

**Rapport de projet DE Stijl 2.0**

21/02/2022

**4 IR A1**

**EMERY Martin** (conception, robot, intégration, rédaction du compte rendu)

**PONTALIER Marie** (conception, robot, intégration, rédaction du compte rendu)

**TOMIETTO Vincent** (conception, robot, intégration, rédaction du compte rendu)

Table des matières

[**1.** **Conception** 3](#_Toc96352028)

[**1.1** **Diagramme fonctionnel général** 3](#_Toc96352029)

[**1.2** **Groupe de threads gestion du moniteur** 4](#_Toc96352030)

[**1.2.1** **Diagramme fonctionnel du groupe gestion moniteur** 4](#_Toc96352031)

[**1.2.2** **Description des threads du groupe gestion moniteur** 5](#_Toc96352032)

[**1.2.3** **Diagrammes d’activités du groupe gestion moniteur** 5](#_Toc96352033)

[**1.3** **Groupe de threads gestion du robot** 7](#_Toc96352034)

[**1.3.1** **Diagramme fonctionnel du groupe gestion robot** 7](#_Toc96352035)

[**1.3.2** **Description des threads du groupe gestion robot** 7](#_Toc96352036)

[**1.3.3** **Diagrammes d’activité du groupe gestion robot** 8](#_Toc96352037)

[**1.4** **Groupe de threads vision** 11](#_Toc96352038)

[**1.4.1** **Diagramme fonctionnel du groupe vision** 11](#_Toc96352039)

[**1.4.2** **Description des threads du groupe vision** 11](#_Toc96352040)

[**1.4.3** **Diagrammes d’activité du groupe vision** 12](#_Toc96352041)

[**2.** **Transformation AADL vers Xenomai** 15](#_Toc96352042)

[**2.1** **Thread** 15](#_Toc96352043)

[**2.1.1** **Instanciation et démarrage** 15](#_Toc96352044)

[**2.1.2** **Code à exécuter** 15](#_Toc96352045)

[**2.1.3** **Niveau de priorités** 15](#_Toc96352046)

[**2.1.4** **Activation périodique** 15](#_Toc96352047)

[**2.2** **Donnée partagée** 16](#_Toc96352048)

[**2.2.1** **Instanciation** 16](#_Toc96352049)

[**2.2.2** **Accès en lecture et écriture** 16](#_Toc96352050)

[**2.3** **Port d’événement** 16](#_Toc96352051)

[**2.3.1** **Instanciation** 17](#_Toc96352052)

[**2.3.2** **Envoi d’un événement** 17](#_Toc96352053)

[**2.3.3** **Réception d’un événement** 17](#_Toc96352054)

[**2.4** **Ports d’événement-données** 17](#_Toc96352055)

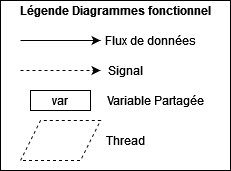
[**2.4.1** **Instanciation** 17](#_Toc96352056)

[**2.4.2** **Envoi d’une donnée** 18](#_Toc96352057)

[**2.4.3** **Réception d’une donnée** 18](#_Toc96352058)

[**3.** **Analyse et validation de la conception** 18](#_Toc96352059)

1. **Conception**
   1. **Diagramme fonctionnel général**



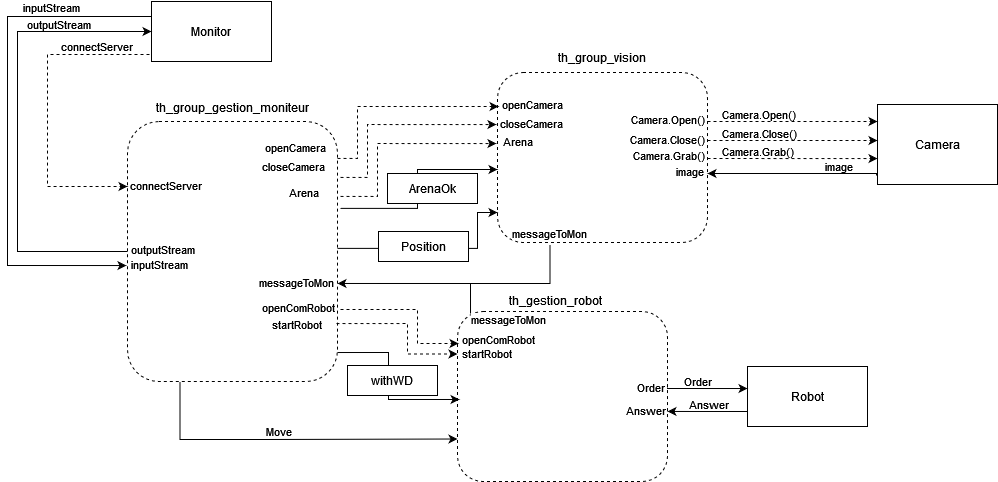


Figure 1: Diagramme fonctionnel général

* 1. **Groupe de threads gestion du moniteur**
     1. **Diagramme fonctionnel du groupe gestion moniteur**

