INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

RAPPORT DE PROJET

Antoine ALAVERDOV, Clémence LEMEILLEUR, Clément VIGAND

Promo 56, Année 2021/2022 – 4IR-SI-B1

« Projet Robot, Temps Réel »

Février 2022

Encadrant : J.Tury

RAPPORT DE PROJET

Antoine ALAVERDOV, Clémence LEMEILLEUR, Clément VIGAND
Promo 56, Année 2021/2022 – 4IR-SI-B1

"Projet Robot, Temps réel"
Février 2022

Encadrant: J.Tury

SOMMAIRE :

I- Conception du système

A- Diagramme fonctionnel général

Voici un diagramme fonctionnel qui présente les principaux blocs de notre conception. On y retrouve pour chaque groupe de thread les données et ports partagés. Les 3 groupes de threads son visibles : th_group_gestion_moniteur, th_group_vision, th_group_gestion_robot.

Voici la légende qui sera appliquée à tous nos diagrammes :

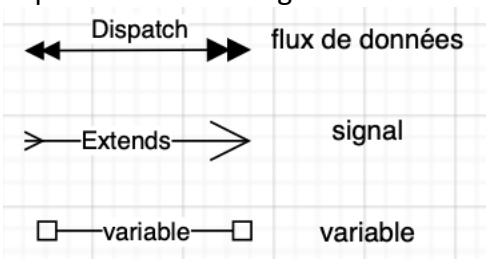

Fig. 1: Légendes des diagrammes

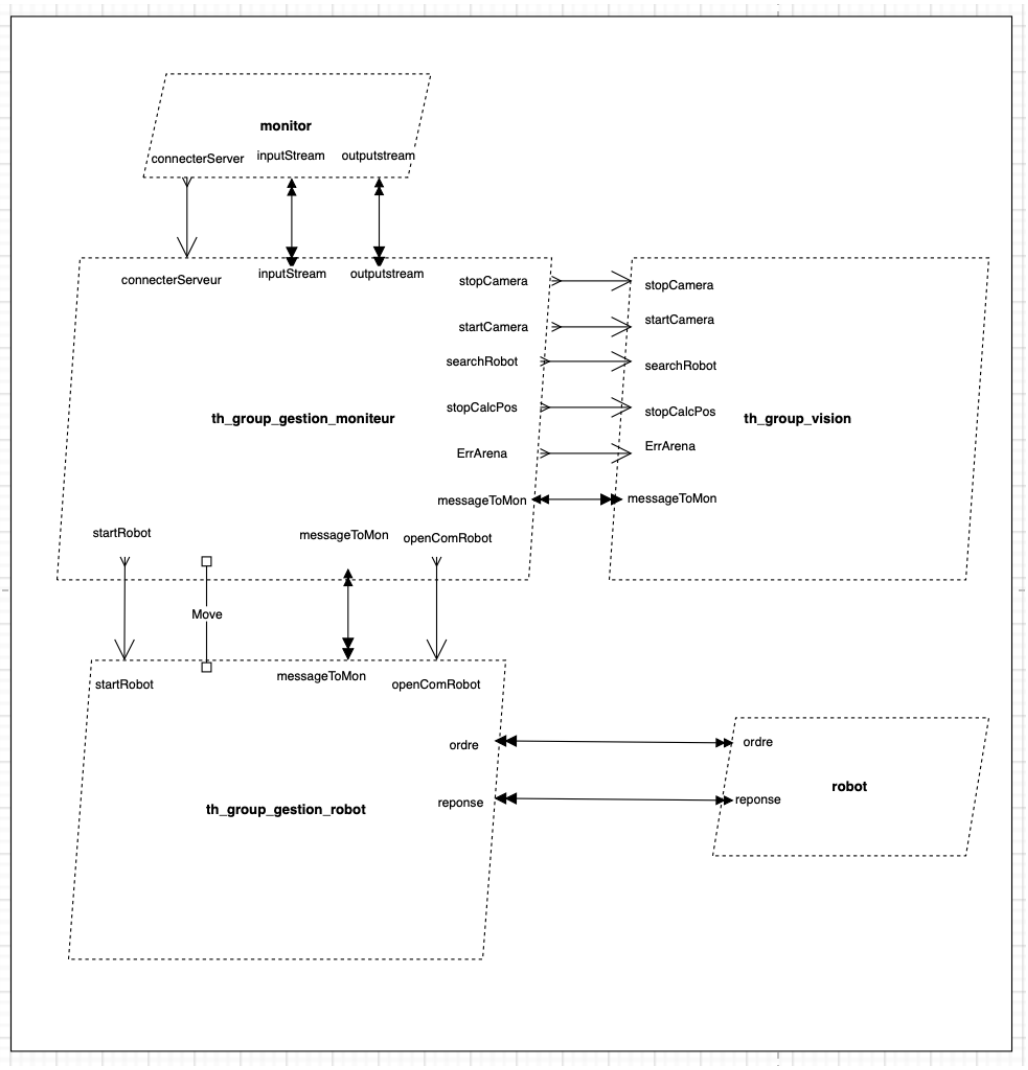

Fig. 2: Diagramme fonctionnel du système

B- Groupe de threads gestion du moniteur

Dans cette partie, nous retrouvons le diagramme fonctionnel en AADL décrivant le groupe de threads de gestion du moniteur et les diagrammes d'activité de chaque thread de ce groupe.

