Département de génie informatique et de génie logiciel

INF3995

Projet de conception d'un système informatique

Conception d'un système aérien minimal pour exploration

Rapport final de projet

Équipe No. 200

Babin Philippe
Lauzon Nicolas
Lharch Nour El Houda
Martineau William
Quiquempoix Theo

21 avril 2021

Table des matières

1. Objectifs du projet

Il y a quelques mois, une agence spatiale a émis un appel d'offres pour la conception d'un système d'exploration utilisant des drones. Ceux-ci sont de petite taille, car ils permettent de couvrir plus de terrain qu'un drone de grande taille, qui serait limité dans ses mouvements.

Les drones doivent être en mesure d'envoyer des données relatives à leur exploration à une interface utilisateur. Ces données portent par exemple sur l'environnement détecté, l'état interne des drones (batterie, vitesse, etc.) ainsi que des logs permettant de vérifier leur bon fonctionnement. Pour effectuer cette communication, nous devions utiliser une station au sol fournie par l'Agence. Celle-ci communique par radio et permet d'envoyer des commandes ainsi que de recevoir des informations des drones. Cette même station au sol doit être en mesure de faire différentes actions, telles que sauvegarder des données de l'expérience et relayer de l'information à l'interface utilisateur. Pour conclure, ceci est l'idée générale du système et nous voyons très rapidement qu'il est découpé en trois parties, soit le code embarqué, la station au sol ainsi que l'interface utilisateur. Afin de pouvoir tester notre système de façon sécuritaire, nous utilisons un simulateur. Il permet d'expérimenter avec nos algorithmes avant de les implémenter sur les drones physiques.

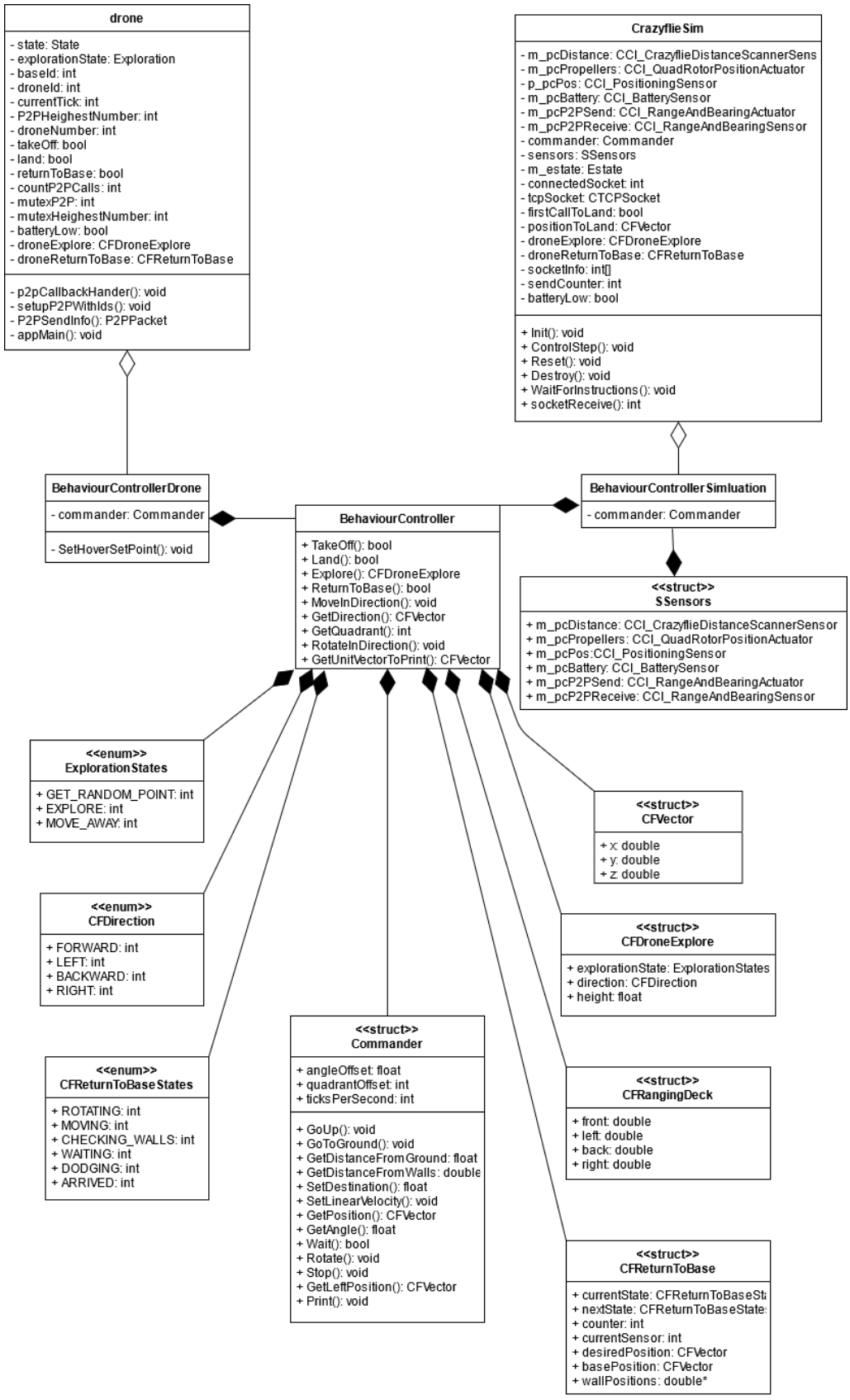
2. Description du système

2.1 Logiciel embarqué (Q4.5)

Comme expliqué dans le *Critical Design Review* (CDR), notre simulation ainsi que notre code embarqué utilise une librairie commune. Cette librairie est codée en C [22] pour qu'elle soit compatible avec notre code embarqué qui est en C lui aussi. Nous avons choisi ce langage, car un fichier C peut importer un fichier C, mais il ne peut pas importer un fichier C++ [16]. Ces deux librairies contiennent des fonctions de haut-niveau où qui sont communes autant à la simulation qu'au code embarqué. Comme nous pouvons voir dans la figure 1, cette librairie appelle des fonctions de deux autres librairies qui sont spécifiques à la plateforme utilisée (drone et simulation). Ces deux librairies, appelées *BehaviourControllerDrone* et *BehaviourControllerSimulation*, implémentent les fonctions de la structure *Commander* en utilisant à leur tour les librairies de ARGoS [3] et de Bitcraze [1]. Nous pouvons simplement importer le bon fichier dans notre contrôleur et le passer aux méthodes de la librairie et le tour est joué.

Nous voyons dans la figure 1 que plusieurs structures sont utilisées par la librairie. Celles-ci offrent des interfaces communes et sont utilisées de façon similaire pour les deux côtés. Elles sont souvent passées en paramètre aux fonctions de haut niveau ou retournées par celles-ci. Un exemple concret de leur application est de convertir le système de coordonnées des drones physiques vers les CVector3 de ARGoS. Nous pouvons donc passer en paramètre le même type dans les deux contrôleurs.

