INF6600 Rapport TP4
Améliorations au système de contrôle d'un drone fermier

Daniel Lussier-Lévesque et Ian Gagnon

***Abstract*—L'implémentation d'un système de contrôle de drone fermier a été réalisé progressivement tout au long de la session en commençant par un module TrueTime avec un système continu déjà implémenté en Simulink. Par la suite, l'implémentation a continué sur une machine virtuelle QNX roulant sous VMware, avec certaines fonctions déjà fournies. Ce rapport se concentre sur ce qui a été fait sur cette base de code pour améliorer les aspects les plus importants. La première étape a été d'implémenter un système de log qui nous permettent de voir comment se comporte l'ordonnancement des tâches et la performance du système. Par la suite, malgré le manque de droits d'accès qui nous permettraient de comprendre l'ensemble des problèmes de performances, nous avons toutefois été capable de les mesurer et de les documenter. Un système graphique vivant en dehors de la machine virtuelle nous permettant de valider le comportement du système a aussi été implémenté.**

I. Introduction

Le système de contrôle considéré est celui d'un drone autonome capable de se déplacer sur trois axes et ayant une caméra fixe pour prendre en photo l'ensemble d'un champs. Le drone doit naviguer les champs et prendre des photos jusqu'à ce que la mémoire soit pleine, puis transmettre les photos à une station base. Lorsque la batterie est presque vide (10%), le drone doit retourner à la station base pour être rechargé.

Le système de contrôle implémente le contrôle de la navigation (où le drone va), le contrôle de la batterie (est-il temps d'aller recharger la batterie?) et le contrôle mission (quelle séquence d'étapes doit être exécutés pour opérer avec succès?).

Le système de contrôle de drone sur lequel nous avons apportés nos modifications est un système simple où la communication entre le système continu et discret est effectué par des variables atomiques et par l'enregistrement de *callbacks*. La communication en dehors du système s'effectue par des messages de log sur la console de debug où à intervalle régulier (chaque seconde) l'état du drone est envoyé. Toutes les tâches sont asynchrones les unes par rapport aux autres et roulent sur leur propre fil d'exécution. Nous avons choisi de mettre à jour le système de contrôle du drone toutes les 100 ms.

De plus, l'implémentation du système de contrôle doit être accompagné d'une plateforme de simulation d'un environnement dans lequel opérer, qui comprend le temps de réponse de la caméra, les variations d'orientation et de vitesses, le temps de transmission de photo à la station base et la charge et décharge de la batterie. Le système continu est mis à jour toutes les 20ms de sorte qu'il soit significativement plus rapide que le système de contrôle. De cette manière, le système discret ne voit pas énormément de différence dépendamment s'il se fait juste avant ou juste après une mise-à-jour du système continu.

L'ensemble des photos prises peut être visualisé à travers une carte du monde qui est mis à jour à chaque transmission et montre quelle région a été prise en photo. La carte doit être accédée en allant chercher les fichiers sur le système de fichier de la machine virtuelle.

La politique d'ordonnancement utilisée est un systèmes de priorités fixes et un ordonnancement FIFO pour chaque niveau de priorité. Le choix a été fait de placer la priorité du contrôle caméra avant celle du contrôleur de navigation parce qu'il est plus facile de compenser pour une échéance manquée dans le cas de la navigation (une photo manquée implique de défaire les dernières mise-à-jour de navigation pour reprendre la photo).

La machine virtuelle roule sous QNX 7.0 sous VMware Workstation avec un hôte Windows 10. La machine hôte à un processeur d'Intel à quatre coeurs, le i7 3770, avec 16 GB de mémoire DDR3 et utilise un Seagate ST1000DM003 comme disque dur.

II. Lacunes du système de base

Le système décrit plus haut a plusieurs lacunes importantes. Nous allons ici nous focaliser sur les trois principales faiblesses qui nous ont empêcher de réaliser correctement plusieurs améliorations auxquels nous avions pensés. Il va s'en dire que nous avons penser plus important de régler ces lacunes au meilleur de nos capacités en priorité.

