POLYTECHNIQUE MONTRÉAL

UNIVERSITÉ D'INGÉNIERIE

Département de génie informatique et génie logiciel

INF3995

Projet de conception d'un système informatique

Critical Design Review
H2021-INF3995 du département GIGL.

Conception d'un système aérien minimal pour exploration

Équipe No. 200

Babin
Lauzon
Lharch
Martineau
Quiquempoix

8 Mars 2021

Table des matières

1. Vue d'ensemble du projet

1.1 But du projet, porté et objectifs

Le but de ce projet est de créer un ensemble de logiciels qui permettront à un essaim de drones miniatures de cartographier l'espace d'un bâtiment. Les drones vont cartographier la salle grâce à des capteurs laser et doivent se déplacer de manière autonome. Les drones utilisés sont des Bitcraze Crazyflie 2.1 [1] qui communiquerons entre eux en *peer-to-peer* (P2P) [2] et avec une station au sol munie d'un Crazyradio PA [1].

La station au sol hébergera une interface web qui permettra d'afficher la cartographie de la salle et d'envoyer des commandes basiques aux drones. L'objectif général du projet est d'avoir un essaim de drones autonomes capable d'explorer l'espace d'un bâtiment et de retourner les données de l'exploration à une console centrale.

Au début de ce projet, nous avons commencé par la PDR (Preliminary Design Review). La PDR consiste à remettre un prototype minimal avec une simulation ARGoS [3] de deux drones qui suivent un parcours. Ce livrable nécessite aussi de remettre une interface Web de la station au sol qui montre le niveau de batterie et qui communique avec les drones pour allumer et éteindre leur DEL (vous pouvez trouver en annexe une vidéo de démonstration de prototype).

Pour le deuxième livrable, qui est le CDR (Critical Design Review), l'objectif est de remettre un système avec toutes les fonctionnalités demandées pour la simulation. Ainsi, nous aurons une simulation des drones qui volent dans un environnement aléatoire en évitant les obstacles grâce à des capteurs lasers. Le système comportera aussi un serveur Web avec les commandes pour décoller et revenir à la base. Le serveur web devra aussi afficher les informations des drones ainsi que la carte générée par ces drones.

Enfin, le dernier livrable est le RR (Readiness Review). L'objectif de ce livrable est de remettre un système complètement fonctionnel avec tous les logiciels et la documentation. Nous implémenterons donc le code des drones simulés sur les drones physiques. De plus, nous aurons tous les boutons demandés sur la partie web.

1.2 Hypothèses et contraintes

Notre première hypothèse, est que nous allons avoir besoin de seulement quatre modules : une interface web, un backend sur la station au sol, une programmation embarquée sur chacun des drones physiques et la simulation des drones.

Nous planifions de faire l'interface web en Angular [11] puisque nous possédons déjà des connaissances sur cette plateforme et cela facilitera le développement de ce module. Ensuite, pour ce qui est du backend, il sera responsable de collecter les données que les drones vont envoyer, de faire les

calculs pour la génération de la carte et de transférer les données transformées à l'interface web. Le backend sera codé en python [10], car nous voulons qu'il soit performant et simple à développer avec un langage facile d'utilisation. De plus, les drones Crazyflie possèdent déjà une librairie Python, ce qui nous a permis de finaliser notre choix sur ce langage. Enfin, nous savons que le logiciel pour les drones (le système embarqué) sera en C/C++ [25]. De plus, la simulation sera elle aussi programmée en C/C++.

Pour ce qui est des contraintes, la station au sol devra être capable de communiquer avec les drones par l'intermédiaire de la *CrazyRadio* PA sur un canal de communication de 2.4 GHz. Nous supposons qu'au moins un drone est toujours en communication avec la *CrazyRadio* PA de la station au sol. De plus, la station au sol devra aussi héberger une interface web qui servira d'interface pour l'utilisateur. Cette interface web devra être visualisable par plusieurs appareils (ordinateurs, appareils mobiles, etc.) en même temps via le réseau. Cette interface devra afficher des informations sur les drones, afficher la carte générée par les drones en temps réel et donner quelques options à l'usager pour le contrôle des drones : décoller, atterrir, mettre à jour le logiciel des drones et retour à la base. Les informations qui devront être présente sur l'interface web devront être mis à jour à une fréquence minimale de 1 Hz. Pour la réception et le traitement des informations recueilli par les drones, elles devront être effectuées par la station au sol. De plus, pour faciliter la génération de la carte, la position et l'orientation des drones au début de la mission doivent être connues par la station. De plus, la distance des drones par rapport à la station au sol devra être estimée à l'aide de la puissance du signal (RSSI [4]) reçu par le *CrazyRadio* PA. Le backend devra générer des registres pour permettre la vérification du bon fonctionnement du système (collecte et traitement des données).

Le prototype doit être réalisé à l'aide de deux drone *Crazyflie* 2.1 avec le « *STEM Ranging bundle* » complètement installé. Toutefois, le système doit être capable de supporter un nombre arbitraire de drones. Les drones devront avoir une programmation embarquée pour fonctionner automatiquement sans contrôle de l'utilisateur une fois que la mission est commencée. L'algorithme d'exploration n'a pas de contrainte, mais nous planifions d'utiliser un algorithme d'exploration aléatoire. Une fois la mission commencée, les drones devront être capables d'explorer une pièce d'une superficie maximale de $100\,m^2$ et devront pouvoir éviter les obstacles automatiquement. Un seul drone sera constamment connecté à la station au sol à l'aide de la *Crazyradio* PA. Les autres drones devront faire de la communication *peer-to-peer* à l'aide de la librairie P2P [5] de *BitCraze*. Enfin, une contrainte que nous pourrions avoir serait un problème de surcharge si plusieurs personnes se connectent à l'interface web. On pourrait avoir un problème de surcharge si trop de clients font de requêtes en même temps, le serveur ne pourrait potentiellement pas pouvoir répondre à tout le monde avec des délais raisonnables. On pourrait aussi avoir des problèmes de

concurrence si deux clients s'amusent à débuter et arrêter la mission en même temps.

1.3 Biens livrables du projet

Nous allons générer plusieurs artefacts durant le projet. À chaque livrable, nous avons un certain nombre d'artefacts à remettre. Pour le livrable de *Critical Design Review* (CDR), le premier artefact à remettre est le serveur web interfacé avec la simulation *ARGoS* affichant les différentes métriques des drones et la carte générée par les drones. Le deuxième artefact est le code embarqué sur les drones qui envoie les mesures du *ranging deck* [6] à la station au sol. Le dernier artefact est la simulation *ARGoS* des 4 drones qui explore un environnement tout en évitant les obstacles. Les artefacts du premier livrable sont à remettre le 8 mars 2021. Pour le livrable final, nous avons tous les logiciels, embarqués et non embarqués, avec la documentation comme premiers artefacts. Ensuite, nous devons remettre les drones qui explorent un environnement en évitant les obstacles ainsi que l'interface web a la station au sol qui affiche la carte. Enfin, une vidéo qui montre le fonctionnement du système en simulation et avec les vrais drones. Ces artefacts sont à remettre pour le 12 avril 2021.

Tableau 1 : Biens livrables du projet

Milestones	Artefacts	Date de publication
Critical Design Review	L'interface web avec : • Informations des drones. • Carte générée par les drones.	8 mars 2021
	Le code embarque des drones qui envoie les mesures du ranging deck.	
	Simulation ARGos avec 4 drones qui évitent les obstacles.	
Readiness Review	Tous les logiciels avec la documentation (le code) : • Code embarqué. • Code du serveur (Backend). • Code de l'interface web (frontend).	12 avril 2021
	Les drones qui explorent un espace tout en évitant les obstacles.	
	L'interface web qui affiche la carte générée par les vrais robots	
	Une vidéo qui montre le fonctionnement de notre système.	