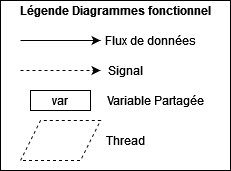
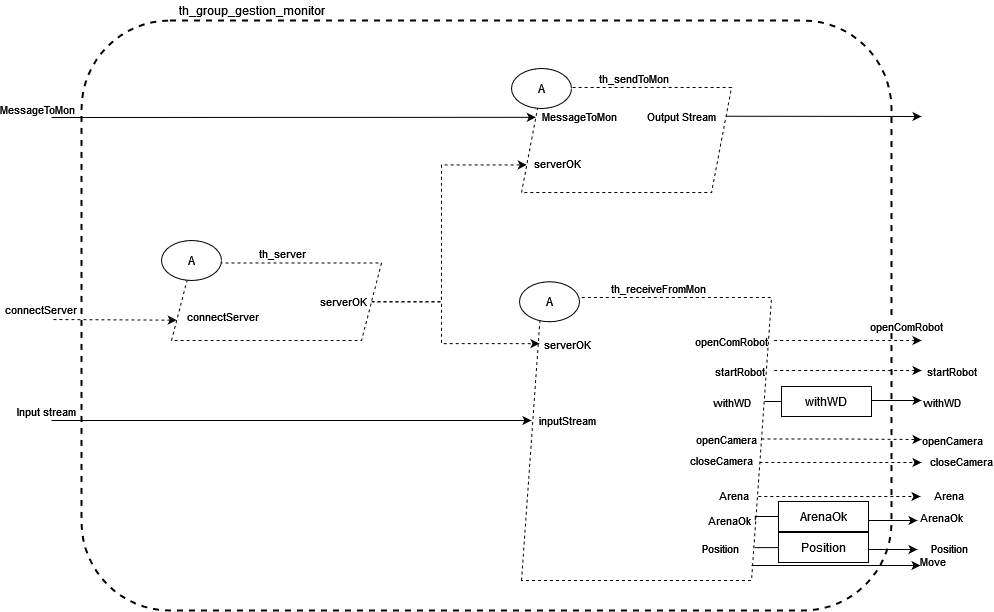


Figure 2: Diagramme fonctionnel du groupe gestion moniteur

* + 1. **Description des threads du groupe gestion moniteur**

|  |  |
| --- | --- |
| **Nom du thread** | **Rôle** |
| th\_server | Ouvre la connexion avec le moniteur, lève en erreur en cas d’échec |
| th\_sendToMon | Permet l’envoi de messages au moniteur |
| th\_receiveFromMon  Fonctionnalités 3 et 5, partiellement 6, utile à toutes les autres qui dépendent de messages du moniteur | Traite les messages reçus depuis le moniteur, permet l’activité des threads correspondants aux messages, coupe caméra, robot et moniteur en cas de perte du moniteur |