Figure 1: Diagramme de classes des drones



Pour résumer la figure 1, nous avons notre *BehaviourController* au cœur de la librairie. Chacune de ses méthodes utilise un *Commander* qui est passé en paramètre. Il y a aussi un pointeur vers les capteurs des drones de la simulation qui est passé en paramètre ainsi que des structures utilisées dans les algorithmes. Celles-ci sont représentées par les structures commençant par CF dans notre schéma. Les contrôleurs sont les entités qui se trouvent dans les coins supérieurs droits et gauches. Ils importent les *BehaviourController* spécifiques et assignent le *commander* pour qu'ils se correspondent. Ensuite, nous pouvons directement appeler les méthodes de haut-niveau de *BehaviourController.*

Du côté de la simulation, nous avons utilisé ARGoS3. Celui-ci permet de définir un environnement ainsi que de contrôler le comportement de nos drones par l'intermédiaire de contrôleurs. Le but ultime de la simulation est de vérifier le bon fonctionnement de nos algorithmes. C'est pourquoi lorsque nous avions un résultat satisfaisant après la CDR, nous avons commencé à tout implémenter sur les drones physiques. Le fonctionnement de la simulation est assez simple. Nous générons une expérience dans un fichier .argos et y insérons une référence vers notre contrôleur. Lors de la construction de l'expérience, ARGoS va lier notre contrôleur aux drones placés dans l'expérience. Chaque contrôleur possède une méthode nommée *controlStep* qui est appelée plusieurs fois par seconde. C'est donc dans cette méthode que la plupart de notre code se trouvera. Nous utilisons une machine à état qui indique aux drones quoi faire selon les commandes reçues de l'interface utilisateur. Afin de communiquer avec notre serveur, nous utilisons des sockets [6]. Ceux-ci envoient les informations relatives au drone à chaque 10 frames, ce qui correspond à 2 fois par secondes puisque la simulation roule à 20 frames par secondes. À chaque frame, nous regardons si nous avons reçu une commande de la part du serveur. Si oui, nous changeons l'état de la machine à état.

Du côté des drones, nous avons quelques modifications par rapport à la CDR. Naturellement, nous avons implémenté la librairie sur les drones, ce qui a ajouté une bonne partie de code. Nous avons déplacé toute la partie P2P [5] dans 3 méthodes : *p2pCallbackHandler*, *setupP2PWithIds* et *P2PSendInfo*. Ces méthodes gèrent le fait de recevoir un paquet, ainsi que la mise en place du P2P et son setup, c'est-à-dire que les drones vont s'autoassigner leurs IDs en fonction des paquets qu'ils reçoivent. Ensuite, dans le *appMain*, nous avons mis un switch-case, comme celui dans la simulation, pour gérer les différents états nécessaires pour que les drones explorent et fassent les différentes fonctionnalités demandées. Mais, le fait d'avoir ajouté tout ce code nous a fait rajouter plus de variables globales, car nous devons utiliser ces variables dans les différentes méthodes de P2P, ainsi que dans les *Log Configs* et *Param Configs* [20] de Crazyflie pour envoyer et recevoir des informations avec le serveur. Enfin, du côté de *BehaviourControllerDrone*, nous avons implémenté les méthodes nécessaires pour l'utilisation de la librairie avec l'API de Bitcraze, mais nous avons dû rajouter une méthode appelée *SetHoverSetPoint*. Cette méthode est tout simplement nécessaire

pour faire déplacer les drones. En effet, c'est celle-ci qui va dire au drone de se déplacer dans une certaine direction avec une certaine vitesse et angle.

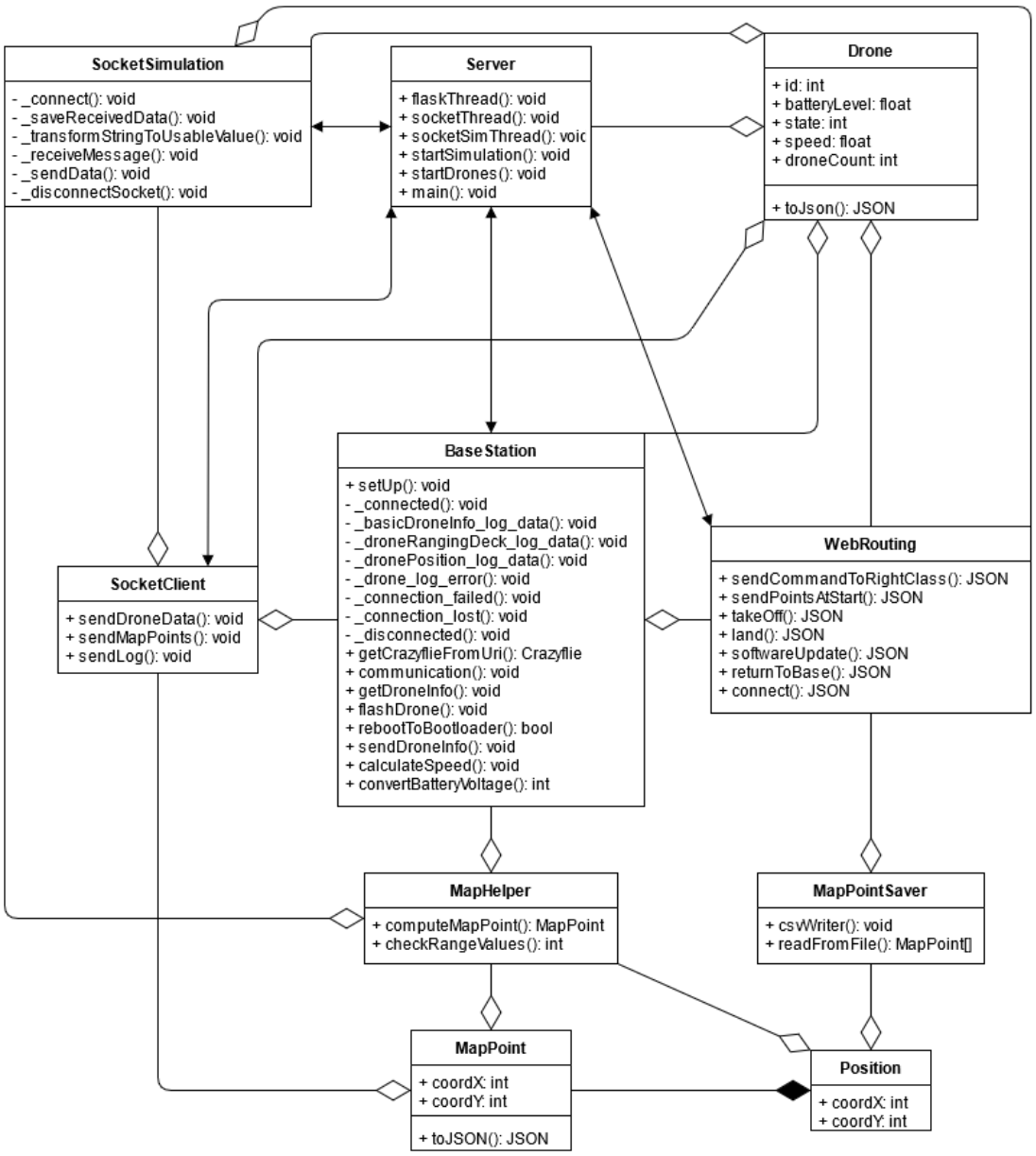
Pour ce qui est de la librairie commune, les méthodes de commandes contiennent pour la plupart une autre machine à état permettant d'indiquer à quelle étape de la commande le drone est rendu. Toutes ces méthodes prennent des pointeurs vers des objets. Ceux-ci contiennent souvent des références vers les objets. Nous pouvons modifier l'objet dans la méthode de la librairie. Cela permet de conserver les modifications entre chaque appel des fonctions de la librairie. Nous utilisons des void* pour passer les objets C++ vers la librairie commune en C et ensuite en C++ vers les librairies spécifiques. Pour envoyer des commandes aux librairies spécifiques, nous utilisons un *commander* qui contient des pointeurs vers les fonctions des librairies spécifiques. Nous initialisons cette structure dans les contrôleurs appropriés et le passons en paramètres aux fonctions de la librairie. Il y a aussi différentes structures qui ont été déclarées, comme *CFDroneExplore*, *CFReturnToBase* et *CFVector* qui offrent une interface commune entre la simulation et les drones.

