Dernière modification: 8 mars 2021

INF3995: Projet de conception d'un système informatique
Hiver 2021
Rapport CDR

Hivexplore - Système aérien minimal pour exploration

Proposition répondant à l'appel d'offres no. H2021-INF3995 du département GIGL

Équipe No. 102

Misha Krieger-Raynauld

Nathanaël Beaudoin-Dion

Rose Barmani

Samer Massaad

Simon Gauvin

Yasmine Moumou

Table des matières

1 Vue d'ensemble du projet

1.1 But du projet, portée et objectifs

Le but principal du projet est de réaliser une preuve de concept pour l'Agence Spatiale démontrant que l'utilisation de robots équipés de capteurs rudimentaires pour une mission d'exploration d'une pièce d'au plus 100 m^2 est une solution fonctionnelle. Pour ce faire, un prototype de niveau de maturité 4 (NMS 4) de la solution doit être réalisé.

Ce prototype consiste en une station au sol avec une interface opérateur ainsi qu'une partie embarquée, soit un essaim d'un nombre arbitraire de drones explorateurs. La station au sol doit permettre à un opérateur de monitorer l'état, la vitesse et la batterie des drones, de débuter ou de mettre fin à une mission d'exploration et de mettre à jour le système à bord des drones. Quant aux drones, ceux-ci doivent pouvoir explorer une pièce d'un bâtiment de dimension moyenne, éviter des obstacles incluant les autres drones et communiquer entre eux. La mission d'exploration a comme objectif la collecte des données sur les lieux par les capteurs des drones. Ainsi, à l'aide des données récoltées, une carte de l'endroit exploré est générée par la station au sol et affichée à l'opérateur.

Pour ce qui est du suivi du projet, trois livrables sont attendus : le « Preliminary Design Review », le « Critical Design Review » et le « Readiness Review ». Le « Preliminary Design Review » inclut la remise d'un prototype de base, d'une simulation simple ainsi que de la présente réponse à l'appel d'offres. Ensuite, le « Critical Design Review » consiste en la remise d'un système partiellement completé, mais implémentant la grande majorité des fonctionnalités. Finalement, le « Readiness Review », implique la remise du prototype complètement finalisé respectant tous les requis, un rapport technique détaillant le développement du prototype ainsi qu'une présentation orale du produit fini.

1.2 Hypothèse et contraintes

Avant de détailler le plan de la réalisation du projet, il importe de mentionner que celui-ci repose sur certaines hypothèses afin de simplifier l'étendue des connaissances préalables nécessaires. En premier lieu, nous présumons que la Crazyradio est capable de se connecter à plus d'un drone à la fois, chose qui nous sera nécessaire dans le but de diffuser les commandes à tous les drones et à récolter les données de chaque drone en simultané. En considérant que la documentation de la Crazyradio mentionne une portée d'un kilomètre, nous supposons également que la connexion avec les drones sera toujours fiable : la pièce mesure au plus 100 m^2. Les obstacles dans celle-ci ne devraient pas non plus représenter une entrave à la communication avec les drones selon nos expérimentations.

Lors de la réponse à l'appel d'offres, nous avions posé l'hypothèse que toutes les fonctionnalités physiques des drones et du *firmware* Crazyflie pourraient être reproduites fidèlement avec la simulation ARGoS. Depuis, nous avons effectivement constaté que ce ne sont pas tous les modules du *firmware* et de l'API Crazyflie qui sont disponibles dans ARGoS. Certaines fonctionnalités comme la communication *peer-to-peer*, le RSSI et l'API de *logging* et de *param* ont dû être réécrites ou adaptées pour les tester dans une simulation ARGoS. De même, dans l'optique où nous souhaitons réaliser le requis optionnel de l'affichage de la position des drones dans la station au sol, nous supposons toujours aussi que les capteurs des drones nous offrent suffisamment d'information pour déterminer la position absolue de ceux-ci.

Sinon, en se tournant vers des considérations externes à l'équipe, nous devons aussi émettre l'hy-

pothèse qu'il continuera d'être possible de se transférer le drone entre les membres de l'équipe malgré les restrictions pour contrer la COVID-19. Jusqu'à maintenant, ceci ne s'est pas avéré être problématique.

Au-delà des hypothèses, plusieurs contraintes nous sont aussi imposées pour encadrer le projet. Une fois la mission commencée, soit une fois qu'ils reçoivent la commande de début de mission, les drones doivent être autonomes et parcourir le périmètre de la pièce à explorer sans recevoir d'autres commandes de la station au sol (R.F.1). Au niveau des drones, au moins un d'entre eux doit être en tout temps en communication avec la station au sol (R.F.5) avec comme seul moyen de communication la Crazyradio connectée par USB (R.M.2, R.M.4). Pour les programmer et pour la communication P2P, l'API de BitCraze doit être utilisée (R.L.1, R.L.2). Aussi, l'utilisation d'un serveur pour la communication entre les drones est interdite (R.L.2). Il y a toujours un minimum de deux drones, mais un nombre arbitraire de drones peuvent être ajoutés (R.F.2). Leurs systèmes peuvent être mis à jour par l'API BitCraze seulement s'ils sont au sol (R.F.4.1). Au retour à la base, les drones doivent être à un mètre de la station au sol (R.F.4.2) et si leur niveau de batterie est inférieur à 30%, ceux-ci ne peuvent pas débuter une mission et doivent retourner à la base s'ils sont déjà en mission (R.F.4.3). La distance avec la station au sol doit obligatoirement être déterminée par la puissance du signal RSSI que la Crazyradio reçoit (R.L.3). Le prototype doit nécessairement être fait avec le « STEM Ranging bundle » sur deux drones Bitcraze Crazyflie 2.1 (R.M.1) et seulement le *ranging deck* et le *optical flow deck* peuvent y être installés (R.M.3). Enfin, l'interface utilisateur doit pouvoir être visualisée sur divers appareils et les mises à jour des informations doivent être d'une fréquence de 1 Hz au minimum (R.F.5).

Pour ce qui est de la conception, il est obligatoire de faire une simulation ARGoS avant d'expérimenter avec les drones (R.C.1). Des tests unitaires et de régression doivent être faits (R.C.2, R.C.3) et il doit être possible d'effectuer une simulation sur ARGoS de notre système par une seule commande (R.C.4).

En bref, Le prototype final remis à l'Agence doit respecter tous les requis et contraintes stipulés par le contrat suite à la réponse de l'appel d'offres. Si un requis n'est pas respecté, cette décision doit être justifiée auprès de l'Agence. Suite au travail que nous avons réalisé depuis la réponse à l'appel d'offres, nous avons justement choisi de ne pas respecter certaines de ces contraintes. Plus spécifiquement, les requis R.C.2 et R.F.4.1 seront enlevés afin de nous permettre de respecter un budget limité et fixe tout en réalisant les fonctionnalités les plus importantes. Toutes les raisons justifiant ces choix sont approfondies aux sections 4.2 et 5.3 de ce rapport.

Finalement, il existe aussi des contraintes plutôt organisationnelles et externes à l'équipe. Nous sommes contraints à ne pas dépasser la limite de 630 heures-personnes et le projet s'échelonne sur dix semaines, du 1er février au 12 avril. De plus, nous devons avancer dans le projet en priorisant les requis spécifiés par chacune des trois remises du 15 février, 8 mars et 12 avril ; remises qui seront faites sur le répertoire Git. De plus, pour permettre à l'Agence Spatiale de prendre connaissance des avancements, un suivi du déroulement du projet doit être documenté. Finalement, dû au contexte actuel de pandémie, le projet doit être réalisé à distance et les membres de l'équipe devront communiquer à l'aide de diverses plateformes pour le travail collaboratif, ce qui limite l'utilisation des drones à un ou deux membres en même temps.

1.3 Biens livrables du projet

Au cours du projet, trois principaux livrables sont attendus. Le premier, soit le « Preliminary Design Review », était à remettre le 15 février 2021 et comprenait la remise d'un prototype simple, d'une simulation ARGoS et la réponse à l'appel d'offres. Le prototype correspondait à une station au sol avec une interface permettant de monitorer le niveau de batterie des drones et d'allumer ou éteindre leurs DEL. La simulation ARGoS devait, quant à elle, simplement simuler deux drones se déplaçant avec une trajectoire quelconque.

Ensuite, le « Critical Design Review », attendu pour le 8 mars 2021, implique la remise d'un prototype assez avancé. En effet, les commandes de communication aux drones pour commencer et terminer une mission doivent être implémentées et l'interface utilisateur doit permettre de monitorer les drones et de visualiser la carte générée. Ainsi, les drones doivent être en mesure d'explorer un lieu et de collecter des données pour générer la carte. Une simulation ARGoS de quatre drones qui utilisent des capteurs de distance pour éviter les obstacles et qui explorent une carte générée aléatoirement est aussi à remettre pour ce livrable. Joint à ce livrable ce trouve le présent rapport.

Finalement, la dernière remise est pour le 12 avril 2021 : le « Readiness Review », qui correspond à la remise finale du prototype. Celle-ci doit respecter tous les requis et inclure une simulation ARGoS complète. Les fichiers remis par le biais du répertoire Git seront donc le code source du système embarqué, de la station au sol, de l'application Web et de la simulation ARGoS, ainsi que les fichiers de conteneurisation Docker et toute documentation. Accompagnant la remise du code sera également un rapport technique détaillé du développement du système et une vidéo mettant en scène le fonctionnement complet de la solution. Enfin, une présentation orale du prototype et du processus viendra clore ce dernier jalon du projet.

2 Organisation du projet

2.1 Structure d'organisation

Nous avons organisé le projet en *sprints* d'une semaine en nous inspirant de la méthodologie Agile. La remise de chaque sprint est le lundi midi, à l'heure du début de la séance de cours. Nous avons donc minimalement une rencontre par semaine le lundi après-midi pour clore le *sprint* de la semaine et débuter le prochain. Durant ces rencontres, nous avons un Google Doc partagé que tous les membres de l'équipe peuvent visualiser et modifier. Ce document contient les notes de nos réunions, soit les ordres du jour. Chacune de ces réunions est introduite par un court *standup* durant lequel chacun peut mettre à jour ses coéquipiers de ses avancements de façon formelle.

Les réunions de *sprint* du lundi se divisent en trois étapes : la rétroaction, le *grooming* et l'assignation des tâches pour le *sprint* à venir. La rétroaction correspond à un tour de table rapide de 20 minutes afin de mentionner les éléments s'étant bien ou moins bien passés pour apporter les correctifs nécessaires pour la suite. Ensuite, pendant l'étape de *grooming* de 20 minutes, nous réévaluons la répartition des tâches établie au début du projet, en fonction de ce qui a été accompli à présent, ce qu'il reste à accomplir et les nouvelles difficultés anticipées pour les prochains sprints du projet. Finalement, nous planifions le *sprint* à venir en 30 minutes en évaluant notre progrès, le temps mis sur les tâches du *sprint* précédent et les priorités pour la suite. Le tout en gardant un oeil sur les livrables et les requis à respecter sur le long terme. Nous documentons ces modifications sur le tableau de gestion GitLab de l'équipe.

En plus des rencontres du lundi, nous nous rencontrons le jeudi matin. Le but de cette rencontre, plus flexible, est de faire un autre suivi avec les membres pour s'assurer d'aider ceux qui sont en difficulté. En effet, nous avons établi qu'il est important de privilégier le déblocage des membres dont l'avancement des tâches est bloqué par une difficulté quelconque. Aussi, si d'autres points doivent être partagés avec l'équipe au complet, nous pouvons tenir des réunions le samedi.

Pour communiquer entre nous, nous utilisons la plateforme Discord. Dans celle-ci, nous avons créé plusieurs chaînes textuelles pour mieux organiser nos messages (réunions, ressources, annonces, etc.). De plus, dans le but de travailler ensemble, nous avons créé plusieurs chaînes vocales. Dans celles-ci, les membres peuvent facilement rejoindre d'autres membres dans des discussions et demander de l'aide. Le tout afin de favoriser la collaboration et l'entraide au sein de l'équipe.

Pour ce qui est de notre méthode de travail, nous restons ouvert à la programmation en binôme pour le tâches plus complexes et les décisions architecturales. De plus, pour chaque *merge request*, deux personnes autres que son auteur doivent approuver la branche afin de pouvoir l'intégrer à la branche principale. Ainsi, nous nous assurons que la grande majorité des membres aient compris et approuvé les changements.