1- Diagramme fonctionnel du groupe de gestion du moniteur

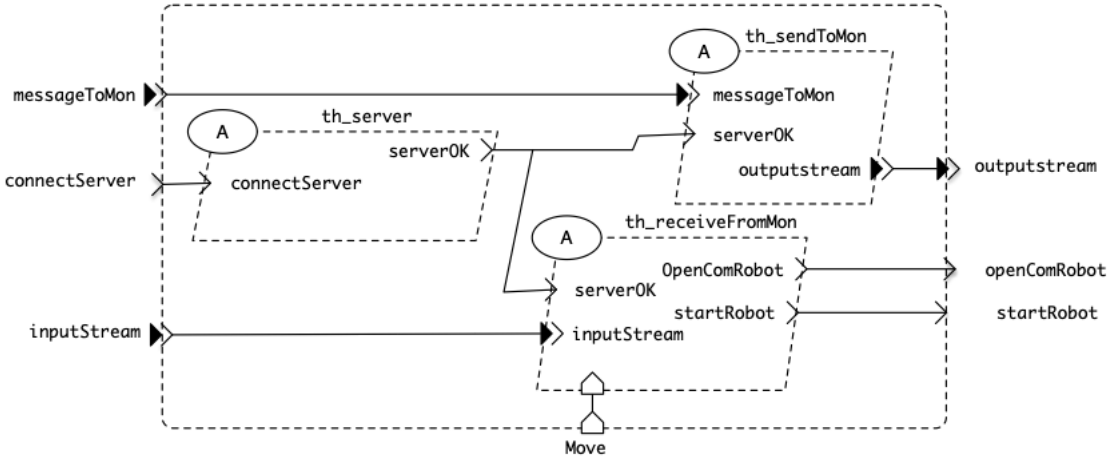

Fig. 3: Diagramme fonctionnel du groupe de threads gestion du moniteur

2- Diagramme d'activité du groupe gestion du moniteur

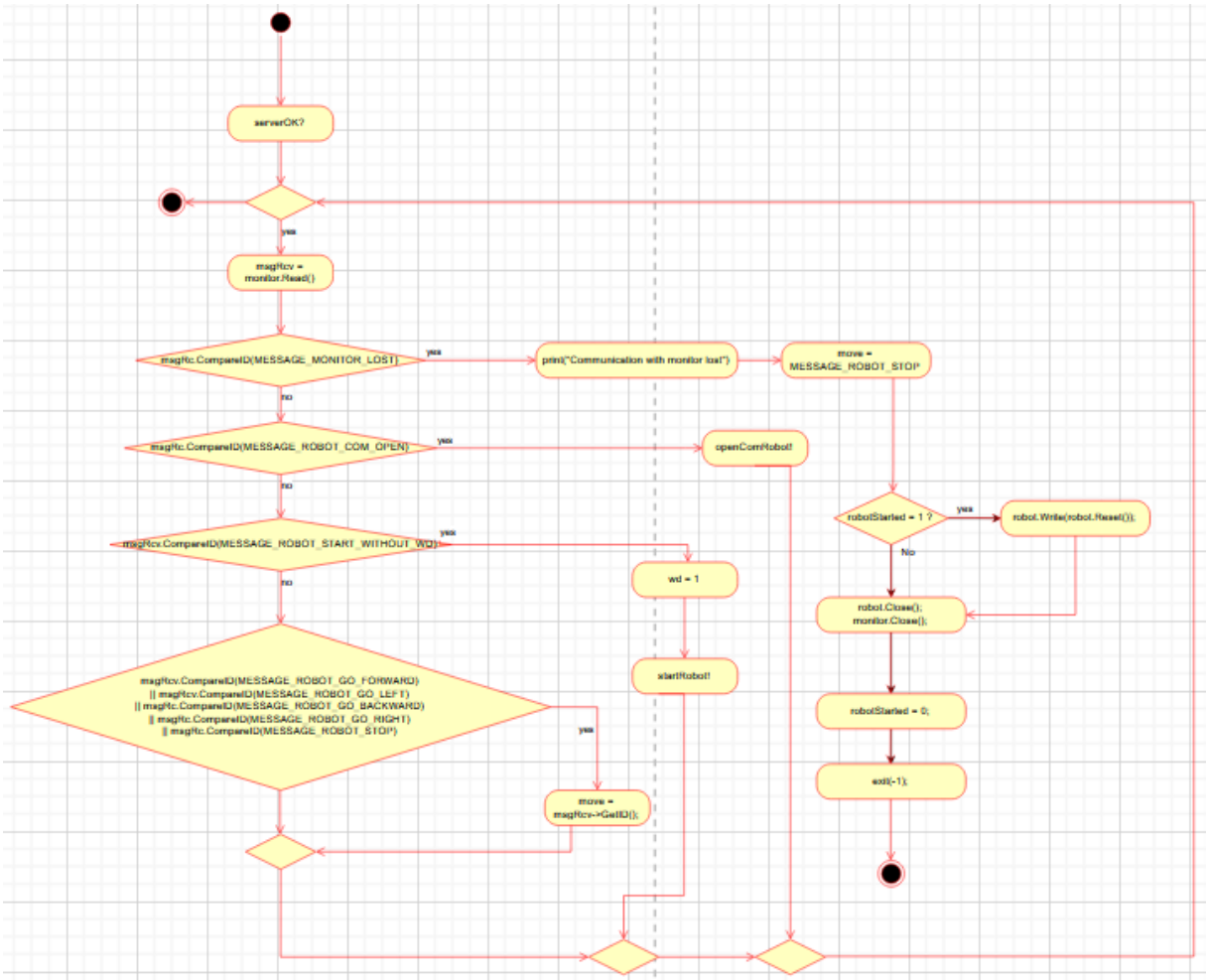

Fig. 4: Diagramme d'activité du thread th_receiveFromMon

Le moniteur envoie des messages au superviseur qui les agit en conséquence en débloquant les sémaphores correspondants et en mettant à jour les variables partagées. Lorsque la communication entre les deux est coupée, le message correspond à "MESSAGE_MONITOR_LOST", alors on réinitialise le système complet.