Il y a eu quelques changements depuis la CDR du côté de la librairie. Premièrement, puisque nous avons maintenant implémenté le code embarqué des drones, nous avons dû faire beaucoup de changement parce que parfois, nos algorithmes n'étaient pas adaptés pour les drones physiques. Par exemple, l'algorithme du retour à la base fonctionnait bien dans la simulation puisque les drones ont accès à leur position absolue, mais ce n'est pas le cas pour les vrais drones. Les drones de la simulation peuvent utiliser leur position absolue dans l'arène pour s'orienter, ce qui n'est pas le cas dans la vraie vie. Nous avons donc changé l'algorithme pour que les drones utilisent leur position relative. En faisant ce changement, nous avons fait en sorte que les drones vérifient autour d'eux lorsqu'ils évitent des obstacles. Cela permet de prendre en compte des situations rares, mais qui auraient pu faire échouer la mission. Ensuite, nous avons changé l'algorithme d'exploration pour qu'il fasse une petite rotation lorsque le robot arrive face à un mur. Nous avons donc des robots dans des angles différents ce qui permet de détecter une plus grande partie de la carte, en particulier les coins. De plus, lorsque les drones arrivent face à des murs, ils regardent s'ils peuvent se déplacer dans la direction définie aléatoirement. Ils vont donc moins souvent rester bloqués contre des murs. Aussi, nous avons bougé plusieurs méthodes de BehaviourControllerSimulation vers BehaviourController, car nous nous sommes rendu compte que celles-ci allaient être réutilisées autant pour la simulation que pour les drones physiques. Cela a donc grandement réduit la répétition de code et la complexité de BehaviourControllerSimulation.

2.2 La station au sol (Q4.5)

La station au sol est le serveur de notre système, c'est l'élément central du système et permet les bonnes communications entre les drones et l'interface de contrôle. Nous avons décidé de concevoir notre serveur avec le langage Python [8]. Nous utilisons Python, car c'est un bon langage pour faire des calculs et c'est un des langages les plus faciles à utiliser avec la Crazyradio.

Figure 2: Diagramme de classes du serveur



Le diagramme de classe pour le serveur est représenté avec la figure 2. Sur ce schéma, une double flèche représente la communication entre deux classes, une flèche

d'agrégation représente l'utilisation d'une classe par une autre classe et une flèche de composition représente un héritage.

Tout d'abord, nous remarquons une classe centrale au serveur. Celle-ci n'a pas changé depuis la CDR, et est la classe *Server*. Elle s'occupe d'initialiser tous les différents composants du serveur. En fonction du paramètre entrée lors de l'exécution du Dockerfile [14], la classe *Server* va exécuter les composants qui sont rattachés à ce paramètre. Ce paramètre permet au serveur de s'exécuter de deux manières distinctes : en mode simulation, ou en mode drone physique. Dans chacun des deux modes, le serveur va exécuter le socket du client (*SocketClient*) et le routage web des requêtes REST/HTTP [7] (*WebRouting*). Ces deux classes permettent de communiquer avec le client, ce qui explique pourquoi ils sont toujours exécutés par le serveur. Si nous voulons exécuter le serveur en mode simulation, alors il s'occupe d'initialiser et de lancer le socket avec la simulation (*SocketSimulation*) et n'exécute pas la classe qui gère le *Crazyradio* (*BaseStation*). Si nous voulons exécuter le serveur en mode drone physique, alors le serveur s'occupe de démarrer la classe *BaseStation* et de ne pas initialiser la classe du socket avec la simulation.

Ensuite, nous avons plusieurs classes secondaires dans le serveur, et elles servent principalement à communiquer avec les deux autres modules de notre solution (le client et les drones). En premier, la communication avec le client, qui se fait avec deux classes, *WebRouting* et *SocketClient*. La classe *WebRouting* permet de gérer le routage des requêtes REST/HTTP. Nous pouvons notamment remarquer qu'elle possède une méthode par commande possible sur le client, et aussi deux nouvelles méthodes depuis la CDR qui sont *sendCommandToRightClass* et *sendPointsAtStart*. Cette classe est très importante, car elle permet de réceptionner les différentes requêtes, puis d'exécuter les classes et méthodes nécessaires au bon fonctionnement de la requête sur tout le système. De plus, nous avons la classe *SocketClient*, qui permet d'initialiser le socket entre le client et le serveur. Sans cette classe, nous ne pourrions pas communiquer de manière continue avec le client. Elle permet notamment d'envoyer les données reçues des drones vers le client et aussi d'envoyer les points calculés de la carte vers le client. Nous remarquons que cette classe comporte une nouvelle méthode depuis la CDR, qui est la méthode *sendLog* et nous sert à envoyer les logs en continu au client. Ensuite, nous avons les deux autres classes de communication, qui s'occupe de communiquer avec les drones. La première est *SocketSimulation*, qui permet de communiquer avec les drones de la simulation. Elle reçoit en continu les données qui sont envoyées par les drones de la simulation. Par la suite, elle s'occupe de communiquer les commandes aux drones. Elle communique avec la simulation ARGoS grâce à un socket, car c'est le meilleur moyen de communiquer en continu sans problème. En outre, nous avons aussi la classe *BaseStation*. Cette classe nous permet de communiquer avec les drones physiques à travers le *CrazyRadio*. Elle initialise d'abord la communication avec les drones puis communique les commandes aux drones lorsque nécessaire. Elle est aussi constamment en écoute des drones, qui envoient leurs données en continu à la *CrazyRadio*.

Finalement, nous avons les autres classes du serveur, que nous pouvons classifier comme des classes de calculs ou de formatage des données. Ces classes