Finalement, pour une meilleure organisation et efficacité, nous avons décidé de répartir entre nous les rôles suivants : animateur, secrétaire, porte parole, avocat du diable, coordonnateur et maître du temps [12]. L'animateur, rôle pris par Misha Krieger-Raynauld, se charge d'identifier les points à discuter et de les inscrire dans l'ordre du jour. Aussi, il s'occupe de gérer les tours de parole et de s'assurer que l'on ne s'éloigne pas du sujet lors des discussions. Pour ce qui est du secrétaire, Samer Massaad, son rôle est de prendre des notes durant les réunions et d'organiser les documents produits, comme les ordres du jour. Quant au porte-parole, rôle assumé par Simon Gauvin, ses responsabilités sont de représenter notre équipe lorsque nous devons contacter l'Agence afin d'obtenir des clarifications sur les requis. Le rôle de l'avocat du diable, détenu par Yasmine Moumou, est de poser des questions et proposer des arguments qui vont à contresens des idées ou des propositions. Ceci permet de souligner les faiblesses, les incohérences ou les problèmes dans nos raisonnements ainsi que dans les solutions proposées afin de les éviter. Ensuite, le rôle de coordonnateur, détenu par Nathanaël Beaudoin-Dion, est de s'assurer que l'équipe respecte les échéanciers dans sa réalisation des tâches. Il s'assure de signaler tout retard et d'encourager l'équipe à aller de l'avant. En bref, il garde un oeil sur les dates de remises et rappelle l'équipe des objectifs à atteindre dans le temps. Cela nous permet de nous assurer que le projet avance au rythme prévu. Enfin, Rose Barmani a le rôle du maître du temps. Sa responsabilité principale est la gestion du temps durant les réunions. Le maître du temps assigne un certain temps aux points de l'ordre du jour durant la réunion et signale au groupe si ces limites ne sont pas respectées. Finalement, pour ce qui est du réseau de communication, nous favorisons un réseau décentralisé, c'est-à dire un pouvoir réparti entre tous les membres de l'équipe pour la prise des décisions et la direction générale. Nous bénéficions ainsi des connaissances de tous et faisons ressortir les meilleures idées. À travers tout cela, nous essayons le plus possible de prendre nos décisions par unanimité pour assurer une meilleure cohésion et satisfaction au sein des membres.

2.2 Entente contractuelle

Pour le projet, nous proposons le type d'entente contractuelle livraison clé en main, aussi appelé contrat à terme. Ce contrat implique la livraison d'un produit final qui doit être accepté par le client avant que le paiement soit émis. Cette méthodologie correspond exactement à ce qui est attendu par l'Agence. Effectivement, il est très important pour l'Agence que nous remettions un prototype

fini pour le 12 avril 2021, sans quoi la proposition sera rejetée. De plus, le contrat de livraison clé en main implique une étude approfondie du déroulement du projet préalablement à sa réalisation. Il faut donc planifier et établir les défis technologiques à surmonter, les coûts, la durée, le matériel nécessaire, les acteurs impliqués, le mode organisationnel de l'équipe et plus encore à l'avance. Cet aspect permet une plus grande assurance par rapport aux coût finaux et permet au promoteur de suivre le déroulement du projet. Les critères au niveau du degré de risque pour les coûts et les échéanciers, l'incertitude et la complexité des spécifications, le niveau de de compétition, l'estimation des coûts et l'urgence de la livraison permettent de conclure que le contrat livraison clé en main est le plus approprié pour notre projet.

Dans un premier temps, le degré de risque pour les coûts et échéanciers est assez élevé. En effet, l'Agence Spatiale veut maximiser ses investissements afin de stimuler l'industrie spatiale avec leurs ressources limitées. Ainsi, il est important pour elle d'avoir une solution fonctionnelle et innovatrice au plus bas coût possible et en un délai fixe. Si la proposition du contrat dépasse la limite des coûts et de l'échéancier, celle-ci sera alors rejetée ou devra assumer les dépassements sans rémunération.

Ensuite, pour ce qui est du critère des requis techniques, l'appel d'offres envoyé par l'Agence décrit précisément la majorité des fonctionnalités essentielles à la solution proposée pour le projet. Cependant, un certain degré de liberté est permis à l'entrepreneur pour des requis comme la précision de la carte, puisque ce projet reste après tout un prototype et agit en tant que preuve de concept.

En ce qui concerne le niveau de compétition, il est relativement faible considérant que, malgré que plusieurs équipes aient reçu l'appel d'offres, l'Agence se réserve le droit d'accepter plus d'un projet.

L'Agence nous laisse la liberté d'établir un budget et une estimation des coûts du projet. Cependant, ceux-ci doivent être basés sur des taux fixes horaires pour les développeur-analystes et le coordonnateur de projet avec un maximum de 630 heures-personnes non négociables. De surcroît aux contraintes matérielles nous limitant à des modèles spécifiques, le projet ne permet pas une grande flexibilité budgétaire.

Finalement, il est très important de réitérer que la remise du prototype fini à l'Agence doit se faire en date du 12 avril 2021. Cette date non discutable est importante à la planification du projet.

Ainsi à l'aide de ces cinq critères, nous pouvons facilement confirmer le type de contrat approprié pour ce projet. Un contrat livraison clé en main, aussi appelé contrat à terme, satisfait parfaitement les exigences présentées par l'appel d'offre de l'Agence Spatiale. En effet, la durée maximale du projet ainsi que son budget sont fixes et ne peuvent pas changer. Par contre, les requis importants sont connus d'avance et si l'entrepreneur veut modifier ces requis, une négociation sera alors requise et permise. C'est d'ailleurs dans cette optique que notre CDR vient proposer certains amendements en matière des requis pour respecter le budget.

3 Solution proposée

3.1 Architecture logicielle générale

L'architecture logicielle de la solution proposée comprend quatre composantes principales et une logique de machine à états commune. Celles-ci sont représentées à la figure 1 par différentes couleurs pour les distinguer.

La première est un client Web développé avec Vue.js 3. L'équipe a choisi de développer un client Web

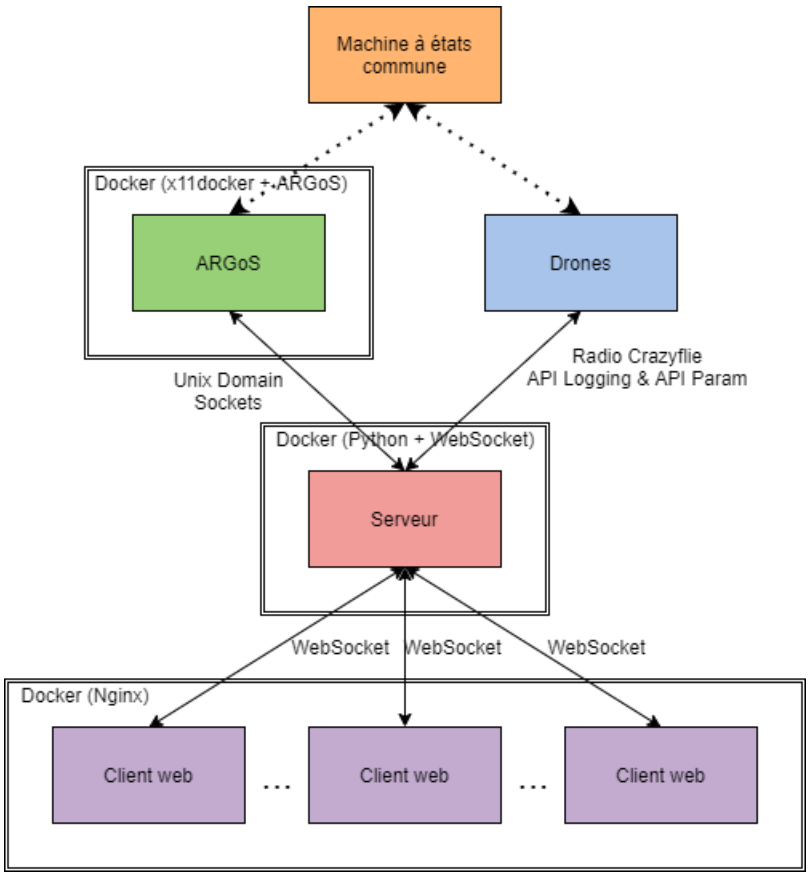


Figure 1 – Diagramme de l'architecture logicielle générale

pour des raisons de portabilité. Effectivement, il doit être possible de visualiser l'interface utilisateur sur différents appareils (R.L.4). Une application Web permet de réutiliser le même client pour un ordinateur et pour un appareil mobile, dans la mesure où celle-ci est conçue avec des composantes adaptatives (*responsive*).

La deuxième composante logicielle correspond au programme en C embarqué sur les drones. Les drones communiquent avec le serveur avec la Crazyradio. Ils explorent un environnement et recueillent des données avec leurs capteurs. Parallèlement, ils envoient ces données au serveur et reçoivent les commandes reliées aux étapes de la mission de celui-ci.

La troisième composante est la simulation des drones avec ARGoS. Cette composante logicielle permet de valider les fonctionnalités dans un environnement simulé avant de les implémenter sur les drones. Cette composante simule aussi la communication avec le serveur en l'absence des drones et de la Crazyradio. Elle lui envoie des données sur l'environnement simulé et reçoit ses commandes.

La dernière composante logicielle est le serveur. Celui-ci utilise la Crazyradio afin de communiquer avec les drones et communique par *sockets* Unix avec la simulation ARGoS. Le serveur permet la connexion simultanée de plusieurs clients pour la visualisation des données et le contrôle des drones. Le serveur est en charge de recevoir et de traiter les données reçues des drones et de les envoyer aux différents clients Web connectés. Il a aussi la responsabilité de recevoir les commandes provenant des clients et de les transmettre aux drones pour changer l'état de la mission.

Initialement, il avait été planifié d'encapsuler la machine à états commune aux drones et à ARGoS

dans une librairie qui serait incluse dans le code embarqué et le code ARGoS à la compilation. Cependant, plusieurs complexités propres à ARGoS et aux drones ont obligé cette idée à être reconsidérée. En effet, le contrôle des drones dans la simulation et le contrôle des Crazyflies ne s'effectuent pas de la même manière. Du côté de la simulation, le contrôle se fait avec des positions. Pour les Crazyflies cependant, le contrôle se fait avec des vitesses pour faire usage de la technique la plus adaptée et fidèle à notre algorithme.

En effet, la librairie de contrôle de Crazyflie offre une abstraction qui permet de se défaire des complexités associées à la stabilisation du drone [15]. Ceci réduit grandement la logique de contrôle et permet de créer un algorithme d'exploration plus puissant. Du côté de la simulation, aucun de ces mécanismes de stabilisation ne sont disponibles ; seules des commandes de positionnement relatif ou absolu sont définies. Malgré que les drones de la simulation et les Crazyflies puissent tous deux être contrôlés avec des positions relatives, la complexité que ce contrôle amène afin d'avoir un comportement réactif à l'environnement n'est pas négligeable : pour un contrôle par positions relatives, des états intermédiaires doivent être utilisés pour attendre la fin d'un mouvement. Cette logique complexifie significativement l'écriture d'une logique réactive au niveau du drone, ajoutant une source d'erreurs supplémentaires. Nous ne pensons pas qu'ajouter de la complexité à notre réel système embarqué simplement afin de faciliter sa simulation est justifiable, surtout si d'autres éléments tels que le manque d'abstraction de la stabilisation du drone sont absents de la simulation.

Ces différences de contrôle rendent la tâche de factorisation du code en commun beaucoup plus complexe et moins utile. Après avoir passé 50 heures supplémentaires à investiguer des solutions possibles pour une couche d'abstraction au niveau de la simulation, il a été décidé de ne pas créer de librairie commune à la simulation et aux drones. Cependant, la même machine à états est toujours utilisée dans les deux composantes logicielles, même si chacune aura une implémentation spécifique à sa plateforme.