A. Peu de visibilité sur l'état du système

Le principal problème est qu'avec le système de base, nous ne savons pas si le système est *correct*, c'est à dire que le système effectue les actions que l'ont s'attend à ce qu'il fasse. Les messages de logs qui sont envoyés sont répétitifs et il est facile de manquer un comportement anormal dans les 1800 messages que comportent une séquence d'exécution. De plus, il est impossible de savoir ce qui se passe à l'intérieur de cette période d'une seconde. une période plus courte n'est toutefois pas envisageable: il serait impossible d'analyser l'entièreté du message en temps réel et donc encore plus facile de manquer une information inattendue. Il n'est pas non plus envisageable d'analyser les données à la fin de la simulation: la console de debug ne peut contenir qu'un nombre limité de messages.

En conséquence, il n'est pas possible de qualifier le système d'aucune manière: un problème de comportement du système demanderait une correction importante au code qui invaliderait l'ensemble des mesures effectués. Comme une séance de simulation dure 30 minutes, il serait difficile d'arriver à un ensemble de résultat suffisant dans un temps raisonnables si nos mesures devaient être invalidés. Sans mesures fiables, il est impossible de déterminer si une tentative d'amélioration du système améliore réellement le système. Pour cette raison, cette lacune est considérée comme la plus importante du système implémenté.

B. Aucune visibilité sur la qualité de l'ordonnancement

En assumant que le comportement du système est correct, nous ne pouvons toujours pas penser à améliorer les performances du système ou rajouter de nouvelles fonctions utiles parce que nous n'avons aucune métrique de performance de disponible. Nous ne savons pas si les tâches manquent leur échéance de manière régulière et arrivent tout juste à rester fonctionnels, ou au contraire si il y a largement de ressources disponibles pour de nouvelles tâches.

Dans le cas où le comportement du système n'est pas celui attendu, les métriques de qualité d'exécution nous sont aussi utile pour comprendre les problèmes. Si une tâche est en famine parce que les tâches plus prioritaires prennent toutes les ressources, ce sera facilement visible et évitera une longue séance de debug en aveugle. Nous considérons cette lacune comme fondamentale et bloquante pour des améliorations à d'autres aspect du système.

C. Pics de latence

Après avoir apportés les modifications qui seront détaillés à la section III, nous avons réalisés que le système tel que simulé avait encore une lacune qui bloquait des améliorations que nous avions planifiés: le système est affecté par des pics de latences non-déterministes. Ces pics de latences affectent toutes les tâches en même temps. Plus spécifiquement, ils affectent des tâches qui n'acquirent pas de mutex où autre verrou global et communiquent uniquement par des variables atomiques. On peut donc conclure que ces pics arrivent au niveau système et non au niveau de l'ordonnancement ou au niveau de la communication.

En plus d'affecter les performances du systèmes et de le rendre moins fiable, un impact important de ces pics de latences est que toutes les métriques de pire temps d'exécution ou de déviations standard sont affectés de manière aléatoire et qu'une comparaison avant et après d'une tentative d'amélioration est noyée dans le bruit de ces pics.

Les pics des latences durent plusieurs millisecondes. Plusieurs tentatives de contourner le problème ont été essayé: réduire le nombre de cœurs du système de contrôle, augmenter le nombre de cœurs du système et déplacer la machine virtuelle du disque réseau au disque dur local. Déplacer la machine virtuelle à permis de réduire significativement la durée de ces pics, mais aucune des ces tentatives n'a permis de ramener les variations de latences à des temps raisonnables.

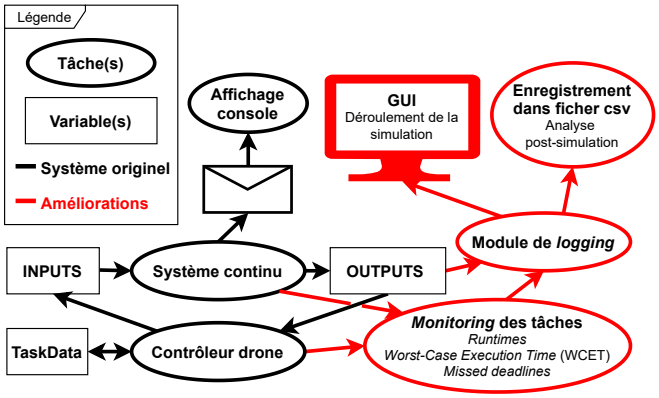

Fig. 1: Diagramme de tâches du système

III. Améliorations apportées

Des lacunes présentées à la section précédente, les deux premières concernant la visibilité du système ont été identifiées comme prioritaires. Pour y remédier, des améliorations sont proposées et illustrées à la Figure 1. Un *monitoring* des tâches récolte d'abord les métriques d'exécution pour chaque tâche, un module de *logging* sert ensuite à regrouper les informations pour finalement les enregistrer en format csv. En supplément, un *Graphical User Interface* (GUI) permet de suivre le déroulement de la simulation.