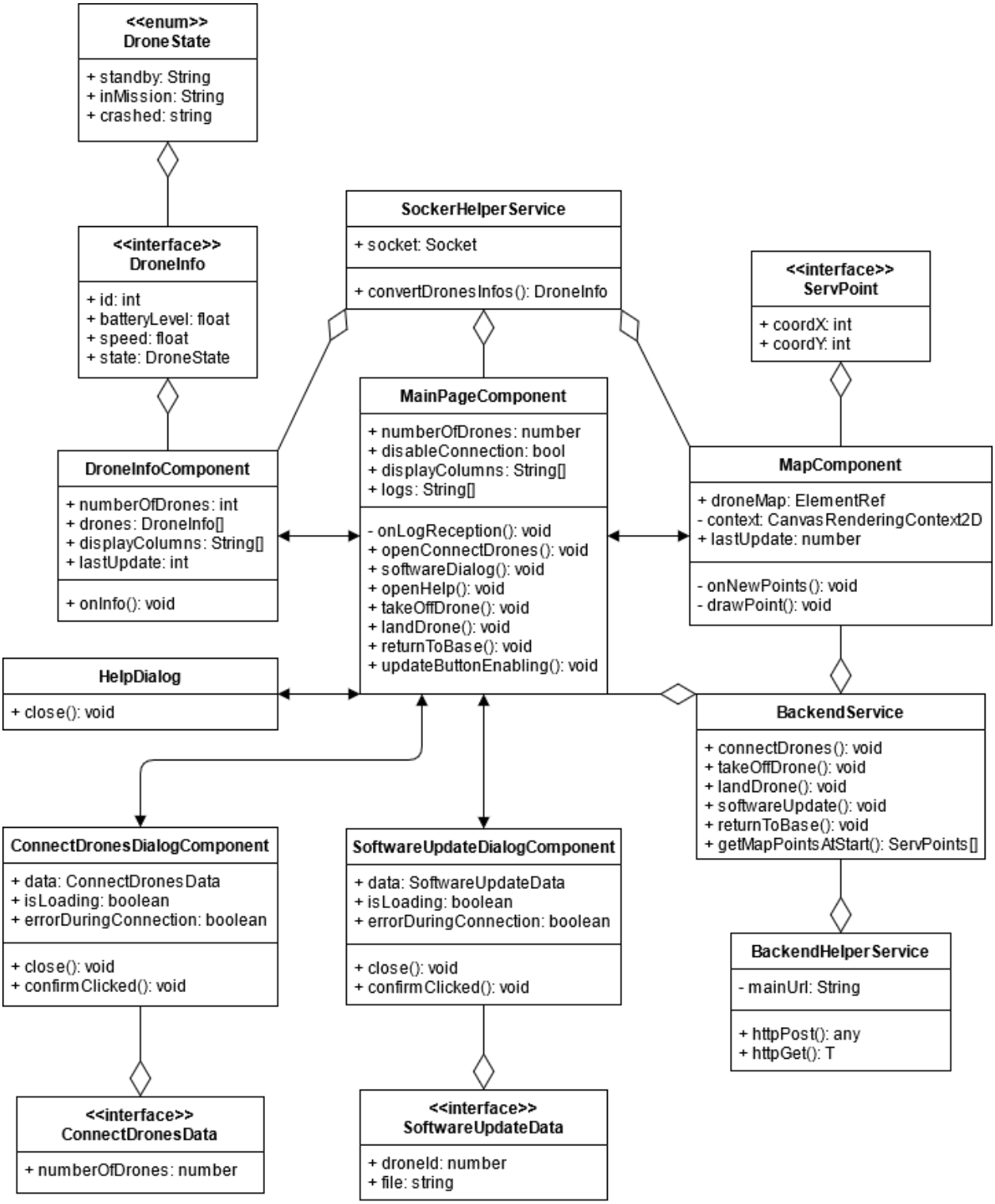
sont les suivantes : *Drone*, *MapPointSaver*, *MapHelper*, *MapPoint* et *Position et custom logger*. La classe *custom logger* est une celle qui permet de créer le fichier de logs, nous avons décidé d'utiliser cette classe à la place du logger de base de Python *logging* [19] car nous voulions avoir des logs plus détaillés et personnalisés. Nous n'avons pas ajouté custom logger dans le diagramme, car il est utilisé dans toutes les classes du serveur et cela rendait le diagramme difficile à lire. Les classes *Drone*, *MapPoint*, et *Position* sont surtout des classes utilisées comme des interfaces. Elles permettent d'encapsuler respectivement les données des drones, des points de la carte et des différentes positions. Grâce à ces classes, les données que nous envoyons au client sont plus compréhensibles et plus propres. Ensuite, nous avons les classes d'aide et de calcul. Ces classes sont *MapPointSaver* et *MapHelper*. La classe *MapPointSaver* nous sert à gérer l'enregistrement des points de la carte dans un fichier CSV [18]. Nous avons décidé de rajouter cette classe et d'enregistrer les points de la carte en cours afin de permettre à l'utilisateur de rafraîchir le client sans perdre les données de la carte. Ensuite, la classe *MapHelper* nous permet de mieux gérer les calculs des points de la carte. Elle permet de transformer les données reçues par les capteurs du drone en point de la carte. Ces points sont ensuite envoyés et affichés sur le client.

On remarque alors plusieurs différences avec ce que nous avions présenté à la CDR. Une des différences notables est le fait que nous n'avons plus la classe *Database*, mais nous l'avons remplacée par la classe *MapPointSaver*. Nous avons réalisé ce changement, car nous avions mal compris un des requis, qui était l'enregistrement des logs. De ce fait, nous n'avons pas de base de données, mais seulement un fichier CSV qui comporte tous les points de la carte en cours d'exploration. Ensuite, un autre changement est l'ajout de méthodes dans *WebRouting* et *SocketClient*. La classe *WebRouting* a deux nouvelles méthodes, *sendPointsAtStart* et *sendCommandToRightClass*. La première méthode nous permet d'envoyer tous les points de la carte enregistrés dans le fichier CSV en un coup. La deuxième méthode nous permet d'éviter de la répétition de code dans les méthodes qui gèrent les requêtes HTTP. Enfin, le dernier changement à noter est le remplacement de la méthode *getInstance* dans *BaseStation* par *setUp*. Ce changement nous permet de reconstruire la classe *BaseStation* lorsque les drones se déconnectent. Cela nous permet de ne pas avoir à relancer notre système au complet lorsqu'un drone se déconnecte et que nous devons le reconnecter. Ce changement est très important, car il permet de faciliter l'utilisation de notre système. De plus, nous remarquons certains ajouts de méthodes dans la classe *BaseStation*. Ces méthodes nous aident à mettre à jour les drones, envoyer des informations entre drone et serveur, puis de faire certains calculs seulement utiles pour les drones physiques.

2.3 L'interface de contrôle (Q4.6)

En ce qui concerne le Client, nous utilisons le cadriciel Angular 11 [9] pour le réaliser. Nous avons décidé d'utiliser Angular, car nous sommes familiers avec ce cadriciel, car il permet d'avoir une interface web esthétiquement propre. Puisque nous utilisons Angular pour faire le client, nous utilisons alors aussi Angular Material [10] pour rendre l'application web plus jolie.

Figure 3: Diagramme de classes du client



La figure 3 illustre nos différentes classes utilisées dans le client avec un diagramme de classe. Les doubles flèches représentent une communication entre deux

composants et les flèches d'agrégation représentent l'utilisation d'une classe par un composant.

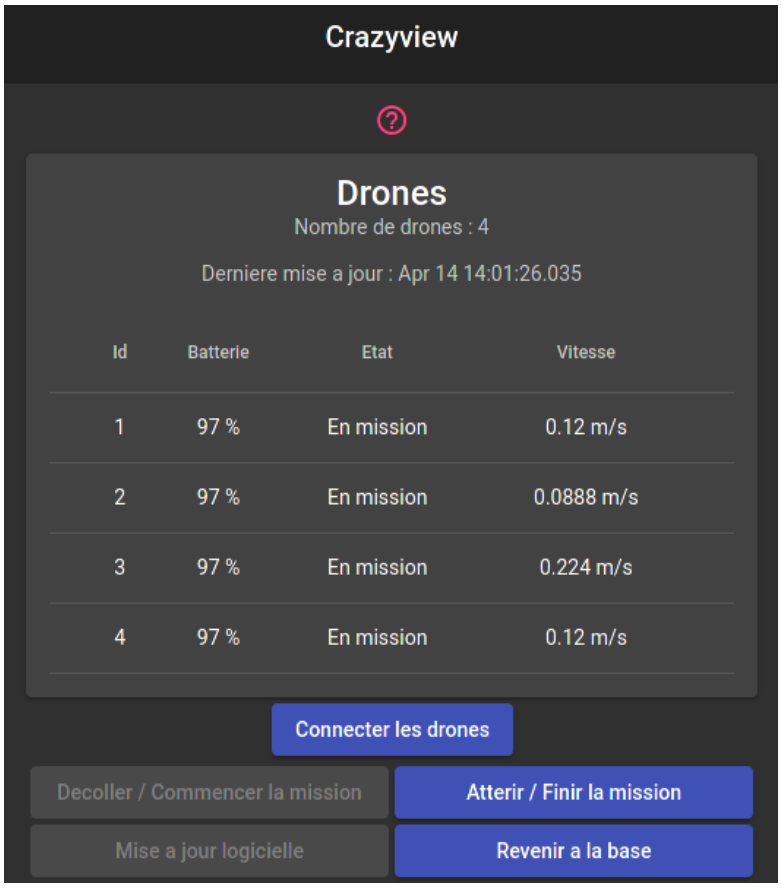
Comme nous pouvons observer sur la figure 3, il y a 3 composants principaux dans notre application web. Ces trois composants sont *DroneInfoComponent*, *MainPageComponent*, et *MapComponent*. Ces trois composants permettent chacun d'afficher une partie différente du client web. Nous avons décidé de séparer l'affichage dans ces trois composants, car cela nous permet de les réutiliser à d'autres endroits, de les déplacer plus facilement, ou même de les modifier plus rapidement. De plus, séparer l'affichage permet aussi d'avoir une meilleure encapsulation de l'affichage. La figure 3 illustre aussi certains changements depuis la CDR. Nous avons rajouté trois nouveaux composants dans l'application pour afficher nos différents dialogues. Ces trois composants sont *ConnectDronesDialogComponent*, *HelpDialogComponent* et *SoftwareUpdateDialogComponent*. Ils permettent respectivement d'afficher le dialogue de connexion des drones, d'afficher le dialogue d'aide qui permet d'expliquer à l'utilisateur comment utiliser notre système d'exploration et d'afficher le dialogue de mise à jour logicielle des drones.