Au niveau de la communication, le serveur et le client s'envoient des informations et des commandes par l'entremise de WebSockets. Cette technologie est parfaite pour répondre au requis R.F.5 puisqu'elle permet au client de recevoir les informations du serveur dès qu'elles lui sont disponibles : le serveur peut communiquer par messages envoyés à des fréquences différentes en fonction de leur importance. Le client n'est donc pas responsable de ce requis. Nous avons opté pour l'utilisation de messages JSON pour la communication entre le client et le serveur, puisque son utilisation est simple et bénéficie d'un support natif en TypeScript et quasi-natif en Python. Nous avons aussi opté d'utiliser un système événementiel par *callbacks*, offrant une abstraction simple afin que les fonctions concernées par un évènement soient automatiquement appelées lors de la réception dudit évènement.

Pour sa part, la communication entre le serveur et la simulation ARGoS est gérée par des *Unix Domain Sockets*. Cette technologie permet une communication inter-processus bidirectionnelle élégante et simple. La raison principale du choix de cette technologie est sa simplicité d'utilisation et sa polyvalence. Pour des raisons de symétrie et de simplicité, nous avons également pris la décision utiliser un système de *callbacks* utilisant des messages JSON entre le serveur et la simulation ARGoS.

Enfin, dans le but d'avoir un environnement reproductible et stable pour le produit final, chaque composante logicielle est conteneurisée avec Docker. L'installation et l'exécution de la solution s'en retrouve grandement simplifiée pour l'utilisateur final. Il sera effectivement possible démarrer les conteneurs pour les différents modules par l'appel d'une seule commande sur Linux (R.C.4). Celle-ci sera constituée d'un script Bash permettant d'initialiser les variables d'environnement pour le *X forwarding* avant d'orchestrer les différentes composantes de l'architecture conteneurisée à l'aide de

Docker Compose.

Pour le conteneur du client, Nginx est utilisé pour répondre aux requêtes et servir la page client. Pour ce faire, le conteneur du client ouvre un port à l'aide d'un *flag* `--port`. Par la suite, un client chargeant la page servie par Nginx peut communiquer avec le serveur par l'entremise d'un autre port exposant le serveur WebSocket. Il est ainsi possible d'avoir plusieurs clients sur différents appareils qui se connectent au même serveur sur le même réseau LAN. Pour sa part, tout dépendant de son mode, le conteneur du serveur doit soit communiquer avec le simulateur ARGoS, soit envoyer ses commandes avec le périphérique Crazyradio pour interfacer avec les drones. Pour les drones, le port USB de la radio est exposé au conteneur Docker via un *flag* `--device`. Pour la simulation ARGoS, le *socket* Unix est accessible au serveur grâce à un volume Docker : le répertoire `/tmp/hivexplore/` est partagé par les conteneurs du serveur et de la simulation à l'aide du *flag* `--volume`. Ceci permet donc la communication bidirectionnelle entre les deux containers. Finalement, étant donné qu'ARGoS nécessite une interface graphique, un conteneur démarré à l'aide de *x11docker* est utilisé afin de pouvoir transmettre l'interface graphique peu importe les pilotes graphiques installés.

3.2 Architecture logicielle embarquée

En se tournant plus particulièrement vers l'architecture spécifique du logiciel embarqué, on peut voir à la figure 2 que trois des quatre composantes sont concernées, soit les drones, le serveur et la simulation ARGoS implémentant tous deux une machine à états commune. Une ligne en pointillé représente la démarcation entre la section ARGoS et la section des drones. Ainsi, si le contexte est celui de la simulation, il faut faire abstraction des boîtes bleues à droite de la ligne, représentant les drones. À l'inverse, si l'on souhaite se pencher sur l'architecture de l'implémentation réelle sur les drones, il faut ignorer les boites vertes à gauche de la ligne, représentant ARGoS.

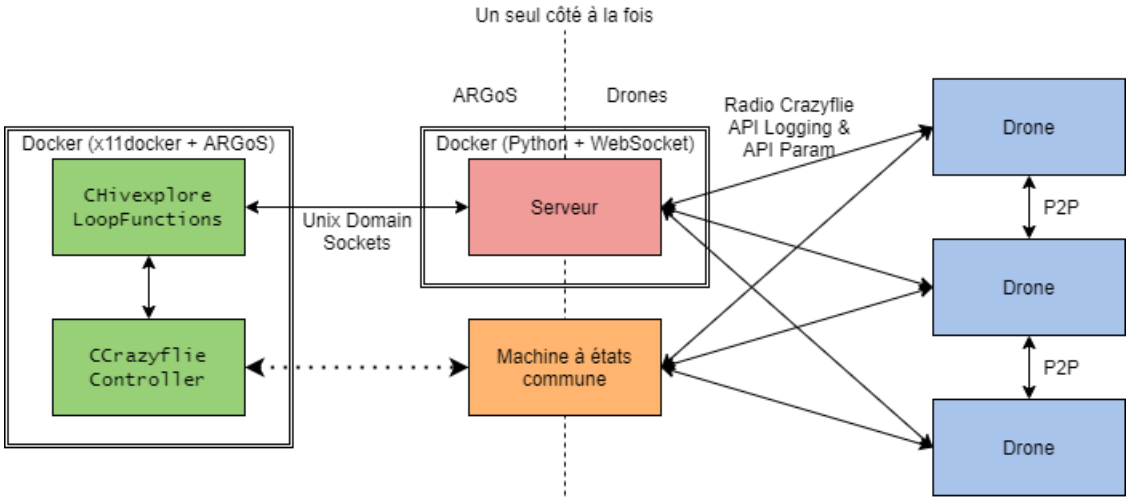


Figure 2 – Diagramme de l'architecture embarquée

En ce qui concerne la communication entre les drones et le serveur, l'hypothèse a été émise qu'un lien radio direct est toujours possible entre la station au sol et les drones. L'information de plusieurs drones peut donc être reçue en même temps par le serveur à l'aide d'une même Crazyradio. Les drones envoient à chaque seconde leur état à la station au sol, où, à l'aide de la radio, il est possible de recevoir les informations désirées de l'essaim en entier. Pour ce qui est de l'envoi des commandes

aux drones, un envoi de type *broadcast* est utilisé. Ainsi, lorsqu'une commande est envoyée - comme « Start mission » - celle-ci est acheminée à tous les drones. Les drones peuvent donc tous commencer la mission de façon autonome et simultanée. Du côté du *firmware*, ces réceptions de logs et envois de commandes sont implémentés à l'aide de l'API de *logging* de Crazyflie ou l'API de *param* configurée avec des macros `ADD_PARAM`, respectivement (R.L.1). Enfin, la mise à jour des informations des drones sur le client est gérée par l'entremise du serveur à l'aide de WebSocket. Ce choix technologique sera approfondi à la section 2.3.

Ensuite, pour faire l'exploration de l'environnement, la communication *peer-to-peer* (P2P) des drones sera mise à profit afin de faciliter la coordination de ceux-ci et éviter l'utilisation du serveur comme relais (R.L.2), puisque la dépendance à un système centralisé limite ce que peuvent accomplir une flotte de drones autonomes [4]. Cette communication peut être implémentée en utilisant l'API *peer-to-peer* de Crazyflie. Pour la simulation avec ARGoS, les capteurs et actuateurs *Range & bearing* permettent un comportement semblable à la communication P2P des Crazyflies. La coordination avec P2P est puissante puisqu'elle délègue à l'essaim de drones une grande marge de manoeuvre pour un algorithme distribué d'exploration.

Étant donné que les drones ont accès au *ranging deck* et au *optical flow deck*, la combinaison de ces modules permet d'avoir la distance dans toutes les directions (avant / gauche / arrière / droite / haut / bas) ainsi qu'un système de positionnement relatif basé sur les variations de position avec un filtre de Kalman.

Du côté de la simulation ARGoS, il va sans dire que celle-ci doit émuler le comportement des drones de la façon la plus réaliste possible. Ceci décrit une des grandes difficultés de la simulation et une contrainte existante dans le domaine de la robotique. Afin de réduire le plus possible les différences entre la simulation et le comportement réel des drones, notre solution prône l'utilisation d'une logique commune pour le déplacement, la communication et la prise de décision. Cependant, tel qu'expliqué dans la présentation de notre architecture logicielle générale, une librairie contenant toute la logique des drones ne sera pas utilisée par les drones Crazyflie et les drones dans la simulation ARGoS à cause de différences dans le contrôle des drones réels et ceux de la simulation. Les implémentations ARGoS des capteurs pour la simulation fournis par l'Agence sont utilisés afin d'émuler le comportement des *decks* des drones. Toutefois, certains capteurs, notamment les capteurs de distance verticale au-dessous et en-dessous du drone ne sont pas disponibles dans la simulation ; il faudrait les implémenter nous-mêmes. La décision a pourtant été prise de ne pas le faire, faute de temps et puisque la position absolue en Z des drones peut être utilisée comme approximation du capteur du bas. Il est aussi présumé qu'il n'y aura jamais d'obstacles venant du haut dans ARGoS, ce qui est une limitation apparente de la simulation.

Par ailleurs, pour ce qui est de la mesure de la puissance de signal (RSSI) reçue par la radio, cette donnée est facilement accessible grâce au *logging framework* de Crazyflie. Or, cette donnée n'est pas accessible dans la simulation ARGoS ; elle a donc été simulée en calculant la distance euclidienne entre les drones et la station au sol. Il est important de noter que la mesure du RSSI contient énormément de bruit avec les Crazyflies, et que ce bruit ne peut pas être facilement simulé avec ARGoS.

Afin de permettre une communication entre le serveur et la simulation ARGoS, les *loop functions* d'ARGoS ont été employées. Celles-ci nous permettent d'y ajouter le code nécessaire pour créer un serveur pour *sockets* Unix et rendre disponible un point d'entrée dans la simulation pour le serveur. C'est là que se trouve le code qui agit d'intermédiaire pour la réception et l'envoi de messages entre le serveur et le contrôleur Crazyflie d'ARGoS.

Un dernier aspect important de l'architecture est l'utilisation d'un code identique pour tous les

drones, puisque cette solution est plus facilement maintenable et extensible avec un grand nombre de drones (R.F.2). Dans l'éventualité où une logique qui requiert des drones avec des rôles spécifiques devrait être utilisée pour les améliorations à l'algorithme d'exploration, n'importe quel drone pourrait assumer les différents rôles de la flotte : les rôles pourraient être échangés entre les drones durant l'exécution de la mission au lieu d'être décidés a priori.

Pour ce qui est du déroulement de la mission, les drones et la simulation ARGoS suivent une même machine à états, présentée à la figure 3.

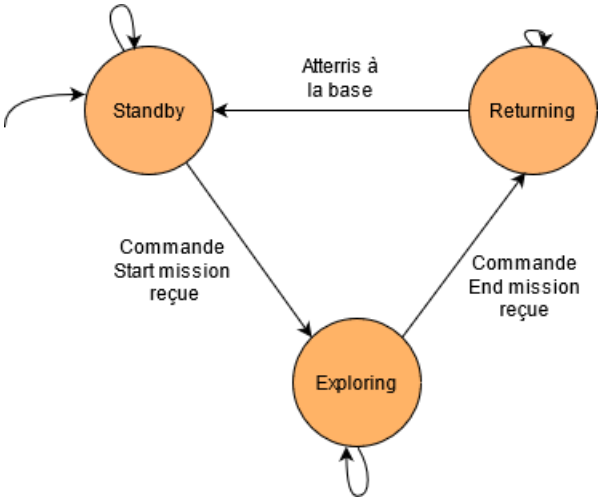


Figure 3 – Machine à états du déroulement de la mission des drones et de la simulation ARGoS

En fonction des commandes envoyées par le client (R.F.4), les drones changent d'état de mission. La logique d'évitement d'obstacles est utilisée pour tous les états du drone qui engendrent un mouvement de celui-ci (R.F.3). Ainsi, en tout temps, les drones s'éloignent des obstacles qui les entourent en appliquant une vitesse dans la direction opposée. Ils débutent à l'état `Standby` et, à la réception de la commande de début de mission, passent à l'état `Exploring`. Une fois à l'état `Exploring`, les drones se retrouvent dans une deuxième machine à états : celle de l'exploration (R.F.1). Cette machine à états est présentée à la figure 4 ci-dessous.