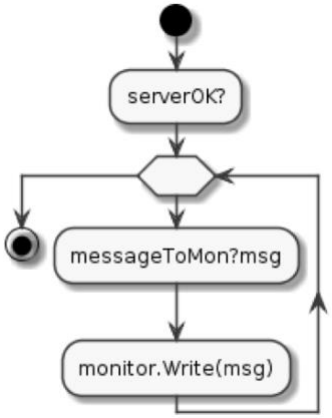

Fig. 5: Diagramme d'activité du thread th_sendToMon

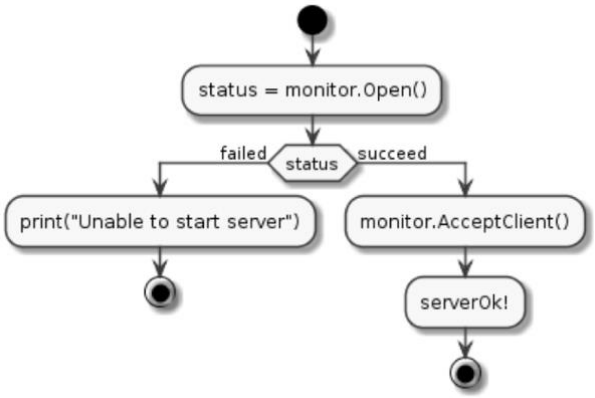

Fig. 6: Diagramme d'activité du thread th_server

C- Groupe de threads gestion du robot

1- Diagramme fonctionnel du groupe de gestion du robot

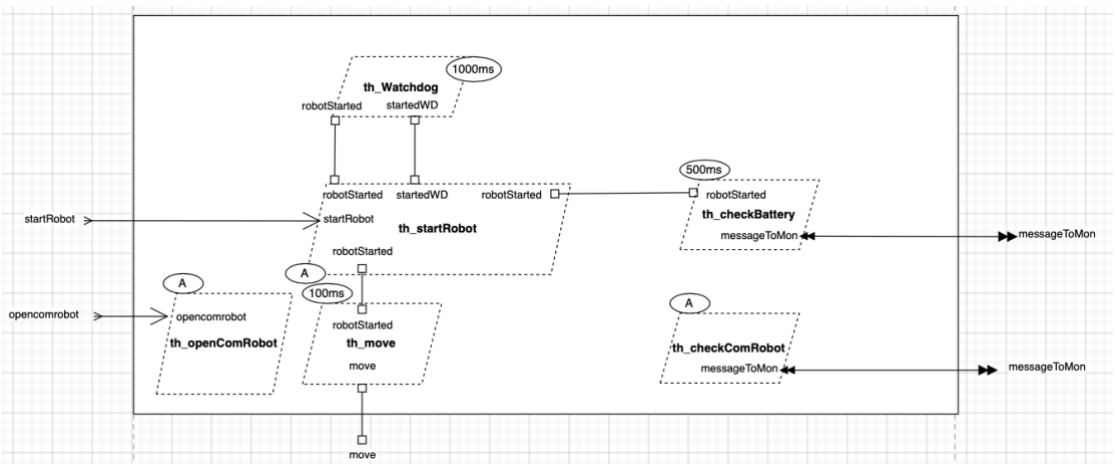

Fig. 7: Diagramme fonctionnel du groupe de gestion du robot

2- Diagramme d'activité du groupe d'activité du robot

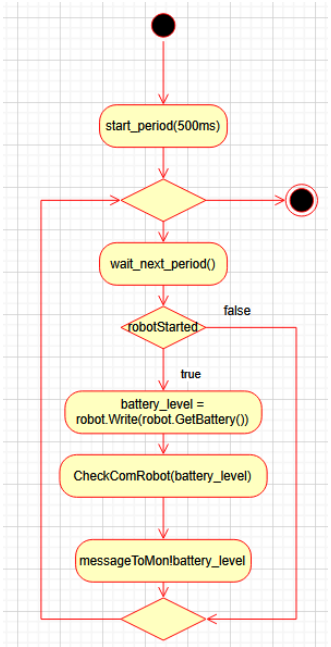

Fig. 8: Diagramme d'activité du thread th_checkBattery

Toutes les 500ms, le superviseur demande le niveau de batterie du robot puis le transmet au moniteur.

