Rapport de projet De Stijl 2.0

Charan NALAKALASALA (conception, camera, rédaction du compte-rendu)

Poonkuzhali PAJANISSAMY(conception, camera, rédaction du compte-rendu)

SHAN Xiaoshan (conception, monitor, robot, rédaction du compte-rendu)

ZHAO Yuanyuan (conception, monitor, robot, rédaction du compte-rendu)

1 Conception

1.1 Diagramme fonctionnel général

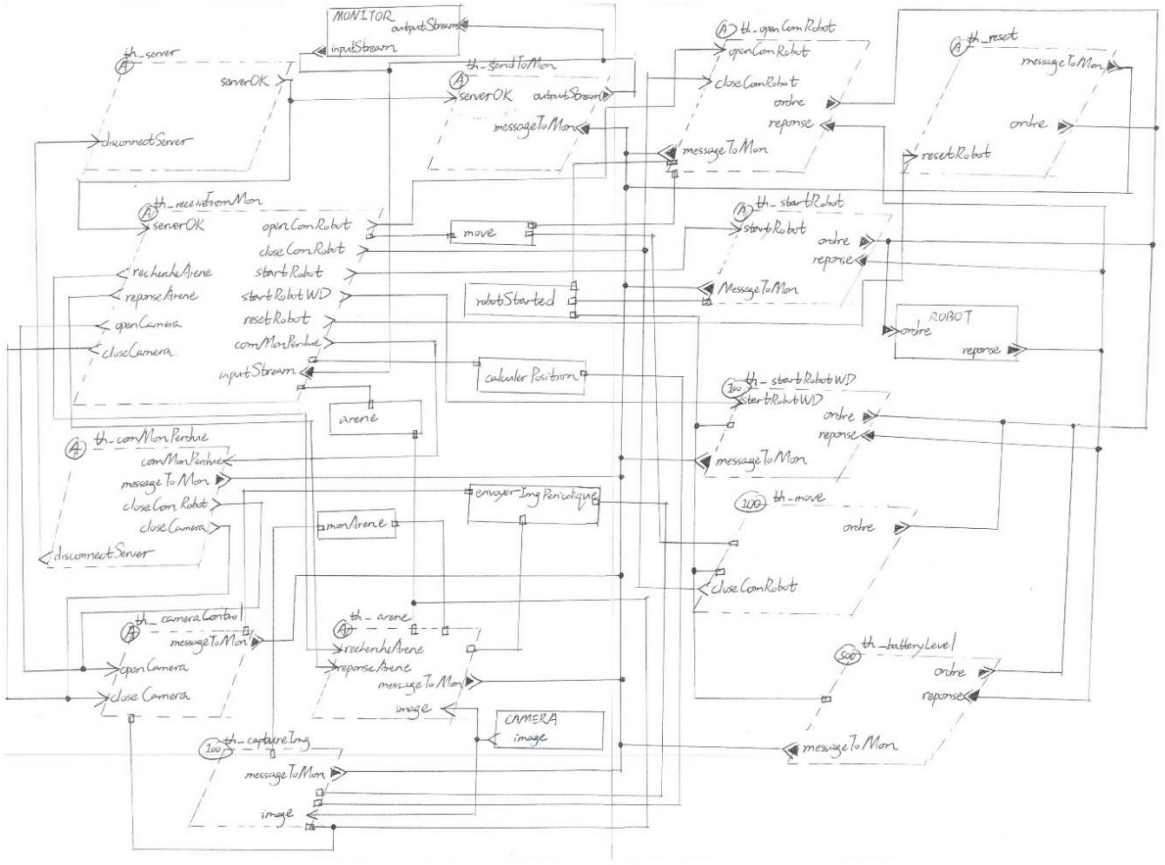

Fig. 1: Diagramme fonctionnel du système

1.2 Groupe de threads gestion du moniteur

1.2.1 Diagramme fonctionnel du groupe gestion du moniteur

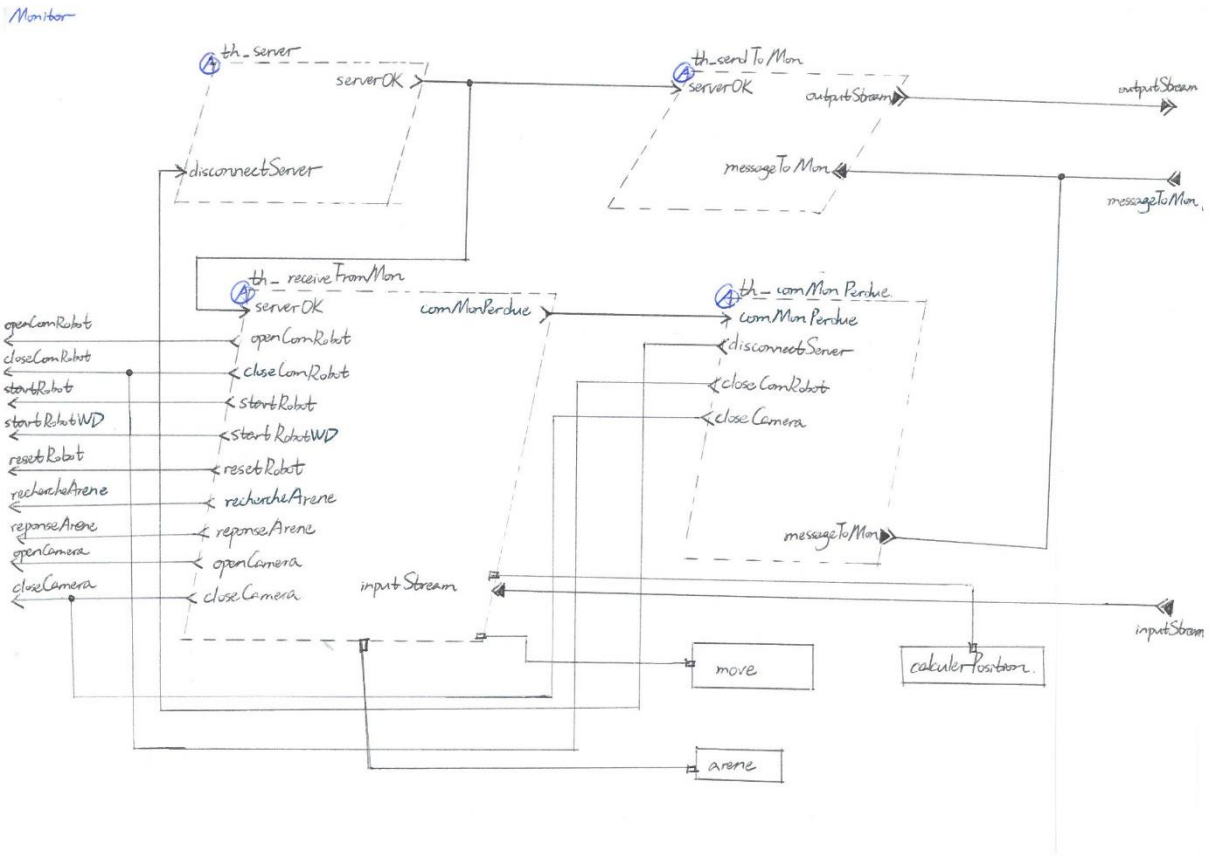

Fig. 2: Diagramme fonctionnel du groupe de threads gestion du moniteur

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	Lancement du serveur et établissement de la connexion. Fermer le monitor quand reçoit l′ événement（disconnectServer).	30
th_receiveFromMon	Envoie des événement	25
th_sendToMon	Réception des messages	22
th_comMonPerdue	Détection de la perte de communication, envoyer message au moniteur, envoyer l'événement (closeComRobot) et (closeCamera), revenir dans le même état qu'au démarrage du superviseur	22