2. Organisation du projet

2.1 Structure d'organisation

Notre équipe se compose de 5 membres. Nous avons un coordonnateur de projet qui est Philippe Babin et 4 développeurs-analystes. Nous avons décidé que le rôle de coordonnateur est purement symbolique puisque notre équipe travaillera en réseau décentralisé. Toutefois, le coordonnateur du projet est celui qui a les drones et qui va s'occuper des taches qui demandent la manipulation des drones. Les autres membres de l'équipe vont s'occuper du frontend, backend et aussi la partie de simulation avec *ARGoS*.

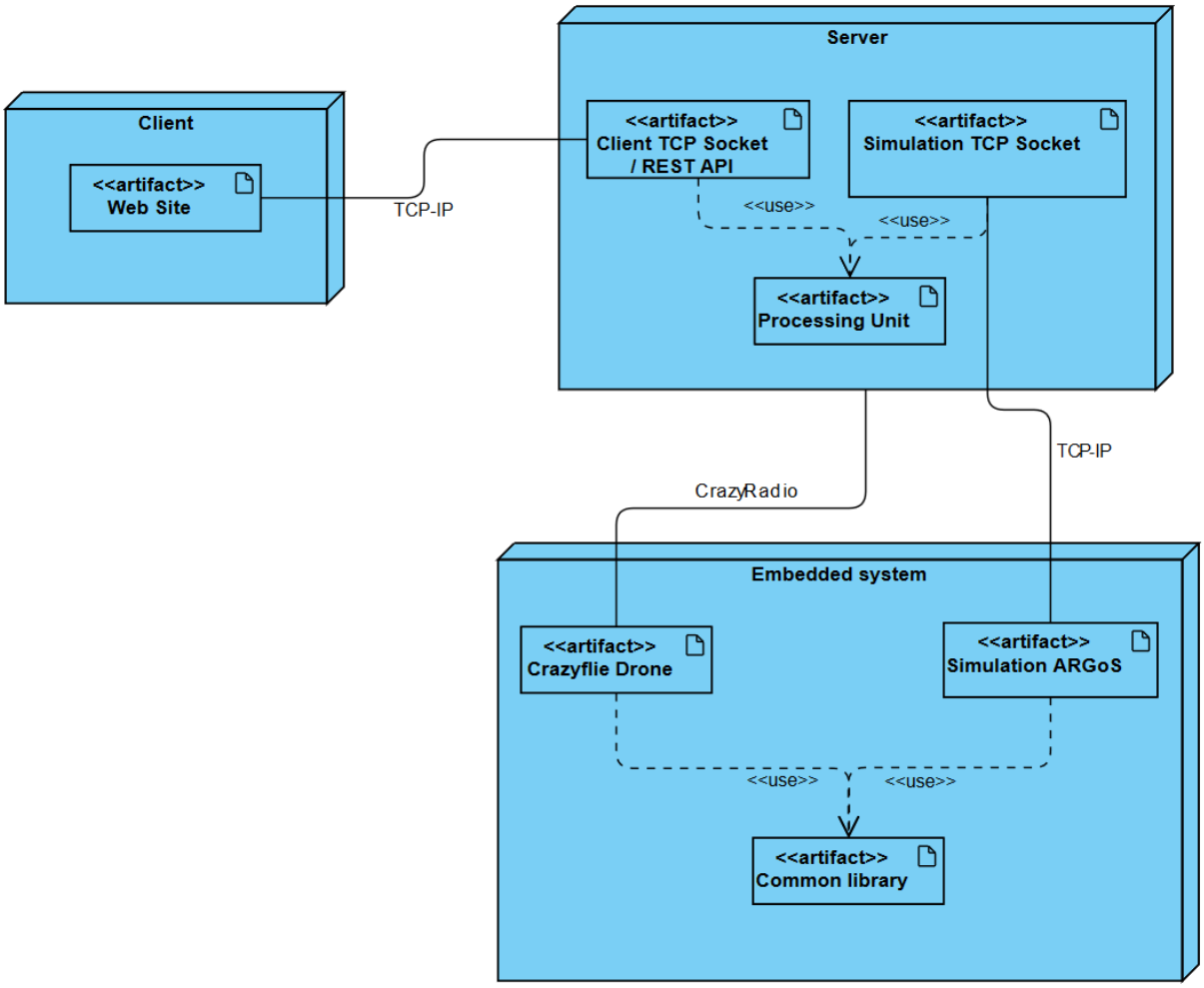
2.2 Entente contractuelle

Dans le cadre de ce projet, le type de contrat qui convient le mieux avec le contexte est le contrat livraison clé en main – Prix ferme. Nous avons choisi ce type de contrat, car c'est un contrat où le contracteur doit livrer un produit final et le paiement se fait après la livraison du produit par le contracteur et l'acceptation du client. Dans notre cas, le produit final est un système qui se compose de drones avec un logiciel embarqué, une interface web et enfin un logiciel *backend* qui s'occupe de générer la cartographie. Nous devons aussi faire une estimation des coûts du projet avec un budget en fonction des lots de travail et une limite pour la charge de travail du projet qui est 630 heures par personne, ce qui veut dire que le prix du projet est ferme. Pendant ce projet, nous avons deux livrables à remettre donc seulement deux dates cibles et ce type de contrat requiert un suivi minimal des travaux par le promoteur, ce qui explique notre choix. Enfin, pour le contrat livraison clé en main – Prix ferme, nous avons besoin de connaître exactement ce qui est demandé et d'avoir des spécifications détaillées pour pouvoir obtenir le contrat. L'appel d'offre et le document des exigences techniques nous donnent toutes les spécifications dont nous avons besoin pour faire ce projet. [7]

3. Solution proposée

3.1 Architecture logicielle générale

Figure 1: Diagramme de déploiement générale du système



Pour concevoir le système, nous l'avons divisé en 3 gros groupes principaux qu'on peut observer sur la figure 1. Chaque groupe du système a un rôle important et a des rôles bien définis. Ces trois groupes sont les suivants : le client, le serveur (server) et le système embarqué (Embedded system).

Tout d'abord, il y a le client. Il s'occupe tout simplement de la partie web du projet. Il va donc faire l'affichage des informations des drones, demandé par le requis R.F.5 ainsi que l'affichage de la carte demandé aussi par le requis R.F.5.

Ensuite, nous avons le serveur. Ce module permet de lier les drones à l'affichage sur le client. Il communique de deux manières avec le client : à travers un socket [8] et à travers des requêtes REST/HTTP [9]. Le socket nous permet de communiquer les informations que nous cherchons à récupérer rapidement,