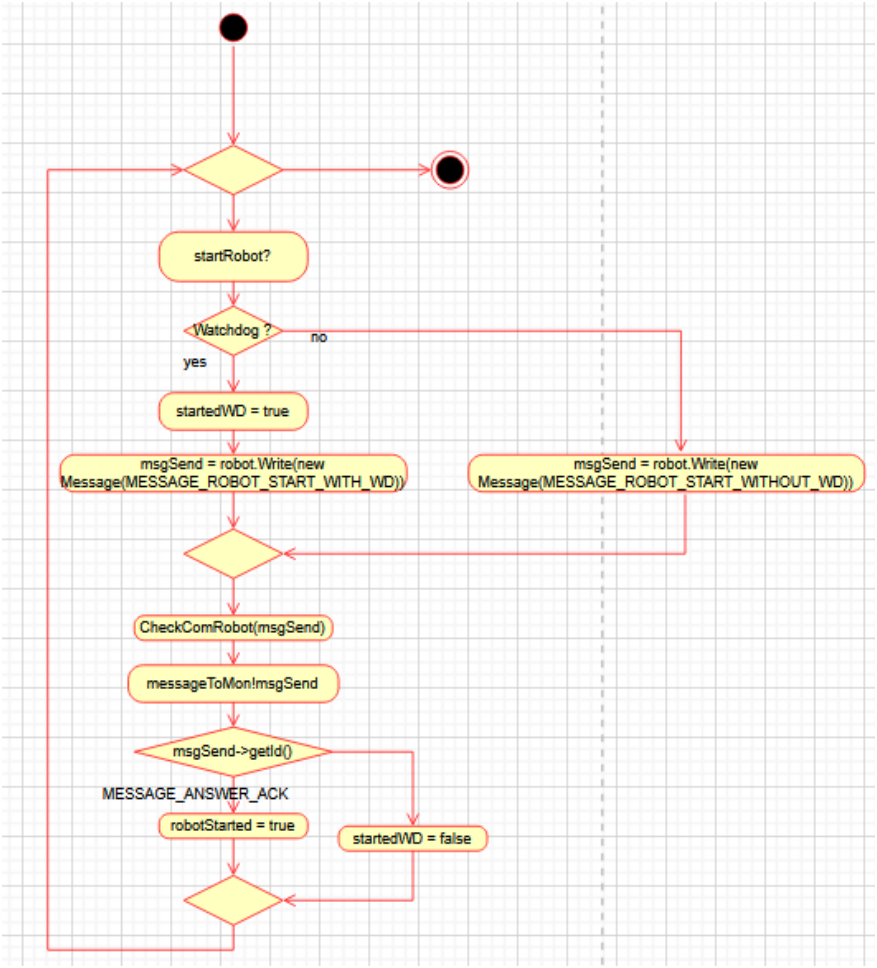

Fig. 9: Diagramme d'activité du thread th_startRobot

La variable watchdog correspond à la demande de démarrage en mode watchdog tandis que la variable startedWD indique que le robot est bien démarré en mode watchdog.
Ce thread démarre le robot dans le mode demandé par le moniteur.

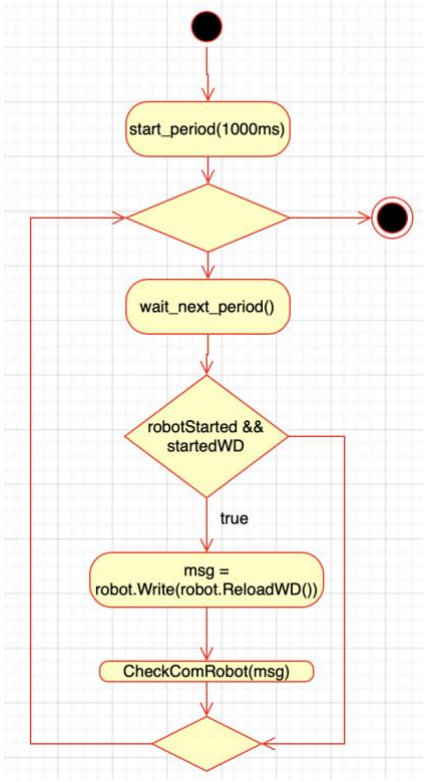

Fig. 10: Diagramme d'activité du thread th_Watchdog

Lorsque le robot est démarré en mode watchdog, toutes les secondes le superviseur envoie l'ordre au robot de réinitialiser le compteur du watchdog.