1.2.3 Diagrammes d'activité du groupe gestion du moniteur

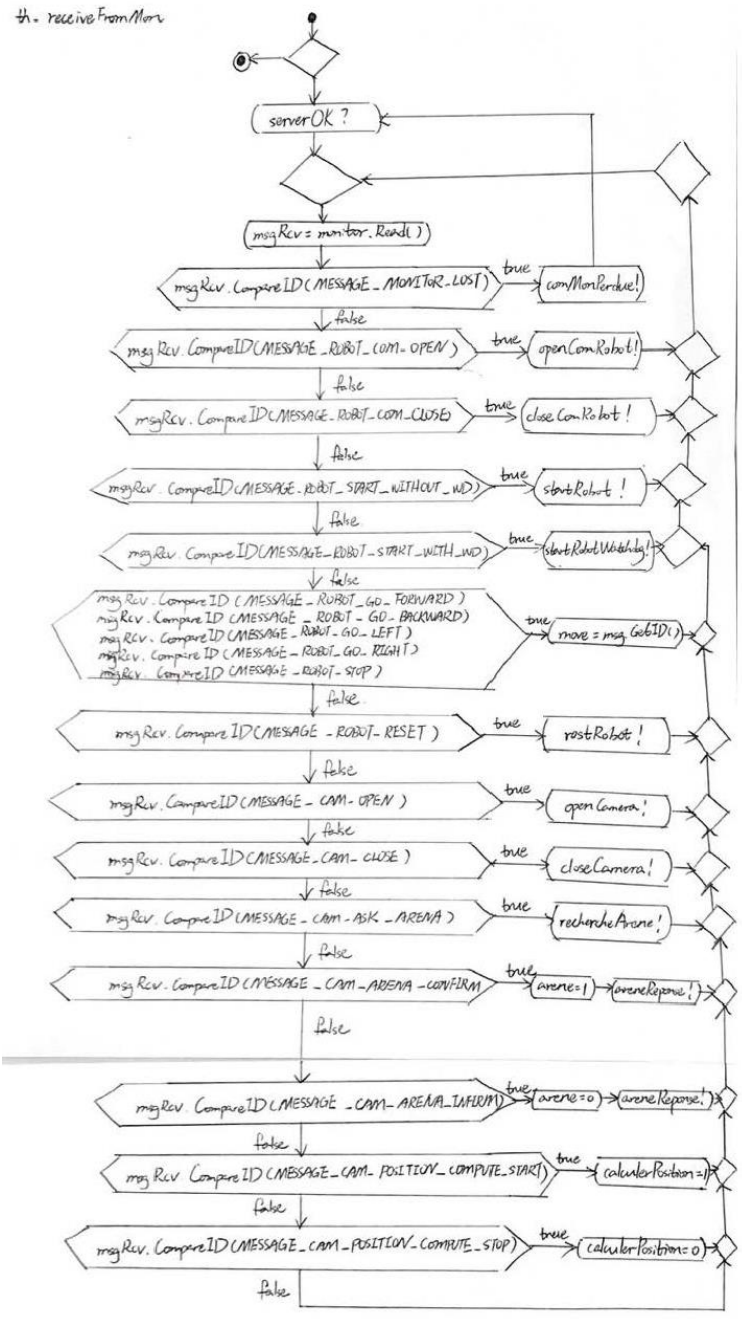

Fig. 3: Diagramme d'activité du thread th_receiveFromMon

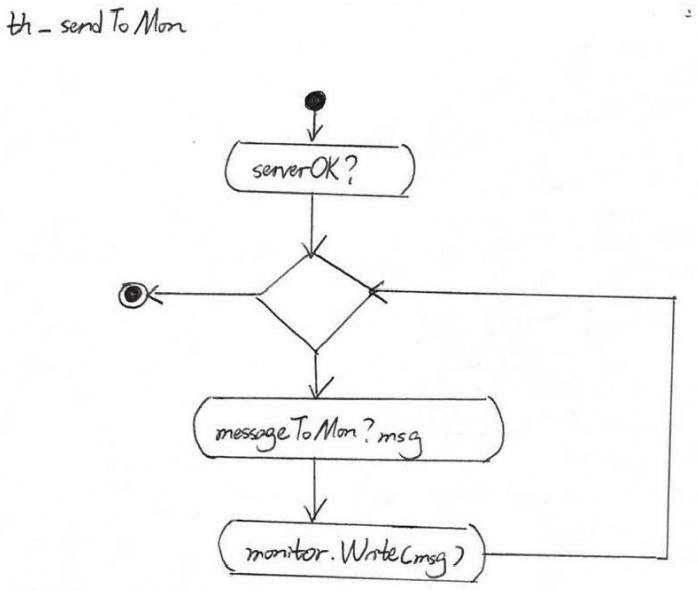

Fig. 4: Diagramme d'activité du thread th_sendToMon

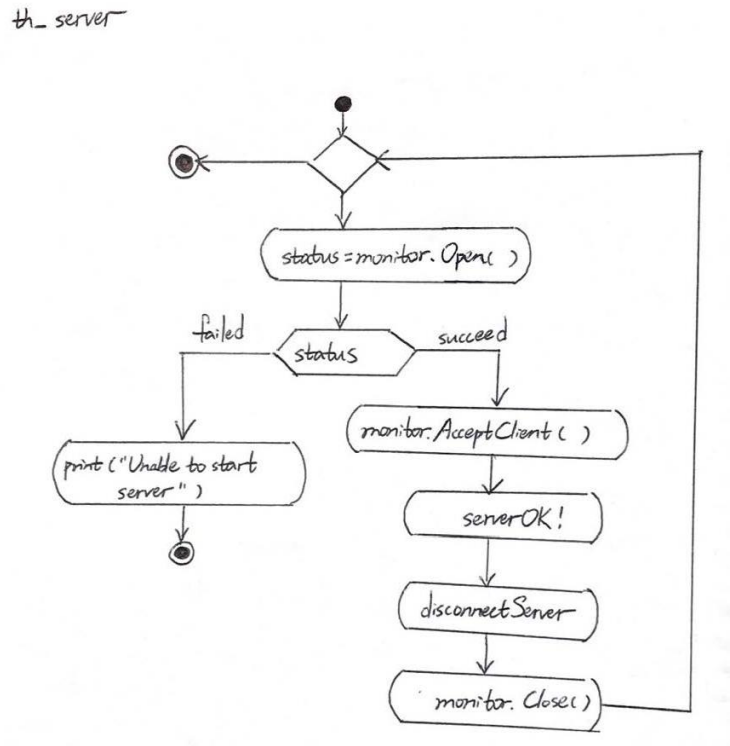

Fig. 5: Diagramme d'activité du thread th_server

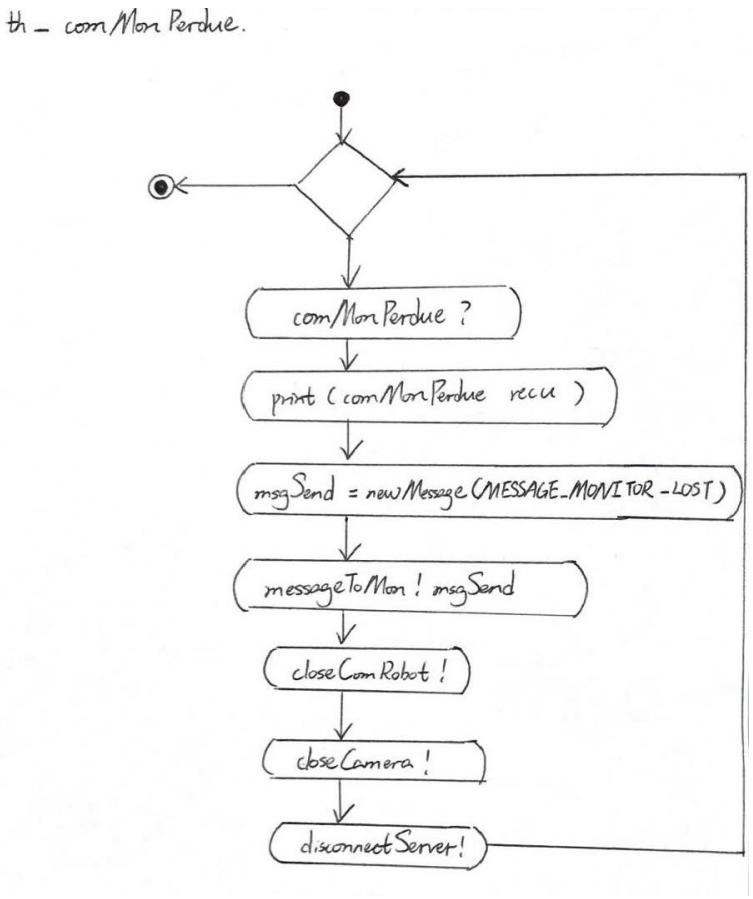

Fig. 6: Diagramme d'activité du thread th_comMonPerdue

1.3 Groupe de threads vision