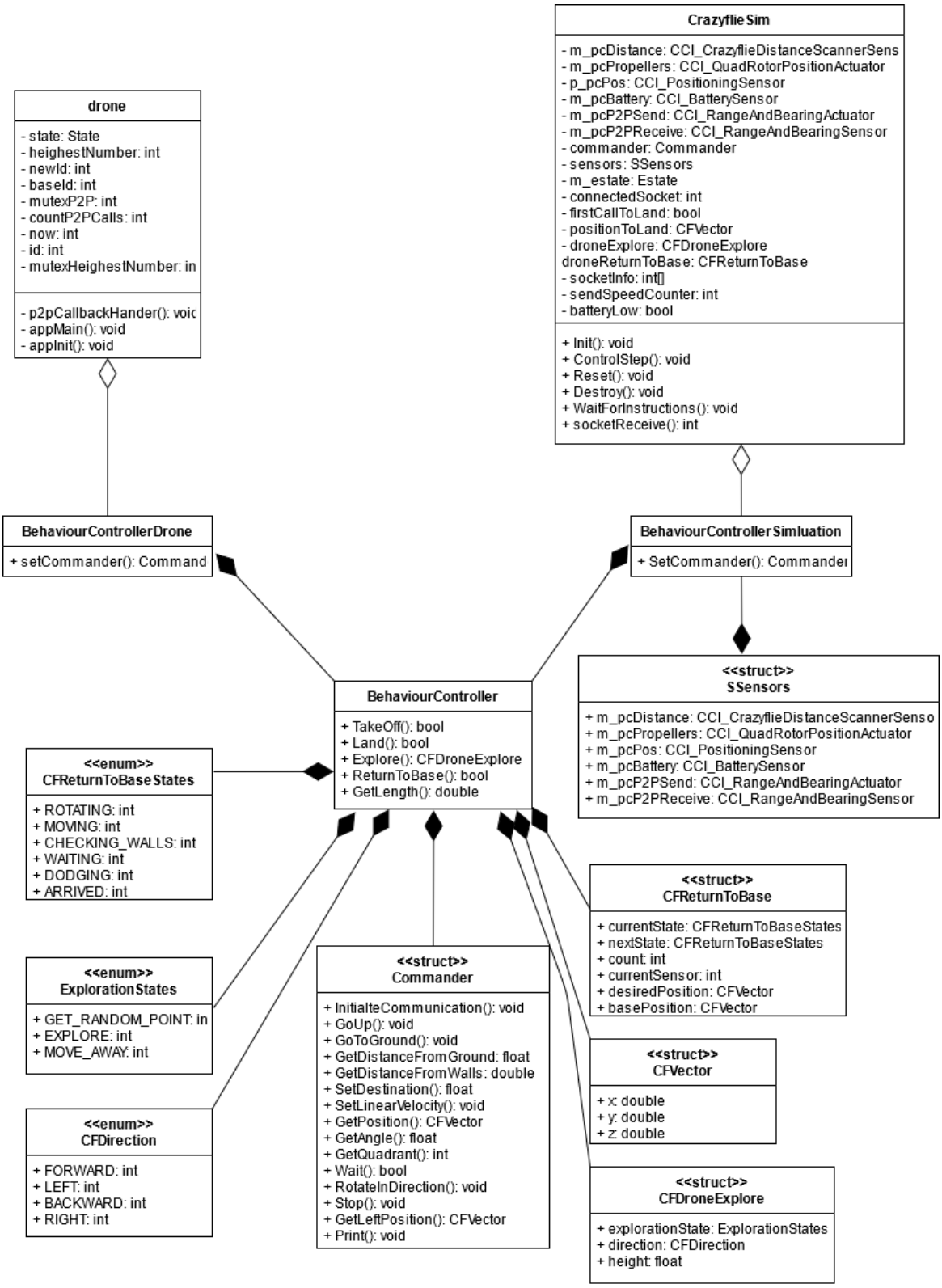
régulièrement et en continue sans devoir faire des « demandes » au serveur. C'est-à-dire, nous envoyons toutes les informations du R.F.5 (données des drones et de la carte) à travers ce socket. C'est donc le serveur qui envoie les données en continue, et non pas le client qui demande puis le serveur répond. Les requêtes REST/HTTP permettent de communiquer des informations saisies ou demandé par l'utilisateur. Ces demandes sont les commandes envoyées aux drones comme « commencer la mission ». Nous utilisons des requêtes REST/HTTP pour communiquer les commandes aux drones, car c'est plus facile d'utilisation, et comporte moins de risque de bug. Avec le module embarqué, nous avons aussi deux types de communications avec le serveur. La première est le Crazyradio. Elle nous permet de communiquer avec les drones physiques. Nous utilisons cette radio, car c'est le seul moyen de communiquer avec les drones en temps réel. De plus, le requis R.M.2 nous impose une seule radio. Ensuite, nous utilisons aussi un socket pour communiquer entre la simulation et le serveur. Son rôle est de simuler la Crazyradio qui envoie et reçoit des données en continu. De plus, la simulation doit envoyer des données en temps réel comme les données des drones ou les données du ranging deck des drones simulés.

Le serveur ne sert pas qu'à lier les drones au client. Il sert aussi à faire les calculs nécessaires au bon affichage de la carte. C'est lui qui s'occupe de faire les calculs, car les drones n'ont pas la capacité de le faire. De plus, ils doivent déjà constamment faire des vérifications pour éviter les murs, et nous ne voulons pas les surcharger. Également, ce n'est pas au client de faire les calculs, car ce n'est pas son rôle et serait une erreur d'architecture.

Troisièmement, nous avons le module système embarqué. Ce module comporte en réalité deux parties qui sont un peu différentes. Il y a la simulation puis les drones physiques. La simulation est la simulation des drones avec ARGoS et nous permet donc de valider nos algorithmes d'exploration ou d'évitement d'obstacle avant de les utiliser sur les drones physiques. Nous avons décidé de créer une librairie commune entre les drones physique et la simulation. Cette librairie nous permet donc d'éviter la répétition de code des algorithmes. Par exemple, elle nous permet d'écrire seulement une fois l'algorithme d'exploration au lieu de la réécrire une fois pour ARGoS et une fois pour les drones.

3.2 Architecture logicielle embarqué

Figure 2 : Diagramme de classe le la partie embarqué



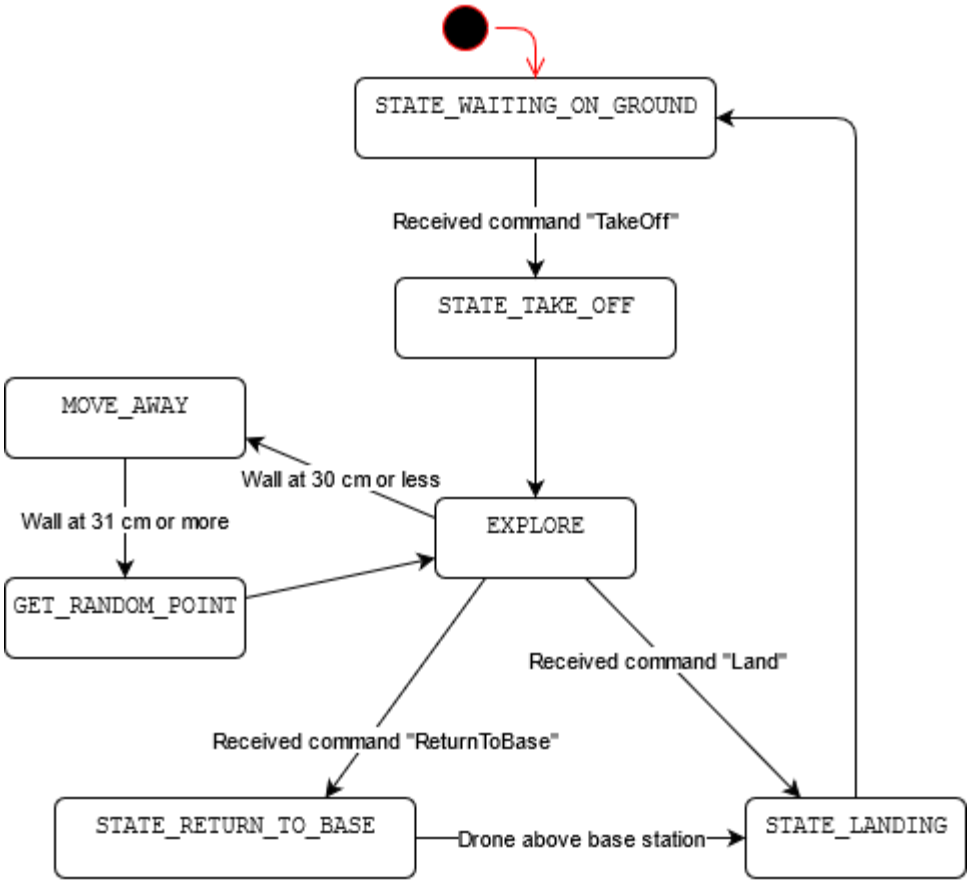
Comme on peut voir dans la figure 2, notre architecture autant côté drone physique que simulation se repose sur la librairie *BehaviourController*. Comme son nom l'indique, elle sert d'interface entre les robots et algorithmes. Elle contient les méthodes de haut niveau qui seront appelées par les contrôleurs. Pour comprendre son utilité, il faut se rappeler que nous utilisons la simulation ARGoS comme un moyen de tester des algorithmes. Par exemple, la méthode *explore*() contient l'algorithme d'exploration qui sera commun au drone et à la simulation. Afin de partager ce code entre les deux côtés, nous avons déclarés une structure contenant toutes les méthodes de base des robots. Celle-ci, appelée *Commander*, contient entre autres des fonctions de déplacement et de collecte de données. Il faut seulement implémenter ces méthodes du côté drone et du côté simulation. Comme on peut voir dans l'exemple suivant, la méthode *GoUp*() est implémentée des deux côtés. Il s'agit du patron de conception Template [24] implémenté sans l'utilisation de classes puisqu'elles ne sont pas disponibles en C.

