque drone et les commandes spécifiques à la mission. Les commandes spécifiques à la mission comprennent les requis **R.F.2** et **R.F.6**. Les commandes spécifiques aux drones sont reliées aux requis **R.F.1**, **R.F.20** et **R.F.12**. Puisque la majorité des commandes dépendent des informations sur les drones, elles possèdent un lien avec le module de requête d'informations sur les drones. Cepen-

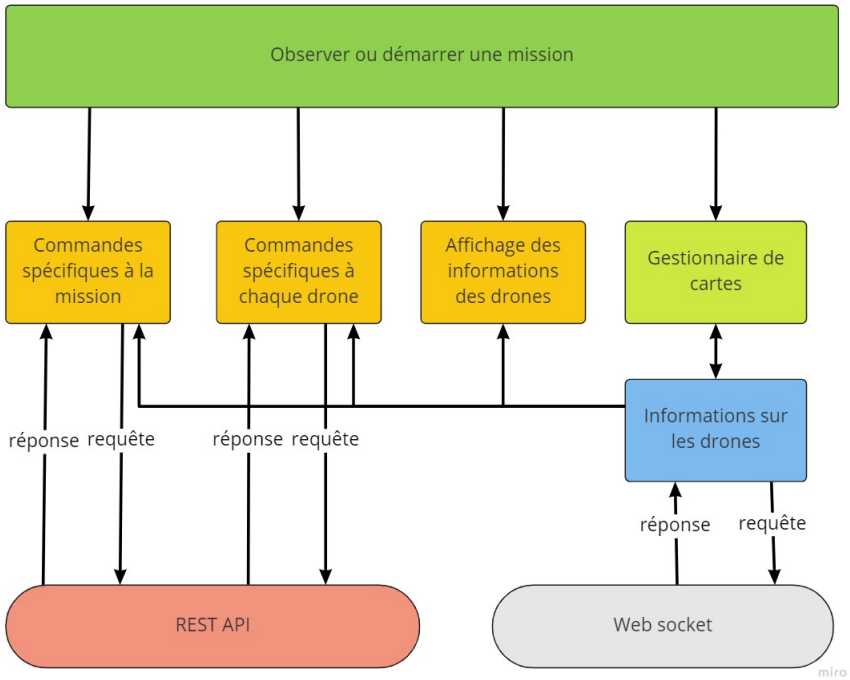

Figure 16 : Options de l'interface utilisateur liées à une mission

dant, il est suffisant d'utiliser une API REST lorsqu'il s'agit de transférer les commandes au serveur.

Par la suite viennent les différents modules associés à l'affichage des informations des drones et à l'affichage des cartes. Ces deux modules utilisent un WebSocket avec le serveur, car selon le requis **R.F.3**, il faut souvent actualiser les diverses informations. Les informations affichées viendront alors directement de cette communication et permettrons donc de remplir les requis **R.F.3**, **R.F.7**, **R.F.9**, **R.F.13** et **R.C.1**. En ce qui concerne le module de gestion des cartes, un schéma plus spécifique peut être retrouvé dans la figure 17.

Comme l'illustre la figure 17, les informations obtenues à partir du serveur permettront de générer les différentes cartes pour chaque drone à partir du client, conformément au requis **R.F.8**. La carte générale sera obtenue en même temps en faisant la combinaison des informations obtenues pour chaque drone remplissant le requis **R.F.11**. Il faut noter qu'il a été décidé qu'une seule carte soit affichée à la fois avec une identification claire, pour éviter de confondre l'utilisateur lors de son utilisation du site, ce qui est relié au requis **R.C.4**.