Ensuite, la figure 3 nous permet aussi de comprendre quels sont les différents services utilisés par le client, et quels sont leurs liens avec les composants. Nous observons 3 services différents dans notre application. Ces trois services sont *SocketHelperService*, *BackendService* et *BackendHelperService*. Ces services servent principalement à gérer les différentes communications avec le serveur. Ces services n'ont pas vraiment changé depuis la CDR. Le premier service, *SocketHelperService*, permet de gérer la communication par le socket avec le serveur. Ce service s'occupe d'initier le socket avec le serveur, puis laisse les différents composants gérés la réception de certaines données, comme les données des drones qui sont gérer par le composant *DroneInfoComponent*. Ensuite, nous avons les deux autres services (*BackendService* et *BackendHelperService*) qui gèrent les requêtes HTTP. Le service *BackendService* permet de faire les requêtes nécessaires vers le backend. Le service *BackendHelperService* permet simplement de rajouter une couche d'abstraction à nos requêtes HTTP, et d'éviter la répétition de code lors d'une requête HTTP.

Depuis la CDR, nous avons effectué certains petits changements. Nous l'avons déjà évoqué, mais nous avons rajouté trois composants pour les dialogues. La raison principale est pour bien encapsuler l'affichage de ces dialogues, et d'améliorer la qualité de notre code. Autrement, notre composant *MainPageComponent* utilise maintenant le *SocketHelperService*. Il utilise ce service, car il doit récupérer les logs qui sont envoyés par le serveur pour bien les afficher sur la page. Ensuite, le *MapComponent* utilise maintenant aussi le *BackendService*. Il utilise ce service, car il fait une requête GET vers le serveur a son initialisation afin de récupérer tous les points de la carte déjà générés. Cela nous permet de rafraîchir la page web sans perdre les données de la carte.

Pour l'interface graphique, nous avons choisi de faire quelque chose de clair et épuré. Nous pouvons séparer notre interface en plusieurs parties importantes.

Figure 4: Première partie de l'interface



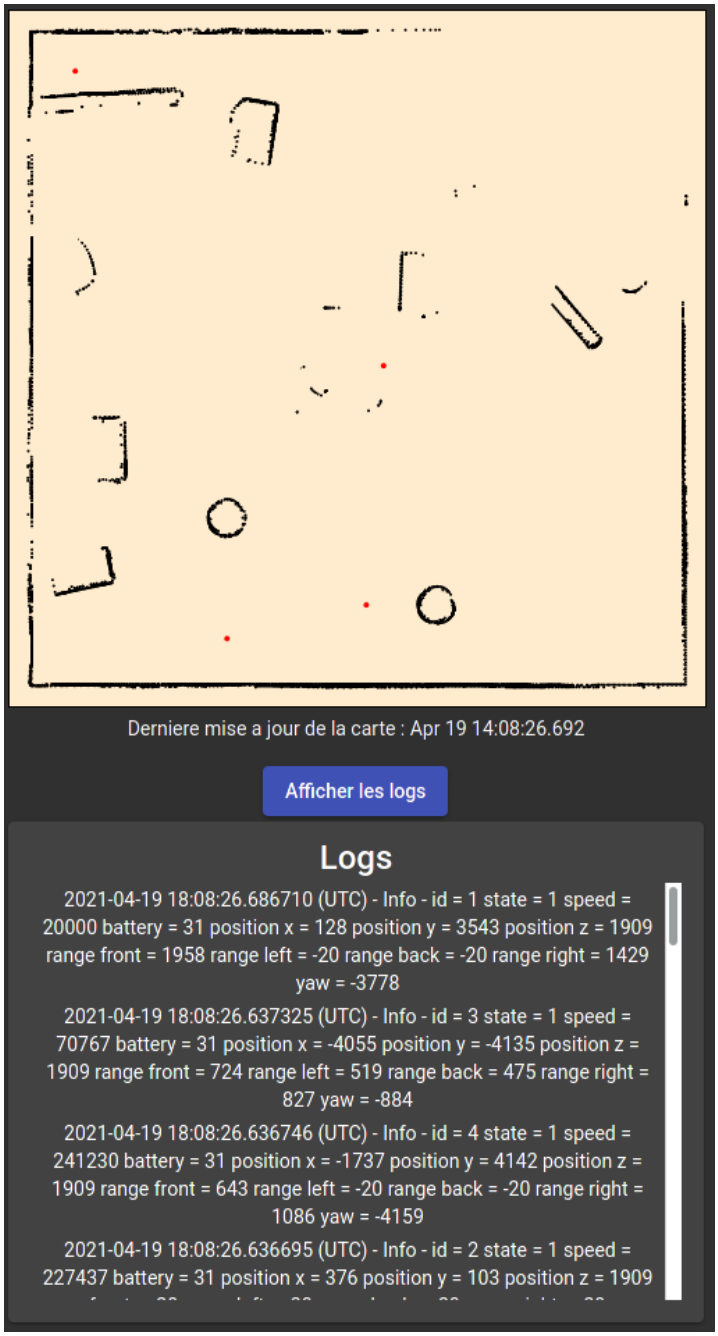
Tout d'abord, nous avons le tableau des drones, qui affiche les différentes informations des drones en temps réel, visible sur la figure 4. Nous retrouvons notamment leurs identifiants (le champ *Id*), leur niveau de batterie, leur état et leur vitesse (en m/s). L'état des drones peut être un des états suivants : en attente, en mission, écrasé, décollage, atterrissage, retour à la base.

Ensuite, nous avons les boutons d'actionnement des drones. Nous pouvons observer ces boutons vers le bas de la figure 4. Ces boutons permettent de contrôler les drones, et de leur envoyer les actions à faire. Les noms des boutons expliquent clairement l'interaction du bouton, comme le bouton Revenir à la Base qui va dire aux drones de revenir à la base.

Par la suite, nous avons la carte d'exploration des drones, que nous observons à la figure 5. Nous avons choisi de construire une carte en 2D vue du haut, car c'est une manière rapide et efficace d'illustrer le terrain exploré.

Finalement, nous avons l'affichage des logs, qui s'observe sur le bas de la figure 5. Ces logs ne sont pas toujours affichés, il faut d'abord cliquer sur le bouton *Afficher les logs* pour les afficher. Lorsque les logs s'affichent, l'utilisateur ne peut qu'observer les 20 derniers logs générés par le système. Si l'utilisateur souhaite observer tous les logs du système, il doit ouvrir le fichier lui-même qui se trouve dans le dossier *logs* dans le serveur.