* + 1. **Diagrammes d’activités du groupe gestion moniteur**

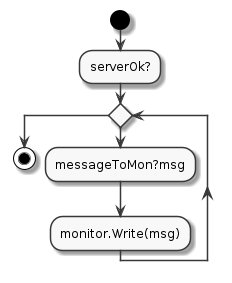


Figure 3: Diagramme d'activité du thread th\_sendToMon

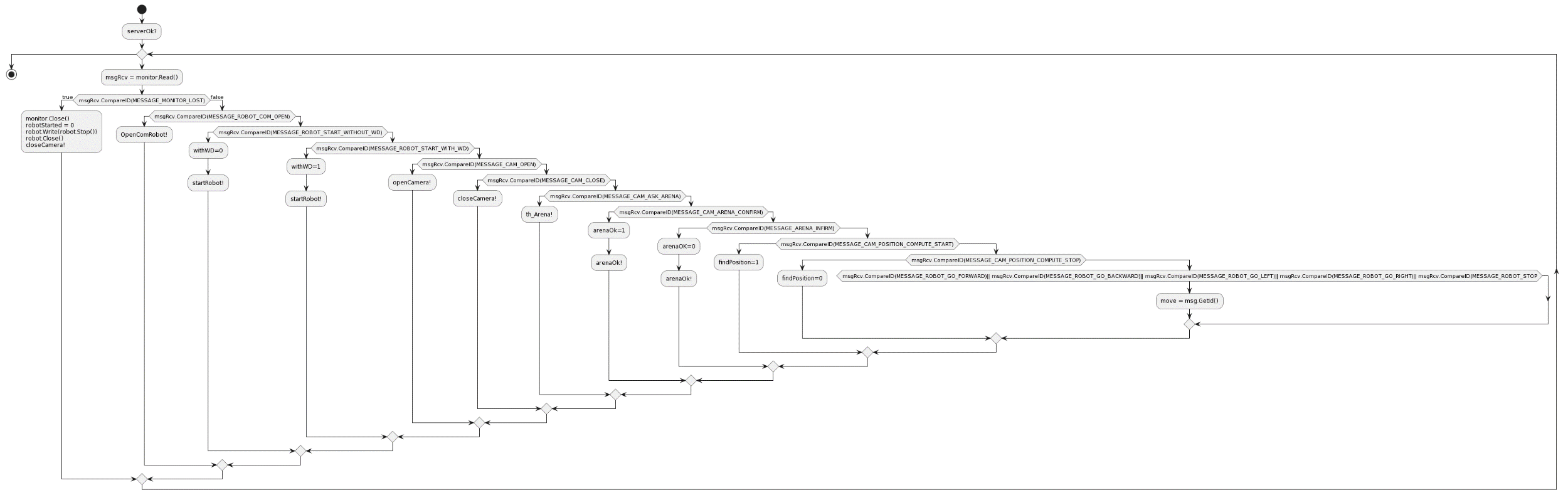


Figure 4: Diagramme d'activité du thread th\_receiveFromMon

* 1. **Groupe de threads gestion du robot**
     1. **Diagramme fonctionnel du groupe gestion robot**

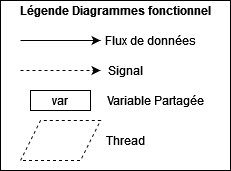
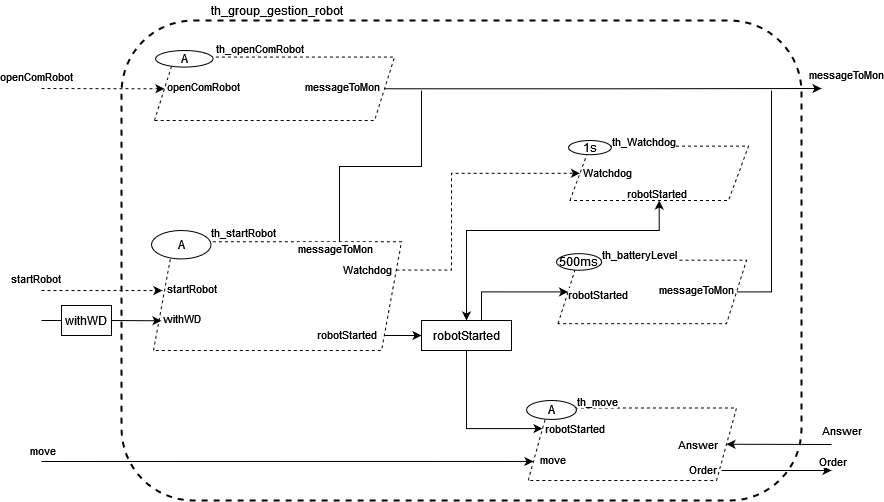


Figure 5: Diagramme fonctionnel du groupe gestion robot

* + 1. **Description des threads du groupe gestion robot**

|  |  |  |
| --- | --- | --- |
| **Nom du thread** | **Rôle** | |
| th\_openComRobot | Ouvre la connexion avec le robot, avec acquittement du succès de l’ouverture | |
| th\_startRobot | Allume le robot avec ou sans Watchdog selon les demandes de l’utilisateur. En cas de demande de WD, débloque le thread th\_Watchdog |
| th\_Watchdog  Fonctionnalité 11 | Une fois lancé, remet à 0 de façon périodique le compteur du WD |
| th\_batteryLevel  Fonctionnalité 13 | Vérifie le niveau de battery et l’envoie au moniteur de façon périodique |
| th\_move | Permet les mouvements du robot |