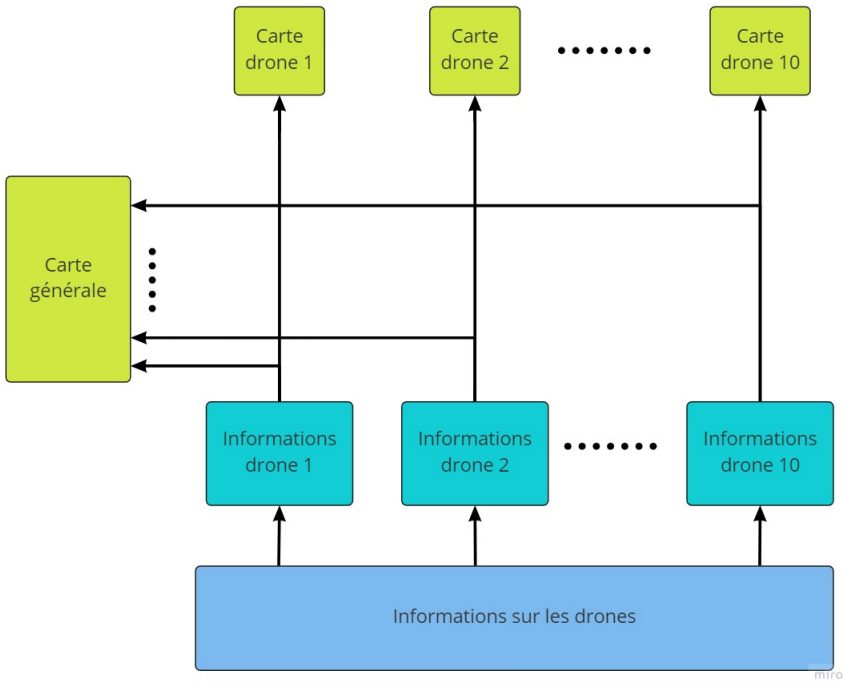

Figure 17 : Système de gestion des cartes de l'interface utilisateur

3.6 Fonctionnement général (Q5.4)

Notre système est composé de quatre modules distincts : le module du client, le module du serveur, le module de la base de donnée et enfin le module du code embarqué. Chacun de ces modules est dans un conteneur Docker qui lui est propre afin de réduire la probabilité que des problèmes surviennent lors du déploiement du système sur différents systèmes d'exploitations.

La communication entre les différents conteneurs est assurée grâce à diverses méthodes. Tout d'abord, pour la communication entre le client et le serveur, chacun des conteneurs expose un port de la machine hôte afin qu'une communication puisse être établie. Le client expose le port 8080 permettant de s'y connecter à l'adresse *http ://localhost :8080*. Le serveur expose le port 5000 permettant de s'y connecter à l'adresse *http ://localhost :5000*. Le client et le serveur communiquent ensuite avec des requêtes suivant le style architecturale REST.

Pour ce qui est de la communication entre le serveur et les drones physique, celle-ci est permis par la *CrazyRadio*. Afin que le conteneur du serveur puisse accéder à ceux-ci, il est nécessaire de monter le fichier */dev/ttyUSB0* de l'hôte, car c'est celui qui représente le port usb connecté à la CrazyRadio.

Enfin, la communication entre le serveur et la simulation se fait à l'aide de *sockets* UNIX. Ceux-ci seront créés dans le dossier */tmp/socket/* de l'hôte et seront monter dans leur conteneur respectif afin que ces deniers puissent y avoir accès.

Pour exécuter le produit final, il faut d'abord commencer par cloner le répertoire git à l'aide de la commande suivante :

```
$ git clone git@gitlab.com:polytechnique-montr-al/inf3995/20221
/equipe-106/INF3995-106.git --recursive
```

Pour que cette commande soit effectuée avec succès il est important de posséder une clé *ssh* enregistrée avec GitLab et GitHub (clone avec HTTPs n'est plus supporté).

Il faut ensuite s'assurer que les dépendances nécessaires soit installées sur le système avant d'exécuter le logiciel.