Pour l'instant, l'algorithme d'exploration implémenté est efficace mais assez rudimentaire. Les drones commencent d'abord par décoller. Une fois la hauteur cible atteinte, les drones se retrouvent à l'état `Explore`. Dans cet état, les drones avancent en ligne droite vers l'avant. Dès qu'un obstacle est détecté par le capteur de distance avant, le drone change à l'état `Rotate`, dans le lequel il effectue une rotation jusqu'à ce qu'il n'y ait plus d'obstacle devant lui. Dès qu'il y a suffisamment d'espace en avant, le drone retourne à l'état `Explore`. À la réception d'une commande du client qui met fin à la mission, les drones sortent de l'état `Exploring` et vont dans l'état de mission `Returning`, peu importe l'état de la machine à états d'exploration. Au sein de ce nouvel état de mission se trouve une nouvelle machine à états très simple. Les drones se déplacent vers la base, et une fois celle-ci survolée, descendent jusqu'à ce que le sol soit atteint. Une fois atterris, les drones retournent dans l'état de mission `Standby` pour attendre une nouvelle commande du client pour une potentielle prochaine mission.

Suivant un processus itératif, le premier algorithme d'exploration qui a été développé est relativement simple et consiste surtout en un évitement des obstacles.La structure flexible de la machine à états d'exploration permet cependant d'étendre la solution et de construire une logique plus so-

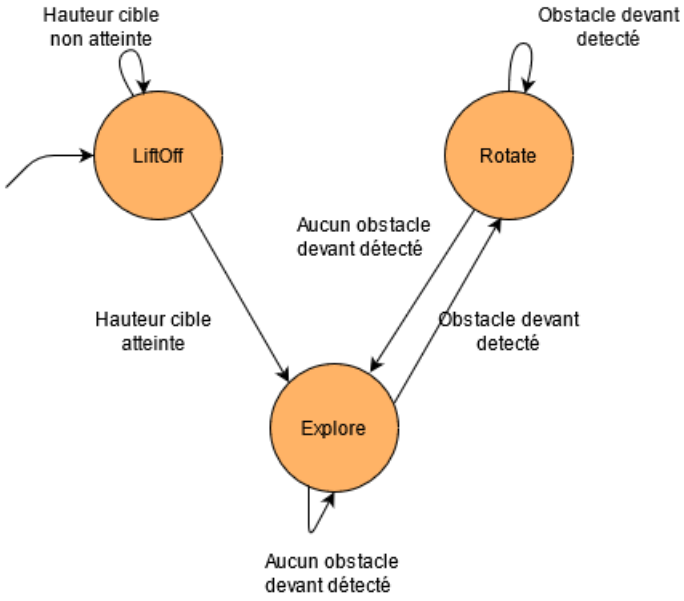


Figure 4 – Machine à états de l'exploration des drones et de la simulation ARGoS

phistiquée pour une exploration plus optimisée. L'ajout d'heuristiques à l'algorithme - par exemple pour rendre la rotation du drone aléatoire ou sinon pour permetttre au drone d'explorer des régions négligées - pourrait pallier plusieurs limitations de l'algorithme actuel.

3.3 Architecture logicielle de la station au sol

Pour l'architecture spécifique de la station au sol, on peut voir à la figure 5 que les quatre composantes sont concernées : ARGoS, les drones, le serveur et le client. Les choix d'architecture et les protocoles de communication sont détaillés dans cette section.

D'abord, le serveur doit pouvoir choisir de communiquer soit avec les drones, soit avec la simulation ARGoS. Pour communiquer avec les drones, `cflib`, la librairie officielle pour Crazyflie, facilite la tâche. Bien qu'il existe des ports en d'autres langues, implémenter le serveur dans la même langue que la librairie officielle est plus simple et évite la possibilité d'erreurs plus rares. Le support à long terme est aussi plus évident en se limitant à une distribution officielle. Il a donc été choisi d'écrire notre serveur en Python.

Pour communiquer avec la simulation ARGoS, trois choix populaires ont été analysés : la mémoire partagée, les FIFOs (*pipes*) ainsi que les *sockets* Unix. Les *sockets* Unix sont reconnus pour leur simplicité d'utilisation et leur puissance, mais parfois aux dépens de leur performance vis-à-vis des alternatives comme les FIFOs. Puisque le système doit pouvoir soutenir une acquisition de données de seulement 1 Hz (R.F.5), la performance de la communication n'est pas un facteur limitant. C'est donc pourquoi les *sockets* Unix ont été choisis en tant que candidat idéal pour la communication bidirectionnelle avec la simulation ARGoS. Le serveur assume le rôle de client et établit une connexion avec le serveur de *sockets* Unix démarré par la *loop function* d'ARGoS. Des paquets associés à des logs ou des paramètres sont alors envoyés pour émuler le comportement de la `cflib` sur ARGoS.

Pour ce qui est de la communication du serveur Python avec le client Web, deux options étaient

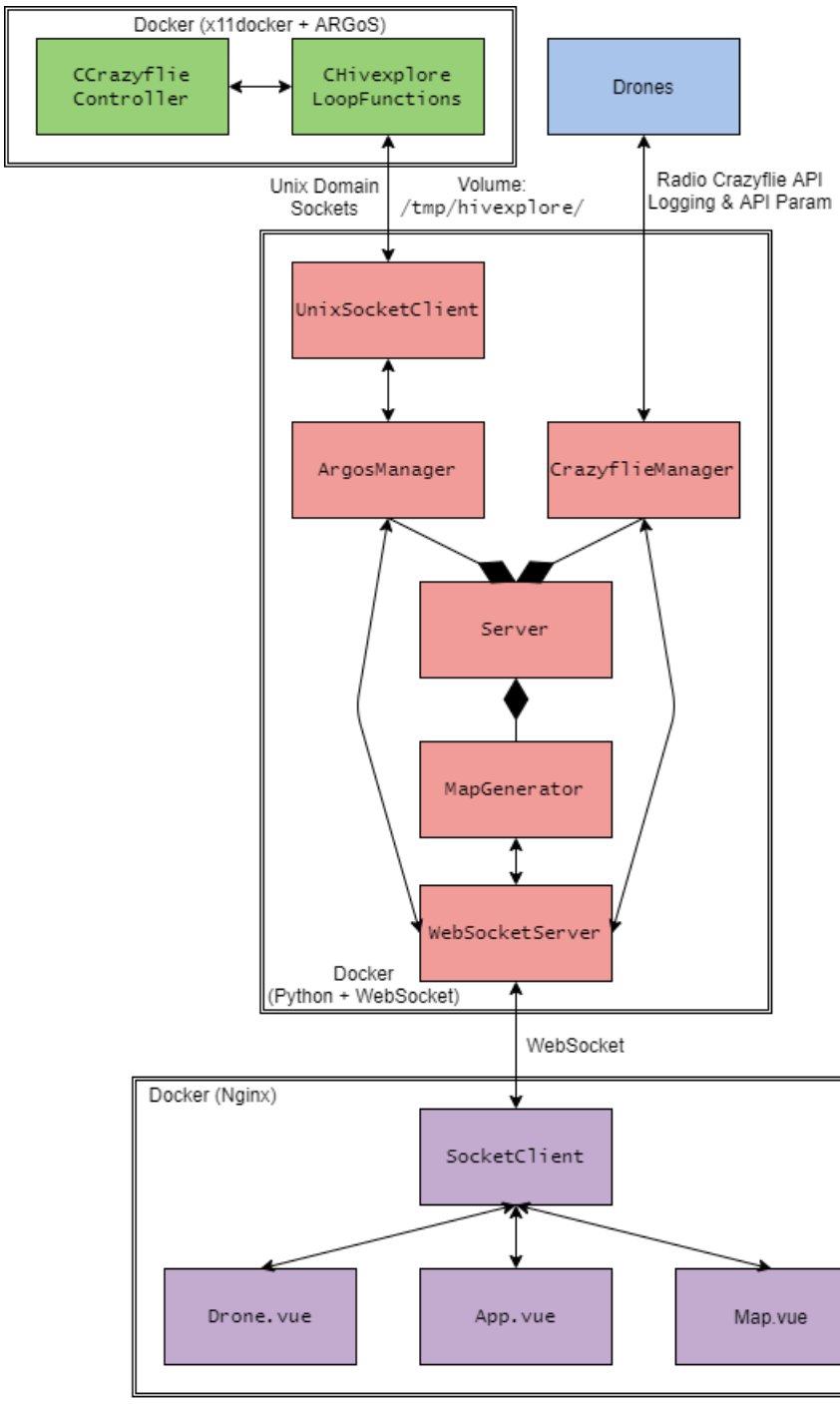


Figure 5 – Diagramme de l'architecture de la station au sol

disponibles, soit les WebSocket ou un API HTTP (potentiellement *RESTful*). Afin d'envoyer les données les plus récentes à tous les clients dès qu'elles sont disponibles, l'utilisation de WebSocket a été retenu comme candidat idéal pour la tâche. En effet, cette technologie permet d'établir une communication bidirectionnelle entre le client et le serveur, selon une interface élégante qui simplifie plusieurs des interactions associées à un état. De cette manière, aucune boucle de requêtes ne doit être maintenue sur le client.

Pour permettre aux divers systèmes de fonctionner de façon simultanée sans se bloquer, les coroutines de Python sont utilisées à leur plein potentiel. La librairie `asyncio` de la librairie standard de Python nous permet de créer des tâches asynchrones qui se partagent équitablement le temps de calcul.

Le serveur est notamment en charge de la génération des points de la carte à partir de la mise en commun des données reçues de chacun des drones (R.F.6, R.L.6). En utilisant les estimations de position 3D et d'orientation (*roll/pitch/yaw*), une matrice de rotation est appliquée sur les points détectés par les capteurs. En générant la carte sur le serveur, la carte est assurée d'être cohérente entre les clients. Les points générés de la carte sont ensuite envoyés au client pour l'affichage. Du côté du client, la carte est visualisée en 3D à l'aide de la librairie Three.js. Cette librairie à été choisie pour sa popularité, sa simplicité, sa performance, et le fait qu'elle puisse afficher de la géométrie en 3D. Ceci nous permet une visualisation très intuitive qui peut être observée de plusieurs angles.

Du côté des données entre le client et le serveur, un nouveau requis requiert une façon d'assurer une persistance des informations des missions. La première partie de ce requis implique de conserver les cartes générées par la mission. Pour ce faire, notre solution propose au client de télécharger la carte à même l'interface Web s'il le désire. Ensuite, la seconde partie du requis concerne la sauvegarde des *logs*. En plus d'être affichés au client au cours de la mission (R.L.5), les logs seront aussi automatiquement enregistrés dans un fichier par le serveur avant d'être envoyés au client. Pour se faire, la librairie standard `logging` de Python sera utilisée. Dans l'éventualité où un utilisateur voudrait avoir accès aux *logs* d'une ancienne mission, il serait toujours possible de se connecter par l'entremise d'une connexion `ssh` ou `ftp` avec le serveur afin de retrouver les fichiers voulus.

Pour le choix de l'interface Web client, l'utilisation du cadriciel Vue.js 3 a été conservée. Cette librairie gagne en popularité depuis quelques années par rapport à ses grands compétiteurs, Angular et React [6]. Une des raisons principales est que Vue.js 3 offre une courbe d'apprentissage moins escarpée. Puisque la complexité du projet est principalement concentrée sur le code embarqué et sur le serveur, il est avantageux d'avoir un cadriciel Web simple d'utilisation. L'utilisation de Vue.js 3 est aussi souvent recommandée pour des petits projets, car le cadriciel offre une expérience de développement élégante et simple [6]. La version utilisée par notre solution est la plus récente de Vue.js, soit Vue.js 3, sortie en septembre 2020. Celle-ci offre un meilleur support Typescript, langue que nous avons choisie pour éviter des erreurs reliées à une mauvaise cohérence des types.

Finalement, une librairie de composantes d'interface graphique pour l'application Web est utilisée afin d'avoir une application visuellement attrayante, uniforme, accessible et adaptative. Pour cette tâche, PrimeVUE a été choisie : pour l'instant, c'est la seule librairie d'interface graphique qui supporte Vue.js 3. Celle-ci simplifie également la création d'une application adaptée aux appareils mobiles (R.L.4).