Étant donné que notre librairie doit être commune entre la simulation en C++ et le drone physique en C, notre librairie devra être écrite en C elle aussi. Cela implique que nous devons nous assurer de connecter correctement, que ce soit le firmware du crazyflie ou ARGoS, avec notre librairie. Le plus gros défi sera de passer nos objets C++ au côté C++ de la librairie puisqu'ils doivent avant passer par la librairie commune en C. Ce problème est facilement réglable en conservant l'adresse de l'objet et en la transformant ensuite en objet au moment voulu. Cela règle en même temps un autre problème majeur, qui est que notre contrôleur côté drone devra envoyer des objets différents à la librairie que le contrôleur côté simulation. Il sera donc possible dans les deux cas de convertir l'objet en pointeur *void** et ensuite de le reconvertir dans le type désiré. La librairie commune n'a donc pas besoin de savoir quels types d'objets sont passés.

Le chemin à suivre pour appeler une fonction est donc le suivant. Nous créons une instance du *Commander* dans un des contrôleurs. Nous appelons la fonction *SetCommander,* qui dépendamment quel fichier est inclut (soit *BehaviourControllerSimulation* ou *BehaviourControllerDrone*), assigne les fonctions au contrôleur. Par la suite, on peut appeler une méthode de la librairie commune, soit par exemple la méthode *TakeOff*. Nous lui passons en paramètres certaines valeur, mais plus précisément le *commander* ainsi qu'une structure convertie en *void** contenant des objets nécessaires pour permettre le contrôle du drone dans la librairie. La méthode *TakeOff* s'occupe ensuite d'appeler *(*commander).GoUp* afin d'appeler la méthode *GoUp* spécifique de la plateforme l'ayant appelée. La méthode *GoUp* convertie le pointer *void** en objet et peut ainsi faire ses opérations.

Nous pouvons voir dans la figure 2 plusieurs structures comme *CFVector*, qui est une interface entre les vecteurs du drone et les vecteurs de la simulation, *CFDirection* qui contient les directions possibles des drones, *ExplorationStates* qui définit les états possibles des drones durant leur exploration, etc. Clairement, cette librairie rajoute son lot de travail et de problèmes, nous pensons cependant qu'elle en vaut la peine, car une fois qu'elle sera terminée, nous aurons créé une interface fonctionnelle entre la simulation et le drone physique qui pourra être réutilisé pour d'autres projets.

Figure 3 : Diagramme d'état des drones



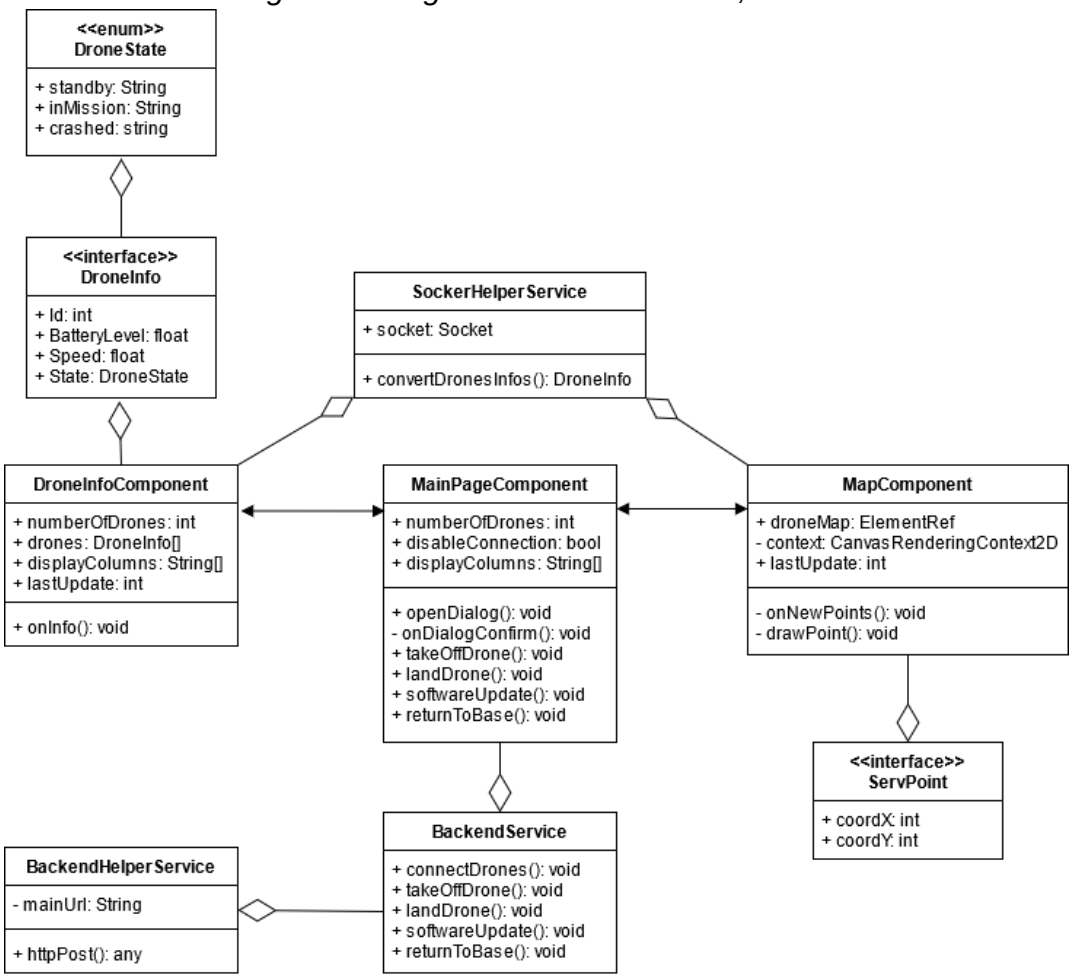
La figure 3 illustre les différents états des drones lors de l'exploration. Lorsqu'un drone démarre, il commence avec un état d'attente (*STATE_WAITING_ON_GROUND*). Lorsque l'utilisateur envoi la commande de début de mission, le drone passe alors à l'état de décollage (*STATE_TAKE_OFF*). Ensuite, une fois, le décollage finit, il passe à l'état d'exploration. L'exploration comporte en réalité 3 états qui sont les suivants : *EXPLORE*, *MOVE_AWAY*, et *GET_RANDOM_POINT*. Le drone reste dans l'état d'exploration en boucle jusqu'à recevoir une commande d'atterrissage (*STATE_LANDING*) ou de retour à la base (*STATE_RETURN_TO_BASE*). Lors de l'exploration, si le drone voit un mur à moins de 30 cm de lui-même, il se met