La première dépendance à installer est Docker. La commande suivante permet d'installer Docker sur Ubuntu.

```
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

La documentation par rapport au téléchargement et à l'installation de Docker se trouve à cette adresse : `https://docs.docker.com/engine/install/ubuntu/`.

Il est important de noter que, si l'utilisateur souhaitant exécuter le projet utilise une carte graphique dédiée Nvidia sur son ordinateur, il doit également installer nvidia-docker. L'utilisateur peut vérifier quelle carte graphique est utilisée en exécutant la commande suivante dans un terminal.

```
$ glxinfo|egrep "OpenGL vendor|OpenGL renderer"
```

Les commandes suivantes permette d'installer nvidia-docker dans le cas d'une carte graphique Nvidia est utilisé par l'ordinateur.

```
$distribution=$(. /etc/os-release;echo $ID$VERSION_ID) \
  && curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | sudo apt-key add - \
  && curl -s -L https://nvidia.github.io/nvidia-docker/$distribution/nvidia-docker.list
      | sudo tee /etc/apt/sources.list.d/nvidia-docker.list
  sudo apt-get update
  sudo apt-get install -y nvidia-docker2
  sudo systemctl restart docker
```

Il est possible de vérifier l'installation de nvidia-docker en exécutant cette commande

```
$ sudo docker run --rm --gpus all nvidia/cuda:11.0-base nvidia-smi
```

L'explication détaillés de ces commandes ce trouve à l'adresse web suivante : `https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/install-guide.html`

La dernière dépendance du projet est le logiciel Docker-Compose. Il peut être installé à l'aide cette ligne de commande.

```
$ sudo curl -L
    "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname
    -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

Il est important de mentionner qu'il est nécessaire d'avoir une version supérieure ou égale à la version 1.29.2 de Docker-Compose dans le cas d'une carte graphique Nvidia. Dans le cas inverse, une version supérieure ou égale à 1.25.0 est nécessaire (bien que la version 1.29.2 est recommandée).

L'application peut ainsi être démarrée en exécutant la commande suivante à la racine du projet :

```
$ ./start.sh
```

Dans le cas d'une erreur que le fichier n'est pas exécutable, il faut exécuter la commande suivante dans le terminal afin de permettre son exécution :

```
$ chmod u+x start.sh
```

Une fois que les conteneurs Docker seront créés (cette étape peut prendre quelques minutes), l'application sera disponible à l'adresse suivante : `http://localhost:8080/`

4 Processus de gestion

4.1 Estimations des coûts du projet (Q11.1)

D'après les calculs effectués en 1, la conception et le développement de ce projet coûtera 78 700$ CAN pour 540 heures de travail. Il a été estimée qu'une heure de développement du système coûte 130$ par personne et une heure de conception 145$ par personne. Le coût total du projet peut ainsi être calculé :

$$
\begin{aligned}
Cot &= (taux\ horaire_{developpeur} \times heures_{developpeur} + taux\ horaire_{coordonateur} \times heures_{coordonateur}) \\
Cot &= (130\$/h \times 505_{heures} + 145\$/h \times 90_{heures}) \\
Cot &= 78700\$
\end{aligned}
\tag{1}
$$

Lors de la planification des tâches, 505 heures de développement et 90 heures de conception ont été allouées, ce qui revient à une heure et demie par semaine par personne pour la coordination du projet.

4.2 Planification des tâches (Q11.2)

En ce qui concerne la planification des tâches, un horaire a été créé afin de donner une idée à l'équipe des différentes tâches devant être effectuées et quand elles doivent être exécutées. Cet horaire a été divisé en *sprints* qui correspondent à une période de 2 à 3 semaines. Les tâches ont ainsi été réparties dans différentes catégories en fonction des composants qui sont affectées par celles-ci. On peut d'ailleurs observer 3 jalons importants lors de la réalisation du projet : le PDR, le CDR et le RR. Puis, les tâches ont été divisées en sous-tâches pour pouvoir mieux les répartir à travers plusieurs *sprints*. Il est important de mentionner que le nombre d'heures indiqué ne correspond qu'à une approximation de la quantité de temps réelle que prendra une tâche.