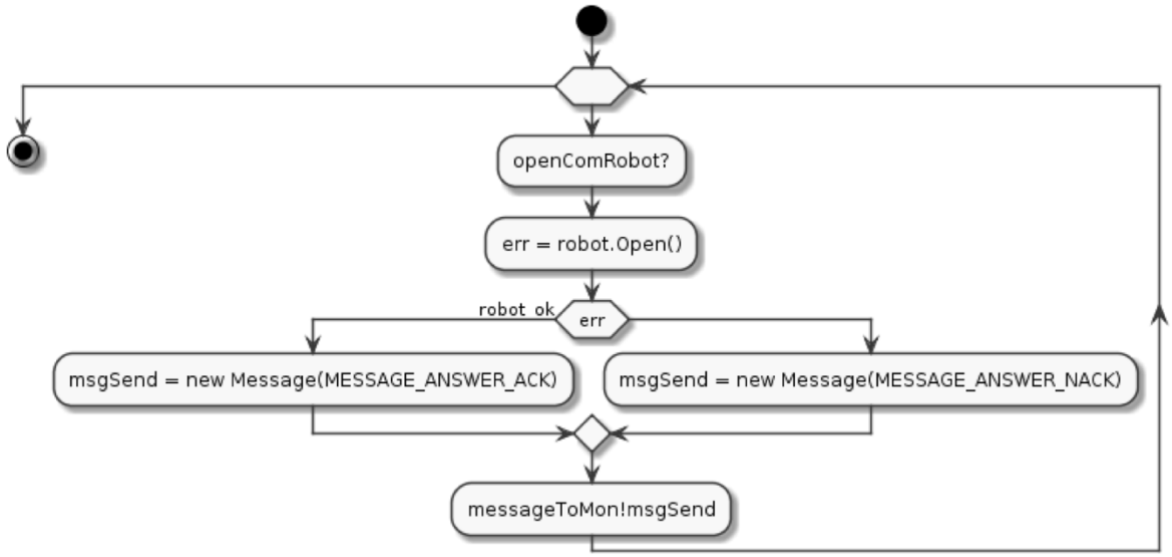

Fig. 11: Diagramme d'activité du thread th_openComRobot

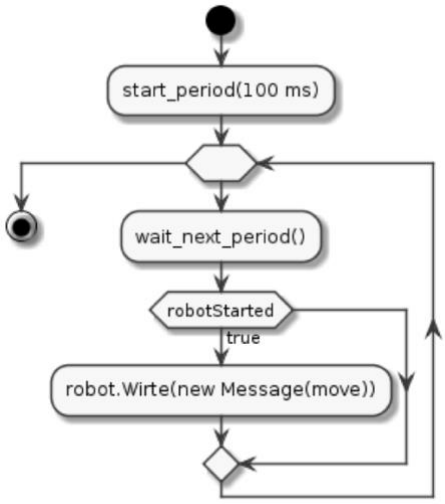

Fig. 12: Diagramme d'activité du thread th_move

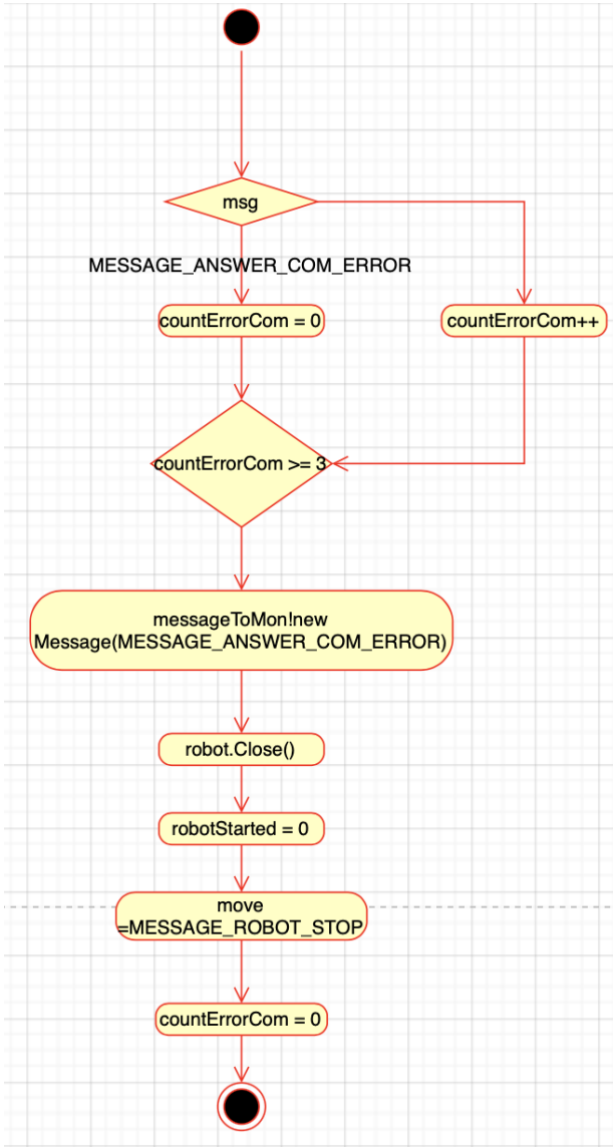

Fig. 13: Diagramme d'activité du thread th_checkComRobot

C'est une fonction qui est exécutée à chaque fois qu'un message est émis vers le robot. Si le message produit une erreur on incrémente un compteur qui indique que la communication a été perdue si ce compteur dépasse 3.