* + 1. **Diagrammes d’activité du groupe gestion robot**

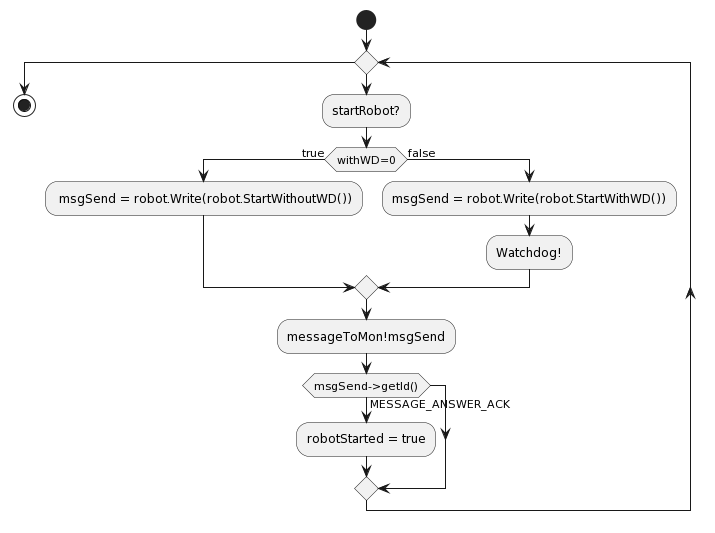


Figure 6: Diagramme d'activité du thread th\_startBobot

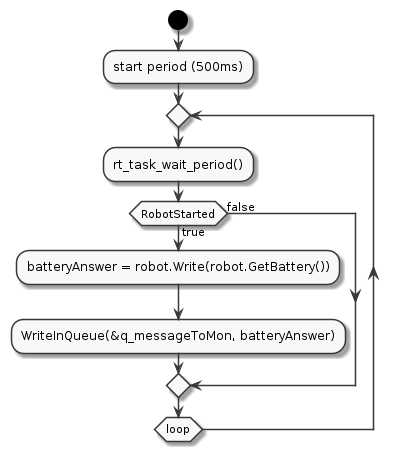


Figure 7: Diagramme d'activité du thread th\_batteryLevel

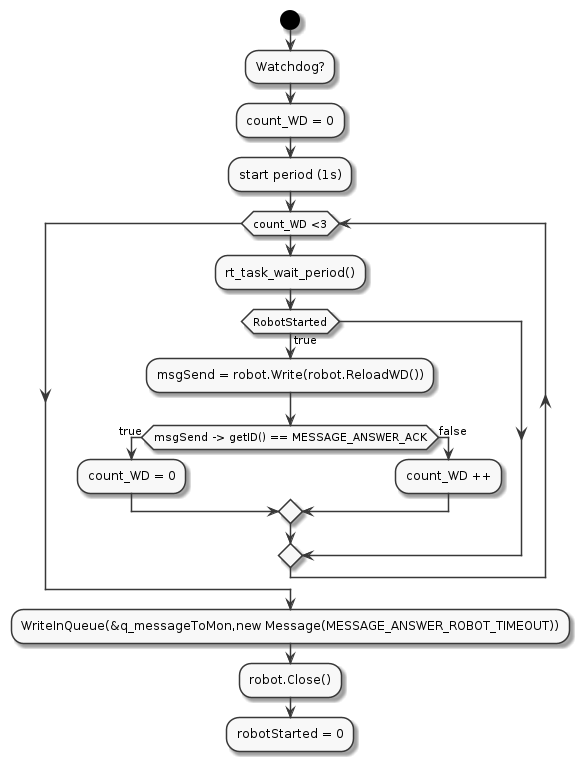


Figure 8: Diagramme d'activité du thread th\_Watchdog

**Lost COM avec le robot ?**

* 1. **Groupe de threads vision**
     1. **Diagramme fonctionnel du groupe vision**

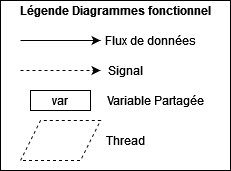
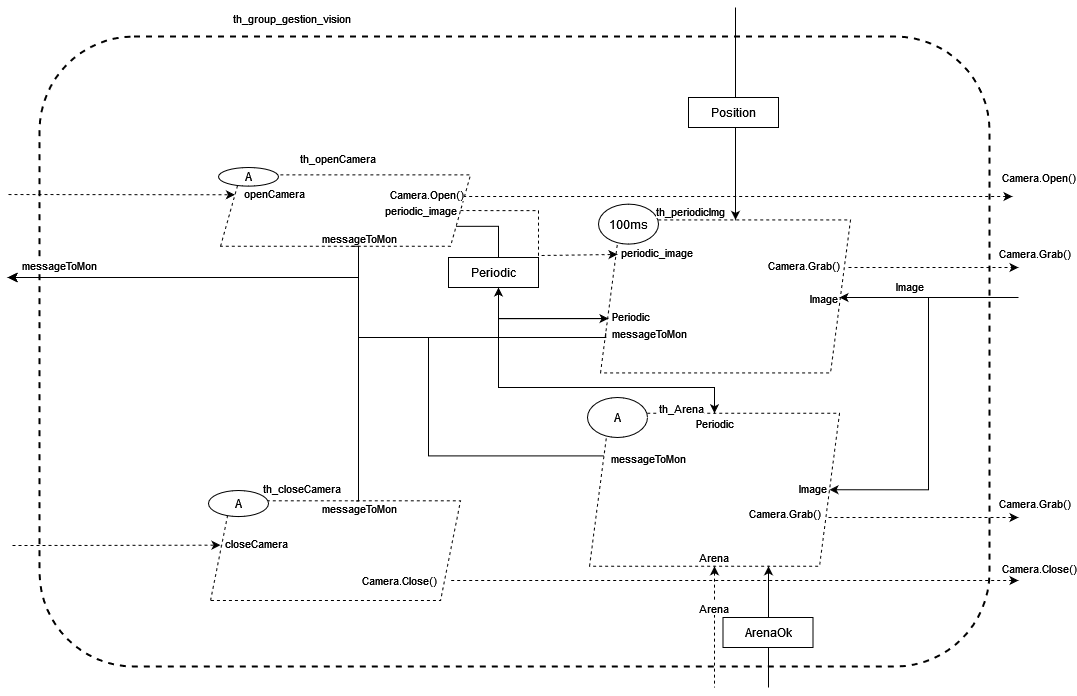


