Rapport de projet De Stijl 2.0

version 1er juin 2019

Laura Bouzereau (conception, vidéo, robot, mission, rédaction du compte-rendu)
Constance Gay (robot, mission, rédaction du compte-rendu)
Louis Jean (conception, robot, intégration)

1 Conception

1.1 Diagramme fonctionnel général

Ci-dessous est le diagramme fonctionnel de notre système, faisant référence aux différents threads que nous détaillerons plus bas.

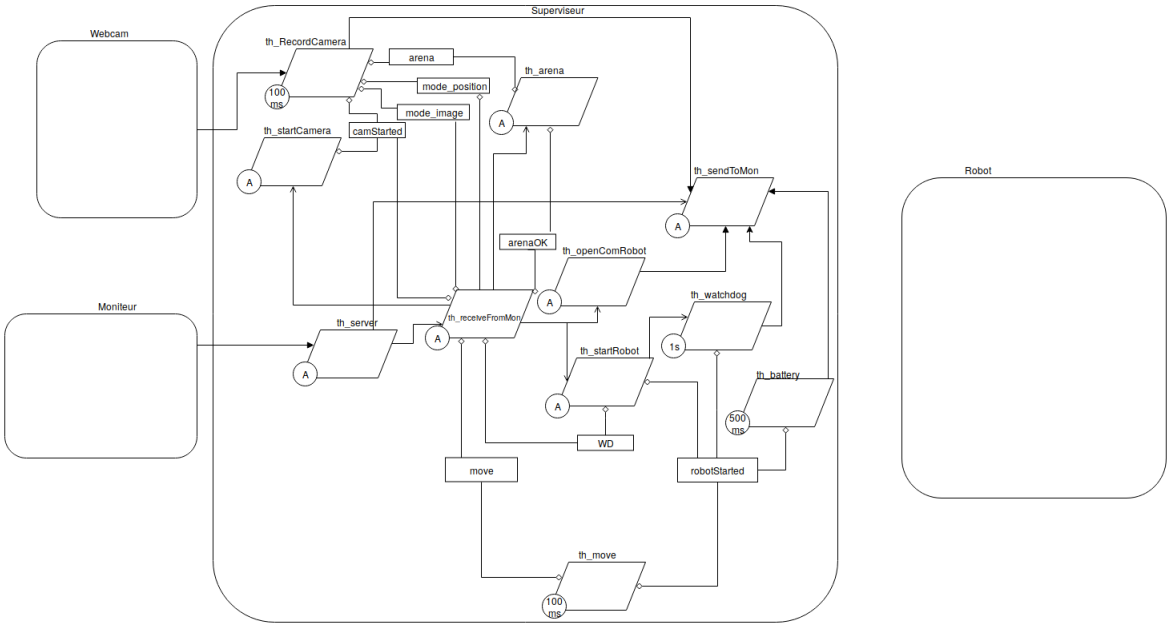

Fig. 1: Diagramme fonctionnel du système

1.2 Groupe de threads gestion du moniteur

1.2.1 Diagramme fonctionnel du groupe gestion du moniteur

Le moniteur a pour fonction de gérer les communication avec le robot, de récupérer les informations (niveau de batterie, perte de connexion...) et d'envoyer des ordres permettant d'effectuer les différentes actions (déplacer robot, allumer camera...).