4 Processus de gestion

4.1 Estimations des coûts du projet

Pour réaliser notre projet, la charge requise est de 630 heures-personnes. Nous sommes une équipe de six développeurs-analystes dont un est aussi coordonnateur de projet. Le taux horaire pour les développeurs-analystes est de 130$/h et de 145$/h pour le coordonnateur de projet. Bien que nous estimons que 60 heures totales seront allouées à la gestion de projet, nous calculons les coûts

du projet en supposant que le taux horaire du coordonnateur de projet ne change pas en fonction de s'il effectue de la gestion ou du développement ; celui-ci reste à 145$/h. En divisant les heures parmi les 6 membres de l'équipe (soit 105 h/personne), nous obtenons un coût de 15 225$ pour le coordonnateur de projet et 68 250$ pour les 5 développeurs-analystes. Ceci équivaut à un coût total relatif aux ressources humaines de 83 475$.

Soixante des 630 heures-personnes sont allouées aux diverses tâches en gestion de projet et les 570 heures restantes seront consacrées au développement. Ceci revient à une moyenne d'un peu plus de 6 heures par sprint (semaine) de gestion et 63 heures de développement hebdomadaire réparti sur les neuf sprints.

4.2 Planification des tâches

Le diagramme de Gantt à la prochaine page détaille visuellement la planification et la répartition des tâches. Il s'agit de celui qui avait été conçu pour la première remise du PDR, maintenant mis à jour avec le progrès des tâches accomplies par l'équipe.

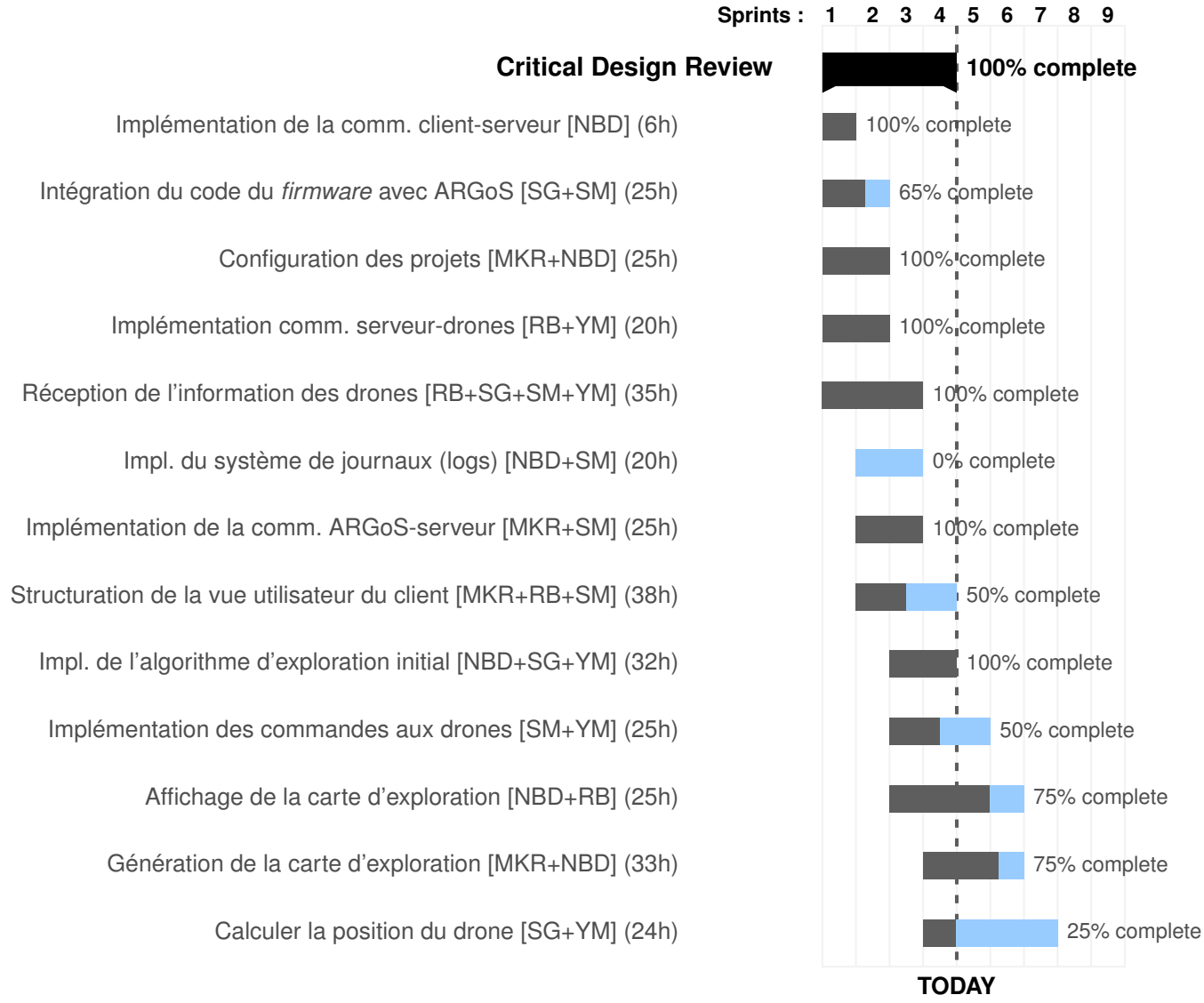
Tel que mentionné dans le rapport du PDR accompagnant le livrable du sprint 2, la charge de travail nécessaire pour le CDR était plus élevée que celle pour le RR. En effet, les tâches qui sont souvent les plus longues en réalité sont de se familiariser avec de nouvelles technologies, définir l'architecture, intégrer les divers modules, implémenter leur communication et avoir un résultat minimalement fonctionnel. En voulant être efficace dans notre méthode de travail, il a été nécessaire de passer plus de temps en début de projet à bien établir établir l'architecture de la communication entre les divers modules de la solution pour avoir une base solide. Sans cela, il nous aurait été nécessaire de faire plusieurs réusinages après le CDR, ce qui aurait été particulièrement inefficace. Étant donné le déséquilibre entre la répartition des tâches pour le CDR et le RR, l'équipe avait proposé de repousser la remise du CDR d'une ou deux semaines.

Afin de pallier ce problème, l'équipe a pris la décision de repousser certaines tâches optionelles, lesquelles seront explicitées plus loin dans cette section. Malgré cela, une énorme charge de travail a dû être assumée : un total de 360 heures de développement et environ 20 heures de réunion en quatre semaines.

Si l'on compare les 360 d'heures dépensées avec la prévision faite pour le PDR, on remarque une différence d'environ 45 heures : il avait été prévu d'allouer que 314 heures de développement pour la remise du CDR et 256 heures pour le RR. Considérant que certaines tâches allouées ont été repoussées pour le CDR, ceci fait que la différence d'heures pourrait être plus grande qu'il ne le semble en réalité.

Légende pour l'assignation des tâches :

- MKR : Misha Krieger-Raynauld
- NBD : Nathanaël Beaudoin-Dion
- RB : Rose Barmani
- SG : Simon Gauvin
- SM : Samer Massaad
- YM : Yasmine Moumou



En effet, en se fiant au diagramme de Gantt, les tâches suivantes ont soit été déplacées à plus tard, soit été annulées :

- Annulée : Intégration du code du *firmware* avec ARGoS (25h)
  - Après avoir passé du temps à essayer de trouver une base commune pour l'algorithme sur le *firmware* Crazyflie et dans la simulation ARGoS, nous avons déterminé que de poursuivre dans cette voie n'en valait finalement pas la peine. Une explication plus approfondie est donnée plus loin.
- Déplacée : Implémentation du système de journaux (logs) (20h)
  - Ce n'était pas un requis nécessaire pour le CDR ; la tâche a donc été déplacée.
- Déplacée : Structuration de la vue utilisateur du client (38h)
  - Il n'était pas nécessaire d'avoir complètement finalisé l'interface utilisateur pour le CDR ; un avancement de 50% a été jugé suffisant pour la démonstration du CDR.

Ainsi, si on enlève les heures prévues mais non dépensées pour ces tâches de la prévision pour le nombre d'heures du PDR, on obtient (314 - 9 - 20 - 19) 266 heures planifiées. Ceci montre clairement que le budget d'heures allouées pour les tâches jusqu'au sprint 4 a été largement dépassé. En effet, ceci représente un écart de (360 - 266) 94 heures investies en supplément à ce qui avait été prévu, ce qui est considérable.

L'écart entre le nombre d'heures prévues (266 heures) et le nombre d'heures réalisées (360 heures) peut surtout être expliqué par trois tâches qui ont pris largement plus de temps que prévu :

- Sprint 1 et 2 : Implémentation de la comm. client-serveur
  - Prévision : 6 heures
  - Dépensé : 32 heures
- Sprint 3 : Implémentation de la comm. ARGoS-serveur
  - Prévision : 25 heures
  - Dépensé : 42 heures
- Sprint 3 et 4 : Impl. de l'algorithme d'exploration initial
  - Prévision : 32 heures
  - Dépensé : 107 heures

En premier lieu, malgré que les membres de l'équipe soient très qualifiés et aient de l'expérience particulièrement pertinente pour ce projet (voir la section 4.4), le projet comporte certains défis technologiques importants, notamment au niveau de l'intégration. En effet, deux des trois tâches qui ont pris beaucoup plus de temps que prévu sont des tâches d'intégration entre le client, le serveur et la simulation ARGoS. L'interaction élégante de plusieurs technologies nécessite un bon travail de recherche technologique et de conception architecturale, choses pour lesquelles la quantité de travail peut être difficile à estimer. Certains défis sont aussi tout simplement complexes, comme la communication entre ARGoS et le serveur, où des *sockets* Unix doivent être utilisés pour créer notre propre protocole applicatif. Globalement, les tâches touchant à ARGoS ont presque toujours été celles qui ont pris plus de temps que prévu. Le logiciel est nouveau pour tous les membres de l'équipe et il amène son lot de complexités additionnelles. Par exemple, pour l'implémentation de l'algorithme d'exploration initial, 70% du temps a été dépensé sur l'implémentation avec ARGoS en tentant de trouver une solution semblable à celle pouvant être réalisée avec le *firmware* Crazyflie.

En deuxième lieu, nous avons souvent eu recours à la programmation en paires *pair programming* pour les tâches effectuées jusqu'à maintenant. Cette technique s'est avérée utile pour résoudre certains problèmes dûs à une complexité technologique, par exemple lors de l'implémentation de l'algorithme d'exploration sur ARGoS. La programmation en paires est très efficace lorsque des décisions d'architecture doivent être prises, lorsqu'un développeur est bloqué sur un tâche ou lorsqu'un réusinage est nécessaire. Cependant, pour d'autres tâches, la programmation en paires peut réduire l'efficacité du temps passé par deux personnes. Par exemple, pour l'implémentation de l'algorithme initial d'exploration, à un certain point, trois développeurs travaillaient en même temps sur la tâche. Si le temps n'est pas utilisé de façon trois fois plus efficace, ceci peut venir augmenter les heures passées sur la tâches d'un facteur important ; si trois développeurs passent une heure sur la tâche, ceci correspond à trois heures du budget.

En dernier lieu, il faut considérer que la très lourde charge de travail nécessaire pour le CDR a sans doute eu un impact néfaste sur le travail effectif réalisé. Les développeurs de l'équipe ont

passé énormément de temps sur le projet - soit (360 heures de développement / 4 semaines) 90 heures/semaine pendant quatre semaines consécutives afin d'atteindre les objectifs du CDR. Si les requis ou la date du livrable du CDR avaient mieux été équilibrés, on devrait normalement s'attendre à (570 heures de développement / 9 semaines) 63 heures/semaine - nettement moins que la moyenne d'heures hebdomadaires des quatre dernières semaines. Sans compter les heures passées en réunion qui sont comptabilisées dans la gestion, ceci représente des semaines 50% plus chargées que le scénario idéal. Ces longues heures de travail nécessaires à l'accomplissement des requis du CDR ont entraîné un stress supplémentaire et une certaine fatigue professionnelle sur chacun des membres. En devant passer autant d'heures dans un si court laps de temps, la productivité a tendance à diminuer. Un cycle vicieux se forme alors rapidement : plus de temps doit être dépensé pour rattraper la productivité perdue ; augmentant alors le temps devant être mis, les heures consécutives de programmation, et réduisant encore plus sévèrement l'efficacité des heures dépensées. Le nombre d'heures nécessaires pour accomplir certaines tâches a donc parfois été victime de la fatigue engendrée par la quantité de fonctionnalités à réaliser pour le CDR.