Figure 9: Diagramme fonctionnel du groupe vision

* + 1. **Description des threads du groupe vision**

|  |  |
| --- | --- |
| **Nom du thread** | **Rôle** |
| th\_openCamera  Fonctionnalité 14 | Ouvre la connexion avec la caméra |
| th\_closeCamera  Fonctionnalité 16 | Ferme la connexion avec la caméra, stoppe l’envoi des images |
| th\_periodicImg  Fonctionnalité 15, 18, 19 | Envoie de façon périodique une image au moniteur, avec ou sans position |
| th\_Arena  Fonctionnalité 17 | Stoppe l’envoi périodique d’images, cherche une arène, la dessine s’il la trouve puis demande validation de l’arène à l’utilisateur, et retourner en mode périodique en ajoutant ou non l’arène à l’image |

* + 1. **Diagrammes d’activité du groupe vision**

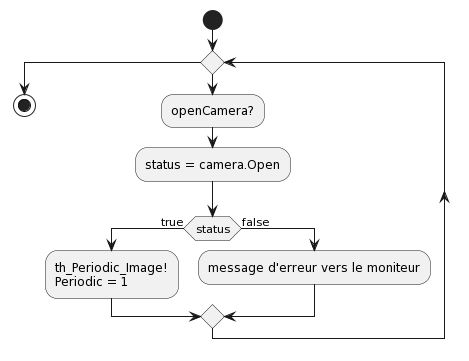


Figure 10: Diagramme d'activité du thread th\_openCamera

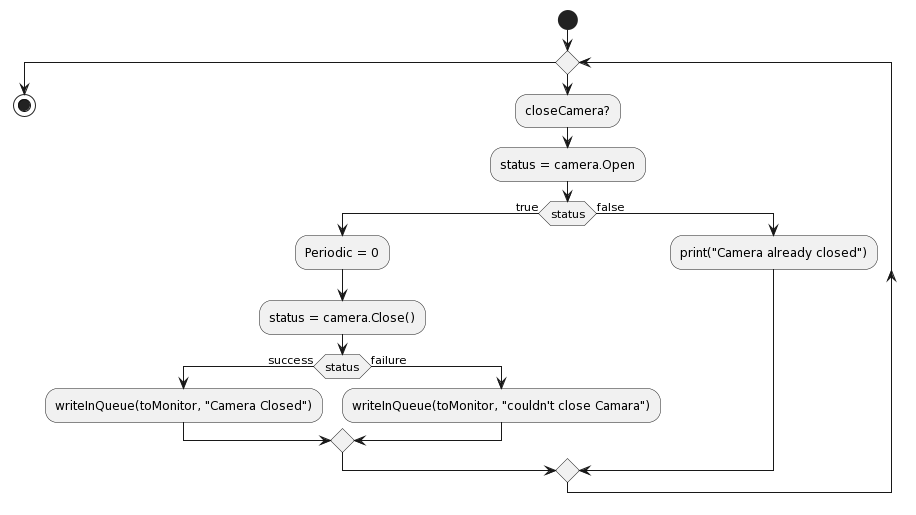


Figure 11: Diagramme d'activité du thread th\_closeCamera

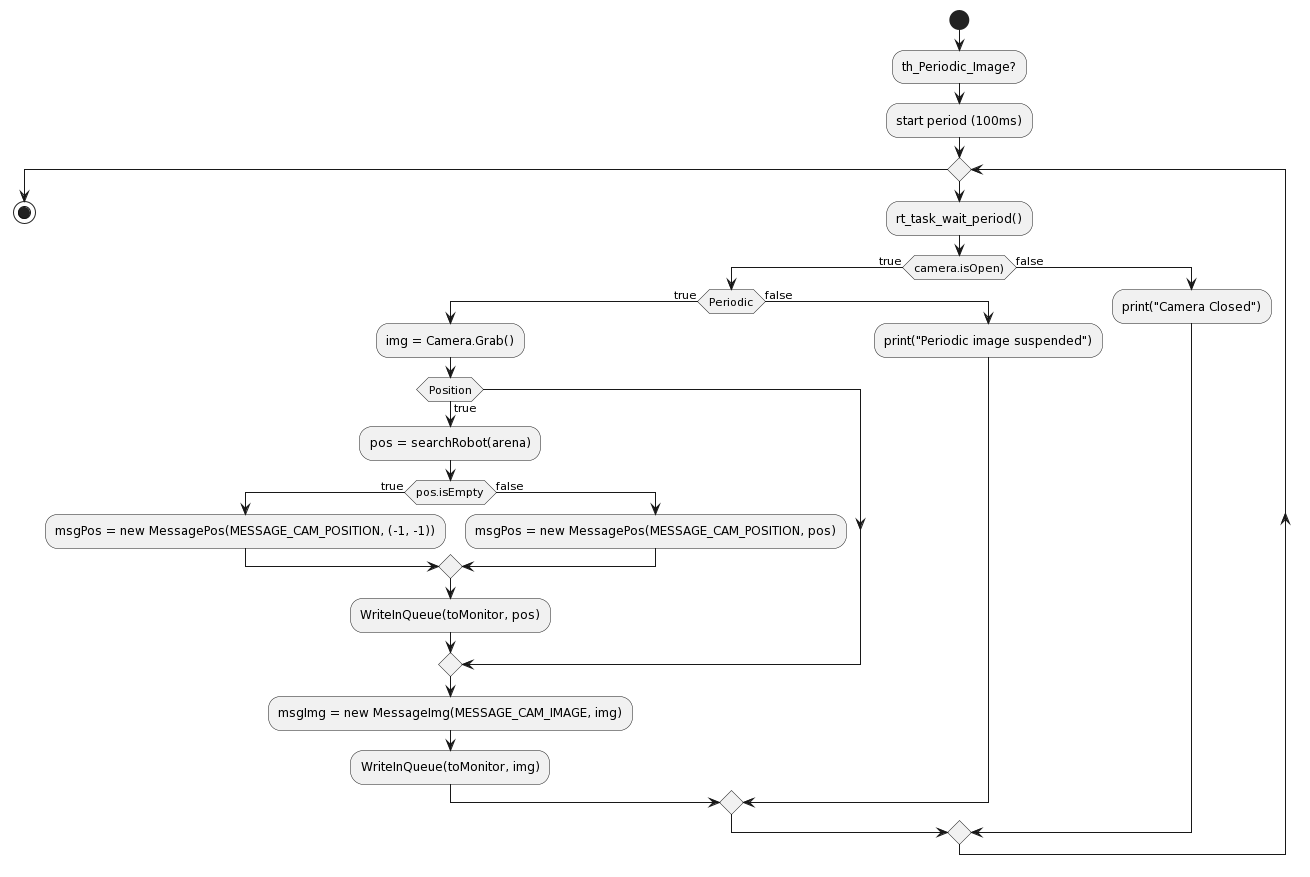


Figure 12: Diagramme d'activité du thread th\_periodicImg

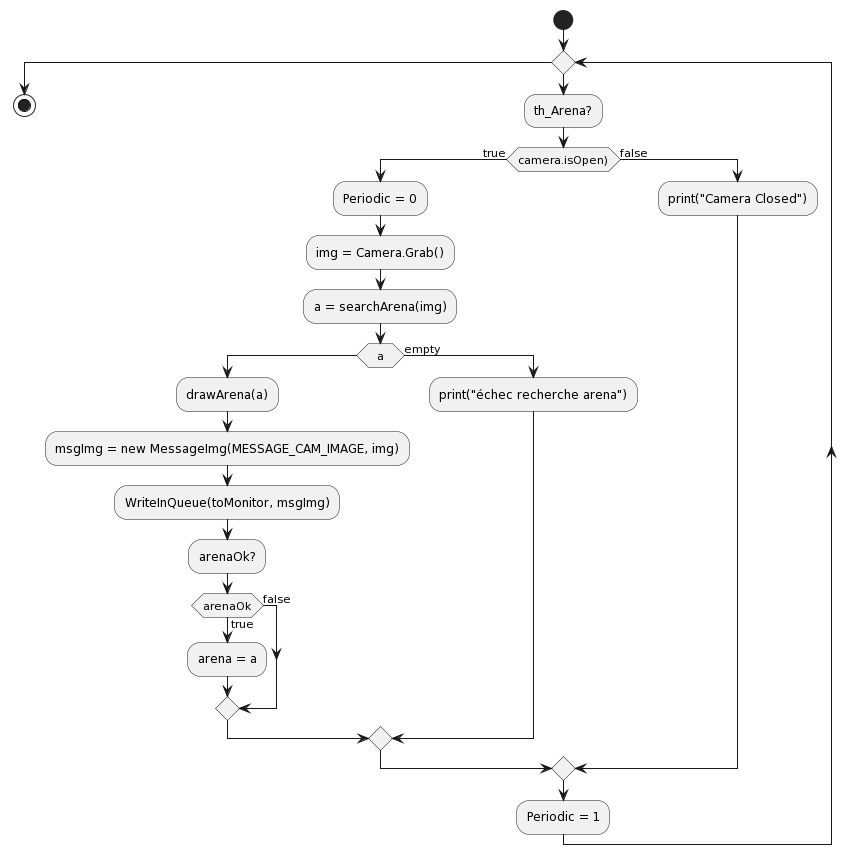


Figure 13: Diagramme d'activité du thread th\_Arena

1. **Transformation AADL vers Xenomai**
   1. **Thread**
      1. **Instanciation et démarrage**