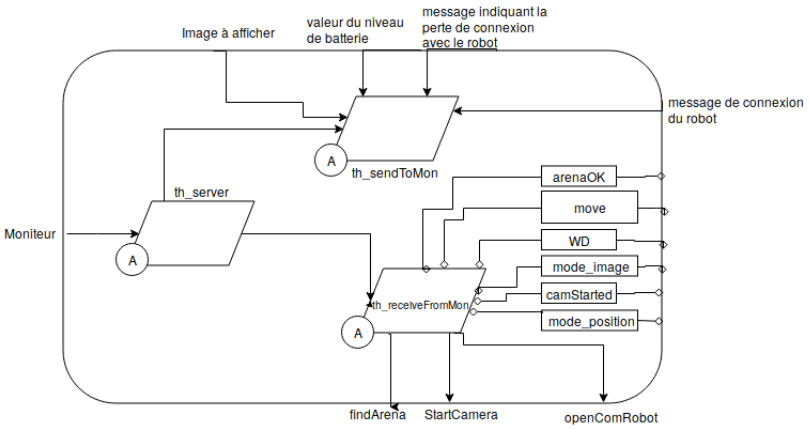

Fig. 2: Diagramme fonctionnel du groupe de threads gestion du moniteur

1.2.2 Description des threads du groupe gestion du moniteur

Tab. 1: Description des threads du groupe `th_group_gestion_moniteur`

Nom du thread	Rôle	Priorité
th_server	S'occupe de créer un server où le moniteur peut se connecter	30
th_receiveFromMon	Gère la reception des messages en provenance du moniteur et l'action demandée est effectuée	25
th_sentToMon	Gère l'envoi des messages du robot au moniteur	23

1.2.3 Diagrammes d'activité du groupe gestion du moniteur

Le thread th_server va ouvrir le moniteur à la connexion, et si ça fonctionne il accepte la connection client et le notifie.

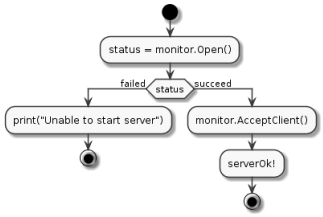

Fig. 3: Diagramme d'activité du thread `th_server`

Le thread th_sendToMon à pour fonction d'attendre des messages destinés au moniteur et de les récupérer à leur arrivée.

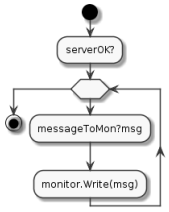

Fig. 4: Diagramme d'activité du thread `th_sendToMon`

Le thread th_receiveFromMon reçoit des messages provenant du moniteur ordonnant une certaine action et le robot va donc les effectuer. Les actions pouvant être demandées sont : ouverture de la communication, démarrage avec et sans watchdog, ouverture de la camera, déplacement du robot, changement du mode de capture de la camera et affichage d'une image.

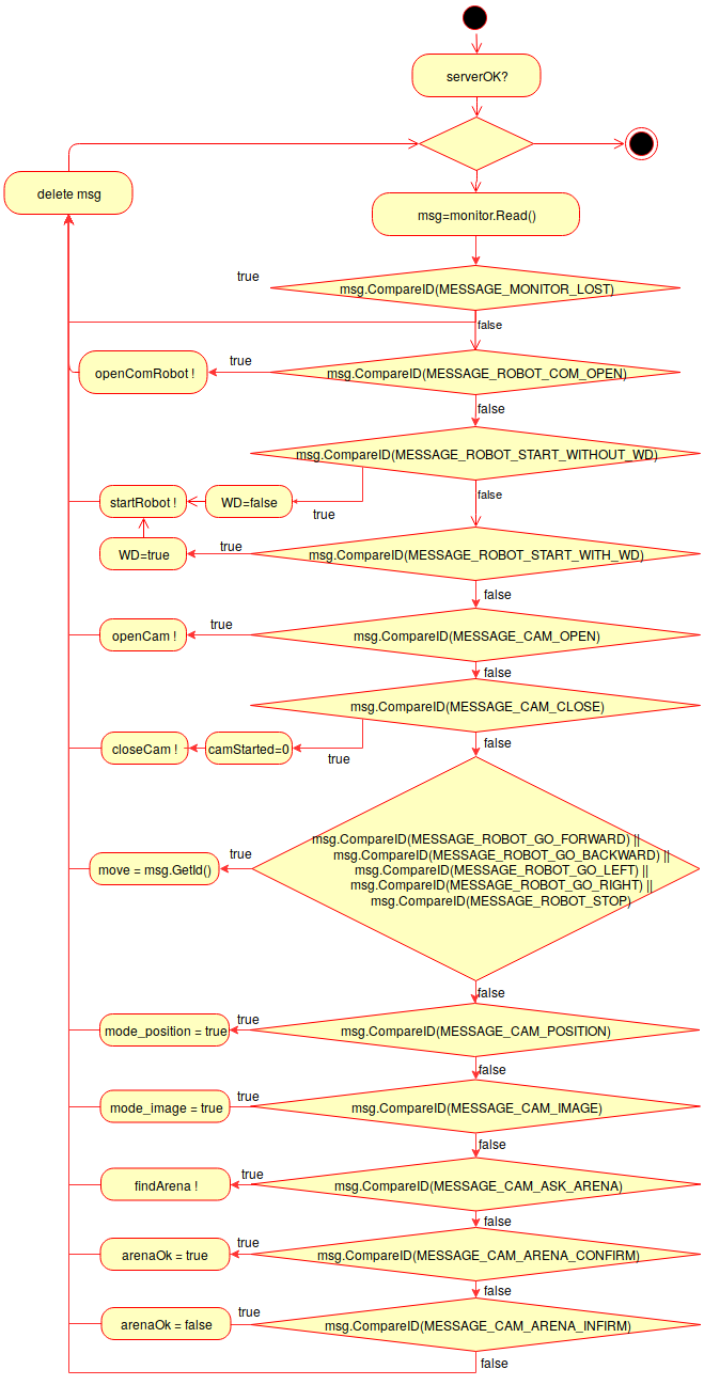

Fig. 5: Diagramme d'activité du thread `th_receiveFromMon`

1.3 Groupe de threads vision

1.3.1 Diagramme fonctionnel du groupe vision

Le groupe vision contient tout ce qui est camera et arène. Il va permettre d'allumer la camèra, de capturer des images dans deux modes différents (position et image) et de chercher une arène sur une photo.

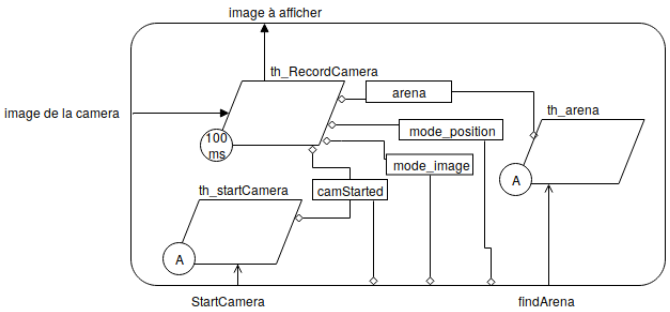

Fig. 6: Diagramme fonctionnel du groupe de threads gestion de la vision

1.3.2 Description des threads du groupe vision

Tab. 2: Description des threads du groupe `th_group_vision`

Nom du thread	Rôle	Priorité
th_startCamera	Démarre la camera	18
th_arena	Cherche une arène sur une image et la sauvegarde si elle est trouvée (demande confirmation à l'utilisateur	17
th_RecordCamera	Renvoi des photos selon le mode (position ou image)	16

1.3.3 Diagrammes d'activité du groupe vision

Le thread th_startCamera reçoit un message du moniteur lui ordonnant de démarrer la caméra, il le fait et notifie le moniteur si il y a une erreur.

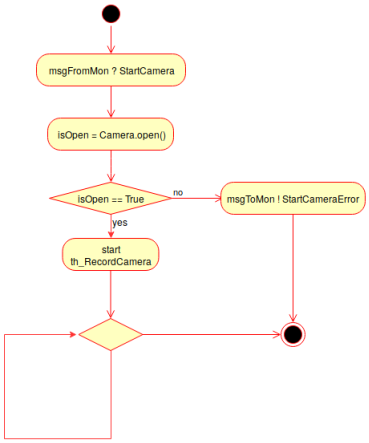

Fig. 7: Diagramme d'activité du thread `th_startCamera`

Le thread th_arena démarre lorsque que l'utilisateur appuie sur un bouton. Ce thread reçoit alors un ordre du moniteur de chercher l'arène sur une image. Le thread va donc mettre en pause la caméra, prendre une photo et chercher l'arène sur cette photo. Si il la trouve, il notifie le moniteur qui va demander confirmation à l'utilisateur. Si l'utilisateur confirme que la photo est bien l'arène, le thread th_arena va enregistrer l'image et s'éteindre.

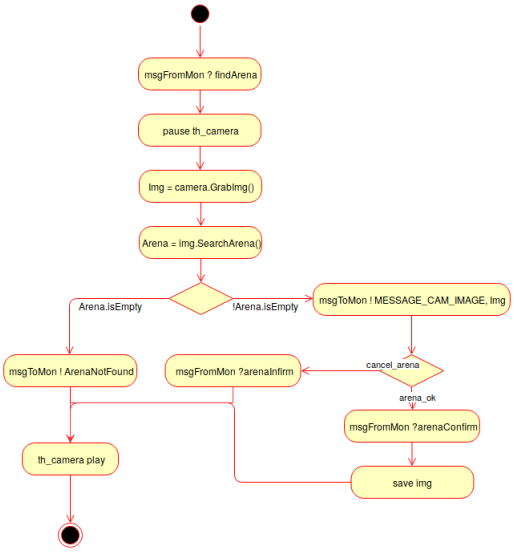

Fig. 8: Diagramme d'activité du thread `th_arena`

Le thread th_RecordCamera gère la caméra périodiquement (100ms). Il permet de lancer la caméra dans deux modes différents. Le premier est le mode position qui permet de chercher un robot dans l'arène et de renvoyer sa position. Le deuxième mode est le mode image qui prend des photos et les renvoit au moniteur. Le thread peut également être mis en pause.

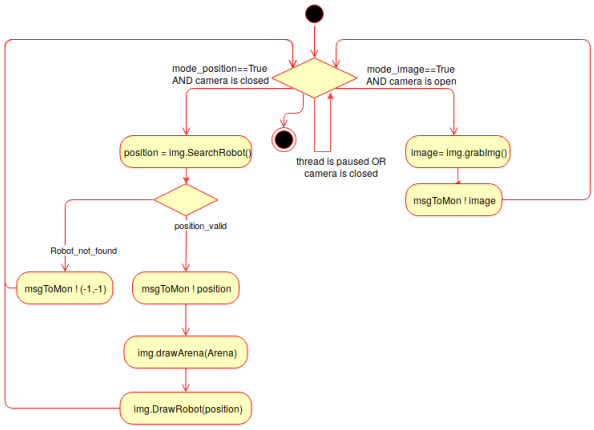

Fig. 9: Diagramme d'activité du thread `th_RecordCamera`

1.4 Groupe de threads gestion du robot

1.4.1 Diagramme fonctionnel du groupe gestion robot

Le groupe de gestion du robot permet d'effectuer les différentes actions du robot, notamment son déplacement, l'ouverture de ses communications, contrôler son niveau de batterie et surveiller sa connexion avec le moniteur.

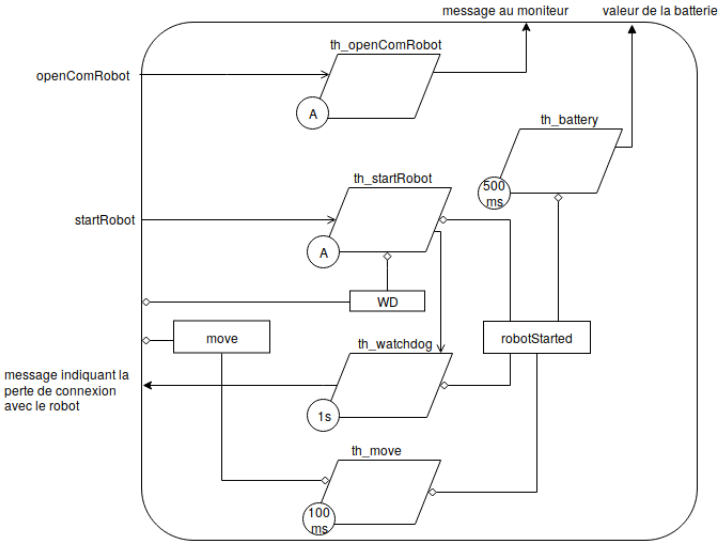

Fig. 10: Diagramme fonctionnel du groupe de threads gestion du robot

1.4.2 Description des threads du groupe gestion robot

Tab. 3: Description des threads du groupe `th_group_gestion_robot`

Nom du thread	Rôle	Priorité
th_watchdog	Vérifie si la connexion entre le moniteur et le robot	22
th_openComRobot	Démarre les communications du robot	20
th_startRobot	Démarre le robot	20
th_move	Permet de déplacer le robot	20
th_battery	Récupère le niveau de batterie du robot	15

1.4.3 Diagrammes d'activité du groupe robot

Nous n'allons pas décrire les threads th_openComRobot et th_move car nous ne les avons pas modifié. Pour ce qui est du thread th_startRobot nous n'avons fait qu'une légère modification afin de la démarrer avec ou sans watchdog.

Le thread th_watchdog va se lancer si le robot est démarré en mode watchdog. Ce thread va envoyer des messages périodiquement (1s) au robot et si 3 messages de suite sont n'ont pas de réponse, le système entier (moniteur, robot...) sera éteint.

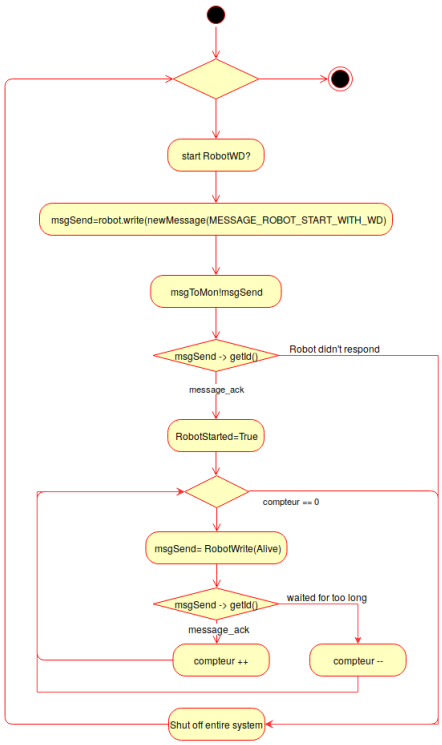

Fig. 11: Diagramme d'activité du thread `th_watchdog`

Le thread th_battery permet d'envoyer un message toutes les 500ms au moniteur contenant le niveau de batterie du robot.

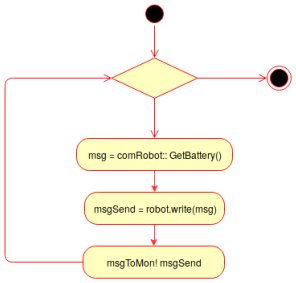

Fig. 12: Diagramme d'activité du thread `th_battery`

2 Analyse et validation de la conception

2.1 Fonctionnalité 1*

Description : Le lancement du serveur doit être réalisé au démarrage du superviseur. En cas d'échec du démarrage du serveur, un message textuel doit être affiché sur la console de lancement de l'application. Ce message doit signaler le problème et le superviseur doit s'arrêter.

2.2 Fonctionnalité 2*

Description : La connexion entre le moniteur et le superviseur (via le socket) doit être établie suite à la demande de connexion de l'utilisateur.

2.3 Fonctionnalité 3*

Description : Tous les messages envoyés depuis le moniteur doivent être réceptionnés par le superviseur.

2.4 Fonctionnalité 4*

Description : L'application superviseur doit être capable d'envoyer les messages au moniteur (via le serveur) avec un délai d'au plus 10 ms.

2.5 Fonctionnalité 5*

Description : Le superviseur doit détecter la perte de communication avec le moniteur. En cas de perte de la communication un message doit être affiché sur la console de lancement du superviseur.

2.6 Fonctionnalité 6*

Description : En cas de perte de communication entre le superviseur et moniteur, il faut stopper le robot, la communication avec le robot, fermer le serveur et déconnecter la caméra afin de revenir dans le même état qu'au démarrage du superviseur.