dans l'état d'évitement du mur (*MOVE_AWAY*). Lorsqu'un drone est dans l'état d'évitement de mur, il part dans le sens opposé au mur. Dès qu'il remarque que le mur est à plus de 31 cm de lui-même, il passe à l'état de décision aléatoire (*GET_RANDOM_POINT*). Cet état de décision aléatoire nous permet de faire décider au drone dans quelle direction, il veut aller. Ensuite, on revient à l'état d'exploration, et on continue ainsi. Lors du retour à la base, le drone cherche à revenir à son point initial. Une fois au-dessus de son point initial, le drone passe à l'état d'atterrissage. Une fois l'atterrissage complété, il repasse dans l'état d'attente.

3.3 Architecture logicielle station au sol

Pour la station au sol, nous avons décidé de séparer la tache en deux. Nous avons donc fait un serveur et un client. Le serveur est en python, car c'est un bon langage pour faire des calculs, et aussi, car c'est un des langages les plus facile à utiliser avec le Crazyradio. Pour le client, c'est une application Angular 11. Nous avons décidé d'utiliser Angular, car nous sommes familiers avec ce cadriciel et aussi, car il permet d'avoir une interface web esthétiquement propre. Puisque nous utilisons Angular pour faire le client, nous utilisons donc aussi Angular Material [12] pour rendre l'application web plus jolie.
Les deux diagrammes suivants expliquent l'architecture du client et du serveur respectivement.

Figure 4: Diagramme station au sol, client

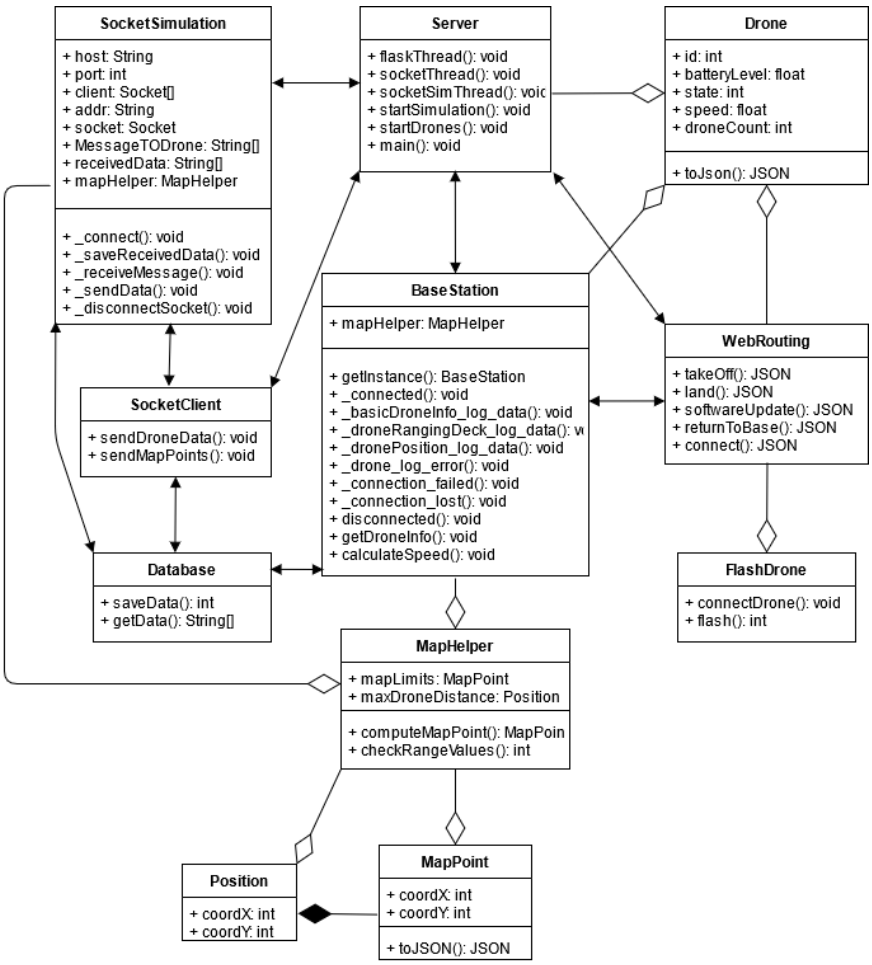


En ce qui concerne le Client, comme évoqué précédemment, nous utilisons le cadriciel Angular 11 pour le réaliser. La figure 4 illustre nos différentes classes utilisées dans le client avec un diagramme de classe. Les doubles flèches représentent une communication entre deux composants et les flèches d'agrégation représentent l'utilisation d'une classe par un composant.

Comme on peut observer sur la figure 4, il y a 3 composants dans notre application web. Ces trois composants sont *DroneInfoComponent*, *MainPageComponent*, et *MapComponent*. Ces trois composants permettent donc 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 endroit, 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. Ensuite, la figure 4 nous permet aussi de comprendre quels sont les différents services utilisés par le client, et quelles 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. Nous avons décidé de séparer les différentes communications afin de mieux séparer les taches et pour avoir une meilleure encapsulation de nos méthodes. 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 gérer 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 POST [13] nécessaires vers le backend. Il implémente différentes méthodes pour l'envoi des différentes commandes possibles depuis l'interface web comme *landDrone* qui permet de faire atterrir les drones. 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.

Figure 5: Diagramme de classe de la station au sol, serveur



Le diagramme de classe pour le serveur est représenté avec la figure 5. 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 est la classe *Server* et s'occupe d'initialiser tous les différent composant du serveur. En fonction du paramètre entrée lors de l'exécution du Dockerfile, 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 (*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. Cette classe est très importante, car elle permet donc 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. 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 continue 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 continue 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 continue 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*, *Database*, *MapHelper*, *MapPoint*, *Position*. Les classes *Drone*, *MapPoint*, et *Position* sont surtout des classes utilisées comme des interfaces. Elles permettent surtout 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 *Database* et *MapHelper*. La classe *Database* nous sert à gérer la base de données. Elle nous permet donc d'enregistrer toutes les données reçues et de les récupérer. 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.

4. Processus de gestion

4.1 Estimations des coûts du projet

Après la planification et la répartition des tâches entre les membres, nous avons défini le nombre d'heures pour chaque membre de l'équipe. Le coordonnateur du projet va passer approximativement 115 heures et 30 minutes pour le projet et avec un budget de 145 $ par heure nous avons un total de 16 747.5 $. Pour les développeurs-analystes, nous avons un total de 427 heures. Le budget pour les développeurs-analystes est de 130 $ par heure, donc le total est de 55 510 $.

Le budget total du projet est donc de 72 257 $.

Pour les taches consacrées à la CDR, nous avons estimé qu'il sera nécessaire d'investir 271 h.

Pour la RR, nous avons estimé qu'il sera nécessaire d'investir 273 h pour accomplir toutes les tâches planifiées.