D-Groupe de threads vision

1- Diagramme fonctionnel du groupe vision

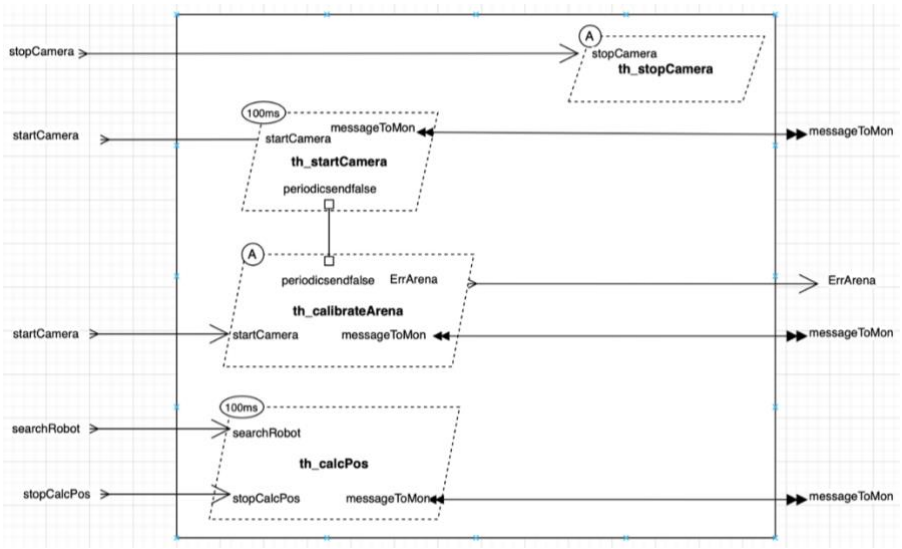

Fig. 14: Diagramme fonctionnel du groupe vision

2- Diagramme d'activité du groupe de gestion du robot

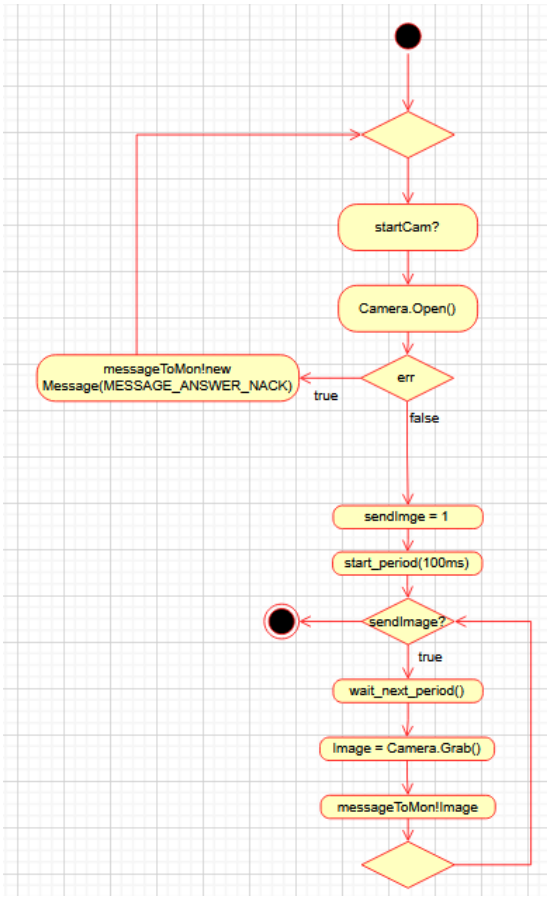

Fig. 15: Diagramme d'activité du thread th_startCamera

Le thread démarre la caméra, si cela réussie alors une image est envoyée au moniteur toutes les 100ms.