Néanmoins, certaines tâches ont été plus courtes que prévu. C'est notamment le cas du calcul de distance avec avec la puissance du signal radio via RSSI, où 10 heures ont été estimées alors que le tout a pu se faire en deux heures. Les tâches de génération des points de la carte et de l'affichage de celle-ci ont aussi été plus efficaces que prévues, en parvenant à avoir un résultat plus avancé que ce qui avait été espéré (voir le diagramme de Gantt). Sur une même note positive, la majorité des tâches complexes de recherche, d'intégration et d'architecture sont maintenant accomplies et nous avons aujourd'hui un logiciel fiable de bonne qualité. Même si la charge de travail reste significative pour le RR, l'équipe considère que le plus complexe a été accompli. Nous sommes plus à l'aise avec les technologies utilisées et la majorité des décisions architecturales ont été prises.

Afin d'être plus efficace pour les prochaines semaines, les trois problèmes soulevés seront adressés. D'abord, puisque les membres de l'équipe sont maintenant assez à l'aise avec les technologies utilisées, une augmentation de l'efficacité est attendue et moins de recherche sera requise à chaque étape. D'ailleurs, la quantité impressionnante de progrès accompli lors de ce quatrième sprint nous offre une première confirmation de cette hypothèse. Naturellement, grâce à cette meilleure compréhension des technologies et une architecture en bonne partie terminée, moins de tâches nécessiteront la programmation en paires ; les heures passées devraient alors être plus efficaces. Finalement, avec cette efficacité renouvelée et les tâches plus complexes derrières nous, la charge de travail pour la remise du RR s'annonce moins sévère que celle qui était demandée pour le CDR. La probabilité de fatigue professionnelle et de perte de productivité devrait s'en voir visiblement réduite. Les trois principaux problèmes du CDR ne devraient donc pas avoir le même impact pour les sprints à venir.

En ce qui concerne la charge de travail du RR, les heures restantes nous contraignent malheureusement à revoir notre planification. Étant donné que 360 heures ont déjà été investies, il ne reste alors que (570 - 360) 210 heures pour le budget. Ceci correspond à une charge de travail de (210 / 5) 42 heures par semaine restante, ce qui est clairement inférieur à la charge de travail du CDR de 90 heures/semaine. De plus, en se rappelant que l'estimation faite pour le CDR n'a pas su allouer assez d'heures pour compenser pour les débordements de certaines tâches, nous jugeons que le budget restant n'est pas suffisant pour accomplir tous les requis restants à implémenter pour le RR. Afin de respecter le budget d'heures restantes, nous avons dû annuler les trois tâches suivantes, choisies afin de maximiser le temps regagné en minimisant l'impact sur

le produit final :

— Annulé : Tests unitaires (R.C.2)

— Annulé : Librairie commune à ARGoS et au *firmware*

— Annulé : Implémentation de la mise à jour sans-fil du *firmware* (R.F.4.1)

Pour chacune des tâches annulées, nous nous sommes assurés de n'annuler que des tâches à la fois lourdes pour le développement et les moins prioritaires possibles pour l'Agence pour maximiser l'impact sur notre temps en minimisant les conséquences sur l'Agence. En ce qui concerne les tests unitaires, ceux-ci ont dû être abandonnés pour ne pas avoir à trop couper dans les autres tâches faute de temps. Les nombreuses justifications derrière ce choix sont approfondies avec plus de détails à la section 5.3.

Quoique ce ne soit pas un requis au sens propre mais plutôt une décision architecturale datant du PDR, nous avons aussi dû abandonner l'idée d'établir une librairie commune pour la logique d'ARGoS et du *firmware* Crazyflie. En effet, tel qu'expliqué avec plus de précisions à la section 3.1, cette tâche s'est avérée beaucoup trop compliquée à réaliser pour très peu d'avantages.

Enfin, le requis de la mise à jour sans-fil (R.F.4.1) doit aussi être délaissé pour plusieurs raisons. D'abord, le requis représente une grande source d'incertitudes. Les potentiels problèmes d'intégration et complexités inhérents aux détails d'implémentation pourraient rendre la tâche beaucoup plus compliquée qu'elle ne semble à première vue. Notamment, ceci implique que le serveur conteneurisé devrait également contenir tout le code embarqué et les outils de développement afin de pouvoir compiler le code et le charger sur les drones. Sinon, on pourrait penser à fournir le *firmware* à mettre à jour à même le client, ce qui deviendrait vite compliqué en essayant de définir les paramètres de format et de transmission.

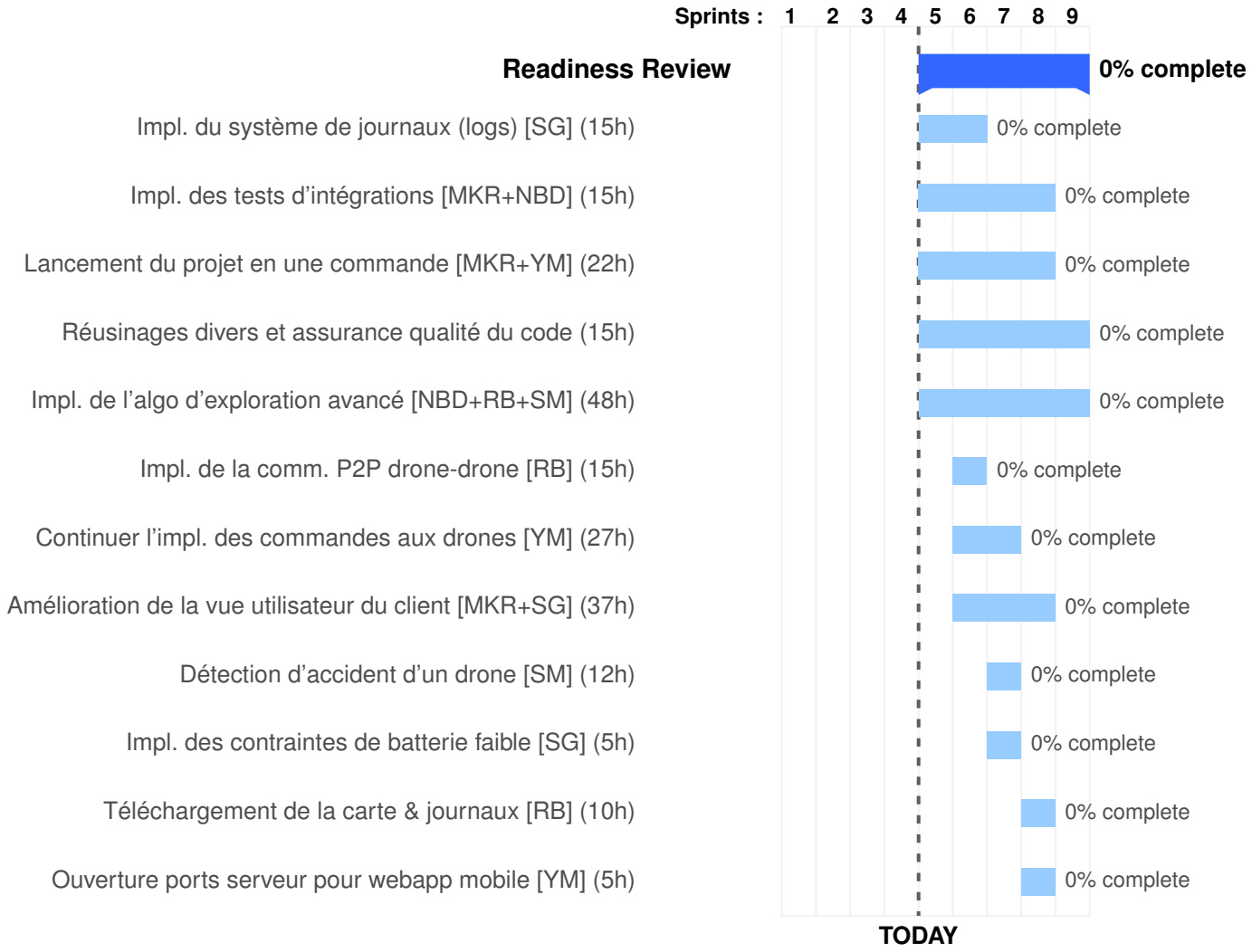
Il faut aussi considérer ce que ce requis apporterait en ce qui concerne la valeur ajoutée pour l'Agence. En effet, pour que la mise à jour du code embarqué sur les drones soit pertinente, ledit code embarqué doit avoir été préalablement été programmé par un quelconque développeur. Dans la mesure où l'on se met à modifier le code, il est très aisé de le charger sur les drones - il suffit de faire appel à la commande `make cload` dans le répertoire du code embarqué ; il est déjà possible de mettre à jour le drone de manière sans-fil. Il serait probablement même plus compliqué de passer à travers le client pour le faire - surtout si le requis précise que le drone doit être à la retourné à la base de toute façon - puisque l'environnement de développement serait nécessairement déjà ouvert. On pourrait aussi penser à des potentielles craintes de sécurité et vulnérabilité : veut-on nécessairement permettre à un opérateur de modifier le code embarqué de façon potentiellement destructrice ? Il n'est donc pas très surprenant que le requis se classe parmi les moins prioritaires selon le document d'évaluation des priorités remis par l'Agence.

Au vu de ces raisons, il a été décidé que le requis de mise à jour du code embarqué à partir du client n'est pas assez important et qu'il constitue le meilleur requis à abandonner. Il aurait sinon fallu couper un autre requis potentiellement plus intéressant pour l'agence.

Sinon, de façon plus générale, certaines tâches ont été simplifiées afin de réduire le nombre d'heures nécessaire. Le requis de la position des drones (R.F.5) pour l'intégration des mesures des différents drones a été simplifié en présumant que la position initiale des drones est connue (R.F.7) et en utilisant la puissance de signal du RSSI (R.L.3). De cette façon, il ne sera pas nécessaire de résoudre un problème d'optimisation pour fusionner les données de drones commençant à des positions de départ différentes. Le téléchargement des journaux (logs) a aussi été simplifié pour se faire automatiquement du côté serveur alors que la carte du client pourra simplement être téléchargée par l'interface utilisateur si l'usager le désire.

En somme, ces coupures sont nécessaires afin de nous permettre de réduire la charge de travail et respecter le budget global de temps alloué au projet.

Le diagramme ci-dessous illustre le travail révisé qui sera effectué d'ici la complétion du projet. Le temps associé aux tâches a été réajusté selon la nouvelle planification et certaines tâches ont été fusionnées par rapport à l'ancien diagramme présenté dans le rapport du PDR. Les heures et l'organisation des sprints peuvent aussi être consultées dans notre tableau GitLab nommé « Backlog » à l'adresse suivante : `https://gitlab.com/polytechnique-montr-al/inf3995/20211/equipe-102/hivexplore/-/boards/2361638`.



La somme des heures allouées pour les tâches jusqu'au RR donne un total de 226 heures. Ceci est légèrement supérieur au temps restant, soit de 210 heures, mais reste sinon relativement fidèle au budget.

Il est bien sûr concevable que certaines tâches puissent prendre plus d'heures qu'initialement estimé, comme cela s'est produit pour le livrable du CDR. Pour combler les incertitudes associées à certaines tâches moins évidentes comme la communication *peer-to-peer* pouvant potentiellement prendre plus de temps que prévu, un tampon supplémentaire de 40 heures en surplus aux 226

heures estimées nous est alloué. Le total estimé revient donc à (226 + 40) 266 heures, soit 56 heures de plus que ce que le budget d'heures de développement nous permet. Ceci représente un dépassement maximal *dans le pire cas* de 10% des 570 heures allouées pour le développement.

Les termes du contrat clé en main doivent néanmoins toujours être respectés, et aucun coût additionnel ne sera imparti à l'Agence. Ainsi, le coût du projet de 83 475$ basé sur 60 heures de gestion et 570 heures de développement maximum restera le même si les heures dépassement légèrement le budget envisagé. Toute heure supplémentaire qui devrait être investie dans le *pire* des cas sera considérée comme de l'*overtime* qui ne sera pas chargé à l'Agence. Les membres de l'équipe ont été consultés face à cette réalité et ils ont tous accepté ces termes. Ainsi, le budget et le contrat sont respectés.

4.3 Calendrier de projet

Le tableau ci-dessous présente un résumé du calendrier général du projet, avec les dates de fin pour chaque jalon principal.