Figure 5: Deuxième partie de l'interface



2.4 Fonctionnement général (Q5.4)

Grâce à notre script [15] appelé *run.sh* il est très facile de lancer tous les différents systèmes de notre projet. Il y a seulement quelques prérequis pour que le tout fonctionne. Il faut s'assurer que la machine qui lance le tout ait les packages Docker [21] et x11docker d'installés. Pour ce qui est de Docker, assurez-vous que votre utilisateur actuel est dans le groupe Docker et que le service est entrain de fonctionner. Ensuite, puisque l'interface utilisateur se connecte sur le port 4200, il faut s'assurer qu'il soit disponible. Même chose pour le serveur qui roule sur le port 5000. Si toutes ces

conditions sont satisfaites, il devrait être possible d'utiliser toutes les commandes du script.

Pour lancer la simulation, il faut simplement exécuter la commande *./run.sh sim.* Cela lance la simulation, le serveur et l'interface. Après que tout a terminé de charger, une fenêtre devrait s'ouvrir avec la simulation ARGoS. Il faut maintenant cliquer sur le bouton *play* pour mettre les robots en attente d'une commande.

Pour envoyer des commandes, nous nous dirigeons vers l'interface qui devrait se trouver à l'adresse *localhost:4200* de n'importe quel fureteur sur l'ordinateur ou nous venons d'executer la commande *run.sh*. Nous devrions y retrouver les quatre commandes ainsi que des éléments comme la carte, les logs et des informations sur les drones. Pour se connecter depuis un autre ordinateur, ou depuis un téléphone, il faut réaliser une étape supplémentaire. Il faut se rendre dans le fichier *environment.ts* qui se trouve dans le dossier *crazysystem/Client/crazyview/src/environments* et changer la variable *LOCAL_IP*. Il suffit de mettre l'adresse IP locale de la station au sol. Vous pouvez trouver cette adresse IP avec la commande *ifconfig*. Ensuite, pour se connecter sur l'interface depuis un autre ordinateur, il faut simplement se rendre sur l'adresse suivante « adresseIP:4200 », ou « adresseIP » correspond à l'adresse IP que vous venez de rentrer dans la variable *LOCAL_IP*.

Pour utiliser les drones, il faut d'abord téléverser une première version de notre code dessus. Pour faire cela, nous avons créé la commande *run.sh flash all* qui permet de compiler et automatiquement téléverser le code. Ensuite, quand le code est téléversé, nous pouvons faire un *run.sh all* qui permet de lancer l'interface et le serveur. Dès que tout est prêt, il y a une séquence de mise en place des drones à suivre pour que tout fonctionne. En effet, comme expliqué dans la section des drones, ils s'autoassignent leurs IDs en fonction des autres drones, grâce au *Peer To Peer* (P2P) [2]. Les instructions se trouvent dans notre interface web. Notez que les commandes *Flash* et *all* nécessitent un paramètre supplémentaire à la fin de la commande, qui est le chemin vers le port USB où se trouve la Crazyradio.

Si jamais certaines informations ne sont pas claires, il est toujours possible de se référer au *README.md* qui donne plus d'informations sur les commandes.

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

Après avoir effectué plusieurs tests pour vérifier le bon fonctionnement de notre système, nous pensons que nous avons un produit final répondant bien aux requis. Il y a quand même certains points qui ne répondent pas directement aux requis.

Pour commencer, nous n'avons pas utilisé le RSSI [4] pour permettre aux drones de s'orienter. Celui-ci n'était pas assez précis et il rendait des fonctionnalités comme le retour à la base impossible à réaliser. Les drones recevaient un signal leur disant qu'ils étaient au-dessus de la base, même s'ils étaient à quelques mètres. Nous n'avons donc pas utilisé cette fonctionnalité et plutôt nous gardons en mémoire la position de début du drone et nous lui demandons de revenir à cette position.

Ensuite, notre algorithme d'exploration n'est peut-être pas optimal. En effet, puisqu'il se base sur une exploration aléatoire, il peut exister des scénarios où certaines parties de la carte ne seront pas explorées. Puisque la carte est généralement assez petite par rapport à la distance vue par les capteurs, il est assez rare qu'un endroit ne sera pas exploré. Cela reste une possibilité qu'il ne faut pas oublier.

Aussi, notre retour à la base contient une grosse lacune. Son fonctionnement est le suivant : Le drone s'oriente vers la base, il s'y dirige, s'il voit un mur, il se déplace à gauche en vérifiant s'il ne rentre pas dans un mur. S'il croise un mur, il se réoriente vers la base et il recommence le processus. S'il se réoriente et il y a encore un obstacle devant lui et à sa gauche, il ne peut pas revenir et il atterrit. Nous avons décidé de ne pas prendre en compte ce cas puisque les obstacles sont généralement loin les uns des autres ce qui réduit énormément les chances que cette situation arrive. C'est toutefois une situation possible qui peut arriver. De plus, lorsque le drone prend trop de temps pour revenir à la base dans la simulation, il passe parfois en dessous du sol. Bien entendu, ce comportement n'est pas désiré, mais puisque ce problème se produit seulement dans la simulation et que c'est un cas isolé, nous avons décidé de ne pas investir de temps dans sa résolution.

Enfin, nous avons fait plusieurs tests pour vérifier le comportement de notre système quand plusieurs clients se connectaient en même temps à l'interface. On se rend compte que si deux clients pèsent en même temps sur des boutons comme *connect* ou *software update*, il est possible d'avoir des résultats non-attendus. Cette situation ne devrait cependant pas se produire et c'est donc pourquoi nous n'avons pas investi de temps pour sa résolution.

4. Déroulement du projet (Q2.5)

Maintenant que le projet est terminé, nous pouvons évaluer si nous avons respecté notre planification. Plusieurs éléments seront évalués, comme la réussite des tests unitaires, l'organisation globale de notre équipe, la gestion des risques et le respect de notre estimation des coûts.

Pour commencer, dans l'appel d'offre, nous avions prévu de faire des tests unitaires pour chacun des systèmes de notre projet. Nous avons bien respecté cette planification pour le frontend et pour la majorité du backend, mais pas pour la simulation, les drones et la librairie. En effet, en cours de projet, nous avons réalisé que ceux-ci ne seraient pas aussi simples que prévus. Pour la simulation, ARGoS ne fournit pas une façon facile de faire des tests unitaires. Même chose pour les drones puisqu'il faudrait charger les tests sur les drones. Pour ce qui est de la librairie, nous tombons face au même problème puisque nous ne pouvons pas "simuler la simulation" ou les drones. La librairie que nous avions prévu utiliser, *CUnit* [17], ne permettait pas de faire ce que nous avions prévu. Pour pallier cela, nous avons fait des tests fonctionnels plus exhaustif. Par exemple, pour la simulation, nous avons généré des expériences permettant de tester chaque commande individuellement. Au lieu de rouler des tests unitaires après des changements, nous roulons ces tests fonctionnels et s'ils passent, nous pouvons assumer que les algorithmes de notre système fonctionnent

correctement. Nous pouvons assumer que si les tests passent pour la simulation, les drones physiques devraient fonctionner correctement. Nous nous assurons néanmoins de faire des expériences pour vérifier leur bon fonctionnement.