	Tâches	PDR	CDR	RR	Temps alloué (heures)
Interface utilisateur	Module d'affichage des anciennes missions		- Options de sélection d'une ancienne mission - Module d'affichage des logs		20
	Module d'édition du code			- Communiquer avec le serveur pour voir les fichiers	15
	Gestionnaire de cartes		- Modélisation de la carte - Rafraîchissement de la carte		20
	Commandes spécifiques aux drones	- Identifier les drones		- Position initiale des drones - Zone de sécurité	15
	Module d'affichage des informations des drones		- Informations sur les drones		10
	Module de communication avec le serveur	- Établir une connexion avec un WebSocket	- Communication pour extraire les anciennes missions		10
	Modélisation de l'interface générale	- Page d'accueil avec boutons "Identifier", "Lancer", "Terminer"	- Développer la page d'anciennes missions - Développer la page de missions	- Développer l'éditeur de code	40
	Commandes spécifiques aux missions	- Lancer la mission - Terminer la mission	- Retour à la base		15
Station au sol	Implémentation de l'algorithme de cartographie		- Implémenter l'algorithme de cartographie		20
	Communication avec l'interface utilisateur	- Établir une connexion avec l'interface utilisateur	- Traitement des requêtes - Transmission des informations		15
	Communication avec la simulation	- Établir une connexion avec le simulateur	- Créer les requêtes spécifiques à la simulation		30
	Communication avec les drones (CrazyRadio)	- Établir une connexion avec les drones	- Interface avec la Crazyradio - Traitement de la réception d'information aux		30
	Communication avec la base de données		- Ajouter une interface sur l'API de la BD - Politique d'envoi de données		20
	Traitement des requêtes des informations des drones		- Décoder les paquets du drone - Traiter les différentes informations reçues		20
	Traitement des commandes des drones	- Traitement des commandes: "Identifier", "Lancer", "Terminer"	- Traitement de la commande "Retour à la base"		10
	Configurer la simulation		- Spécifier la position des drones - Rendre aléatoire la configuration de l'arène		20
	Recompilation des fichiers pour l'édition de code			- Recompiler les fichiers du code embarqué - Envoyer le code compilé sur le drone	20
Embarqué/Simulation	Gestion de la communication (Réception/Envoi)	- Établir la connexion de communication	- Encoder/décoder les paquets - Rediriger les commandes		30
	Établissement de la connexion P2P			- Communication P2P entre les drones	10
	Traitement des commandes	- Traiter "Identifier", "Lancer", "Terminer"	- Traiter le retour à la base		15
	Acquisition des senseurs		- Acquisition des informations reçues des senseurs	- Détection d'un crash	25
	Algorithme de déplacement		- Implémenter l'algorithme de déplacement		25
	Contrôle du mouvement	- Implémenter le mouvement pour "Lancer" et "Terminer"	- Éviter les obstacles - Retour à la base		25
BD	Communication entre la base de données et le serveur		- Enregistrer les logs - Enregistrer les cartes		15
Environnements	Initialiser les conteneurs des sous-systèmes	- Création d'un conteneur pour chaque sous-systèmes			15
	Relier les conteneurs entre eux	- Relier les conteneurs			10
	Création du script de lancement	- Création script bash pour lancer l'application			5
					Total : 505

Figure 18 : Calendrier des tâches

Puis, un autre tableau a été construit pour attribuer les différentes tâches aux membres de l'équipe. Les tâches ont été réparties pour que chaque membre de l'équipe implémente une partie de tous les systèmes présents dans le projet (dans la mesure du possible).

Figure 19 : Répartition des tâches

4.3 Calendrier de projet (Q11.2)

Phases	Requis complétés	Date
PDR / Sprint 1	Création des environnements de développement. Communication entre l'interface utilisateur, le serveur et l'embarqué. Interface utilisateur générale. Identification des drônes.	4 février 2022
Sprint 2	Informations sur les drones. Traitement des requêtes dans le serveur. Traitement des paquets sur le code embarqué. Acquisition des senseurs.	18 février 2022
Sprint 3	Modélisation de la carte dans l'interface utilisateur. Extraction des anciennes missions. Algorithme de cartographie. Communication entre le serveur et la base de donnée. Algorithme de déplacement.	7 mars 2022
CDR / Sprint 4	Affichage des journaux. Interface utilisateur pour les anciennes missions. Implémentation du retour à la base. Contrôle du mouvement.	21 mars 2022
Sprint 5	Édition du code sur l'interface utilisateur. Recompilation des fichiers du code embarqué.	4 avril 2022
RR / Sprint 6	Implémentation de la position initiale des drones et de la zone de sécurité. Communication P2P entre les drones. Détection d'un crash.	25 avril 2022

Tableau 6 : Tableau des jalons du projet

4.4 Ressources humaines du projet (Q11.2)

L'équipe pour réaliser le projet est composée de six développeurs ayant tous des qualifications différentes. Les connaissances des membres de l'équipe sont assez versatiles pour pouvoir développer des fonctionnalités dans tous les sous-systèmes. Cependant, certains membres sont plus spécialisés dans certains domaines. En effet, alors que certains membres ont plus d'expérience en système embarqué, d'autres ont développé plusieurs applications web.

De plus, lors des rencontres, les membres de l'équipe devront s'acquitter des rôles de *Scrum master*, de maître du temps et de secrétaire pour s'assurer que toutes les rencontres soient efficaces et bien organisées. Finalement, un membre de l'équipe sera responsable de s'assurer que tous les requis soient complétés lors des remises énoncés précédemment.

5 Suivi de projet et contrôle

5.1 Contrôle de la qualité (Q4)

Afin de s'assurer de la qualité des livrables, plusieurs tests sont effectués pendant le processus de développement. En effet, lorsqu'une fonctionnalité liée à un sous-système est implémentée, celle-ci doit être testée à l'aide de plusieurs tests unitaires. De plus, lorsqu'elle est prête à être intégrée au code existant, les autres membres de l'équipe doivent réviser le code écrit pour s'assurer qu'elle soit conforme aux requis. Puis, lorsqu'un composant du système est terminé, elle est testée de fond en comble à l'aide de plusieurs tests de composants et de tests d'intégration. Finalement, lorsque le livrable est prêt à être soumis, une révision complète de ce dernier sera effectué par toute l'équipe (autant individuellement qu'en groupe) pour s'assurer que tous les requis ont été livrés.

5.2 Gestion de risque (Q11.3)

Les risques associés à la réalisation du projet sont nombreux et sont principalement reliés au niveau matériel et logiciel du projet. Un des premiers risques associé au projet est lié à l'usure et au bris d'équipement. En effet, lors de la réalisation du projet, il est possible que de l'équipement lié aux drones se brise ou se perde. Dans l'éventualité où de telles situations se produiraient, l'équipe a déjà pris des mesures avec l'Agence Spatiale pour fournir des pièces de remplacement dans le besoin, aux frais de l'équipe. Ce risque est qualifié comme étant important, car l'équipe a besoin de l'équipement pour effectuer divers tests et développer différents logiciels qui seront appliqués sur les drones.

Un autre risque associé au projet est la découverte qu'une technologie serait mal adaptée pour la réalisation de certaines fonctionnalités. En d'autres mots, il se peut que les décisions technologiques initiales ne conviennent pas pour certaines parties du projet. Afin de contrôler le risque associé à une telle possibilité, il a été discuté initialement des différents choix technologiques qui pourraient être utilisés au cours du projet. Cette analyse préparatoire permettra à l'équipe d'avoir une meilleure idée des différentes options technologiques à disposition au cas où un tel problème surviendrait. Ce risque est assez important, car il pourrait causer des très grands changements de logique au niveau logiciel.

En observant de plus près la réalisation du projet, on peut observer qu'il existe un risque au niveau de la communication entre les différentes composantes du projet. En effet, un risque pourrait être une

attaque par inondation du serveur en forçant la connexion d'un très grand nombre d'utilisateurs et donc de limiter la disponibilité de ce dernier. Que cette situation apparaisse de manière accidentelle ou volontaire, il ne faut pas la négliger, même si elle est peu probable. Une technique qui pourrait être implémentée pour palier au problème serait de limiter le nombre maximal de connexions possibles au serveur.

Finalement, un risque extrêmement important du projet qui pourrait survenir serait celui associé à la gestion du temps. Il est possible qu'il y ait une sous-estimation de l'ampleur de certaines tâches, menant alors à des délais et donc empêchant l'équipe de respecter la contrainte de temps établie par l'Agence. La toute première chose sur laquelle l'équipe s'entend pour réduire ce risque est d'attaquer le plus rapidement possible les tâches qui lui sont assignées, sans délai. De plus, afin d'assurer la bonne progression de l'ensemble du projet, des rencontres hebdomadaire sont effectuées où chaque membre de l'équipe parlera de ses avancements sur ses différentes tâches. Non seulement l'équipe entière sera au courant de l'état du projet, mais elle pourra également mieux réagir face aux différents imprévus qui risquent de survenir.

5.3 Tests (Q4.4)

On peut diviser les tests en cinq catégories : les tests de l'interface utilisateur, les tests du serveur, les tests du code embarqué et les tests du robot physique.

Tout d'abord, les tests de l'interface utilisateur incluent tous les tests unitaires en lien avec l'interface qui sera présenté à l'opérateur des drones. Cette section devrait tester toutes les composantes de l'interface comme la carte, les boutons et les menus. De la même manière, la connexion avec le serveur devra être testé à l'aide d'un stub. L'interface utilisateur sera testée à l'aide de la librairie Jest.

Puis, les tests sur le serveur devront s'occuper de vérifier que les différents connexions avec les 3 différentes composantes fonctionnent (interface utilisateur, base de données et drones). De plus, ces derniers devront s'assurer que l'algorithme qui crée la carte à partir des données recueillies par les drones fonctionne. Cela se fera notamment par plusieurs tests unitaires sur chaque étape de l'algorithme choisi ainsi que sur plusieurs tests d'intégration du système au complet. La librairie de test utilisée pour la création des tests du serveur sera Pytest.

Pour tester le code embarqué, la librairie Google Test. Celle-ci sera utilisée pour tester les différents composants à l'aide de tests unitaires du code embarqué comme l'acquisition des capteurs, le contrôle du mouvement, les opérations mathématiques utilisées dans le code et le traitement des commandes. Aussi, des tests d'intégration seront effectués pour s'assurer que les drones répondent aux commandes avec le comportement voulu.

Finalement, en ce qui concerne les tests sur le robot physique, Crazyflie donne une série de tests qui permettent de vérifier l'état du drone, notamment à l'aide de la DEL sur le robot. Ces tests permettent aussi de s'assurer que chacune des hélices fonctionnent.

Il est important de mentionner qu'en plus des tests mentionnés précédemment sur chacun des composants, il y aura aussi des tests d'intégration sur le système complet qui seront effectués. Par exemple, il faudra tester qu'une commande initiée par un utilisateur sur l'interface graphique

se rende bel et bien jusqu'au drone. Aussi, il faudra tester que les informations recueillies par les drones se rendent jusqu'à l'interface utilisateur et la base de données.

5.4 Gestion de configuration (Q4)

Au niveau de la gestion du code, ce dernier est situé dans un répertoire GitLab. À la source du projet, on peut observer quatres dossiers correspondant chacun à chacun des sous-systèmes du projet (interface utilisateur, système embarqué, base de données et station au sol). Chaque sous-système peut être lancé dans un conteneur pour s'assurer que chaque développeur ait le même environnement lors du développement du système. Pour partir le logiciel, un script présent à la source peut être exécuté. Dans chacun des dossiers, un fichier *.gitignore* a été ajouté pour permettre au gestionnaire de version de filtrer les fichiers devant être mis à la disposition de toute l'équipe. Il est aussi important de mentionner que le projet dépend d'un sous-module hébergé sur Github contenant le *firmware* spécifique aux drones.