1.3.1 Diagramme fonctionnel du groupe vision

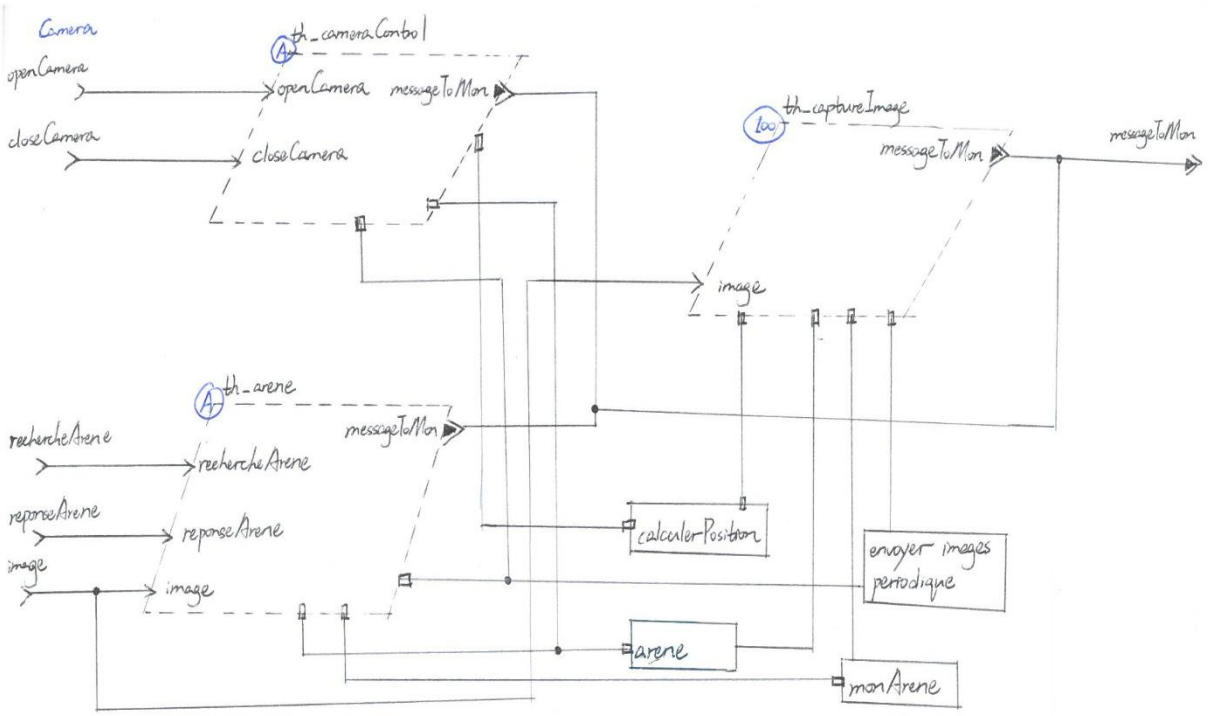

Fig. 7: Diagramme fonctionnel du groupe de threads gestion de vision

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_cameraControl	Contrôler le caméra (ouvrir ou fermer)et envoyer message au moniteur	20
th_captureImg	Obtenir image et envoyer message au moniteur	20
th_arene	Envoyer message au moniteur, recevoir l'événement recherchArene, recevoir la réponse de l'arène , obtenir image	20