Certaines parties du backend n'ont pas pu être testées aussi. Par exemple, certaines fonctions utilisant les sockets faisaient usage de boucles infinies, ce qui rendait les tests très complexes. Nous avons donc préféré investir notre temps dans des tests fonctionnels et manuels plutôt que de passer beaucoup de temps à tester unitairement ces fonctions.

Pour le frontend, presque tout a été testé à l'aide de *Jasmine* [13]. Les seules lignes n'étant pas testées sont celles qui utilisent les sockets. Nous ne voulions pas passer trop de temps à simuler une réponse du backend pour le *callback* des sockets et avons donc préféré laisser ces méthodes non testées.

Ensuite, nous avions prévu faire des tests de régression à chaque fin de sprint. Ce plan a vite été abandonné lorsque nous avons réalisé que la quantité de travail ne nous permettait pas de faire cela. Ces tests auraient pris beaucoup de temps et nous n'aurions pas pu respecter notre budget de temps en ajoutant cela à notre planification déjà bien chargée. À la place, nous avons fait des tests vers la fin du projet. Cela nous a permis de tester notre projet final et de faire des modifications dans le cas approprié.

Côté organisation de l'équipe, nous pensons avoir bien respecté ce que nous avions prévu. Chaque semaine, nous organisions des rencontres qui nous permettaient de faire le point sur la semaine passée et prévoir la semaine suivante. Nous pouvions discuter des problèmes rencontrés et poser des questions si nous en avions. Aussi, le principe de *merge requests* [11] a bien été utilisé pour toute la durée du projet, nous permettant de n'avoir que du code fonctionnel et approuvé sur notre branche principale.

L'un des points que nous n'avions pas prévus originalement est le fait que seulement une personne avait accès aux drones physiques. Nous savions que Philippe serait celui qui manipulerait les drones, mais n'avions pas prévu une façon optimale de fonctionner pour ne pas lui donner trop de travail. Finalement, nous écrivions du code de notre côté et lui demandions de tester le tout, et grâce à notre script de lancement, cela ne lui prenait pas trop de temps. Si nous n'étions pas dans cette situation sanitaire actuelle, le moyen le plus optimal serait de manipuler les drones en présentiel avec toute l'équipe, comme en projet 1.

Enfin, au début du projet, nous avions prévu un total de 616 heures au total. Pour finir, nous avons enregistré 495 heures sur GitLab [12], ce qui est bien en dessous de notre planification. Cependant, nous avons parfois oublié d'enregistrer certaines heures, ou même nous avons fait des résolutions de bogue sans enregistrer les heures. De ce fait, nous estimons notre temps total sur le projet aux alentours des 550 heures de travail. Cela reste quand même moins d'heure que prévu. Nous avons donc surestimé le temps de certaines tâches et d'autres tâches ont simplement été annulées ou changées en cours de route.

5. Travaux futurs et recommandations (Q3.5)

Comme dit précédemment, notre système répond à la plupart des requis demandés. Il y a toute de même quelques améliorations possibles.

Premièrement, notre retour à la base pourrait être grandement amélioré. Comme expliqué précédemment, notre robot bloque s'il rencontre un second obstacle en tentant d'éviter le premier et qu'il n'a pas une ligne de vue sur son point initial. Si nous voulions perfectionner notre système, nous devrions faire demi-tour et aller à droite du premier obstacle. Nous prendrons donc en compte plus de situations. Cependant, s'il y a un autre obstacle de l'autre côté, nous atterrissons où nous sommes en ce moment. Bien sûr, nous pourrions faire un algorithme complexe qui trouve un chemin vers la base peu importe les obstacles autour de lui, mais cela est un problème très complexe et surtout, nous voulons que le robot atterrisse le plus rapidement possible quand le retour à la base est appelé puisque le robot n'a potentiellement plus beaucoup de batterie (Requis retour à la base à 30% de batterie).

Ensuite, nous pourrions améliorer notre algorithme d'exploration pour que les robots couvrent une plus grosse zone lorsqu'ils explorent. Nous pourrions par exemple garder en mémoire les endroits où la recherche est déjà faite et la communiquer aux autres drones. Ensuite, les drones pourraient planifier leur trajectoire pour visiter des endroits non-explorés. Cela serait une grande amélioration à notre algorithme, car en ce moment, le caractère aléatoire de l'exploration fait que certaines zones peuvent demeurer inexplorées pendant un très long moment.

Il y a aussi un requis optionnel que nous aurions pu réaliser. Ce requis était d'afficher un éditeur de code à l'écran. Cela aurait permis de modifier le code et flasher les robots directement sans avoir à redémarrer tous les systèmes. Cependant, nous avons décidé de ne pas le faire par manque de temps.

6. Apprentissage continu (Q12)

Theo : Au début du projet, je ne savais pas ce qu'était Docker. J'ai dû apprendre ce qu'était Docker afin de pouvoir l'utiliser pour notre projet. De plus, certains cours que j'avais en parallèle m'ont permis de mieux comprendre ce qu'est Docker. Grâce à cela, j'ai compris ce qu'était Docker et j'ai pu exercer mes nouvelles connaissances dans ce projet. En outre, certains membres de l'équipe avaient déjà de l'expérience avec Docker, et ils ont pu relire et m'indiquer mes erreurs lors des *merge requests*. J'avais aussi certaines lacunes en C/C++ et en Python. J'ai pu améliorer ces lacunes en m'exerçant et en faisant des recherches sur internet qui m'ont guidé vers les bonnes solutions. Je peux encore m'améliorer dans ces langages en m'exerçant encore plus, et en suivant certains cours sur internet afin de me perfectionner dans ces langages.

William : Au début du projet, je n'étais pas familier du tout avec la conception de systèmes en python ainsi que la conception de serveurs. Je n'avais jamais travaillé avec ces technologies et n'étais pas très efficace. J'ai donc laissé cette partie pour les autres membres de l'équipe qui s'y connaissaient mieux. J'ai pu analyser leur code et mieux comprendre ces concepts. J'ai aussi travaillé de mon côté sur des projets

personnels utilisant des serveurs en python. Maintenant que nous sommes à la fin du projet, j'ai une très bonne compréhension de la conception de systèmes en python ainsi que des serveurs. Si j'avais voulu rendre cet apprentissage plus rapide, j'aurais pu prendre la tâche de concevoir le serveur dès le début. J'aurais donc été obligé d'apprendre tous ces concepts rapidement pour ne pas retarder mon équipe. Ce même phénomène s'est produit avec plusieurs technologies, comme *Docker* et *Gitlab*.

Nour : Au début du projet, je n'avais pas beaucoup d'expérience en Python. Je connaissais le langage en général et j'avais fait quelques travaux pratiques dans ce langage, mais je n'avais jamais fait un projet au complet en python. Quand j'ai dû faire des parties de code dans le backend, j'ai fait un peu de recherche et j'ai aussi demandé de l'aide aux autres membres de l'équipe. Je me suis aussi inspiré du code des autres membres de l'équipe en l'analysant. J'aurais pu améliorer mon apprentissage en essayant de faire des projets personnels en Python pour mieux comprendre la conception d'un système en Python. Je ne savais pas non plus comment les docker files marchait et comment les utiliser, j'ai donc laissé les taches à d'autres membres qui s'y connaissaient un peu plus que moi. J'ai analysé ce qu'ils ont fait et je suivais aussi un cours où nous avons appris à créer des conteneurs avec docker. J'aurais pu essayer de faire les taches moi-même en faisant des recherches pour améliorer mon apprentissage.