Au niveau de l'interface utilisateur, plusieurs fichiers de configuration sont présents dans le dossier respectif. Ces derniers permettent notamment de spécifier les dépendances du projet ainsi que le standard de programmation. Puis, un dossier est alloué au code source et un autre aux tests. En ce qui concerne la station au sol, le dossier est simplement composé du code source nécessaire à l'exécution du serveur. La base de données, quant à elle, est simplement constitué d'un fichier qui extrait une image de MongoDB de DockerHub. Pour le répertoire concernant les drones, ce dernier est divisé en plusieurs parties. Tout d'abord, il est constitué de deux dossiers qui gèrent le code source spécifique à l'embarqué et à la simulation. Aussi, un dossier contient le code partagé entre les deux environnements et il possède aussi un répertoire contenant le code du logiciel du Crazyflie fournie par la compagnie.

Par rapport à la documentation, les fonctions créées dans le code source possèdent en grande majorité une en-tête descriptive. Il sera alors possible de générer de la documentation automatique à l'aide d'outils comme Doxygen.

Finalement, un fichier Markdown a été ajouté dans chacun des dossiers pour décrire la fonction spécifique du répertoire ainsi qu'un tutoriel rapide pour démarrer l'environnement de développement respectif.

5.5 Déroulement du projet (Q2.5)

CDR et RR seulement

Dans votre équipe, qu'est-ce qui a été bien et moins bien réussi durant le déroulement du projet par rapport à ce qui était prévu dans l'appel d'offre initialement.

6 Résultats des tests de fonctionnement du système complet (Q2.4)

CDR et RR seulement

Qu'est-ce qui fonctionne et qu'est-ce qui ne fonctionne pas.

7 Travaux futurs et recommandations (Q3.5)

RR seulement

Qu'est-ce qui reste à compléter sur votre système ? Recommendations et possibles extensions du système.

8 Apprentissage continu (Q12)

RR seulement

Un paragraphe par membre (identifié en début de paragraphe) de l'équipe qui doit aborder chacun de ces aspects de façon personnelle : 1. Lacunes identifiées dans ses savoirs et savoir-faire durant le projet. 2. Méthodes prises pour y remédier. 3. Identifier comment cet aspect aurait pu être amélioré.

9 Conclusion (Q3.6)

RR seulement

Par rapport aux hypothèses et à la vision que vous aviez du système lors du dépôt de la réponse à l'appel d'offre, que concluez-vous de votre démarche de conception maintenant que le système est complété ?

Références

[1] Mike Chan. *SQL vs. NoSQL – what's the best option for your database needs ?* 2019. url : https://www.thorntech.com/sql-vs-nosql/.

[2] *Django Deprecation Timeline*. url : https://docs.djangoproject.com/en/dev/internals/deprecation/.

[3] Matthias Graf. *Which Backend Language Should You Choose ?* Nov. 2020. url : https://medium.com/swlh/which-backend-language-should-you-choose-eb924902a9b3/.

[4] John Hammink. *The Types of Modern Databases*. 2018. url : https://www.alooma.com/blog/types-of-modern-databases.

[5] Sue Lynn. *Understanding Flask vs FastAPI Web Framework*. Juill. 2021. url : https://towardsdatascience.com/understanding-flask-vs-fastapi-web-framework-fe12bb58ee75.

[6] Maarten van Steen et Andrew S. Tanenbaum. *Distributed systems*. 3e éd. Pearson Education, Inc., déc. 2020. 598 p. isbn : 978-90-815406-2-9. url : https://www.distributed-systems.net/index.php/books/ds3/.