Chaque thread a été implémenté par un RT\_TASK déclarés dans le fichier **tasks.h** (voir Figure 14). La création de la tâche se fait à l’aide du service **rt\_task\_create** (voir Figure 15) et son démarrage à l’aide de **t\_task\_start** (voir Figure 16). Toutes les tâches sont créées dans la méthode **Init** de **tasks.cpp** et démarrées dans la méthode **Run**. La création et le lancement d’une tâche sont surveillés et en cas d’échec nous sortons du programme.



Figure 14: déclaration de la tâche **server** dans **tasks.h**

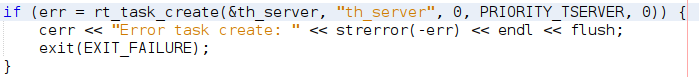


Figure 15: création de la tâche **server** dans **tasks.cpp**

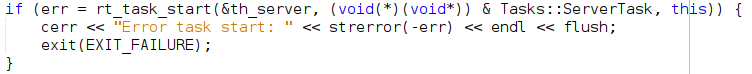


Figure 16: démarrage de la tâche **server** dans **tasks.cpp**

* + 1. **Code à exécuter**

Lors du lancement d’une tâche avec **rt\_task\_start**, on précise la tâche et l’adresse de la routine qui va être exécutée par cette tâche (voir Figure 15).

* + 1. **Niveau de priorités**

Sous Xenomai le niveau de priorité d’un thread peut aller du 0 à 99, avec 0 la priorité la plus faible. Cette priorité est fixée lors de l’appel au service **rt\_task\_create**. Pour faciliter la modification de ces priorités, nous avons utilisé des #define de la forme **PRIORITY\_TTASK** dans le fichier **tasks.cpp** (voir Figure 17).



Figure 17: exemple de définition de priorité pour la tâche **server**

* + 1. **Activation périodique**

Pour rendre périodique l’activation d’un thread, on fait appel aux services **rt\_task\_set\_periodic** pour in Pour rendre périodique l’activation d’un thread, on fait appel aux services **rt\_task\_set\_periodic** pour indiquer la période d’une tâche puis au service **rt\_task\_wait\_period** pour descheduler la tâche en attente du cycle suivant (voir Figure 18). Ces services sont appelés après le lancement de la tâche concernée.

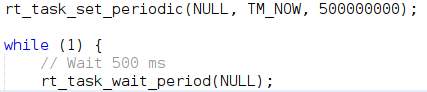


Figure 18: exemple d'activation périodique d'une tâche

* 1. **Donnée partagée**
     1. **Instanciation**

Les données partagées sont définies comme variables globales dans **tasks.h** et instanciées dans **tasks.cpp**.

* + 1. **Accès en lecture et écriture**

Toutes les variables globales partagées sont protégées par des mutex **RT\_MUTEX**, déclarés dans **tasks.h** (voir Figure 19) et instanciés dans **tasks.cpp** de façon analogue aux tâches (voir Figure 20). Pour chaque modification d’une variable globale, le mutex associés est pris via **rt\_mutex\_acquire** puis relâché via **rt\_mutex\_release** (voir Figure 21). Ainsi on évite tout problème de cohérence et d’accès concurrents sur les données partagées.



Figure 19: déclaration d'un mutex dans **tasks.h**

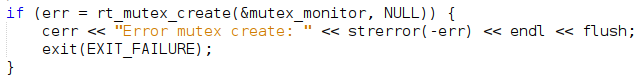


Figure 20: instanciation d'un mutex dans **tasks.cpp**

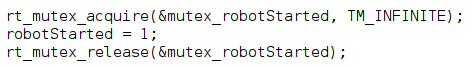


Figure 21: exemple d’utilisation d'un mutex pour protéger la donnée partagée **robotStarted**

* 1. **Port d’événement**

Pour synchroniser nos tâches via des évènements, nous faisons appel à des sémaphores et des queues. Les sémaphores permettent de se synchroniser via des mécanismes d’incrémentation et de décrémentation (p, v et broadcast).

* + 1. **Instanciation**

Les sémaphores sont déclarés dans le **tasks.h** (voir Figure 22). Ils sont instanciés dans **tasks.cpp** de façon analogue à la création des tâches (voir Figure 23).



Figure 22: déclaration d'un sémaphore dans **tasks.h**

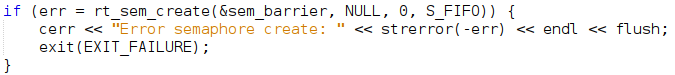


Figure 23: instanciation d'un sémaphore

* + 1. **Envoi d’un événement**

Pour permettre l’activation d’une tâche en l’attente d’un événement, on incrémente le sémaphore correspondant via le service **rt\_sem\_v** (voir Figure 24. Il est aussi possible d’utiliser le service **rt\_sem\_broadcast** si plusieurs tâches se synchronisent via le même événement (voir Figure 25).

