**Rapport de Projet Temps Réel**

**Vincent BAURES** : implémentation des fonctionnalités et codage des fonctions, réalisation de la vidéo

**Alicia CALMET** : conception des diagrammes AADL et d’activité, experte en rédaction du compte-rendu

**Robin MOINE** : conception du diagramme AADL, implémentation des fonctions de fonctionnalités et codage des fonctions, réalisation de la vidéo

**Loïc ROBERT** : implémentation/intégration des fonctions sur le robot, secrétaire et assistant général en conception des diagrammes d’activité, rédaction du compte rendu.

Il est à noter que la relecture du code, du rapport, des diagrammes ainsi que la vérification de la vidéo ont été réalisées par l’ensemble des membres du groupe.

**Lien vers le git :**

Table des matières

[Table des matières 1](#_Toc68627047)

[1 Conception 3](#_Toc68627048)

[1.1 Diagramme fonctionnel général 3](#_Toc68627049)

[1.2 Groupe de threads gestion du moniteur 3](#_Toc68627050)

[1.2.1 Diagramme fonctionnel du groupe gestion du moniteur 3](#_Toc68627051)

[1.2.2 Diagrammes d’activité du groupe gestion du moniteur 3](#_Toc68627052)

[1.3 Groupe de threads gestion du robot 4](#_Toc68627053)

[1.3.1 Diagramme fonctionnel du groupe gestion 4](#_Toc68627054)

[1.3.2 Diagrammes d’activité du groupe 4](#_Toc68627055)

[6](#_Toc68627056)

[8](#_Toc68627057)

[1.4 Groupe de threads vision 9](#_Toc68627058)

[1.4.1 Diagramme fonctionnel du groupe vision 9](#_Toc68627059)

[Nom du Thread 9](#_Toc68627060)

[Rôle 9](#_Toc68627061)

[Périodicité 9](#_Toc68627062)

[Th\_Camera 9](#_Toc68627063)

[Permet de gérer toutes les fonctionnalités qui sont 9](#_Toc68627064)

[liées à la caméra : son ouverture/fermeture, la 9](#_Toc68627065)

[reconnaissance de l’arène, la capture de l’image, la position,... 9](#_Toc68627066)

[Périodicité de 100 ms 9](#_Toc68627067)

[2 Transformation AADL vers Xenomai 10](#_Toc68627068)

[2.1 Thread 10](#_Toc68627069)

[2.1.1 Instanciation et démarrage 10](#_Toc68627070)

[2.1.2 Code à exécuter 10](#_Toc68627071)

[2.1.3 Niveau de priorités 10](#_Toc68627072)

[2.1.4 Activation périodique 10](#_Toc68627073)

[2.2 Donnée partagé 11](#_Toc68627074)

[2.2.1 Instanciation Les données partagées ont été implémentés grâce à l’utilisation de mutex. 11](#_Toc68627075)

[2.2.2 Accès en lecture et écriture 11](#_Toc68627076)

[2.3 Port d’événement 12](#_Toc68627077)

[2.3.1 Instanciation Les port d’évènements sont gérés par le biais de sémaphores. Sous Xenomai, les sémaphores peuvent être instanciés grâce à l’instruction : rt\_sem\_create(&sem\_barrier, NULL, 0, S\_FIFO); sem\_barrier est une variable de type RT\_SEM. 12](#_Toc68627078)

[2.3.2 Envoi d’un événement Pour envoyer un évènement (par exemple la libération d’un sémaphore), on utilise l’instruction : rt\_sem\_broadcast(&sem\_barrier); 2.3.3 Réception d’un événement Grâce à l’instruction présentée en 2.3.2, tout thread qui attend après la libération de ce sémaphore est notifié que le sémaphore est libéré (évènement), et peut continuer de s’exécuter. 12](#_Toc68627079)

[2.4 Ports d’événement-données 12](#_Toc68627080)

[2.4.1 Instanciation Les événement-données sont ici représentés sous forme de pile. Elles sont de type RT\_QUEUE et ont une taille maximale, représentée par la constante MSG\_QUEUE\_SIZE. Pour instancier une pile, on utilise l’instruction : rt\_queue\_create(&q\_example, ‘’example\_queue’’, sizeof(Message\*) \*MSG\_QUEUE\_SIZE, Q\_UNLIMITED, Q\_FIFO) ; 12](#_Toc68627081)

[2.4.3 Réception d’une donnée 12](#_Toc68627082)

[3 Analyse et validation de la conception 13](#_Toc68627083)

[4 Bibliographie 13](#_Toc68627084)