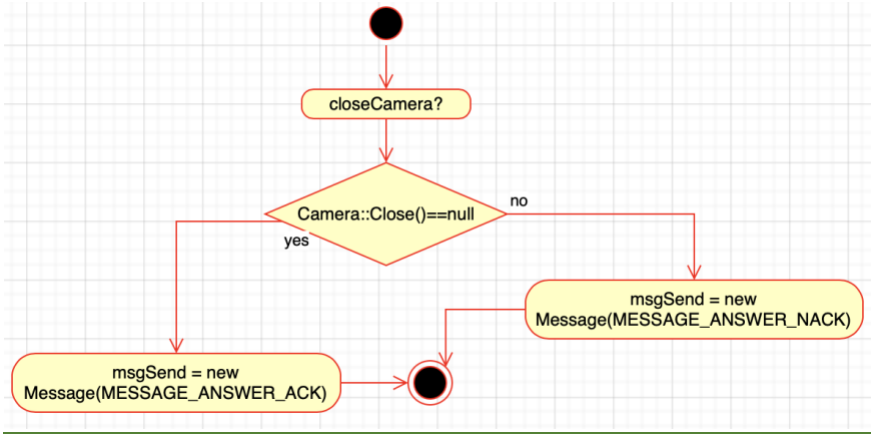

Fig. 16: Diagramme d'activité du thread th_stopCamera

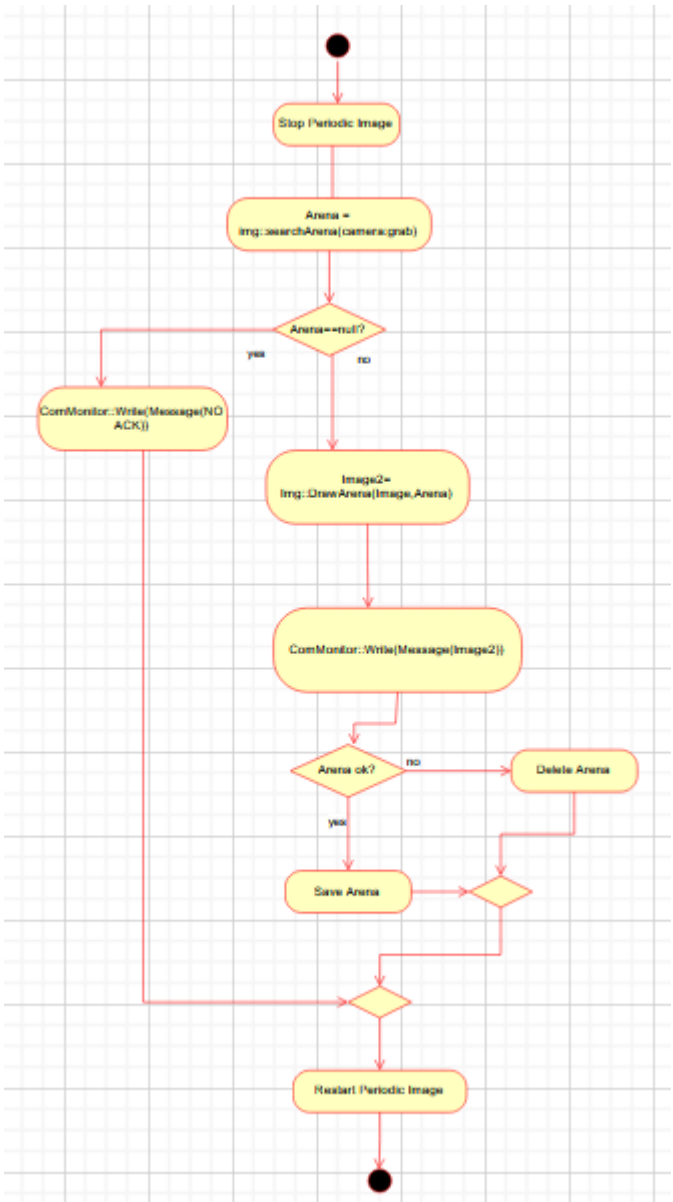

Fig. 17: Diagramme d'activité du thread th_calibrateArena

L'utilisateur fait une demande de recherche d'arène qui envoie et affiche celle-ci. S'il n'y a pas d'erreur à la réception de l'image de l'arène alors l'utilisateur doit valider ou non la calibration. On sauvegarde l'arène si elle est ok, on la supprime sinon.

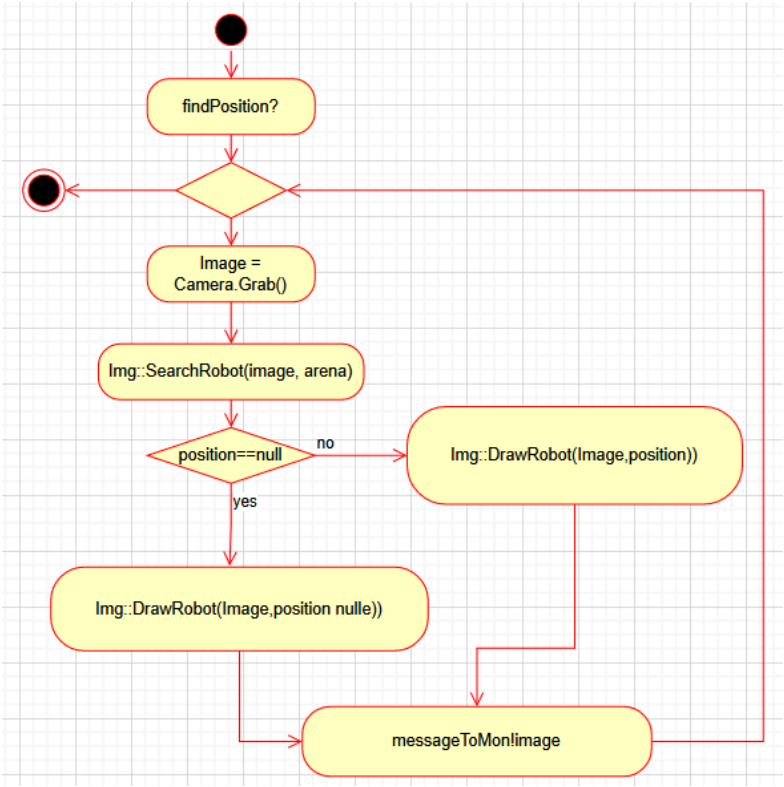

Fig. 18: Diagramme d'activité du thread th_calcPos

II- Transformation AADL vers Xenomai

Dans cette partie on abordera la méthode pour passer d'un modèle AADL à un code sous Xenomai.

A- Thread

1- Instanciation et démarrage

Chaque thread a été implémenté par un RT_TASK déclarés dans le fichier tasks.h. La création de la tâche se fait à l'aide du service rt_task_create et son démarrage à l'aide de rt_task_start.

Toutes les tâches sont crées dans la méthode init de tasks.cpp et démarrées dans la méthode run. Par exemple, pour la tâche th_checkBattery, sa déclaration est faite ligne 83 dans le fichier tasks.h : RT_TASK th_checkBattery;

Sa création ligne 136 de tasks.cpp lors de l'appel de rt_task_create(&th_checkBattery, "th_checkBattery", 0, PRIORITY_TBATTERY, 0) et son démarrage ligne 188 avec rt_task_start(&th_checkBattery, (void(*)(void*)) & Tasks::CheckBattery, this).

2- Code à éxécuter

Le lien sous Xenomai entre le thread et le traitement à exécuter se fait via l'appel à rt_task_start qui permet de lancer le thread et permet au système opérant d'exécuter les instructions présentes dans la tâche.

3- Niveau de priorités

Pour définir le niveau de priorité d'un thread sous Xenomai, nous avons défini des macros représentant leur priorité dans le fichier tasks.cpp : PRIORITY_TSERVER 30. Ensuite cette valeur de priorité est passé en paramètre de rt_task_create lors de la création de la tâche : rt_task_create(&th_server, "th_server", 0, **PRIORITY_TSERVER**, 0). Puis le système opérant s'occupera tout seul de l'ordonnancement des tâches selon leur niveau de priorité.

4- Activation périodique

Un thread est rendu périodique par l'utilisation de la primitive Xenomai rt_task_set_periodic au début de la tâche, par exemple rt_task_set_periodic(NULL, TM_NOW, 500000000); pour rendre la tâche th_checkBattery périodique de 500ms. Il est également nécessaire d'utiliser rt_task_wait_period au début du while(1) pour bloquer l'execution du thread jusqu'au prochain cycle.