A. Monitoring des tâches

Pour chaque tâche qui s'exécute, le temps d'exécution est calculé en faisant la différence entre les temps initial et final obtenus par la fonction `std::chrono::steady_clock::now()` et stockés dans des variables atomiques. Ainsi, la durée de la dernière exécution (*last runtime*), le pire cas (WCET pour *Worst-Case Execution Time*) et les échéances manquées (*missed deadlines*) sont comptabilisés. Notons, que l'horloge du système, avec sa granularité d'une milliseconde, limite la précision des données.

B. Logging

Une fois que nous avons implanté le code pour avoir l'information dans des variables, il faut penser à un moyen de la communiquer. Il a donc fallu implémenter une nouvelle fonction (avec le code de *monitoring* associé) pour lire l'ensemble des variable et les transmettre vers les nouvelles sorties. Nous avons décider d'exécuter la tâche toutes les 100 millisecondes afin de ne pas surcharger le système. Cette fonction est appelée par la tâche de *monitoring* déjà existante qui auparavant ne poussait les donnée que vers la console une fois toute les secondes.

C. Sortie en fichier csv

La sortie qui permet de faire une analyse compréhensive du système est un simple fichier csv qui est sauvegardé en mémoire non-volatile au fur et à mesure que la simulation avance. Le format csv étant supporté par pratiquement tous

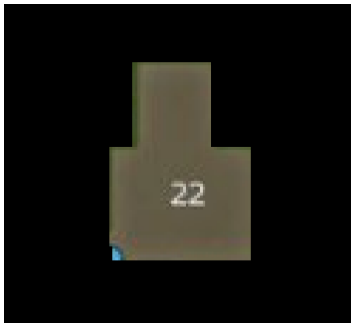

Fig. 2: Image agrandie de la zone couverte par le drone

les outils d'analyse et de base de données, il est donc facile de s'en servir pour obtenir des statistiques sur un grand nombre d'exécutions pour ensuite les formater adéquatement en tableaux, en graphiques ou en générant un rapport (e.g. en JSON). En effet, que ce soit à l'aide de scripts (avec Python, R ou MATLAB) ou manuellement (avec Microsoft Excel ou LibreOffice Calc), l'analyse sera infiniment plus poussée que celle obtenue avec la sortie en console. De plus, cette approche permet d'étudier en détails les circonstances d'un *crash*.

D. GUI

Un GUI a été implémenté en Python afin de suivre l'avancement de la simulation. Comme il s'agit d'une preuve de concept et que le focus de ce projet est l'implémentation sous QNX, l'interface graphique est limitée à un strict minimum (voir section Résultats). L'accent est plutôt mis sur le transfert de données. À cet égard, l'hypothèse (très réaliste) que le drone transmet ces images à la base via une connexion réseau est émise et cette connexion est exploitée pour établir la communication avec le GUI en utilisant un *socket* internet avec le protocole TCP/IP. Comme il n'y a pas de *End Of Transmission* (EOT) sur un *socket*, une convention doit être choisie afin d'assurer un bon transfert de données: message d'une longueur fixe, message délimité (i.e. avec caractères spéciaux de début et fin) ou message avec une entête indiquant sa longueur. Ici, pour garder le tout le plus simple possible, la longueur des paquets est fixée et des caractères de remplissage (i.e. des espaces) sont ajoutés à la fin des trames de données pour ajuster la taille des paquets. De plus, un délai sur la connexion, une gestion de déconnexion ainsi qu'un avertissement de fin de simulation sont ajoutés pour prévenir les erreurs et éviter les *deadlocks*.

IV. Résultats

A. Analyse du système

Avant d'analyser le système en détails à l'aide du fichier csv fournit par nos améliorations, la fonctionnalité de l'implémentation de base est validée à la l'aide de l'image en sortie de la simulation générée par la fonction *transmitPhotos()*. L'image agrandie (Figure 2) démontre effectivement que l'intégralité du champ 22, soit le champ balayé lors de la simulation, a été photographié.