Calendrier	
Jalons	**Dates**
Preliminary Design Review	15 février 2021
Critical Design Review	8 mars 2021
Readiness Review	12 avril 2021

4.4 Ressources humaines du projet

D'abord, comme mentionné lors de la section 4.1, l'équipe est composée de six développeurs-analystes dont un membre est aussi coordonnateur de projet. Les membres sont tous des étudiants en génie informatique à Polytechnique Montréal ; cinq d'entre eux sont en troisième année et le sixième est en quatrième année.

Tous les développeurs ont une certaine expérience en développement Web et en programmation embarquée, deux compétences importantes pour ce projet. Au delà de l'expérience acquise lors du parcours scolaire, chacun a également des connaissances supplémentaires acquises lors de la réalisation de projets personnels, de participation dans des sociétés techniques ou lors de stages en entreprise.

En premier lieu, Nathanaël Beaudoin-Dion a une bonne expérience en embarqué. En effet, il est habile avec les mêmes technologies utilisées que dans ce projet, comme la programmation des microcontrôleurs de la famille STM32F4 en C/C++ et la gestion du système d'exploitation en temps réel avec FreeRTOS. De plus, lors de son dernier stage, il a acquis des connaissances dans plusieurs outils de développement comme GitLab, ses pipelines CI/CD, Docker et `systemd-nspawn`. Finalement, il a aussi une expérience en gestion de projet, étant le directeur du département d'avionique de la société technique Oronos.

Samer Massaad a une bonne expérience avec le développement de tests pertinents et robustes dans le contexte d'un projet Web et d'un serveur. En plus de son expérience en tests, durant son stage, Samer a acquis beaucoup de connaissances sur les différentes méthodes de communication possibles entre un serveur et son client. Pour l'aspect embarqué du projet, sa contribution

sera pertinente avec son expérience au sein de la société technique Oronos dans le département d'avionique.

Simon Gauvin, de son côté, a des bonnes connaissances en C++ grâce à ses plusieurs projets personnels, son stage de recherche et à son expérience professionnelle dans l'industrie du jeu vidéo. Sa capacité de résolution de problèmes et de priorisation des tâches est la bienvenue dans l'équipe.

Yasmine Moumou, quant à elle, a de l'expérience en développement Web Angular, acquise dans le cadre d'un stage. Aussi, étant membre d'une société technique de robotique, elle a de l'expérience en apprentissage machine, plus spécifiquement en « computer vision ». Finalement, elle a aussi auparavant mis le pied dans le domaine du jeu vidéo à travers un projet personnel en C++.

Quant à Rose Barmani, elle est chargée de laboratoire pour le cours « Noyau d'un système d'exploitation ». De plus, elle a de l'expérience en gestion d'équipe, acquis au sein du comité Poly-FI. Aussi, son souci du détail est un atout indispensable à l'équipe. Finalement, elle a également participé à la compétition de robotique CRC.

Enfin, Misha Krieger-Raynauld a beaucoup d'expérience en C++, langue qu'il a utilisée dans plusieurs projets personnels - dont un engin de jeux vidéo - et qu'il a enseignée en tant que chargé de laboratoire pour le cours d'INF1010. Il a également de l'expérience en Python et en développement Web qu'il a acquises à travers son expérience professionnelle en tant que consultant pendant 3 ans. Finalement, il a approfondi son expérience en gestion de projet et de l'utilisation de WebSockets lors de son dernier stage chez Microsoft. Ses connaissances de Linux, des outils de DevOps et les aspects plus poussés du C++ sont un atout à l'équipe.

5 Suivi de projet et contrôle

5.1 Contrôle de la qualité

Afin de répondre aux standards et requis exigés par l'Agence en ce qui concerne les biens livrables, notre équipe suit un processus rigoureux permettant d'assurer le respect et le contrôle continu de la qualité tout au long du développement.

En premier lieu, nous misons sur la mise en place de divers outils de développement afin de prévenir des erreurs de qualité avant même leur inception. Lorsque pertinent, des outils d'analyse statique (*linters*) sont employés pour détecter des erreurs de style, de respect des bonnes pratiques et signaler les portions problématiques de code. Par exemple, du côté du client en TypeScript, nous employons ESLint ; du côté serveur en Python, nous avons opté pour Pylint. En parallèle, des outils de formattage automatique du code source garantissent une certaine uniformité au sein de l'équipe de développement pour nous permettre de consacrer du temps aux éléments plus critiques. À cet effet, nous utilisons Clang-format pour le C et le C++, Prettier pour le TypeScript, le HTML et le CSS, et enfin yapf et mypy pour le Python. Pour que ces outils soient faciles d'accès, ceux-ci sont intégrés au système de compilation (*build system*) et sont exécutables à même l'environnement de développement, soit VS Code dans notre cas.

Bien sûr, les outils d'assurance qualité pour le développement ne sont qu'utiles s'ils sont employés avec assiduité. C'est pourquoi ceux-ci sont appelés dans le pipeline CI/CD afin d'assurer que tout code proposé ait bel et bien passé ces vérifications statiques. Si le code proposé d'une *merge*

request (MR) comporte des erreurs de *linting* ou de compilation, celle-ci ne pourra être acceptée jusqu'à ce que ceci soit résolu.

Dans cette même optique de contrôle de la qualité en prévention des erreurs, nous avons aussi opté pour des choix technologiques qui nous permettent de remarquer des erreurs le plus tôt possible. Du côté du client, le TypeScript a été choisi plutôt que le JavaScript puisque son typage statique nous permet de détecter des erreurs de programmation et d'augmenter la lisibilité du code. Du côté du serveur, nous utilisons mypy en conjonction avec des annotations de types afin d'assurer une certaine cohérence des types en Python.

Toutes ces mesures nous permettent donc d'affirmer avec certitude que les requis de qualité R.Q.1, R.Q.2 et R.Q.3 seront respectés.

Cependant, ces outils ne sont pas une panacée. Un processus de révision sur GitLab par chacun des coéquipiers apporte de grands bienfaits au niveau de l'intégration, de l'organisation du code, de la planification de l'implémentation et une vérification que chaque ajout est bel et bien la solution optimale pour chaque problème rencontré. Pour ce faire, une intégration très fréquente est faite en tout temps à l'aide de *merge requests* (MR) atomiques, spécifiques et encapsulées. En intégrant souvent de plus petites modifications au lieu d'intégrer de grosses modifications à un plus lent rythme, notre équipe atteint une mise en commun plus agile et harmonieuse, peut facilement retracer des régressions, et ce qui est sur la branche principale représente en tout temps un produit déployable pour faciliter la rétroaction. Avec cette approche, les bogues et les problèmes d'intégration sont détectés le plus tôt possible dans le cycle du développement. Par le biais de ces révisions fréquentes de code, chaque membre de l'équipe a la chance et la responsabilité (grâce à un nombre minimal d'approbations) de comprendre les différents aspects du projet. Ceci a l'avantage de mener à une meilleure compréhension globale du projet par chacun pour avoir une solution absente de troubles d'intégration et de problèmes de cohérence. Nous sommes donc confiants que les requis R.Q.3 et R.Q.4 seront bien respectés.

Du côté des techniques de développement, notre approche est basée sur une utilisation extensive du simulateur ARGoS afin de tester le bon fonctionnement de nos algorithmes avant même de les charger sur les drones. Ceci permet également à chaque membre de l'équipe de contribuer au développement du code embarqué sans nécessairement avoir les drones en sa possession. Grâce à ceci, nous pouvons bénéficier de la contribution et des idées de chacun, menant à un meilleur produit qui peut être validé par plusieurs membres de l'équipe lors des révisions.

En terminant, avant chaque remise de livrable, nous vérifierons que tous les objectifs ont bien été atteints en faisant appel à une liste de vérification contenant tous les requis ainsi que certains points additionnels. Cette liste de vérification pourra être conservée dans la section Wiki du répertoire GitLab.

5.2 Gestion de risque

En dépit de nos attentes et notre engagement à livrer un produit à la hauteur des attentes de l'Agence, il faut admettre que le projet comporte certains risques. Il faut avant tout mentionner le contexte inhabituel de la pandémie qui nous impose une méthode de travail à distance. Alors que certaines parties du projet s'en trouvent facilitées, par exemple pour ce qui est de la tenue des réunions, d'autres aspects pourraient s'avérer plus difficiles. On peut par exemple penser aux tests réels des drones dans leur environnement physique. En devant travailler à distance et en isolement les uns des autres, chaque membre de l'équipe n'aura pas un accès facile aux drones pour tester

ce qui a trait au code embarqué. Pour pallier ceci, nous avons planifié que le drone puisse changer de mains pour certains sprints lorsque différents membres de l'équipe souhaitent travailler sur le code embarqué. De plus, notre utilisation du simulateur ARGoS devrait nous permettre de valider une bonne partie du code sur les drones sans avoir à en être en possession, advenant le cas où le drone ne pourrait être transféré à un de nos membres.

Cependant, l'utilisation du simulateur ARGoS ajoute en elle-même une couche supplémentaire de travail. En effet, comme il a été expliqué plus haut, il a été nécessaire de réimplementer certains modules du code embarqué Crazyflie dans ARGoS ou d'utiliser des modules dans ARGoS qui n'ont pas un comportement parfaitement similaire à celui des modules du code embarqué Crazyflie pour pouvoir tester nos drones correctement ; par exemple pour la communication avec le système de contrôle par *sockets* Unix ou le calcul de la distance avec RSSI. Ce genre de travail a ralenti le développement de fonctionnalités qui auraient autrement été assez simples à mettre en place uniquement avec le *firmware* Crazyflie existant.

Certaines de nos hypothèses pourraient aussi s'avérer fausses. Notamment, nous avons supposé que la Crazyradio est capable de rester en contact avec tous les drones en même temps, sans perte ou dégradation de signal. Si ceci se révèle être faux, nous avons un plan de contingence. Il serait possible d'utiliser la communication peer-to-peer pour acheminer les données à un seul drone qui, lui, serait à l'intérieur de la portée de l'antenne de la station au sol. De cette façon, les drones plus éloignés pourraient toujours envoyer leurs informations malgré l'absence de signal direct. Heureusement, nos tests préliminaires montrent que ceci ne sera normalement pas un problème.

Un bris d'équipement pourrait aussi venir retarder le développement. Advenant le cas où un des drones serait mis hors d'usage, nous prévoyons continuer le développement à l'aide du simulateur et le drone restant, de sorte que le progrès ne s'en verrait pas ralenti le temps de commander des pièces de rechange.

Enfin, il peut également être pertinent de mentionner un risque au niveau de la santé et la sécurité : les hélices des drones représentent un danger pour les yeux lorsque les drones sont testés dans leur environnnement réel. Pour éviter toute violation des normes de la CNESST, chaque membre de l'équipe portera systématiquement des lunettes de sécurité lorsque les drones sont allumés et prêts à voler.

5.3 Tests

Afin de respecter le requis R.C.2, il avait initialement été planifié que notre solution logicielle et matérielle serait accompagnée d'une suite de tests unitaires. Cependant, après réévaluation des heures restantes pour le projet, l'analyse des priorités pour le client et le potentiel maximal de valeur ajoutée qu'apporteraient ceux-ci, nous avons à regret dû faire le choix d'annuler l'écriture de tests unitaires. Cette décision n'a pas été prise à la légère et se base sur plusieurs facteurs.

Tout d'abord, il faut considérer la valeur qu'apporterait le développement de tests pour le client, l'Agence Spatiale, en lien avec ses objectifs primaires. Tel que mentionné dans la demande de proposition de l'Agence, le produit recherché consiste en une *preuve de concept* pour un logiciel d'exploration à l'aide d'un essaim de drones munis de capteurs, et ce, principalement à des fins de démonstration pour confirmer la faisabilité du concept. Le prototype fonctionnel de NMS 4 doit également servir à stimuler l'industrie en mettant en valeur des aspects innovateurs de la solution proposée. Il est donc logique de supposer qu'il importe à l'Agence d'avoir un produit à la fine

pointe de la technologie pour maximiser ses investissements en recherche et développement.