B- Donnée partagée

1- Instanciation

Les données partagées sont déclarées et initialisées si besoin dans le fichier tasks.h.

2- Accès en lecture et écriture

L'utilisation d'un mutex est nécessaire pour l'accès à une donnée partagée, pour accéder à la donnée il faut d'abord prendre le mutex, une fois la lecture ou l'écriture finie, il faut le libérer pour laisser d'autres tâches y accéder, par exemple :

```
rt_mutex_acquire(&mutex_robotStarted, TM_INFINITE);
rs = robotStarted;
rt_mutex_release(&mutex_robotStarted);
```

Ici nous lisons la valeur de la donée partagée robotStarted.

Les mutex sont tous définis dans le fichier tasks.h : RT_MUTEX mutex_robotStarted; Ils sont ensuite créés dans la fonction init du fichier tasks.cpp : rt_mutex_create(&mutex_robotStarted, NULL).

C- Port d'évènement

1- Instanciation

Nous avons utilisé des sémaphores pour modéliser les ports d'événement. Ceux-ci sont déclarés dans le fichier tasks.h puis créés dans la fonction init du fichier tasks.cpp via rt_sem_create dans la fonction init.

Par exemple pour le semaphore sem_startRobot, sa déclaration est faite ligne 103 du fichier tasks.h RT_SEM sem_startRobot, sa création ligne 103 du fichier tasks.cpp rt_sem_create(&sem_startRobot, NULL, 0, S_FIFO).

2- Envoi d'un évènement

La libération d'un sémaphore représente l'envoi d'un événement, pour demander le démarrage du robot : rt_sem_v(&sem_startRobot);

3- Réception d'un évènement

La prise de sémaphore représente la réception d'un événement, exemple : rt_sem_p(&sem_startRobot, TM_INFINITE); SI le sémaphore est disponible, il le prend et continue son exécution, sinon il reste bloqué jusqu'à ce que le sémaphore soit libéré.

D-Port d'évènement-données

1- Instanciation

Les données transmises sont de type Message. Pour communiquer entre le moniteur et le superviseur, nous avons utilisé une file d'attente de type RT_QUEUE, celle-ci est déclarée ligne 110 du fichier tasks.h puis créée ligne 149 du fichier tasks.cpp dans la fonction init : rt_queue_create(&q_messageToMon, "q_messageToMon", sizeof (Message*)*50, Q_UNLIMITED, Q_FIFO).

2- Envoi d'une donnée

Pour envoyer une donnée au moniteur, nous avons encapsulé cette donnée dans un message qui est ensuite envoyé au moniteur via le thread WriteInQueue qui écrit le message dans la file, ensuite le thread SendToMonTask lit le dernier message de la queue puis l'envoie au moniteur via monitor.Write().

3- Réception d'une donnée

Le thread ReceiveFromMonTask s'occupe de la réception des données depuis le moniteur, il lit via monitor.Read() les messages reçus par le superviseur depuis le moniteur puis selon le contenu du message envoi ou non l'événement correspondant.

III- Analyse et validation de la conception

Pour cette partie nous avons réalisé une vidéo dans laquelle nous montrons que chacune des fonctionnalités 1 à 13 est opérationnelles. Nous avons particulièrement axé notre démonstration sur les fonctionnalités 5,6, 8, 9, 11 et 13 étant donné que les autres étaient déjà codé et que nous ne les avons pas modifiés.

Cette vidéo est disponible juste ici : https://drive.google.com/file/d/13PasTCHmG0e7Ezg7hzRb-3GW_spO8rIw/view (Elle était trop lourde pour passer sur GitHub).

Conclusion :

Nous avons su gérer le travail en équipe impliquant le partage des tâches et la bonne communication entre les différents membres du groupe. En effet, nous avons débuté la conception tous ensemble afin d'être sûr de partir tous dans la même direction durant le premier TP. Nous nous sommes ensuite partagé les tâches afin d'être plus efficace. Antoine et Clément se sont concentrés sur le code pendant que Clémence a axé son travail sur les diagrammes AADL et les différents détails de conception. Pour ce qui est des diagrammes d'activités, Antoine a pris en charge le groupe vision tandis que Clément et Clémence ont détaillé ceux du groupe robot.

Pour finir nous avons tout repris ensemble en fin de projet afin de vérifier que nous avions bien suivi les directions données en début de projet et corriger les quelques modifications qui avaient été adoptées lors du codage.

Tout au long de ces TP, nous avons réussi à développer les fonctionnalités essentielles du robot et ainsi nous familiariser avec le codage temps réel sous Xenomai. Nous avons réussi à gérer l'approche de conception, codage et gestion des threads. Nous avons pu expérimenter et mieux comprendre les principes de base du temps réel. Cela nous permet donc de nous mettre une fois de plus dans le rôle de l'ingénieur qui est d'utiliser ses connaissances et de les appliquer à des cas réels.

Table des illustrations

Table des annexes

Annexe 1 : Lien GitHub : https://github.com/Clement-INS/Projet_robot

INSA Toulouse
135, avenue de Rangueil
31077 Toulouse Cedex 4 - France
www.insa-toulouse.fr

Université Fédérale Toulouse Midi-Pyrénées
Liberté · Égalité · Fraternité
RÉPUBLIQUE FRANÇAISE
MINISTÈRE
DE L'ÉDUCATION NATIONALE,
DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE