

Rapport de projet De Stijl 2.0

version 2 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)

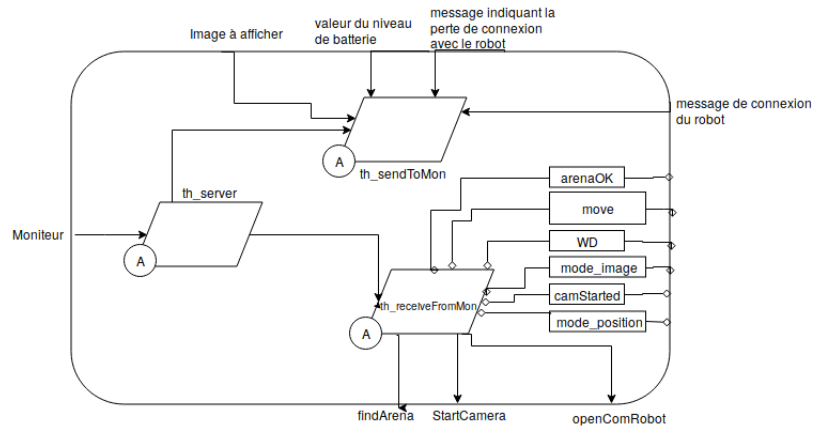


Fig. 2: Diagramme fonctionnel du groupe de threads gestion du moniteur

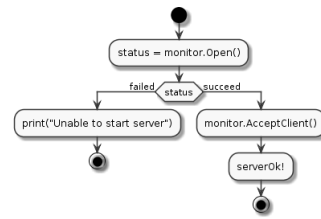
1.2.2 Description des threads du groupe gestion du moniteur

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

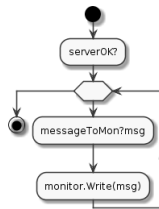
Nom du thread	Rôle	Priorité
<code>th_server</code>	S'occupe de créer un server où le moniteur peut se connecter	30
<code>th_receiveFromMon</code>	Gère la reception des messages en provenance du moniteur et l'action demandée est effectuée	25
<code>th_sentToMon</code>	Gère l'envoi des messages du robot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

1.2.3 Diagrammes d'activité du groupe gestion du moniteur

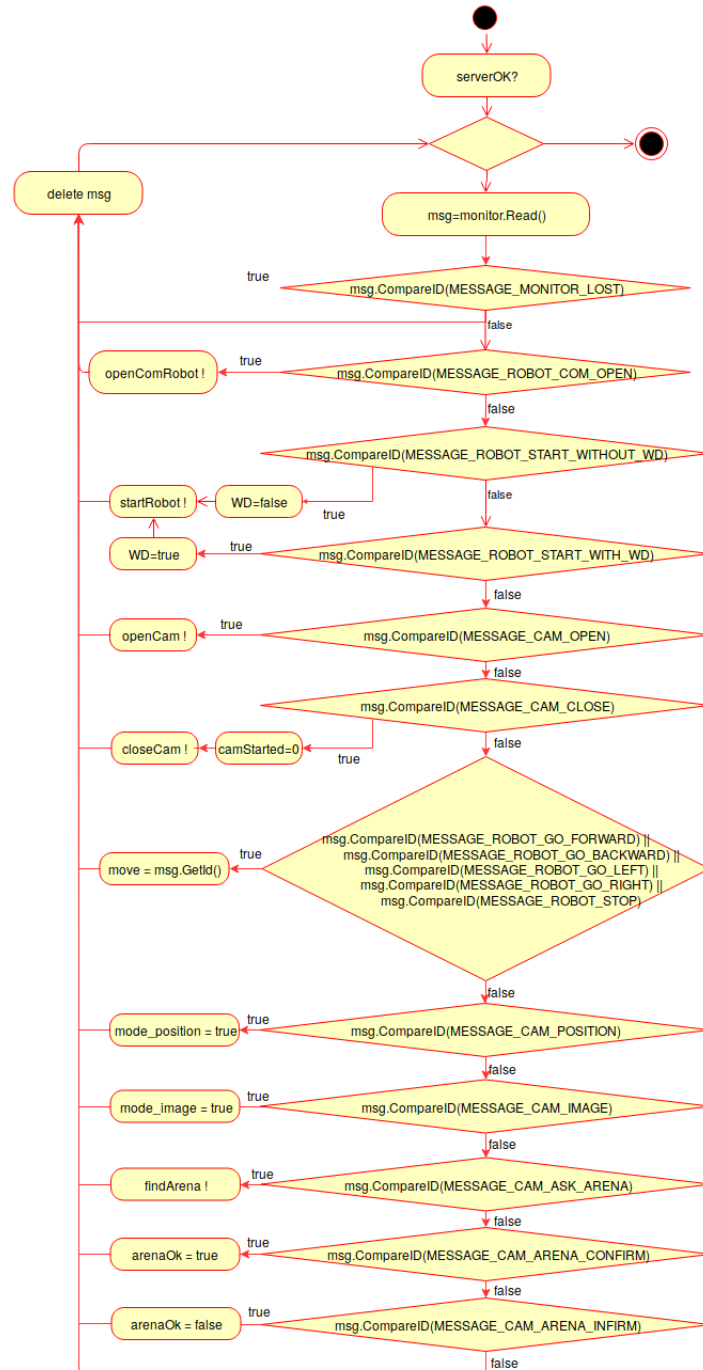
Le thread `th_server` ouvre une connexion pour le moniteur. Quand ce dernier essaye de se connecter en tant que client, sa demande est acceptée et notifie le moniteur.

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

Le thread `th_sendToMon` a 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 indiquant les actions à effectuer par le biais de messages prédéfinis. Les actions pouvant être demandées sont : ouverture de la communication, démarrage avec et sans watchdog, ouverture de la caméra, déplacement du robot, changement du mode d'acquisition de la caméra 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 concerne la caméra et l'arène. Il va permettre d'allumer la caméra, de capturer des images dans deux modes différents (position et image) et de chercher une arène sur une photo.

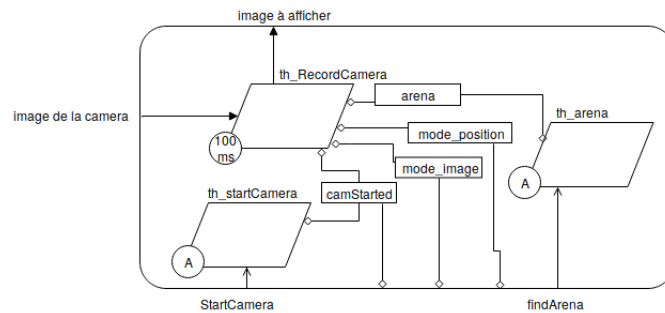


Fig. 6: Diagramme fonctionnel du groupe de threads gestion de la vision

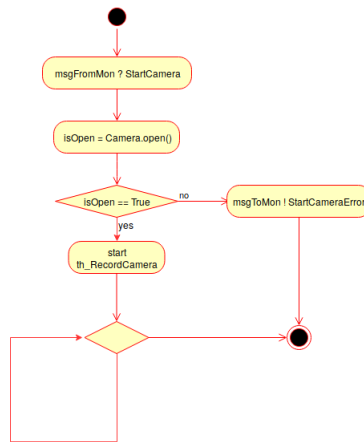
1.3.2 Description des threads du groupe vision

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

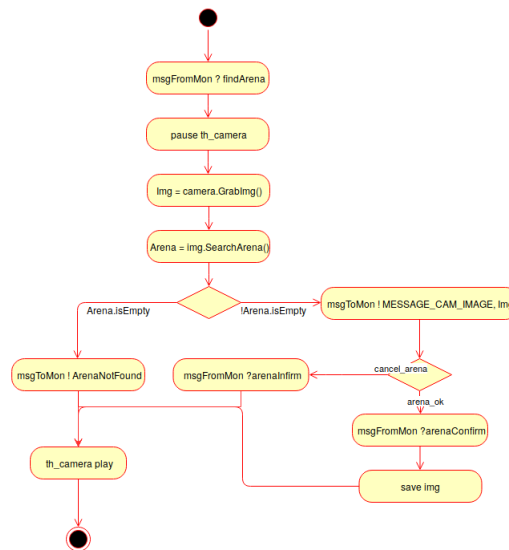
Nom du thread	Rôle	Priorité
<code>th_startCamera</code>	Démarre la caméra	18
<code>th_arena</code>	Cherche une arène sur une image et la sauvegarde si elle est trouvée (demande confirmation à l'utilisateur)	17
<code>th_RecordCamera</code>	Renvoie des photos selon le mode (position ou image)	16

1.3.3 Diagrammes d'activité du groupe vision

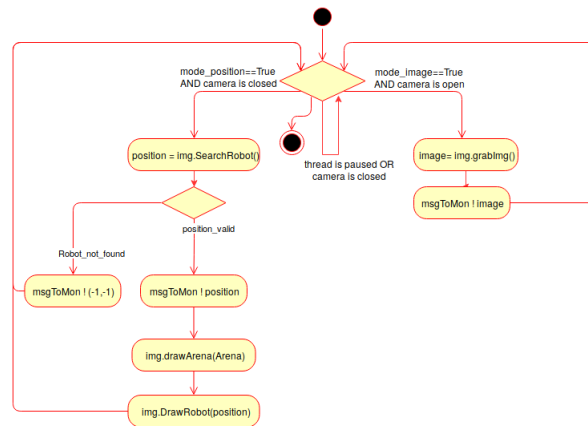
Le thread `th_startCamera` reçoit un message du moniteur, allume la caméra et notifie le moniteur en cas d'échec.

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 l'ordre 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. S'il la trouve, il notifie le moniteur qui va demander une confirmation de l'utilisateur. Si l'utilisateur confirme que l'arène est bien sur la photo, le thread `th_arena` va enregistrer l'image et s'arrêter.

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 ou plusieurs robot(s) dans l'arène et de renvoyer sa position. Le deuxième mode est le mode image qui prend des photos et les renvoie au moniteur. Le thread peut également être mis en pause, notamment par le thread `th_arena`.

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.

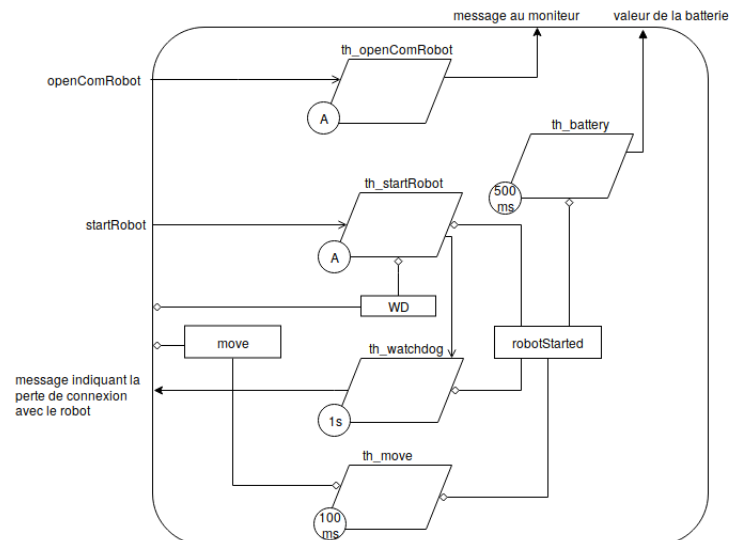


Fig. 10: Diagramme fonctionnel du groupe de threads gestion du robot

1.4.2 Description des threads du groupe gestion robot

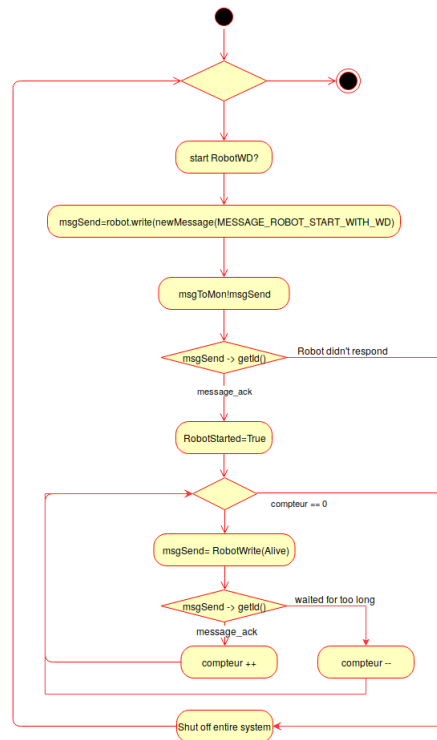
Tab. 3: Description des threads du groupe `th_group_gestion_robot`

Nom du thread	Rôle	Priorité
<code>th_watchdog</code>	Vérifie la connexion entre le moniteur et le robot	22
<code>th_openComRobot</code>	Démarre la communication avec le robot	20
<code>th_startRobot</code>	Démarre le robot	20
<code>th_move</code>	Permet de déplacer le robot	20
<code>th_battery</code>	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 aucune modification n'a été appliquée. En ce qui concerne le thread `th_startRobot` nous n'avons fait qu'une légère modification afin de choisir le mode de démarrage, c'est-à-dire avec ou sans watchdog.

Le thread `th_watchdog` se lance si le robot est démarré en mode watchdog. Ce thread envoie des messages périodiquement (toutes les secondes) au robot et si 3 messages de suite ne sont pas acquittés, le système entier (moniteur, robot...) se réinitialisera puis s'éteindra.

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.

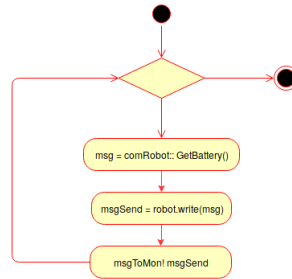


Fig. 12: Diagramme d'activité du thread `th_battery`

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’acquiescement 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 envoie un message périodiquement au robot (toutes les 500ms). Dans notre diagramme, nous avons décidé d’incrémenter notre compteur à chaque réception de message et décrémenter à chaque perte. Lors de la réalisation, nous avons plutôt choisi de déclarer la connexion comme perdue uniquement si trois messages de suite n’ont pas de réponse. Le watchdog est complètement rechargé si la communication est rétablie. Si la connexion est perdue le système entier s’éteint, comme décrit ci-dessous avec la 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 connexion est détectée, un message est affiché 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. Le système est donc complètement réinitialisé.

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’acquiescement 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’acquiescement 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 connexion. Dans ce thread, un message est envoyé au moniteur lui indiquant dans quel mode la connexion a été établie.

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 renvoie 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` est lancé suite à la demande de l'utilisateur via le moniteur. Le moniteur envoie un message à ce thread ce qui 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 : Le thread `th_RecordCamera` est lancé avec une période de 100ms. Il vérifie que la caméra est toujours ouverte et le mode d'acquisition. Si la variable partagée `mode_image` vaut `true`, alors une image est renvoyée au moniteur à chaque itération.

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'acquiescement de la demande. L'envoi périodique des images doit alors être stoppé.

Réalisation : Si une demande de fermeture de caméra est faite par le moniteur (`MESSAGE_CAMERA_CLOSE`), alors la caméra sera fermée par le thread `th_ReceiveFromMon`. Ainsi, le thread `th_RecordCamera` détectera que la caméra a été fermée et cessera d'enregistrer des images. Nous ne renvoyons pas d'acquiescement 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 recherche d'arène arrive du moniteur, le thread `th_arena` démarre. Il prend ensuite le sémaphore `sem_recordCamera`, afin de 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 ensuite chercher l'arène. S'il la trouve, l'image contenant l'arène sera envoyée au moniteur `MESSAGE_CAM_IMAGE`. Sinon le moniteur sera notifié de l'échec.

L'utilisateur doit ensuite confirmer la validité de l'image. Si c'est le cas, le moniteur enverra le message `MESSAGE_CAM_ARENA_CONFIRM` et l'image sera enregistrée. Sinon le moniteur enverra le message `MESSAGE_CAM_ARENA_INFIRM`.

Après cela, le sémaphore `sem_recordCamera` est relâché pour libérer le thread `th_RecordCamera`.

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 d'acquisition (image ou position) est géré par la variable partagée `mode_image`. Cette variable est passée à `true` dans le cas où le moniteur envoie `MESSAGE_CAM_IMAGE` et `false` avec le message `MESSAGE_CAM_POSITION`. A chaque itération, `th_RecordCamera` vérifie le mode afin d'effectuer les actions correspondantes. Le mode par défaut est le mode image (détaillé ci-dessus).

Le mode position récupère une image, dessine l'arène précédemment enregistrée et cherche le(s) robot(s) présents dans l'arène. L'image ainsi dessinée est ensuite renvoyée au moniteur. Ainsi, les positions ne sont pas renvoyées, il n'y a pas d'envoi de (-1,-1) en cas d'échec. En effet, dans le cas où aucun robot n'est détecté, rien n'est dessiné. Cette fonctionnalité n'a pas pu être testée.

2.19 Fonctionnalité 19

Description : Suite à une demande de l'utilisateur d'arrêter 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 itération du thread `th_RecordCamera`. Si l'utilisateur le change, ce changement sera pris en compte grâce à la variable `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 créées dans la méthode `init` de `tasks.cpp` et démarrées dans la méthode `run`.

Par exemple, la tâche `th_server` est déclarée ligne 73 dans le fichier `tasks.h`

```
RT_TASK th_server;
```

il est créé ligne 102 de `tasks.cpp` lors de l'appel de

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

et son démarrage est effectué 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.

Nous avons défini les priorités lors de la création des threads :

```
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 dans un premier temps déclarer sa période

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

puis déclarer l'attente d'une période 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 pouvons utiliser des sémaphores pour bloquer et libérer des tâches, que ce soit périodiquement ou occasionnellement. Par exemple, pour le thread `th_arena`, le sémaphore `sem_arena` permet de bloquer le thread et de le débloquent occasionnellement quand ce dernier est disponible

3.2 Port d'événement

3.2.1 Instanciation

Comment avez-vous instancié un port d'événement ?

Comme décrit précédemment, nous avons utilisé les sémaphores couplées aux mutex. L'envoi de messages au moniteur nous a également été utile.

3.2.2 Envoi d'un événement

Quels services ont été employés pour signaler un événement ?

Afin de signaler un événement, plusieurs options étaient disponibles : écrire dans une queue, libérer un sémaphore ou mettre à jour une variable partagée.

3.2.3 Réception d'un événement

Comment se fait l'attente d'un événement ?

La réception d'un événement peut se faire par une attente de libération d'un sémaphore, le changement de valeur d'une variable partagée ou de la réception d'un message particulier.

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. Certaines sont également instanciées avec une valeur par défaut dans `tasks.h`

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.

Nous avons principalement utilisé une `RT_QUEUE` 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 une `RT_QUEUE`.

3.4.3 Réception d'une donnée

Quels services avez-vous employés pour recevoir des données ?

Pour recevoir des données, il faut lire dans une `RT_QUEUE`.