2.7 Fonctionnalité 7*

Description : Dès que la communication avec le moniteur est en place, la communication entre le superviseur et le robot doit être ouverte. Si la communication est active, il faut envoyer un message d'acquittement au moniteur. En cas d'échec, un message le signalant est renvoyé au moniteur.

2.8 Fonctionnalité 8

Description : La communication entre le robot et le superviseur doit être surveillée par un mécanisme de compteur afin de détecter une perte du médium de communication.

Réalisation : Nous avons traité cette fonctionnalité par le développement d'un thread périodique th_watchdog. Celui-ci indique au robot qu'il doit démarrer en mode watchdog, et attend que celui-ci confirme son démarrage. Ensuite le moniteur envoit des messages périodiquement au robot (toutes les 500ms). Dans notre diagramme nous avons déclaré incrémenté notre compteur à chaque réception de message et décrementé à chaque perte de message. Au final nous n'avons pas implémenté notre code comme sur notre schéma de conception. Nous avons choisis de déclarer la connexion comme perdue uniquement si trois messages de suite n'ont pas de réponse ce qui parait plus logique. Si la connexion est perdue le système entier s'éteint, comme on peut ci-dessous (fonctionnalité 9).

2.9 Fonctionnalité 9

Description : Lorsque la communication entre le robot et le superviseur est perdue, un message spécifique doit être envoyé au moniteur. Le système doit fermer la communication entre le robot et le superviseur et se remettre dans un état initial permettant de relancer la communication.

Réalisation : Cette fonctionnalité a été traitée à la suite de la précédente dans le thread th_watchdog, si une perte de connection est détectée, un message s'écrit sur la console, un message MESSAGE_ANSWER_ROBOT_TIMEOUT est envoyé au moniteur, la communication avec le robot est fermée et la variable RobotStarted est remise à 0.

2.10 Fonctionnalité 10*

Description : Lorsque l'utilisateur demande, via le moniteur, le démarrage sans watchdog, le robot doit démarrer dans ce mode. En cas de succès, un message d'acquittement est retourné au moniteur. En cas d'échec, un message indiquant l'échec est transmis au moniteur.

2.11 Fonctionnalité 11

Description : Lorsque l'utilisateur demande, via le moniteur, le démarrage avec watchdog, le robot doit démarrer dans ce mode. Un message d'acquittement est retourné au moniteur. En cas d'échec, un message indiquant l'échec est transmis au moniteur.

Une fois le démarrage effectué, le robot doit rester vivant en envoyant régulièrement le message de rechargement du watchdog.

Réalisation : Cette fonctionnalité est implémentée dans le thread th_startRobot. Une variable partagée (WD) nous permet de stocker le mode de connection, et dans ce thread, un message est envoyé au moniteur lui indiquant avec quel mode s'est connecté le robot.

2.12 Fonctionnalité 12*

Description : Lorsque qu'un ordre de mouvement est reçu par le superviseur, le robot doit réaliser ce déplacement en moins de 100 ms.

2.13 Fonctionnalité 13

Description : Le niveau de la batterie du robot doit être mis à jour toutes les 500 ms sur le moniteur.

Réalisation : Nous avons implémenté cette fonctionnalité avec le thread th_battery qui renvoit toutes les 500ms le niveau de batterie du robot au moniteur.

2.14 Fonctionnalité 14

Description : La caméra doit être démarrée suite à une demande provenant du moniteur. Si l'ouverture de la caméra a échoué, il faut envoyer un message au moniteur.

Réalisation : Le thread th_StartCamera sera lancé suite à l'appui d'un bouton et un message du moniteur à ce thread entrainera le démarrage de la caméra. Si ce démarrage échoue, un message est renvoyé au moniteur (StartCameraError) et le thread se termine. Si le démarrage fonctionne le thread th_RecordCamera est démarré.

2.15 Fonctionnalité 15

Description : Dès que la caméra est ouverte, une image doit être envoyée au moniteur toutes les 100 ms.

Réalisation : Une fois que le thread th_RecordCamera est lancé, il boucle périodiquement toutes les 100ms. Il vérifie que la caméra est toujours ouverte, et si oui il teste le mode de capture. Si la variable partagée mode_image est vrai, alors une image est renvoyée au moniteur à chaque boucle.

2.16 Fonctionnalité 16

Description : La caméra doit être fermée suite à une demande provenant du moniteur. Un message doit être envoyé au moniteur pour signifier l'acquittement de la demande. L'envoi périodique des images doit alors être stoppé.

Réalisation : Si une demande de fermeture de caméra est faites par le moniteur (MESSAGE_CAMERA_CLOSE) alors la camera sera fermée par le thread th_ReceiveFromMon. Ainsi le thread th_RecordCamera va voir que la caméra a été fermée et cessera de prendre des images. Nous ne renvoyons pas d'acquittement au moniteur.

2.17 Fonctionnalité 17

Description : Suite à une demande de recherche de l'arène, le superviseur doit stopper l'envoi périodique des images, faire la recherche de l'arène et renvoyer une image sur laquelle est dessinée cette arène. Si aucune arène n'est trouvée un message d'échec est envoyé.

L'utilisateur doit ensuite valider visuellement via le moniteur si l'arène a bien été trouvée. L'utilisateur peut :

- valider l'arène : dans ce cas, le superviseur doit sauvegarder l'arène trouvée (pour l'utiliser ultérieurement) puis retourner dans son mode d'envoi périodique des images en ajoutant à l'image l'arène dessinée.
- annuler la recherche : dans ce cas, le superviseur doit simplement retourner dans son mode d'envoi périodique des images et invalider la recherche.

Réalisation : Lorsqu'une demande de rechercher d'arène arrive du moniteur, le thread th_arena se met en route car son sémaphore est libéré. Il prend ensuite le sémaphore sem_recordCamera, ce qui va mettre en pause le thread th_RecordCamera jusqu'à la libération du sémaphore. L'envoi périodique des images est donc stoppé.
Le thread th_arena va donc chercher l'arène et si il la trouve une image va être envoyé au moniteur MESSAGE_CAM_IMAGE. Sinon le moniteur sera notifié de l'échec.
L'utilisateur va ensuite confirmer si l'image est valide. Si c'est le cas, le moniteur enverra le message MESSAGE_CAM_ARENA_CONFIRM et la variable arenaOK deviendra true. Sinon le moniteur enverra le message MESSAGE_CAM_ARENA_CONFIRM et la variable arenaOK deviendra false.
Le thread th_arena regardera ensuite la valeur de la variable arenaOK, si elle est true la photo sera sauvegardée dans la variable partagée ImgArena. Sinon il ne se passe rien. Après cela le semaphore sem_recordCamera est relaché et le thread th_RecordCamera n'est donc plus en pause et l'envoi périodique des images recommence.

2.18 Fonctionnalité 18

Description : Suite à une demande de l'utilisateur de calculer la position du robot, le superviseur doit calculer cette position, dessiner sur l'image le résultat et envoyer un message au moniteur avec la position toutes les 100 ms. Si le robot n'a pas été trouvé, un message de position est envoyé avec une position (-1,-1).

Réalisation : La gestion du mode de capture (image ou position) est géré par les deux variables partagées mode_image et mode_position. Ces variables sont respectivement mises à true lorsque le moniteur envoi MESSAGE_CAM_IMAGE ou un message MESSAGE_CAM_POSITION. A chacune de ces boucles, le thread th_RecordCamera va vérifier laquelle des deux variables est vrai, et va entrer dans le mode correspondant. On a déjà détailler le mode image plus haut.

Le mode position que nous avons implémenté prend une image , dessine l'arène précédemment prise en photo dessus et ensuite cherche les robots. L'image ainsi dessinée est ensuite renvoyée au moniteur. Ainsi comme nous ne renvoyons pas la position du robot il n'y a pas d'envoi (-1,-1) en cas d'échec. Nous n'avons pas réussit à tester cette fonctionnalité mais nous prévoyons des problèmes en cas d'échec vu qu'il n'est pas traité.

2.19 Fonctionnalité 19

Description : Suite à une demande de l'utilisateur de stopper le calcul de la position du robot, le superviseur doit rebasculer dans un mode d'envoi de l'image sans le calcul de la position.

Réalisation : Comme décrit dans la fonctionnalité précédente, le mode de prise d'image est testée à chaque boucle du thread th_RecordCamera, donc si l'utilisateur le change, ce changement sera pris en compte avec les variables mode_position et mode_image.

3 Transformation AADL vers Xenomai

3.1 Thread

3.1.1 Instanciation et démarrage

Expliquer comment vous implémentez sous Xenomai l'instanciation et le démarrage d'un thread AADL.

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_server`, sa déclaration est faite ligne 73 dans le fichier `tasks.h`

```
RT_TASK th_server;
```

sa création ligne 102 de `tasks.cpp` lors de l'appel de

```
rt_task_create(&th_server, "th_server", 0, PRIORITY_TSERVER, 0)
```

et son démarrage ligne 146 avec

```
rt_task_start(&th_server, (void(*)(void*)) & Tasks::ServerTask, this)
```

3.1.2 Code à exécuter

Comment se fait le lien sous Xenomai entre le thread et le traitement à exécuter.

La fonction suivante permet de faire the lien entre un thread et son traitement

```
int rt_task_start(RT_TASK * task, void(*entry)(void*cookie),void*cookie);
```

3.1.3 Niveau de priorités

Expliquer comment vous fixez sous Xenomai le niveau de priorité d'un thread AADL.

Il y a plusieurs façon de la faire mais nous avons adopté la suivante lors de la création des tâches :

```
rt_task_create(RT_TASK * task,const char *name, int stksize, int prio, int mode);
```

3.1.4 Activation périodique

Expliquer comment vous rendez périodique l'activation d'un thread AADL sous Xenomai.

Afin de rendre périodique un thread il faut déjà déclarer sa période

```
rt_task_set_periodoc(RT_TASK * task,RTIME idate, RTIME period);
```

Ensuite il faut mettre la ligne suivante dans le while du thread :

```
rt_task_wait_period(NULL);
```

3.1.5 Activation événementielle

Expliquer les moyens mis en œuvre dans l'implémentation sous Xenomai pour gérer les activations événementielles d'un thread AADL.

Sous Xenomai nous pouvions utiliser des sémaphores (pour bloquer/libérer des tâches) et des mutex (pour proteger des variables partagées).

3.2 Port d'événement

3.2.1 Instanciation

Comment avez-vous instancié un port d'événement ?

Une fois de plus, sémaphore et mutex sur variables partagées.

3.2.2 Envoi d'un événement

Quels services ont été employés pour signaler un événement ?

Afin de signaler un événement on peut : écrire dans une queue, libérer un sémaphore ou mettre à jours une variable partagée.

3.2.3 Réception d'un événement

Comment se fait l'attente d'un événement ?

La reception d'un événement peut se faire par une attente d'un sémaphore très longue (TM_INFINITE) que nous avons utilisé pour le thread th_RecordCamera. Normalement le premier élément à se mettre en attente pour un sémaphore sera le premier à l'obtenir.

3.3 Donnée partagée

3.3.1 Instanciation

Quelle structure instancie une donnée partagée ?
Une donnée partagée est une variable globale, protégée par un mutex donc n'importe quel thread peut l'instancier.

3.3.2 Accès en lecture et écriture

Comment garantissez-vous sous Xenomai l'accès à une donnée partagée ?

Pour garantir l'accès sécurisé à une donnée partagée sous Xenomai il faut utiliser un mutex de type RT_MUTEX.

3.4 Ports d'événement-données

3.4.1 Instanciation

Donnez la solution retenue pour implémenter un port d'événement-données avec Xenomai.

Il faut utiliser une RT_QUEUE comme on a pu voir dans le thread th_receiveFromMon.

3.4.2 Envoi d'une donnée

Quels services avez-vous employés pour envoyer des données ?

Pour envoyer des données il faut écrire dans le RT_QUEUE.

3.4.3 Réception d'une donnée

Quels services avez-vous employés pour recevoir des données ?

Pour envoyer des données il faut lire dans le RT_QUEUE.