1.3.3 Diagrammes d'activité du groupe vision

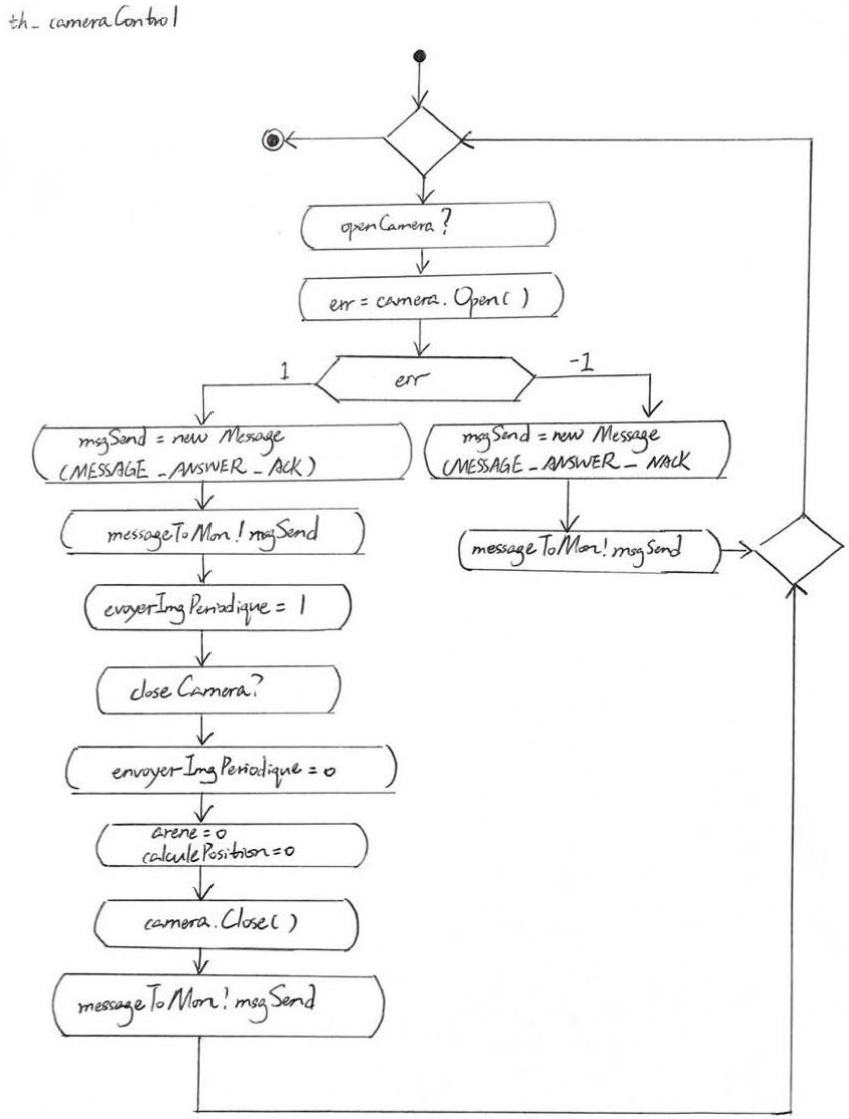

Fig. 8: Diagramme d'activité du thread th_cameraControl

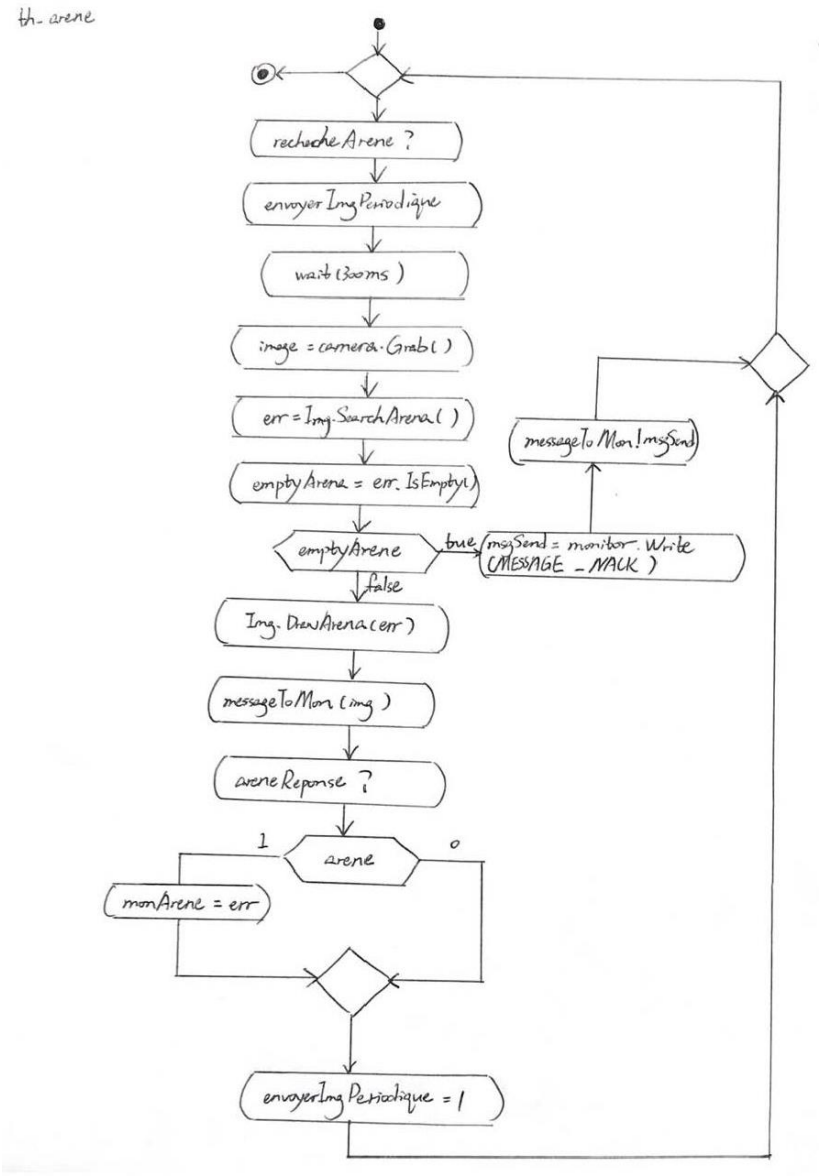

Fig. 9: Diagramme d'activité du thread th_arene

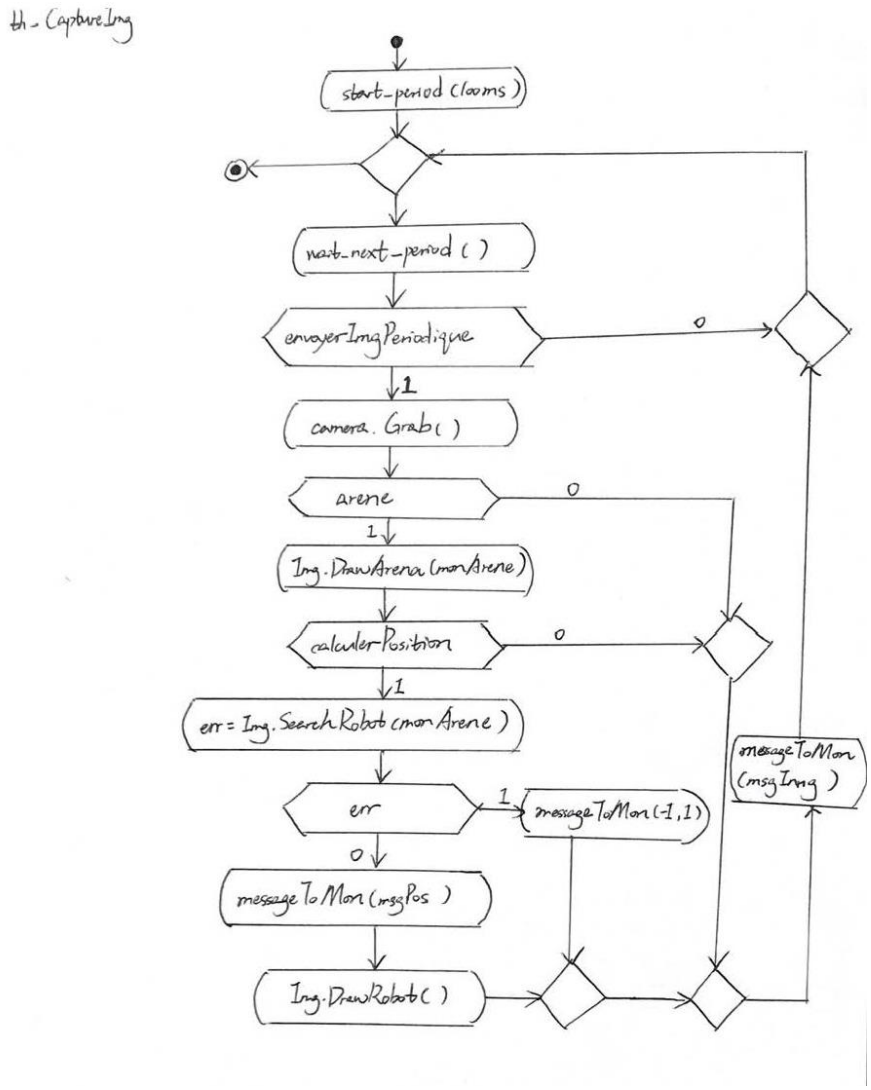

Fig. 10: Diagramme d'activité du thread th_captureImg

1.4 Groupe de threads gestion du robot

1.4.1 Diagramme fonctionnel du groupe gestion robot

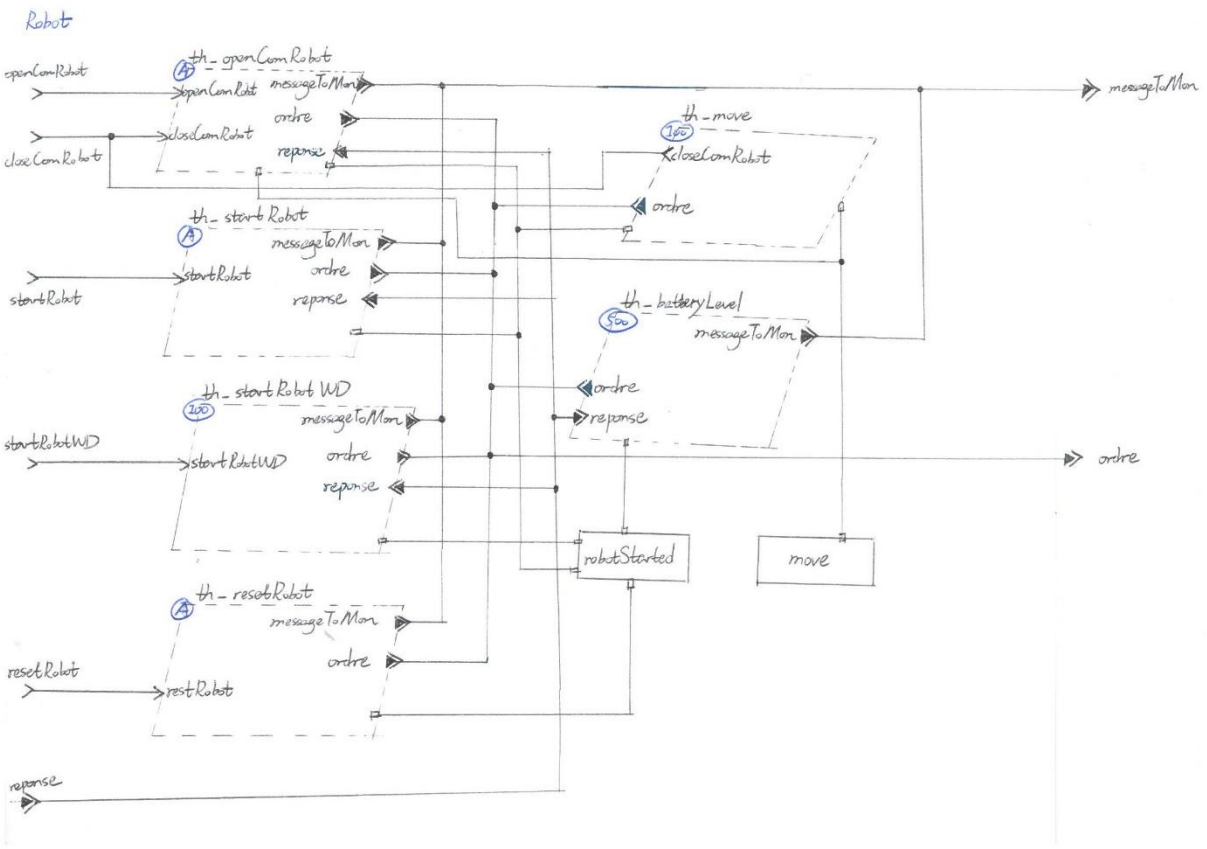

Fig. 11: Diagramme fonctionnel du groupe de threads gestion du robot

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_openComRobot	Ouvrir ou fermer communication robot, envoyer message au moniteur, donner ordre et recevoir réponse	20
th_startRobot	Démarrage du robot sans watchdog, envoyer message au moniteur, donner ordre et recevoir réponse	20
th_startRobotWD	Démarrage du robot avec watchdog, envoyer message au moniteur, donner ordre et recevoir réponse	20
th_resetRobot	Reset robot, envoyer message au moniteur, donner ordre	20
th_move	Donner ordre, envoyer l'événement (closeComRobot) quand le robot perte communication	20
th_batteryLevel	Quand le robot est démarré, il envoie toutes les 500 ms un message de niveau de la batterie du robot au monituer	20