4.2 Planification des tâches

Dans l'annexe 1, nous présentons deux tableaux (Tableau 3: Tâches effectué durant la CDR, Tableau 4: Tâches planifiés pour la RR) qui montrent les tâches effectuées durant la CDR et les tâche qui sont planifiés pour la RR. Dans les tableaux sont mentionnés toutes les tâches, les personnes assignées à chaque tâche, le temps alloués pour chaque tâche et quel système de notre projet est touché par chaque tâche (Drone ou simulation ou serveur ou interface client ou autre).

Pour la CDR et la RR, nous avons créé un diagramme de Gantt, présenté dans l'annexe 2, pour visualiser dans le temps les diverses tâches composant le projet et pour montrer l'avancement.

De plus, dans l'annexe, nous présentons un histogramme (Annexe 1, Figure 6) montrant le nombre d'heures travaillé par chaque membre de l'équipe pendant la CDR ainsi que le nombre d'heures planifié pour chaque personne pour la RR.

4.3 Calendrier de projet

Tableau 2: Remises des livrables

Livrable	Date de remise	Description
Critical Design Review	8 mars 2021	Remise d'un système avec fonctionnement partiel
Readiness Review	12 avril 2021	Remise d'un système avec toute la fonctionnalité du système

4.4 Ressources humaines du projet

Pour Nicolas, il possède des connaissances en Angular qui viennent du deuxième projet intégrateur. Il possède aussi de l'expérience avec des scripts Windows et sur l'utilisation des machines virtuelles. Il possède aussi de l'expérience sur les problèmes matériels et est capable de faire de la soudure sur les systèmes informatiques.

Pour William, ses connaissances en *Unity* [14] sont sans aucun doute un grand atout pour ce projet. En effet, la simulation *ARGoS* ressemble beaucoup à une simulation *Unity*, surtout dans la façon dont le code est exécuté. De plus, William a travaillé sur quelques projets personnels en C++ ce qui lui a permis d'être familier avec les concepts importants.

Theo, quant à lui, a de nombreuses expériences avec Angular, notamment lors d'un stage. Étant le membre avec le plus d'expérience en Angular dans l'équipe, il a été convenu qu'il soit le spécialiste frontend de l'équipe. De plus, il possède quelques expériences avec le langage Python. Il va donc pouvoir prêter main forte pour le développement du backend en Python. Il possède aussi quelques expériences en C++, ce qui va lui permettre d'aider au développement des Drone et de la simulation.

Pour Philippe, il a plus de 1 an et 3 mois d'expérience en Java [15] et en tant que QA. Cela sera très utile pour l'architecture ainsi que pour la mise en place de tests unitaires et d'intégration. De plus, il a de l'expérience au PolyFab Normand Brais [16] se trouvant à Polytechnique Montréal, et a fait de nombreux projets en rapport avec le système embarqué.

Pour Nour, elle a des connaissances en Angular qu'elle a acquis du deuxième projet intégrateur. Elle a aussi quelques connaissances dans le langage python ainsi que le langage C et C++. Elle pourra donc participer au développement du backend ainsi qu'au développement du système embarqué pour les drones.

Malgré toutes les connaissances des membres, la conception du projet demande plusieurs compétences que les membres de l'équipe n'ont pas. Tous les membres de l'équipe ont fait le projet 2, qui nous a tous appris à être indépendant et à acquérir des compétences en faisant des recherches sur

internet. Pour ce qui est du processus d'acquisition de ces compétences, nous comptons utiliser les mêmes méthodes d'apprentissage. Cette méthode a été très efficace pour les compétences que nous avons eues besoin d'acquérir pour les taches de la CDR.

5. Suivi de projet et contrôle

5.1 Contrôle de la qualité

Chaque fonctionnalité sera complémentée d'un ou de plusieurs tests unitaires correspondant pour assurer le bon fonctionnement de ces nouvelles fonctionnalités. Pour assurer la qualité de notre code avant chaque remise, nous avons décidé de finaliser notre développement au minimum deux jours avant la remise. Cela permet de nous donner du temps pour peaufiner le code et corriger les petits bugs qui peuvent rester.

En plus des tests unitaires, nous allons faire des tests de régressions en équipe à chaque fin de sprint. Pour ce faire, nous allons avoir une personne dans l'équipe qui teste le système en entier en partagent son écran pour que tous les membres de l'équipe évaluent le système.

Nous allons aussi faire des tests d'intégration tout au long du développement pour assurer le bon fonctionnement de chaque fonctionnalité du système.

Pour contrôler la qualité tout au long du développement, le code que nous écrivons est révisé par chaque membre de l'équipe lorsque nous faisons des merge request [17]. Nos merge request doivent être approuvés par tous les membres de l'équipe. Cela va aussi nous aider à comprendre le code des autres et peut nous permettre de proposer de meilleures solutions après avoir compris le code de la personne qui a programmé la fonctionnalité.

5.2 Gestion de risque

Dans le contexte actuel de la pandémie du Covid-19, plusieurs risques liés à la gestion de projet et au travail à distance doivent maintenant être pris en compte dans notre gestion de risque.

Un premier potentiel risque est si une hélice sur un drone brise et que nous sommes en attente de recevoir une hélice supplémentaire pour la réparation. Dans ce cas, plusieurs options sont disponibles selon l'avancement du projet. Premièrement, si nous sommes simplement en train de développer la détection et le transfert des points pour générer la carte, nous pouvons continuer de travailler avec un seul drone. De plus, nous pouvons aussi bouger le drone endommagé dans nos mains, car ce que nous testons est la détection et la transmission des points et non l'algorithme de vol. Une autre option est de continuer le développement en utilisant la simulation Argos. Deuxièmement, si nous sommes en train de développer la communication entre les drones et avec la station au sol, il n'est pas nécessaire que les drones soient en vol. Dans ce cas, le fait qu'une hélice soit endommagée ne change rien à notre développement et ne nous ralentira pas le temps de recevoir une nouvelle hélice.

Un second potentiel risque similaire au premier est si une autre partie d'un drone ne fonctionne plus. Dans ce cas, comme pour le premier risque, tout dépendant de la pièce endommagée, nous pouvons tout de même continuer le développement de fonctionnalités qui ne touchent pas à la partie endommagée en attendant de recevoir la ou les pièces de remplacement. De plus, tout dépendant de la situation, nous pourrions aussi développer en utilisant Argos en attendant de recevoir la pièce de remplacement si nécessaire.

Un troisième potentiel risque est si la personne qui a les deux drones a des problèmes avec son ordinateur et ne peut plus tester et développer avec les vrais drones ou si on veut qu'une autre personne dans l'équipe expérimente avec les drones. Dans ce cas, puisqu'une seule personne aura en sa possession les deux drones, il sera alors nécessaire qu'une personne dans l'équipe se déplace pour aller chercher les drones. Pour le déplacement, une personne dans l'équipe à une voiture et peut se déplacer si nécessaire.

Un quatrième potentiel risque existe s'il y a des changements au requis du projet. Dans ce cas, la meilleure façon de régler ce problème sera de se rassembler en équipe et refaire notre planification pour tenir en compte des changements. Cela inclut un changement dans les tâches dans notre planification sur le Board sur Gitlab [18]. Ensuite, la tâche sera assignée à une personne de l'équipe.

Un cinquième potentiel risque est que si une personne dans l'équipe à un problème personnel comme une maladie ou un problème familial et ne peut pas compléter sa tâche ou ne peut simplement pas travailler pour une certaine durée de temps. Pour régler ce problème, nous devrons faire une rencontre d'équipe d'urgence pour discuter de la situation et assigner la tâche à une autre personne pour compenser pour la personne qui ne peut pas travailler.