****

Figure 24: incrémentation d'un sémaphore

****

Figure 25: broadcast d'un sémaphore

* + 1. **Réception d’un événement**

Une tâche en attente d’un événement est bloquée jusqu’à l’occurrence de cet événement. En pratique, le service **rt\_sem\_p** est bloquant tant que le sémaphore en argument vaut 0. Quand l’événement a lieu et que le sémaphore est modifié (voir 2.3.2), ce service décrémente le sémaphore et le code de la tâche situé après le service peut s’exécuter. Dans une boucle infinie, ce qui est le cas de nos tâches, après exécution du code la tâche se remet ainsi en attente de l’événement à chaque boucle (voir Figure 26).

****

Figure 26: décrémentation d'un sémaphore

* 1. **Ports d’événement-données**

Les ports d’événements-données sont à la fois des ports d’événements (déclenchement d’actions dans des tâches à la suite d’événements, comme dans le 3.2) et de données (des données sont en même temps transmises à ces tâches). De simples sémaphores ne sont ainsi plus adaptés. Nous utilisons donc ici des files de messages.

* + 1. **Instanciation**

Sous Xenomai, nous allons utiliser des RT\_QUEUES, déclarées dans **tasks.h** (voir Figure 27) et instanciées dans **tasks.cpp** via le service **rt\_queue\_create** qui prend en paramètres un pointeur sur la file de messages, son nom, sa taille, le nombre maximum de messages dans la file et un mode (voir Figure 28).



Figure 27: déclaration d'une file de messages dans **tasks.h**

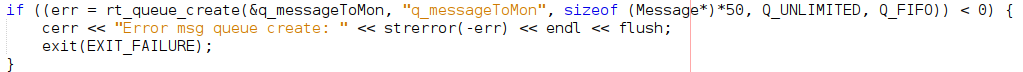


Figure 28: instanciation d'une file de messages dans **tasks.cpp**

* + 1. **Envoi d’une donnée**

Pour envoyer des données et lancer les tâches en attente d’éléments sur ces files, nous utilisons la tâche **WriteInQueue** déjà implémentée. Elle permet de mettre des messages dans une queue (voir Figure 29).



Figure 29: exemple d'utilisation de **WrinteInQueue** pour envoyer un message **msgSend** dans la queue **q\_messageToMon**

* + 1. **Réception d’une donnée**

Les tâches en attente d’une donnée pour poursuivre leur exécution ont un comportement similaire à celui d’une tâche en attente sur un sémaphore. En effet, la tâche **ReadInQueue** utilisée pour lire un message est bloquante tant qu’aucun élément n’est disponible dans la file. Lorsqu’un élément est disponible, la tâche est débloquée et lit la donnée en la retirant de la queue (voir Figure 30).



Figure 30: exemple d'utilisation de **ReadInQueue** pour récupérer un message dans la file et le stocker dans **msg**

1. **Analyse et validation de la conception**

## **Fonctionnalité 3**

**Description** : Tous les messages envoyés depuis le moniteur doivent être réceptionnés par  
le superviseur.  
**Réalisation** : Fonctionnalité déjà implémentée pour certains messages, nous l’avons donc étendue à tous les autres messages utiles pour implémenter les fonctionnalités qui suivent.

## **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, et les différentes connexions doivent être fermées (caméra, robot et  
moniteur).  
**Réalisation** : La tâche receiveFromMon, quand elle reçoit un MESSAGE\_MONITOR\_LOST du read demandé au moniteur, détecte la perte de connexion et ferme toutes les connexions ouvertes.

## **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** : Réalisation partielle dans la tâche receiveFromMon : le robot et les connexions sont bien arrêtées, mais la réinitialisation n’a pas été implémentée pour l’instant. Il faudrait pour cela changer un certain nombre de threads pour les transformer en boucle, et ainsi permettre plusieurs exécutions. En l’état ce n’est pas possible, par exemple avec the thread chargé de l’ouverture du serveur.

## **Fonctionnalité 8**

## **Fonctionnalité 9**

## **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** : Nous avons modifié le démarrage déjà implémenté. Si la variable partagée withWD est à 1, le watchdog est lancé avec son thread associé qui envoie un message au robot toutes les secondes.

## **Fonctionnalité 13**

**Description** : Le niveau de la batterie du robot doit être mis à jour toutes les 500 ms sur le  
moniteur.  
**Réalisation** : Nous utilisons une tâche périodique. Après vérification de l’état du robot (démarré ou non), nous demandons au robot son niveau de batterie et l’envoyons au moniteur.

## **Fonctionnalités 14 à 19**

**Réalisation :** Conçues mais non implémentées.