Philippe : Au début du projet, je n'avais jamais fait de python et j'avais des lacunes en C/C++. Mais, je me suis mis dès le début sur le code du serveur en python, et pour y arriver, j'ai lu beaucoup de documents de python et des tutoriels. Grâce à tout cela, j'ai réussi à bien apprendre le python. Pour le C/C++, j'ai amélioré mes connaissances grâce au fait que j'avais les drones, et donc je devais beaucoup toucher à notre librairie et le code embarqué, ainsi que la documentation de Crazyflie. Niveau python, je ne vois pas ce que j'aurais pu faire de plus pour améliorer mon apprentissage. Mais, du côté du code C/C++, j'aurais pu participer à la conception de la libraire et plus m'investir dans la création de notre simulation en C/C++.

Nicolas: Au début du projet, je possédais déjà des connaissances de base en Python, en C/C++ et en Angular. Toutefois, me connaissances dans ces langages étaient très basiques. De plus, je n'avais pas d'expérience en gestion et planification de projet et je n'avais aucune expérience avec GitLab et joce n'avais pas de connaissances avec Docker. Dès le début du projet, je me suis mis à travailler avec GitLab pour me familiariser avec les fonctionnalités de cet outil. Avec l'aide de Philippe et de Theo qui m'ont grandement aidé, j'ai rapidement progressé avec GitLab et j'ai rapidement appris à bien utiliser cet outil. En apprenant à utiliser GitLab, j'ai appris en parallèle comment faire de la gestion de projet à l'aide de la matière du cours et à l'aide de mes coéquipiers. Pour ce qui est de Docker et des conteneurs, c'est principalement William et Philippe qui se sont occupés de cette partie, mais j'ai touché un peu sur cette partie en leur demandant des conseils. De plus, lorsque j'étais bloqué, je faisais des recherches par moi-même sur internet pour aller lire la documentation existante. Finalement, j'ai principalement approfondi mes connaissances en Python puisque la majorité de mes tâches concernaient le serveur. J'ai donc principalement appris des choses sur Python, par exemple: faire un socket de communication, faire des classes et des classes singleton, bien utiliser des fichiers .log et .csv, etc. Toutefois, j'aurais dû

augmenter ma participation pour ce qui est du développement du drone (C/C++) ainsi que du frontend (Angular) pour approfondir ces connaissances.

7. Conclusion (Q3.6)

Pour conclure, l'organisation de l'équipe a été efficace dans son ensemble. Nous avons très bien réussi à utiliser l'outil de gestion qui est GitLab. Nous avons réussi à bien estimer le temps de nos tâches malgré le fait que nous avons réalisé moins d'heures que prévues sur le projet dans sa globalité. Nous avons donc bien réussi à nous organiser pour concevoir notre système, et nous sommes satisfaits du système que nous avons réalisé. De plus, nous avons été efficaces pour concevoir ce projet. Nous avons réussi à utiliser les outils rapidement et sans trop de problèmes.

Ensuite, nous avons réussi à construire notre librairie commune entre la simulation et les drones physiques. Cependant, cette librairie nous a demandé plus de temps que prévu à la concevoir, car la compatibilité entre les deux langage C et C++ nous a demandé à revoir certains aspects de cette librairie. Malgré cela, nous avons quand même réussi à la développer, et elle nous permet alors de ne pas répéter les algorithmes communs entre la simulation et les drones physiques. La conception de cette librairie et son utilisation permanente est donc une très grande réussite pour l'ensemble de l'équipe.

Autrement, maintenant, que le système est complété, nous pouvons dire que nous n'avons pas forcément fait les bons choix de cadriciel au début du projet en ce qui concerne la simulation ou les drones embarqué. En effet, nous n'avons pas réussi à développer des tests unitaires que ça soit pour la simulation ou pour les drones embarqués, car nous n'avons pas pensé à cela lors de la création du projet.

Finalement, nous pouvons dire que nous avons réussi à faire plus que prévu initialement. En effet, nous avons réussi à rajouter la position des drones sur la carte dans le client. Nous n'avons pas prévu de faire cela initialement, c'est un bonus pour l'équipe d'avoir réussi à le faire. De plus, nous avons aussi ajouté un bouton pour connecter les drones pour que ce soit facile à les connecter pour un utilisateur. Initialement, nous ne pensons pas faire ce bouton, mais nous avons très vite implémenté celui-ci pour nous permettre de connecter les drones plus rapidement et plus facilement.

8. Références

[1] Bitcraze. (2021) Stem Ranging Bundle. [En ligne]. Disponible : https://store.bitcraze.io/products/stem-ranging-bundle

[2] "Peer to peer", dans *Wikipédia,* 27 fév. 2021. [En ligne]. Disponible : https://en.wikipedia.org/wiki/Peer-to-peer

[3] Pinciroli, C. (s.d.) ARGoS. [En ligne]. Disponible : https://www.argos-sim.info/

[4] "Received signal strength indication", dans *Wikipédia,* 1 fév. 2021. [En ligne]. Disponible : https://en.wikipedia.org/wiki/Received_signal_strength_indication

[5] Bitcraze. (2020) Peer to Peer API. [En ligne]. Disponible : https://www.bitcraze.io/documentation/repository/crazyflie-firmware/2020.02/p2p_api/

[6] IBM (s.d.) What is a socket? [En ligne]. Disponible : https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.cbcpx01/ovsock.htm

[7] Redhat. (2021) What is a REST API? [En ligne]. Disponible : https://www.redhat.com/en/topics/api/what-is-a-rest-api

[8] "Python (programming language)", dans *Wikipédia*, 6 mars 2021. [En ligne]. Disponible : https://en.wikipedia.org/wiki/Python_(programming_language)

[9] Angular. (s.d.) Angular. [En ligne]. Disponible : https://angular.io/

[10] Angular. (s.d.) Angular Material. [En ligne]. Disponible : https://material.angular.io/

[11] GitLab. (s.d.) Merge requests. [En ligne]. Disponible : https://docs.gitlab.com/ee/user/project/merge_requests/

[12] GitLab. (s.d.) About GitLab. [En ligne]. Disponible :https://about.gitlab.com/company/

[13] "Jasmine (JavaScript testing framework)", dans *Wikipédia*, 22 déc. 2020. [En ligne]. Disponible : https://en.wikipedia.org/wiki/Jasmine_(JavaScript_testing_framework)

[14] Docker. (2021) Dockerfile reference. [En ligne]. Disponible : https://docs.docker.com/engine/reference/builder/

[15] "Shell script", dans *Wikipédia*, 30 déc. 2020. [En ligne]. Disponible : https://en.wikipedia.org/wiki/Shell_script

[16] "C++", dans *Wikipédia*, 28 fév. 2021. [En ligne]. Disponible : https://en.wikipedia.org/wiki/C%2B%2B

[17] CUnit. (s.d.) CUnit. [En ligne]. Disponible : http://cunit.sourceforge.net/

[18] " Comma-separated values", dans *Wikipedia,* 29 sept. 2020. [En ligne]. Disponible: https://fr.wikipedia.org/wiki/Comma-separated_values

[19] Python. (s.d.) logging - Logging facility for Python [En ligne]. Disponible: https://docs.python.org/3/library/logging.html

[20] Bitcraze (2021) Table of content (TOC). [En ligne]. Disponible : https://www.bitcraze.io/documentation/repository/crazyflie-firmware/2019.09/logparam/

[21] Docker (2021) Docker. [En ligne]. Disponible : https://www.docker.com/

[22] "C (programming language)", dans *Wikipédia*, 20 avril 2021. [En ligne]. Disponible : https://en.wikipedia.org/wiki/C_(programming_language)

Annexe 1

Pour visualiser notre système en fonctionnement sur les drones physiques, nous avons réalisé une vidéo. Vous pouvez visionner cette vidéo à l'adresse suivante :

https://youtu.be/zcl6CX4cGpY