5.3 Tests

Pour tester le code embarqué des drones, nous allons utiliser la simulation d'ARGoS pour assurer le bon fonctionnement de nos drones. Pour s'assurer du bon fonctionnement, nous devons nous vérifier que l'algorithme d'exploration fait son travail et ne manque aucune partie de la salle à explorer. De plus, nous devons nous assurer que les drones sont capables d'éviter les obstacles et les collisions avec l'environnement et avec les autres drones.

Nous allons utiliser Jasmine [19] pour tester l'interface web qui est codé en utilisant Angular. L'utilisation de Jasmine nous permet d'avoir une couverture de code de 100 % et nous permet donc de nous assurer que notre code est testé en entier. Pour ce qui est de la partie *backend*, puisque celle-ci est codée en Python, nous allons utiliser *Unittest* [20] qui est fourni par défaut avec Python pour faire les tests unitaires et les tests d'intégration. Finalement, pour tester le code embarqué sur les drones et simulation, nous allons utiliser CUnit [26] pour faire les tests unitaires pour le code C. Les tests unitaires et les tests d'intégrations seront complétés par la ou les personnes qui ont développé la fonctionnalité. Nous allons seulement pouvoir tester notre librairie, car le code ARGoS et Crazyflie n'est malheureusement pas testable automatiquement, c'est-

à-dire que nous pouvons faire des tests manuels, mais pas avec une librairie. En effet, nous ne pouvons pas « mock » un drone ou une simulation, et donc nous ne pouvons pas écrire de tests.

Nous allons aussi faire des tests de régression en équipe à chaque fin de sprint pour tester directement le système. Une personne dans l'équipe exécutera le code normalement et partagera son écran pour que le reste des membres de l'équipe puissent valider le bon fonctionnement du système.

5.4 Gestion de configuration

Afin de s'assurer que notre code est lisible et facilement maintenable, nous avions l'intention d'indiquer en haut de chaque header file l'utilité de la classe. Aussi, nous allons laisser des commentaires au-dessus de chaque fonction pour expliquer leurs paramètres et leur utilité. Ces mesures devraient rendre le code bien lisible pour tout le monde et faciliter sa maintenance.

Ensuite, nous voulons ajouter un Read me à la base du projet pour indiquer comment lancer chacune des parties. Il expliquera par exemple comment lancer le server, le *frontend*, la simulation et comment télécharger le code sur le drone. Cela devrait permettre à n'importe qui de pouvoir lancer toutes les parties du projet très facilement sans devoir trop chercher.

Nous avons séparé notre projet en 3 dossiers qui correspondent à chacune des parties du projet, soit Drone, Server et Client. Cela nous permettra d'avoir un code bien organisé et facilitera le développement.

Pour s'assurer que seulement du code de qualité se ramasse dans les remises, nous optons pour un système de *merge request*. Cela veut dire que du code doit être vérifié par tous les membres de l'équipe avant d'être mis sur notre branche de remise, soit la branche master. Les imperfections et erreurs sont donc captées et communiquées à l'auteur pour qu'il fasse les changements nécessaires. Une fois les changements terminés, les autres membres peuvent regarder à nouveau et s'ils jugent que le code est parfait, ils acceptent la *merge request* et le code est jumelé avec le reste. Si jamais nous avons des questions par rapport à une partie du code, nous pourrons nous organiser des rencontres afin d'expliquer ou même prendre des décisions côté architecture et design.

Comme dit précédemment, notre code est séparé en trois dossiers. Pour s'assurer que ce code peut rouler sur n'importe quelle machine, nous avons ajouté un *dockerfile* [21] dans chacun de ces dossiers. Ceux-ci créent un environnement contenant toutes les dépendances nécessaires et les instructions sur comment construire les éléments nécessaires. Ces trois *dockerfiles* sont reliés par un fichier appelé *docker-compose* [22]. Celui-ci construit les trois conteneurs d'un coup permettant d'avoir un environnement complet en une ligne de commande.

Nous comptons aussi avoir un script bash [23] pouvant lancer chaque partie du projet. Nous pourrons lui passer des arguments qui lui indiqueront quelle partie du code à lancer. Ces arguments seront indiqués dans le « Read me » ce qui permettra à n'importe qui de lancer n'importe quelle partie du projet.

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] Bitcraze. (2021) Multi ranger deck. [En ligne]. Disponible : https://www.bitcraze.io/products/multi-ranger-deck/

[7] J. Collin, *Cycles d'acquisition et ententes contractuelles*, 2021. [En Ligne]. Disponible : https://moodle.polymtl.ca/course/view.php?id=1703

[8] 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

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

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

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

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

[13] MDN contributors. (2020) POST. [En ligne]. Disponible : https://developer.mozilla.org/fr/docs/Web/HTTP/Methods/POST

[14] "Unity (game engine)", dans *Wikipédia*, 28 fév. 2021. [En ligne]. Disponible : https://en.wikipedia.org/wiki/Unity_(game_engine)

[15] "Java (programming language)" dans *Wikipédia*, 4 mars 2021. [En ligne]. Disponible : https://en.wikipedia.org/wiki/Java_(programming_language)

[16] PolyFab Normand Brais. (2021) PolyFab. [En ligne]. Disponible : https://polyfab.polymtl.ca/

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

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

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

[20] Python. (2021) Unit testing framework. [En ligne]. Disponible : https://docs.python.org/3/library/unittest.html

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

[22] Docker. (2021) Overview of Docker Compose. [En ligne]. Disponible : https://docs.docker.com/compose/

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

[24] "Template method pattern", dans *Wikipédia*, 19 fév. 2021. [En ligne]. Disponible : https://en.wikipedia.org/wiki/Template_method_pattern

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

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

ANNEXE 1

Tableau 3 : Tâches effectué durant la CDR

Tâches	Temps alloué en heures	Personne(s) responsable	Système concerné (Drone/ Simulation/Serveur/Interface client/ Autre)
Retour à la base simulation	2	William	Simulation
Retour à la base à 30% de batterie simulation	2	William	Simulation
Vitesse du drone simulation	2	Nicolas	Simulation
Implémenter getDistanceFromWalls	3	William	Simulation
Décoller/Commencer mission simulation	2	Nicolas	Simulation
Atterrir/ Finir Mission simulation	2	Nicolas	Simulation
Acceder au UI par plusieurs appareils	3	Theo	Serveur, Interface client
Créer librairie done/simulation	5	William	Drone, Simulation
Test unitaire Python	3	Theo, Nour	Serveur
Générer la simulation Argos aléatoirement	20	William, Nour	Simulation
Afficher les informations des drones (tableau)	3	Nour	Interface client
Ajouter les boutons de commande (4 boutons)	5	Nour	Interface client
Afficher la carte	33	Nicolas, Theo	Interface client
Calculer la position du mur avec les infos reçu du drone	13	Nicolas, Theo	Serveur
Envoyer les commandes aux drones quand le frontend le demande	3	Nour	Serveur, Simulation
Récupérer les données des drones (autres que les points) (crazyradio)	13	Philippe, William	Serveur
Transférer les points au frontend	3	Theo	Serveur
Récupérer les points	20	Theo, Philippe, William	Serveur

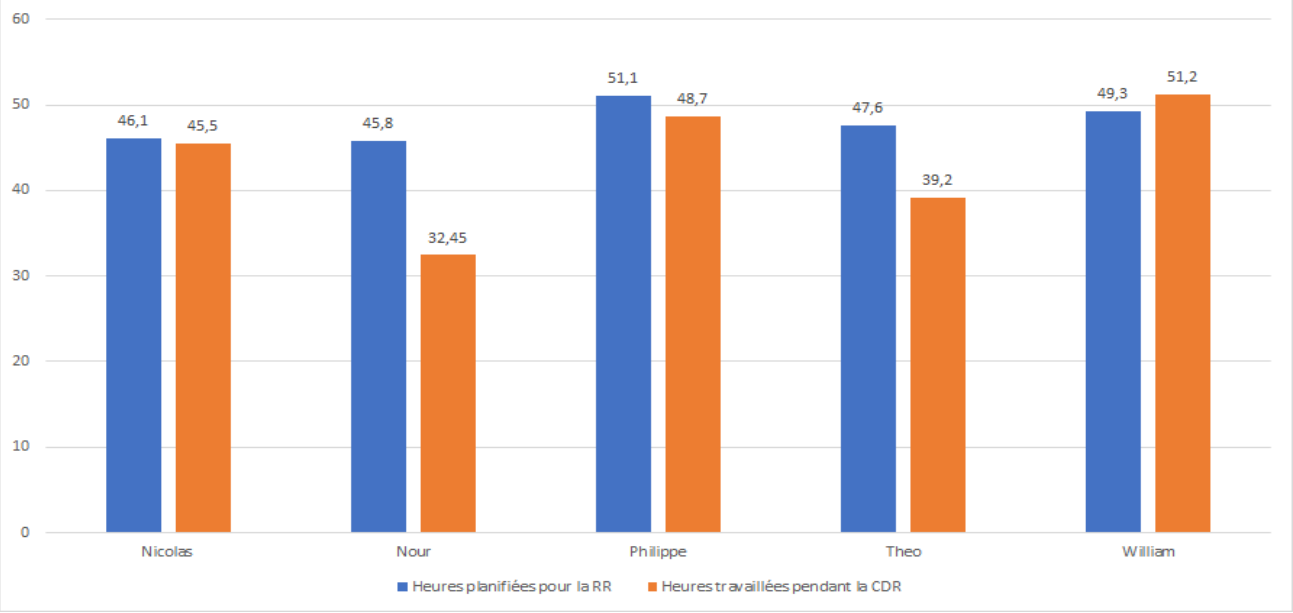
des drones (crazyradio)			
Communiquer en P2P	**20**	**Philippe, Nour**	**Drone, Simulation**
Communiquer avec le backend avec la simulation	**20**	**Nicolas**	**Serveur, Simulation**
Algorithme d'exploration et éviter les obstacles et les autres drones	**13**	**Theo, Nour**	**Simulation**
Écrire les rapports hebdomadaires	**20**	**Nicolas, Nour, Philippe, Theo, William**	**Autre**
Rédiger le rapport de CDR	**53**	**Nicolas, Nour, Philippe, Theo, William**	**Autre**
Créer/maintenir script d'automatisation	**8**	**Nicolas, Nour, Philippe, Theo, William**	**Autre**
Total d'heure pour la CDR	**271**		

Tableau 4 : Taches prévues pour la RR

Nom de la tâche	Temps alloué en heures	Personne(s) responsable	Système concerné (Drone/ Simulation/Serveur/Interface client/ Autre)
Rédiger le rapport de RR	**53**	**Nicolas, Nour, Philippe, Theo, William**	**Autre**
Estimer la distance des drones avec le RSSI	**20**	**Philippe, Nour**	**Serveur, Drone**
Communiquer les commandes aux drones quand le frontend le demande	**13**	**Theo, Nicolas, Philippe**	**Serveur, Drone**
Enregistrer les points de la carte	**5**	**Nicolas, Nour**	**Serveur**
Générer des logs pour assurer le bon fonctionnement	**13**	**Nicolas, Nour**	**Serveur**
Retour à la base à 30% de batterie	**1**	**William**	**Drone**
Implémentation du code embarqué sur les drones physiques	**20**	**Theo, Philippe**	**Drone**
Revenir à la base (RSSI)	**8**	**William**	**Drone**
Mise à jours logicielle	**33**	**Philippe, William**	**Drone**

sur les drones			
Atterir/finir la mission	**8**	**Nicolas**	**Drone**
Décoller/commencer la mission	**8**	**Nicolas**	**Drone**
Algorithme d'exploration (améliorations)	**20**	**Theo, Nour**	**Drone**
Mise à jours logicielle (bouton sur le frontend)	**9**	**Theo, Nicolas, Nour**	**Interface client**
Implémentation de la libraire côté drone	**13**	**Theo, Philippe**	**Drone**
Faire des tests de régression	**33**	**Nicolas, Nour, Philippe, Theo, William**	**Autre**
Mettre en place les dockerfiles/docker-compose	**8**	**Nicolas, Nour, Philippe, Theo, William**	**Autre**
maintenir script d'automatisation	**8**	**Nicolas, Nour, Philippe, Theo, William**	**Autre**
Total d'heure pour la RR	**273**		

Figure 6 : Temps passé en heure pendant la CDR (orange) et temps planifié en heure pour la RR (bleu) pour chaque membre de l'équipe



ANNEXE 2:

Milestone

15 févr. 2021 · 16 févr. 2021 · 17 févr. 2021 · 18 févr. 2021 · 19 févr. 2021 · 20 févr. 2021 · 21 févr. 2021 · 22 févr. 2021 · 23 févr. 2021 · 24 févr. 2021 · 25 févr. 2021 · 26 févr. 2021 · 27 févr. 2021 · 28 févr. 2021 · 1 mars 2021 · 2 mars 2021 · 3 mars 2021 · 4 mars 2021 · 5 mars 2021 · 6 mars 2021 · 7 mars 2021 · 8 mars 2021

Milestone : CDR

Drones

- Communiquer avec backend
- Atterir / Finir la mission
- Decoller / Commencer la mission
- Algorithme d'exploration
- Revenir a la base
- Retour a la base a 30% de batterie
- Communiquer en P2P
- Eviter les obstacles et les autres drones

Serveur

- Recuperer les points des drones
- Transferer les points au frontend
- Envoyer les commandes aux drones lorsque le frontend le demande
- Calculer la position du mur avec les infos recues par le drone
- Recuperer les informations des drones

Frontend

- Ajouter les boutons de commande
- Afficher la carte
- Afficher les informations des drones

Simulation

- Generer la simulation Argos aleatoirement

Légende

Philippe · Nour · Nicolas · William · Theo

Mar 8, 2021 · Mar 9, 2021 · Mar 10, 2021 · Mar 11, 2021 · Mar 12, 2021 · Mar 13, 2021 · Mar 14, 2021 · Mar 15, 2021 · Mar 16, 2021 · Mar 17, 2021 · Mar 18, 2021 · Mar 19, 2021 · Mar 20, 2021 · Mar 21, 2021 · Mar 22, 2021 · Mar 23, 2021 · Mar 24, 2021 · Mar 25, 2021 · Mar 26, 2021 · Mar 27, 2021 · Mar 28, 2021 · Mar 29, 2021 · Mar 30, 2021 · Mar 31, 2021 · Apr 1, 2021 · Apr 2, 2021 · Apr 3, 2021 · Apr 4, 2021 · Apr 5, 2021 · Apr 6, 2021 · Apr 7, 2021 · Apr 8, 2021 · Apr 9, 2021 · Apr 10, 2021 · Apr 11, 2021 · Apr 12 2021

Milestone : RR

Drones

- Implementer le code embarqué
- Mise a jour logicielle
- Revenir a la base
- Retour a la base a 30% de batterie
- Decoller / Commencer la mission
- Atterir / Finir la mission

Frontend

- Ajouter details du bouton Mise a jour logicielle

Serveur

- Estimer la distance des drones avec la puissance du signal (RSSI)
- Generer les logs des points des drones
- Enregistrer les points de la carte
- Communiquer les commandes aux drones