Or, l'écriture de tests unitaires engendre un surcoût non négligeable à l'inclusion de fonctionnalités. En effet, le fait de devoir mettre en place des tests unitaires pour chaque fonctionnalité entraîne une augmentation du temps de développement d'au moins 50% selon nos estimations et notre expérience. Cela est sans compter le fait que certains tests deviennent vite désuets et doivent être réécrits en début de projet dirigé selon un processus itératif aux remaniements fréquents. Dans un contexte où le nombre d'heures maximal limite l'allocation des ressources, l'inclusion obligatoire de tests unitaires pour chaque fonctionnalité serait nuisible au résultat innovateur espéré : une plus grande partie de requis fonctionnels devrait être abandonnée pour consacrer le temps nécessaire à l'écriture de tests unitaires pour les requis conservés.

En considérant les intérêts de l'Agence et la priorité plus faible qu'elle accorde au requis R.C.2 (qui n'est pas en gras selon le document de priorité des requis), nous pensons donc qu'il est préférable de se concentrer sur la réalisation d'un prototype innovateur plutôt que sur la conception de tests unitaires.

Tout n'est pas pour autant perdu du point de vue de la valeur ajoutée pour les tests. Nous avons déjà pris le temps de mettre en place l'infrastructure de tests unitaires. Du côté du client, le cadriciel de tests pour Vue.js a été configuré à l'aide de Jest, que nous avons favorisé par rapport à une combinaison de Mocha + Chai par désir de simplicité. Pour le serveur en Python, nous avons configuré Pytest puisqu'il offre une flexibilité maximale avec ses fonctionnalités modernes tout en étant facile pour les cas plus simples. Enfin, le cadriciel de tests intégré au *firmware* Crazyflie [1] est disponible pour tout ce qui touche au code embarqué. Ainsi, dans la mesure où l'Agence souhaiterait poursuivre le développement de notre solution en lui ajoutant des tests unitaires, elle peut être assurée que l'architecture logicielle est dans un bon état pour le lui permettre.

D'autre part, même si nous avons opté de déprioriser les tests unitaires au sens propre, notre solution sera toujours testée de plusieurs façons alternatives. Dans certains scénarios, il peut être très efficace et simple de procéder à une vérification manuelle dirigée selon une procédure préétablie sous la forme d'une liste de contrôle. C'est justement cette approche qui sera employée pour tester les éléments visuels, le comportement des drones, la simulation ARGoS et certains aspects d'intégration.

La simulation ARGoS est en effet utilisée en priorité à la programmation sur les drones afin de tester le bon fonctionnement de notre solution avant tout essai avec les vrais drones dans leur environnement physique qui pourrait les endommager (R.C.1). On pourrait cependant se questionner sur l'impact de l'absence d'une librairie commune pour la logique du drone sur la qualité de l'expérimentation avec ARGoS, situation introduite plus tôt dans l'architecture. Heureusement, une approche simultanée de *software-in-the-loop* [13] et *hardware-in-the-loop* [14] permet de remédier à ce problème. Malgré deux implémentations différentes, la machine à états pour la logique du drone demeure le même. Notre approche se détaille comme suit.

En premier lieu, la simulation ARGoS permet de valider le bon comportement de l'algorithme de vol des drones. En implémentant la logique dans ARGoS, il est très aisé de détecter les problèmes possibles avec la solution en la testant dans un environnement simulé ; ceci reflète l'approche *software-in-the-loop* qui permet d'éviter de programmer les drones avec une logique dangereuse.

Ensuite, si la simulation fonctionne bien et donne le résultat escompté, il est temps de passer à l'implémentation de cette même machine à états sur le réel code embarqué des drones. En guise de précaution supplémentaire, les actuateurs du drone ne sont pas immédiatement allumés. Grâce à l'interface de haut niveau que nous offre le *firmware* Crazyflie [15] , il est très simple

de faire abstraction de la logique de contrôle des hélices pour se concentrer uniquement sur la validation de la machine à états. En utilisant nos mains pour simuler les obstacles avec un drone statique et en inspectant les messages de statut envoyés à la console pour déterminer l'état du drone, on utilise une méthode s'apparentant à une simulation *hardware-in-the-loop*. De cette façon, une quantité maximal de problèmes de logique sont évités à chaque étape avant d'utiliser le drone avec ses actuateurs.

Pour que les simulations ARGoS servent également à tester de façon cohérente, reproductible et exhaustive à l'avenir, nous créerons une liste de contrôle détaillant toutes les conditions à vérifier manuellement dans la simulation. Cette vérification pourra également être ajoutée en tant qu'étape manuelle du pipeline CI/CD de GitLab pour agir de rappel contre les oublis. Enfin, une génération aléatoire d'environnements dans la simulation ARGoS permet de tester des situations plus rares et détecter davantage de problèmes rapidement (R.C.5).

Malgré l'absence de tests unitaires, nous prenons plusieurs mesures afin de détecter des régressions évidentes pour respecter une version simplifiée du requis R.C.3. En effet, le pipeline CI/CD permet de valider automatiquement la compilation, le passage de l'analyse statique (*linting*) et la cohérence des types en Python (*type checking*). Grâce à ceci, chaque nouvelle fonctionnalité sera automatiquement validée afin d'empêcher qu'elle cause une régression entraînant des erreurs de qualité ou de compilation, sans quoi le code ne pourra être accepté.

5.4 Gestion de configuration

Du côté de notre système de gestion de configuration, notre solution est organisée sous un même répertoire Git divisé en quatre sous-projets : le client, le serveur, le code embarqué pour les drones et la simulation ARGoS. Nous avons opté pour le style *monorepo* - c'est-à-dire un seul répertoire contenant les sous-projets qui forment un tout - plutôt que différents répertoires pour plusieurs raisons. L'utilisation d'un seul répertoire diminue le risque que des changements asymétriques brisent accidentellement l'intégration entre les sous-systèmes à cause de *merge requests* séparées pour des modifications touchant à plusieurs projets. De plus, certains éléments communs et centraux tels que la configuration du pipeline CI/CD, les `README` de documentation, les configurations pour les outils de développement et l'intégration du tout sous un script de composition sont plus faciles à gérer sous un même répertoire plutôt qu'avoir des éléments dupliqués ou isolés d'un projet à l'autre.

Pour la documentation, notre projet mise sur les `README`s afin de communiquer toute information essentielle au développement et à l'utilisation de Hivexplore, organisés de façon hiérarchique pour donner que l'information pertinente à chaque projet. Pour de la documentation optionnelle plus poussée que nous souhaitons écrire à l'interne, nous ferons appel aux pages Markdown du Wiki de notre répertoire GitLab. C'est là que ce trouverons nos listes de contrôle pour nos tests manuels, nos documents de conventions, les informations supplémentaires pour certaines interfaces, ou enfin même des collections de liens et de ressources utiles.

Pour ce qui est du code source, nous priorisons avant tout le respect des conventions propres à chaque langue de programmation. Du côté du client, nous avons suivi la hiérarchie des projets Vue 3 telle que générée par le Vue CLI. Pour le serveur en Python, nous avons suivi l'organisation recommandée pour des *packages*. Enfin, les projets C et C++ suivent la structure déjà donnée par le code du *firmware* Crazyflie et des exemples de simulations ARGoS, respectivement.

Du côté de l'environnement de développement, nous avons opté pour une configuration intégrée

de plusieurs outils à travers des mécanismes d'orchestration de commandes. Pour le client, nous utilisons les scripts `npm`, et pour le serveur, la simulation ARGoS et le code embarqué, des Makefiles de commandes permettent d'appeler tous les outils allant du compilateur jusqu'au formatteur. Plusieurs de ces outils sont également configurés pour fonctionner à même l'environnement de développement, soit VS Code dans le cas des développeurs de notre équipe. Avec un *multi-root workspace*, des configurations de débogage et l'intégration des options d'extensions pour qu'elles fonctionnent avec notre projet, nous pouvons ainsi profiter de nos outils à leur plein potentiel et avoir une bonne cadence.

À ces outils s'ajoute enfin le choix d'utiliser Docker pour la conteneurisation. En effet, certains programmes comme ARGoS nécessitent beaucoup de dépendances qu'il peut être incommode d'installer sur une machine hôte. Pour ce cas, nous utilisons alors une image Docker de développement pour conteneuriser ARGoS pour qu'il soit identique peu importe la machine hôte et qu'il puisse s'intégrer à VS Code. Pour notre client Vue, les dépendances identiques sont assurées au moyen d'un `package-lock.json` et `node_modules` avec npm. Pour le serveur en Python, nous utilisons de façon analogue un fichier `requirements.txt` et un `venv` pour les dépendances. Enfin, tel que précisé dans l'architecture, des conteneurs Docker sont utilisés pour conteneuriser le client, le serveur, et la simulation ARGoS pour un environnement de production.

La conteneurisation avec Docker des divers modules nous facilitera l'implémentation de l'objectif final, soit le démarrage aisé de l'entièreté de Hivexplore à l'aide d'un seul script sous Linux (R.C.4). Plus particulièrement, ce script Bash s'occupera d'initialiser certaines variables d'environnement avant de déléguer l'orchestration des conteneurs à Docker Compose.

Références

[1] "Unit testing," Bitcraze. [en ligne]. Disponible : `https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/development/unit_testing/`. [Accédé : 12-Fev-2021].

[2] "Flow deck v2," Bitcraze. [en ligne]. Disponible : `https://www.bitcraze.io/products/flow-deck-v2/`. [Accédé : 11-Fev-2021].

[3] "Multi-ranger deck," Bitcraze. [en ligne]. Disponible : `https://www.bitcraze.io/products/multi-ranger-deck/`. [Accédé : 10-Fev-2021].

[4] St-Onge D., Varadharajan V., Svogor I. et Beltrame G., "From Design to Deployment : Decentralized Coordination of Heterogeneous Robotic Teams", Mai 2020. [en ligne] : `https://www.frontiersin.org/articles/10.3389/frobt.2020.00051/full` [Accédé : 9-Fev-2021]

[5] A. Topiwala, P. Inani, et A. Kathpal, "Frontier Based Exploration for Autonomous Robot", 2018. [en ligne] : `https://arxiv.org/ftp/arxiv/papers/1806/1806.03581.pdf` [Accédé : 27-Fev-2021]

[6] S. Daityari, "Angular vs React vs Vue : Which Framework to Choose in 2021," CodeinWP, 18-Dec-2020. [en ligne]. Disponible : `https://www.codeinwp.com/blog/angular-vs-vue-vs-react/`. [Accédé : 7-Fev-2021].

[7] Bitcraze, "bitcraze/crazyflie-lib-python," GitHub. [en ligne]. Disponible : `https://github.com/bitcraze/crazyflie-lib-python`. [Accédé : 9-Fev-2021].

[8] The ARGoS website. [en ligne]. Disponible : `https://www.argos-sim.info/`. [Accédé : 9-Fev-2021].

[9] Vue.js. [en ligne]. Disponible : https://v3.vuejs.org/. [Accédé : 10-Fev-2021].

[10] "Crazyflie 2.1," Bitcraze. [en ligne]. Disponible : https://www.bitcraze.io/products/crazyflie-2-1/. [Accédé : 11-Fev-2021].

[11] "Web technology for developers," Web APIs | MDN. [en ligne]. Disponible : https://developer.mozilla.org/en-US/docs/web/API/WebSocket. [Accédé : 10-Fev-2021].

[12] "Les rôles et leurs avantages," Travailler en équipe | Attribuer des rôles dans l'équipe | Les rôles et leurs avantages. [en ligne]. Disponible : https://ernest.hec.ca/video/DAIP/travailler-en-equipe/etudiants/attribuer-roles/roles.html. [Accédé : 13-Fev-2021].

[13] M. S. Xiang Chen, "Real Time Software-in-the-Loop Simulation for Control Performance Validation - Xiang Chen, Meranda Salem, Tuhin Das, Xiaoqun Chen, 2008," SAGE Journals, 01-Jan-1970. [en ligne]. Disponible : https://journals.sagepub.com/doi/10.1177/0037549708097420. [Accédé : 01-Mar-2021].

[14] "On hardware-in-the-loop simulation," IEEE Xplore. [en ligne]. Disponible : https://ieeexplore.ieee.org/document/1582653. [Accédé : 02-Mar-2021].

[15] "Controllers in the Crazyflie," Bitcraze. [en ligne]. Disponible : https://www.bitcraze.io/documentation/repository/crazyflie-firmware/2020.04/functional-areas/controllers/. [Accédé : 26-Fev-2021].