# 1 Conception

## Diagramme fonctionnel général

## Groupe de threads gestion du moniteur

### Diagramme fonctionnel du groupe gestion du moniteur

### Diagrammes d’activité du groupe gestion du moniteur

|  |  |  |
| --- | --- | --- |
| **Nom du Thread** | **Rôle** | **Périodicité** |
| th\_receiveFromMon | Permet de recevoir des messages depuis le moniteur | Non |
| th\_SendTo Mon | Permet d’envoyer un message au moniteur | Contrainte de temps de 10 ms |
| th\_Server | Permet la connexion au serveur | Non |

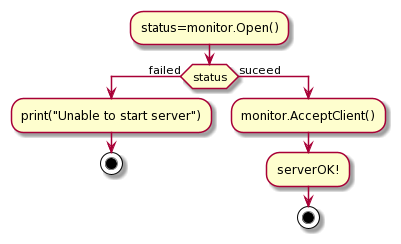


Figure 1: Diagramme d'activité du thread th\_server

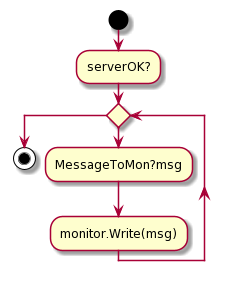


Figure 2: Diagramme d’activité du thread th\_sendToMon

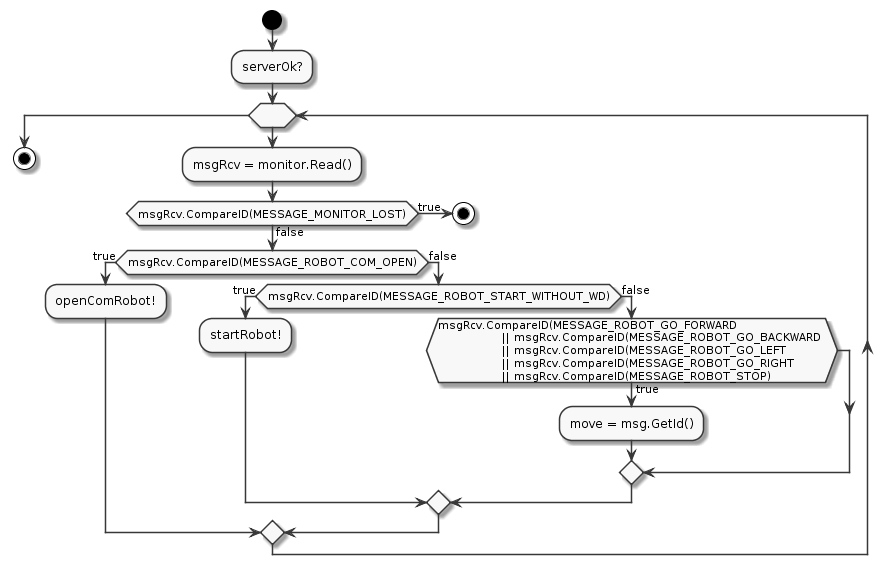
Il est a noté que nos diagrammes d’activité ne sont pas exhaustifs. Afin de ne pas surcharger les digrammes, nous n’avons donc pas représenter les nœuds d’exécution des sémaphores et des mutex.

Figure 3:Diagramme d’activité du thread th\_receiveFromMon

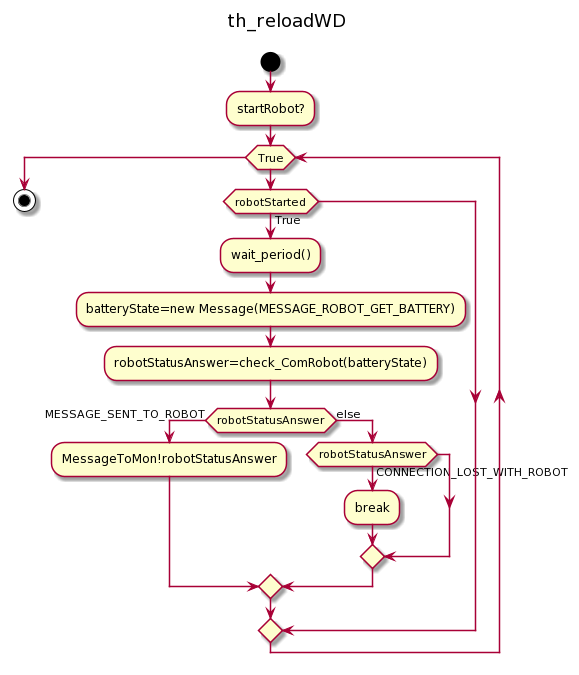
## 1.3 Groupe de threads gestion du robot