#TODO Parler des vrais résultats et mettre les graphiques

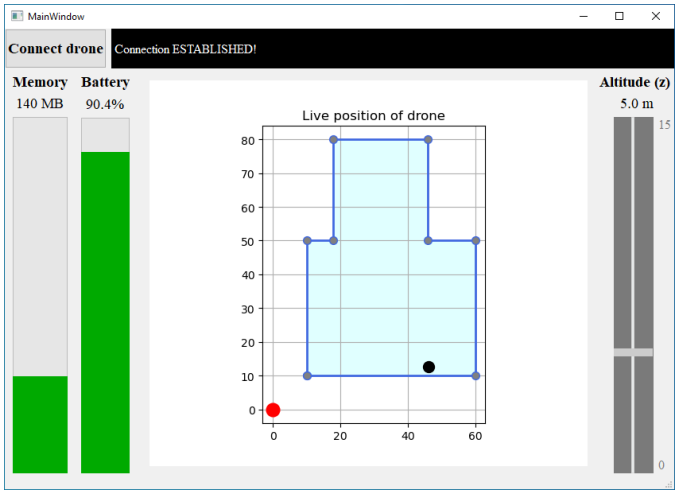

Fig. 3: GUI durant la simulation

B. Description du GUI

Dans la communication TCP/IP, le serveur roule sur QNX et le client sur le GUI. Après avoir ouvert le GUI, il faut donc d'abord lancer la simulation QNX (qui va partir le serveur) et ensuite appuyer sur le bouton "Connect drone" pour établir la connexion. Le GUI, présenté à la Figure 3, affiche les paramètres de base de la simulation, soit: la mémoire, le niveau de batterie, la position du drone (x, y) et son altitude (z). De plus, une vidéo montrant le GUI en fonction (i.e. la simulation en accélérée) est disponible: INF6600-GUI-DEMO[1].

V. Conclusion

Dans cet article, nous avons décrit les améliorations apportées à l'implémentation d'un système de contrôle de drone fermier. L'analyse du système de base a soulevé des lacunes importantes en ce qui concerne les visibilités de l'état du système et de la qualité de l'ordonnancement. En effet, ce manque d'informations rend l'analyse du système et de ses performances laborieuse pour ne pas dire impossible.

Afin de pallier à ce manque de données, nous avons dans un premier temps implémenté un *monitoring* des tâches avec un module de *logging* permettant de recueillir de précieuses informations et de les enregistrer pour les analyser ultérieurement. Cela nous a permis d'étudier le système et de réaliser que...

#TODO Parler brièvement des conclusions de l'analyse

Dans un second temps, nous avons ajouté un GUI communiquant avec le drone via une connexion réseau utilisant le protocole TCP/IP. Ce dernier affiche l'état du système durant la simulation et aide l'utilisateur à repérer un comportement inattendu du drone. Dans l'optique d'un déploiement réel du système sur une ferme, cette interface graphique roulant au niveau de la base (vraisemblablement la maison du fermier) s'avérerait utile puisqu'elle permettrait au fermier du suivre le drone lors de sa mission et de réagir en cas de problèmes.

Cependant, les améliorations apportées ont des limites...

Il y a plusieurs directions pour les travaux futurs. Premièrement, il serait intéressant d'analyser les performances du système en utilisant plus de *cores* (8 par exemple). Une

telle parallélisation des tâches, si bien réalisée, pourrait considérablement augmenter la rapidité du système.

Deuxièmement, la sécurité du système pourrait être rehaussée par l'utilisation de partitions. En isolant les processus de cette manière, on éviterait que la défaillance d'une tâche fasse planter tout le système. Par exemple, une erreur du système de communication n'aurait aucun impact sur le système de navigation, ce qui n'est pas une certitude présentement.

Dernièrement, d'un point de vue sécurité pour une utilisation réelle du système, il serait pertinent d'ajouter un bouton d'urgence à l'interface graphique pour arrêter le drone de force ou le ramener immédiatement à la base dans le cas où l'algorithme de contrôle aurait un raté (e.g. un mauvais calcul qui enverrait le drone à 1000 m d'altitude).

REFERENCES

[1] Ian Gagnon, Daniel Lussier-Lévesque, *Demonstration du GUI pour drône*,
https://drive.google.com/file/d/
1W9yt0Z4YxCwr0TjXkgv12WVO9TSmWGtv/view?usp=sharing.