1.4.3 Diagrammes d'activité du groupe robot

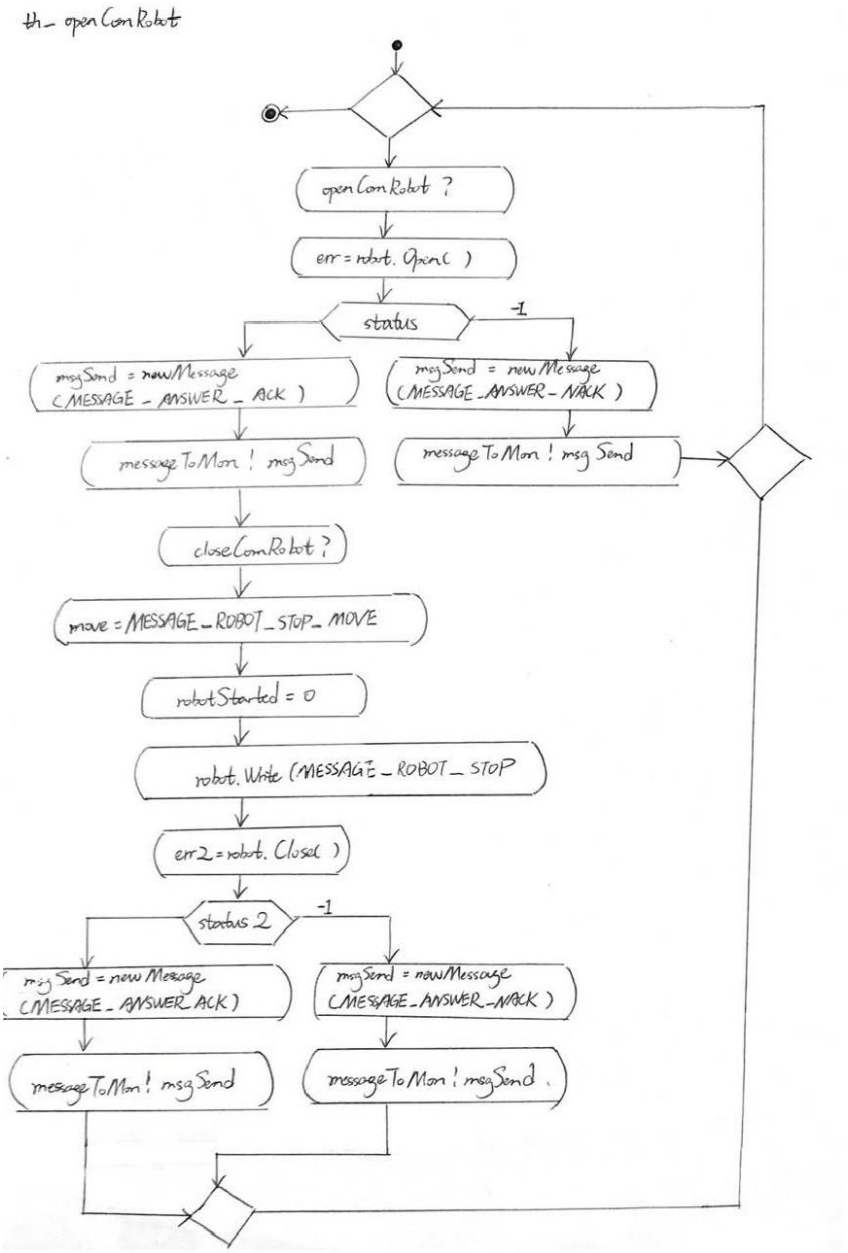

Fig. 12: Diagramme d'activité du thread th_openComRobot

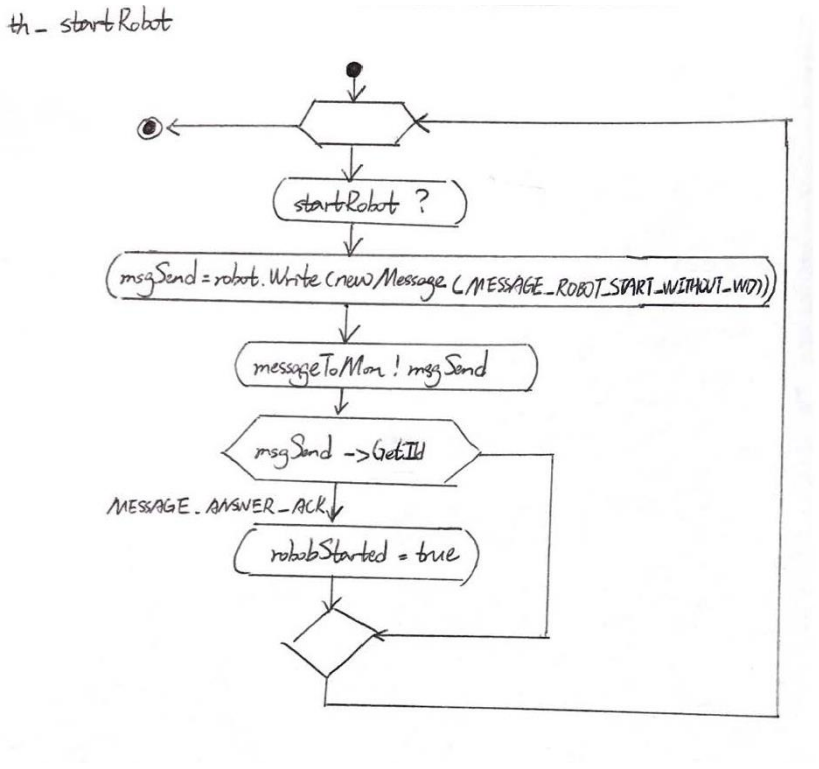

Fig. 13: Diagramme d'activité du thread th_startRobot

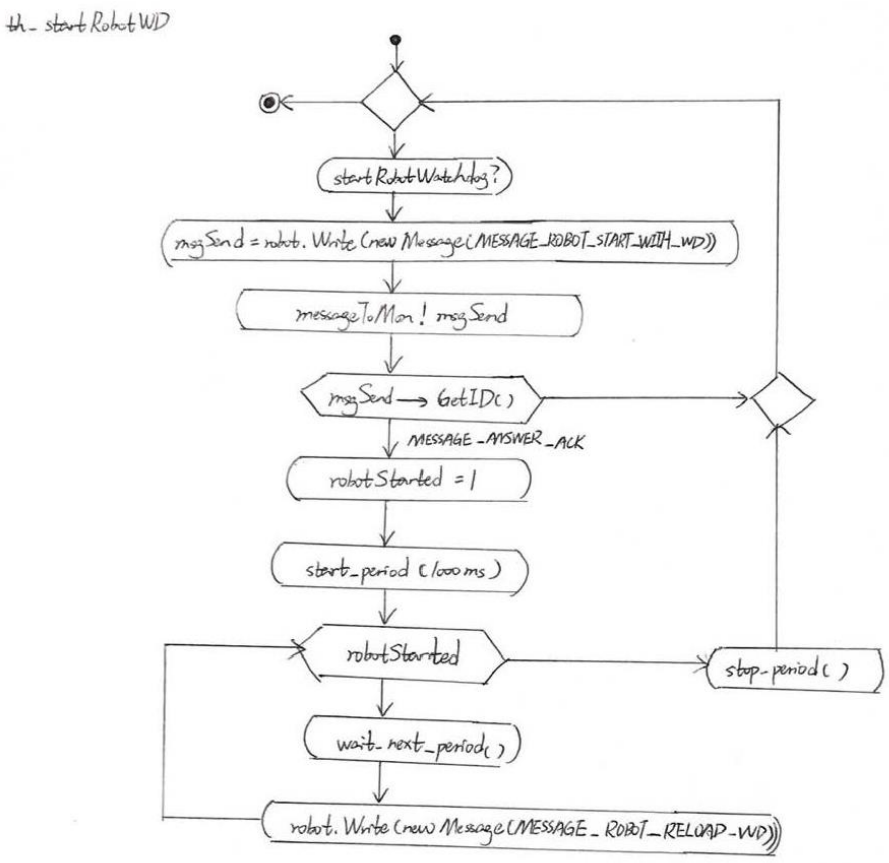

Fig. 14: Diagramme d'activité du thread th_startRobotWD

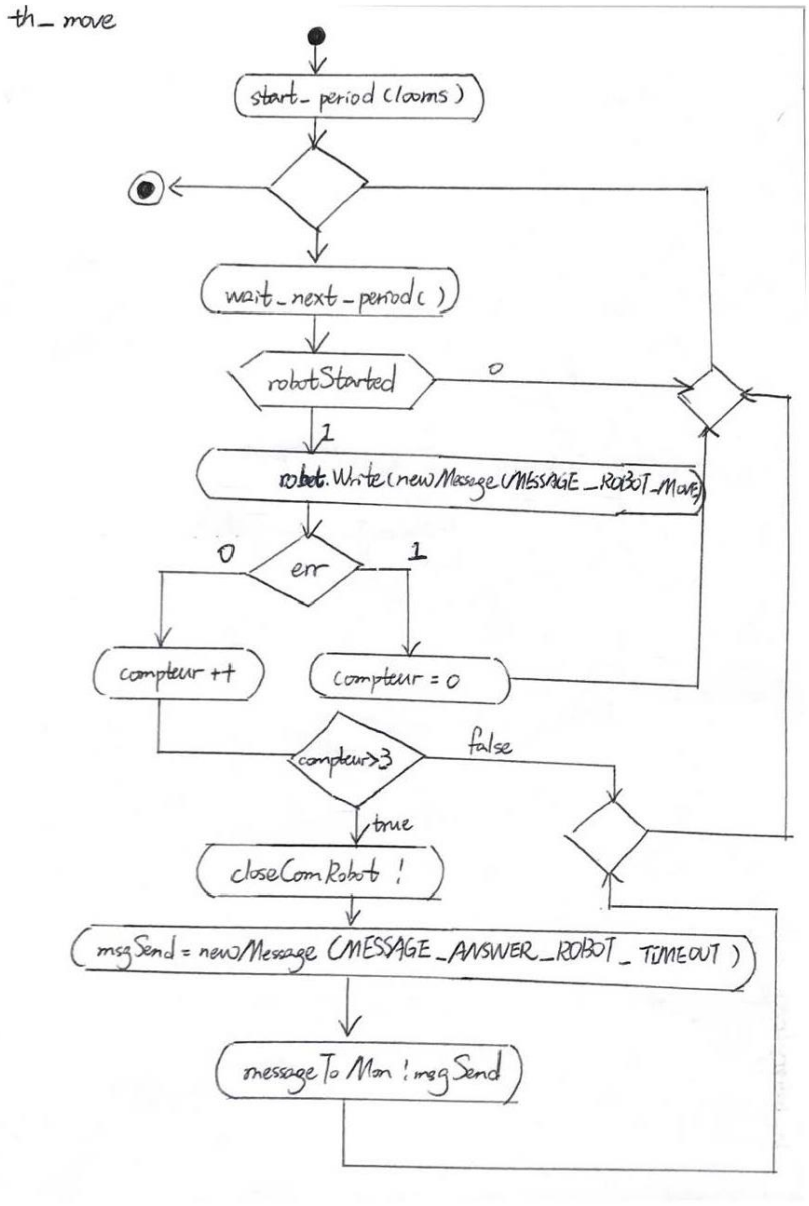

Fig. 15: Diagramme d'activité du thread th_move

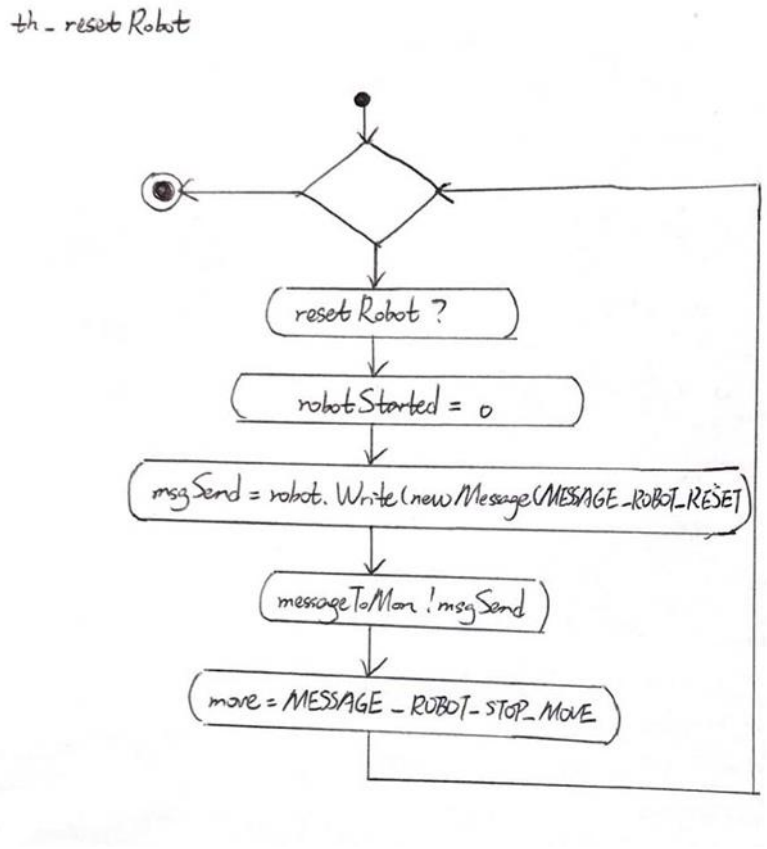

Fig. 16: Diagramme d'activité du thread th_resetRobot

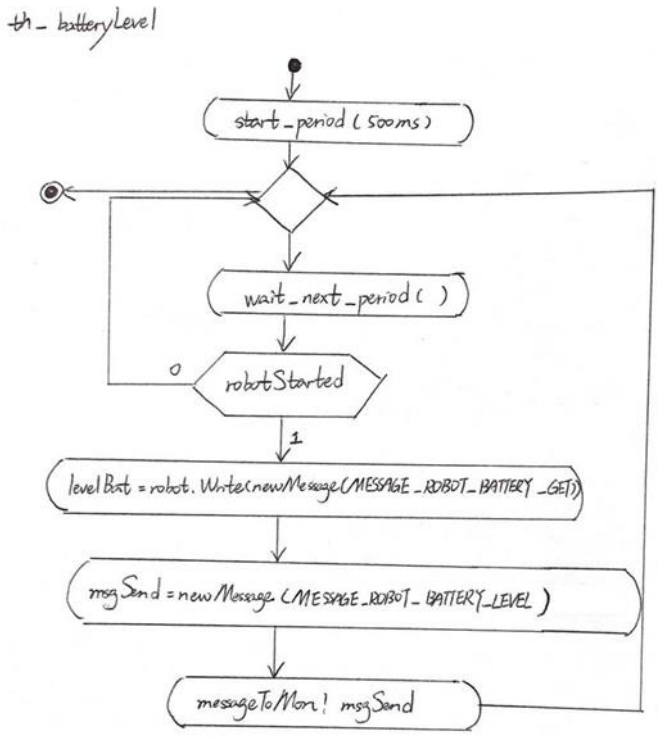

Fig. 17: Diagramme d'activité du thread th_batteryLevel

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.

Réalisation : La fonctionnalité est implémentée à l'aide d'une tâche(tâche :th_server).

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.

Réalisation : La fonctionnalité est implémentée à l'aide d'une tâche(tâche :th_server). Pour ce thread on a ajouté le fermeture du monitor quand reçoit l'événement (disconnectServer).

2.3 Fonctionnalité 3*

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

Réalisation : Cette fonctionnalité a été implémentée à l'aide d'une tâche (tâche :th_receiveFromMon) qui reçoit les messages pré -formaté depuis le moniteur. Sur le thread origine, on a complété la partie robot, on a aussi ajouté les parties concernant le move et le caméra.

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.

Réalisation : Cette fonctionnalité a été implémentée à l'aide d'une tâche (tâche :th_sendToMon). Pour le délai d'au plus 10 ms, on n'a pas ajouté ce contraint.

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.

Réalisation :
Cette fonctionnalité a été implémentée à l'aide d'une tâche(tâche :th_comMon_perdue) qui est responsable d'envoyer les événements d' arrêter les tâches quand il recevoir l' événement(communication_perdue). Th_ comMon_perdue va attendre jusqu'à réception de signal communication_perdue qui est envoyé par th_receiveFromMon. Après,un message MESSAGE_MONITOR_LOST doit être envoyé à th_senToMon.

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.

Réalisation : Cette fonctionnalité est sur la même tâche(tâche :th_comMon_perdue) avec la fonctionnalité 5.Quand l' événement(communication_perdue) est reçu,signals vont être envoyé pour stopper le robot, la communication avec le robot, et déconnecter la caméra. Et cette fonctionnalité ne peut être mise en place que lorsque l'ensemble de l'application est déjà réalisée,donc avant d'envoyer l' événement (disconnectServer),on doit attendre une période du temps (wait(1s)).Et la priorité de th_comMon_perdue doit être inférieur.Finalement,quand th_server recoit l' événement (disconnectServer),monitor va être fermé(monitor.close()) et reprendre la communication.

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.

Réalisation : Cette fonctionnalité a été implémentée à l'aide d'une tâche(tâche :th_openComRobot). Dans cette tâche, on a aussi ajouté la fermeture de la communication avec le robot.

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 : Cette fonctionnalité a été implémentée dans le thread(th_move). Car la tâche th_move doit envoyer les ordres de movements au robot(robot.Write(new Message(move))),on peut déclarer un objet err de type message pour déterminer si il a réussi ou pas. Pour détecter une perte de communication, on utilise un compteur. Et un varibale compteur est utilisé pour compter chaque échec. Quand il y a un succès, on remet le compteur à 0. Si il dépasse 3, on ferme la communication robot. On envoi un message (MESSAGE_ ANSWER_ROBOT_TIMEOUT) à moniteur. On le met dans état initial.

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 implémentéeà l'aide de la tâche th_resetRobot. Quand on reçoit l'événement resetRobot, on met la donnée partagée robotStarted=0. On envoie le message (MESSAGE_ROBOT_RESET)au moniteur. On envoie aussi les ordres de movement au robot(move=MESSAGE_ROBOT_STOP_MOVE).

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.

Réalisation : Cette fonctionnalité a été implémentéeà l'aide de la tâche th_startRobot. Quand on reçoit l'événement startRobot. On envoie le message (MESSAGE_ROBOT_START_WITHOUT_WD) au moniteur. Quand on reçoit la réponse (MESSAGE__ANSWER__ACK), le robot démarre (la donéé partagé robotStarted=1). En cas d'échec(MESSAGE__ANSWER__NACK), on revient à l'état initial.

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 a été implémentée à l'aide de la tâche th_startRobotWD. Quand on a reçu l'événement startRobotWD. On envoie le message (MESSAGE_ROBOT_START_WITH_WD) au moniteur. Quand on reçoit la réponse (MESSAGE__ANSWER__ACK), le robot démarre (la donnée partagée robotStarted=1), on ajoute une période (start_period(1000ms)), et le robot envoie régulièrement le message de rechargement du watchdog (MESSAGE_ROBOT_RELOAD_WD). Quand la donéé partagé robotStarted !=1, on arrête la période (stop_period), et revient à l'état initial.

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.

Réalisation : Cette fonctionnalité a été implémentée à l'aide d'une tâche qui envoie toutes les 100 ms un ordre de mouvement au robot une fois que celui-ci est démarré (tâche th_move). Cette implémentation ne garantit pas que le temps soit inférieur à 100 ms entre la réception du message et sa prise en compte par le robot. En effet, le temps de traitement de la réception par la tâche th_receiveFromMon, le temps de traitement de la tâche th_move et celui de l'envoi de l'ordre via le Xbee ne sont pas considérés. Afin de réduire ces délais, les priorités de th_receiveFromMon et de th_move sont élevés mais ne permettent pas de garantir l'exigence de temps. Augmenter la fréquence de la tâche th_move permettrait de tenir cette contrainte, mais risque de surcharger la communication avec le robot (une version avec l'envoi de l'ordre que s'il a changé serait souhaitable). Finalement, une version asynchrone (attente d'un événement-donnée entre th_receiveFromMon et de th_move) aurait été préférable. Cependant, après discussion avec le client, la version périodique à 100 ms est cependant validée.

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 :
Cette fonctionnalité a été implémentée sur la tâche th_batteryLevel.Quand le robot est démarré(la donéé partagé robotStarted==1),il envoie toutes les 500 ms un message de niveau de la batterie du robot au monituer par la tâche th_sendToMon.

2.14 Fonctionnalité 14

Description : The camera must be started following a request from the monitor. If the camera opening has failed, a message must be sent to the monitor.

Réalisation :
Method used :
camera.Open() – Opens the camera
MESSAGE_CAM_OPEN is the id used and it waits for the acknowledgement.

2.15 Fonctionnalité 15

Description : As soon as the camera is open, an image must be sent to the monitor every 100 ms.
Réalisation :
Method used:
camera.Grab() – Captures the image
The image is sent for every 100 ms to the monitor by fixing a parameter while instantiating the Camera object.

2.16 Fonctionnalité 16

Description : The camera must be closed following a request from the monitor. A message must be sent to the monitor to signifier acknowledge the request. The periodic sending of images must then be stopped.

Réalisation :
Method used:
camera.Close() – Stops the Camera
When the supervisor receives the message from monitor to close the camera, it should be closed using camera.Close() method and id used id is MESSAGE_CAM_CLOSE
The periodic sending of the images are also closed.

2.17 Fonctionnalité 17

Description : Following a request to search for the arena, the supervisor must stop the periodic sending of images, search for the arena and send back an image on which the arena is drawn. If no arena is found, a failure message is sent.

The user must then visually validate via the monitor if the arena has been found. The user can:

- validate the arena: in this case, the supervisor must save the found arena (for later use) and then return to his periodic image sending mode by adding the drawn arena to the image.

- cancel the search: in this case, the supervisor must simply return to his periodic sending mode of images and invalidate the search.

Réalisation : It is treated using the id's:
MESSAGE_CAM_ASK_ARENA is the id used for Arena detection request
MESSAGE_CAM_ARENA_CONFIRM is the id that tells that the arena is correct
MESSAGE_CAM_ARENA_INFIRM is the id used to tell that the arena is not correct
Methods used:
SearchArena()- Performs Calculation
DrawArena() - Draws the arena on the image

2.18 Fonctionnalité 18

Description : The processing of an image to find the robot's position is done using Image SearchRobot method. It is possible to draw the found position on the image using the DrawRobot method. The position is sent from the supervisor to the monitor using a message with a dedicated header.

Réalisation : It is treated using the id:
MESSAGE_CAM_POSITION_COMPUTE_START that is used for Calculation of the robot position.
Methods used:
SearchRobot()- Finds the position of an image
DrawRobot() - Draws the found position.

2.19 Functionality 19

Description: Following a request from the user to stop the calculation of the robot's position, the supervisor must switch back to a sending mode of the image without the calculation of the position.

Realisation:

It is treated using the id:

MESSAGE_CAM_POSITION_COMPUTE_STOP that is used for stopping the calculation of the robot's position.

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.

Avec void Tasks::Run() {}

Par exemple pour le task th_server

```
void Tasks::Run() {
rt_task_set_priority(NULL, T_LOPRIO);
  int err;

  if (err = rt_task_start(&th_server, (void(*)(void*)) & Tasks::ServerTask, this)) {
    cerr << "Error task start: " << strerror(-err) << endl << flush;
    exit(EXIT_FAILURE);
  }
}
```

3.1.3 Niveau de priorités

La priorité de chaque tâche sont définis dans le fichier tasks.cpp quand une tâche est creé. Par exemple, pour la tâche th_server :

```
#define PRIORITY_TSERVER 30
rt_task_create(&th_server, "th_server", 0, PRIORITY_TSERVER, 0)
```

3.1.4 Activation périodique

Pour rendre périodique l'activation d'un thread AADL sous Xenomai.

Par exemple, pour th_move, on met une période de 100ms :

```
rt_task_set_periodic(NULL, TM_NOW, 100000000);
```

3.1.5 Activation événementielle

Pour gérer les activations événementielles ,on doit déclarer semaphors dans le fichier task.h. Toutes les semaphors sont crées dans la méthode init.

Par exemple, pour l' événement serverOk , sa déclaration est faite ligne 73 dans le fichier tasks.h：

```
RT_SEM sem_serverOk ;
```

sa création dans le fichier task.cpp :

```
if (err = rt_sem_create(&sem_serverOk, NULL, 0, S_FIFO)) {
    cerr << "Error semaphore create: " << strerror(-err) << endl << flush;
    exit(EXIT_FAILURE);
  }
```

3.2 Port d'événement

3.2.1 Instanciation

On peut déclarer dans le fichier task.h.Par exemple, pour l' événement serverOk :

```
RT_SEM sem_serverOk ;
```

3.2.2 Envoi d'un événement

Par exemple, pour l' événement serverOk :

```
rt_sem_v(&sem_serverOk, TM_INFINITE);
```

3.2.3 Réception d'un événement

Par exemple, pour l' événement serverOk :

```
rt_sem_p(&sem_serverOk, TM_INFINITE)
```

3.3 Donnée partagée

3.3.1 Instanciation

Pour instancier une donnée partagée,on peut déclarer une variable dans le fichier task.h.

Par exemple pour move :

```
int move = MESSAGE_ROBOT_STOP;
```

3.3.2 Accès en lecture et écriture

On utilise mutex pour obtenir un accès exclusif à des ressources partagées

Par exemple pour move ,sa déclaration de mutex est faite dans lefichier tasks.h :

```
RT_MUTEX mutex_move ;
```

Sa création de tasks.cpp :

```
if (err = rt_mutex_create(&mutex_move, NULL)) {
cerr << "Error mutex create: " << strerror(-err) << endl << flush;
exit(EXIT_FAILURE);
}
```

Et dans le méthode moveTask :

```
rt_mutex_acquire(&mutex_move, TM_INFINITE);
cpMove = move;
rt_mutex_release(&mutex_move);
```

3.4 Ports d'événement-données

3.4.1 Instanciation

On peut déclarer dans le fichier task.h :

```
RT_QUEUE        q_messageToMon ;
```

Et on doit écrire deux métodes pour envoyer et recevoir d'une donnée:

```
void WriteInQueue(RT_QUEUE *queue, Message *msg) ;
Message *  ReadInQueue(RT_QUEUE *queue) ;
```

3.4.2 Envoi d'une donnée

Par exemple pour messageToMon :

```
WriteInQueue(&q_messageToMon, msgSend);
```

3.4.3 Réception d'une donnée

Par exemple pour messageToMon :

```
ReadInQueue(&q_messageToMon, msgSend);
```