### 1.3.1 Diagramme fonctionnel du groupe gestion

### 1.3.2 Diagrammes d’activité du groupe

|  |  |  |
| --- | --- | --- |
| **Nom du Thread** | **Rôle** | **Périodicité** |
| Th\_battery | Correspond fonctionnalité 13. Permet de retourner le niveau de batterie du robot toutes les 500 ms au moniteur | 500ms |
| Th\_move | Correspond à la fonctionnalité 12. Permet le déplacement du robot : un message de déplacement est envoyé du moniteur au superviseur puis du superviseur au robot | Contrainte de temps de 100 ms |
| Th\_startRobot | Ce thread est toujours en attente d’une demande de mode de démarrage pour le robot. Dès qu’il reçoit une requête de démarrage sans watchdog il permet d’allumer le robot, s’il reçoit une requête avec watchdog, il démarre en mode watchdog. | Non |
| Th\_OpenComRobot | Permet d’ouvrir la communication entre le robot et le superviseur | Non |
| Th\_ReloadWD | Permet de confirmer la présence du watchdog au superviseur | Contrainte de temps de |

Figure 4: Digramme activité du thread th\_battery



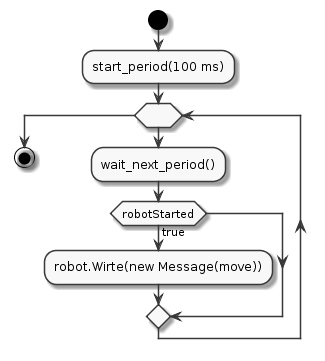


Figure 5: Diagramme d'activité du thread th\_move

## 

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

## 

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

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

## 1.4 Groupe de threads vision

1.4.1 Diagramme fonctionnel du groupe vision   
1.4.2 Diagrammes d’activité du groupe vision

|  |  |  |
| --- | --- | --- |
| Nom du Thread | Rôle | Périodicité |
| Th\_Camera | Permet de gérer toutes les fonctionnalités qui sont  liées à la caméra : son ouverture/fermeture, la  reconnaissance de l’arène, la capture de l’image, la position… | Périodicité de 100 ms |

# Transformation AADL vers Xenomai

## 2.1 Thread

### 2.1.1 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 créés dans la méthode Tasks ::Init() du fichier tasks.cpp puis démarrées dans la méthode Tasks ::Run().

Prenons l’exemple du lancement du thread th\_server qui est l’un des premiers utilisés :  
- sa déclaration est faite ligne 85 dans le fichier tasks.h : *RT\_TASK th\_server*  
- sa création ligne 136 de tasks.cpp lors de l’appel de *rt\_task\_create(&th\_server, "th\_server", 0, PRIORITY\_TSERVER, 0)*  
- son démarrage ligne 196 avec *rt\_task\_start(&th\_server, (void(\*)(void\*)) & Tasks::ServerTask, this)*

### 2.1.2 Code à exécuter

Chaque traitement à exécuter a été implémenté par une fonction dans tasks.cpp, et le lien avec le thread se fait à l’aide des paramètres passées au service *rt\_task\_start*.

Ainsi, pour le thread th\_server, son code à exécuter se trouve dans la fonction *Tasks::ServerTask* à la ligne 256. Cette fonction est appelée en deuxième argument lors du démarrage du thread : *rt\_task\_start(&th\_server, (void(\*)(void\*)) & Tasks::ServerTask, this).*

### 2.1.3 Niveau de priorités

Les priorités sont définies par un numéro allant de 0 à 99 (99 étant la priorité la plus élevée). L’attribution d’une priorité d’un thread se fait dans le fichier tasks.cpp en utilisant un *#define* puis en allouant la priorité voulue. Cette priorité est passée en paramètre lors de la création du thread dans le service *rt\_task\_create.*

Par exemple reprenons la création et la gestion du thread th\_server. Nous lui définissons une priorité de 30 à la ligne 30 du fichier tasks.cpp grâce à la déclaration suivante : *#define PRIORITY\_TSERVER 30*. Ensuite, cette priorité est allouée lors de la création du thread *rt\_task\_create(&th\_server, "th\_server", 0, PRIORITY\_TSERVER, 0),* où l’avant dernier paramètre est la priorité définie ligne 30.

Pour choisir le numéro de priorité de chaque thread, nous avons comparé l’importance de leur exécution. Ainsi, le thread th\_server est plus prioritaire aux autres threads car sans lui nous ne pouvons pas établir une connexion avec le monitor et donc communiquer avec le robot.

### 2.1.4 Activation périodique

Pour qu’un thread s’exécute de manière périodique, il est nécessaire d’utiliser le service *rt\_task\_set\_periodic* fourni par Xenomai. Les paramètres passés sont alors le thread en question, le délai initial et la valeur de la période en ns. Lors du traitement périodique à réaliser, on utilise *rt\_task\_wait\_period* pour faire attendre la tâche en question jusqu’au prochain cycle.

Prenons l’exemple du thread th\_battery qui doit s’exécuter toutes les 500 ms. Le thread est rendu périodique à la ligne *683 rt\_task\_set\_periodic(NULL, TM\_NOW, 500000000)*. La ligne 690, *rt\_task\_wait\_period(NULL)*, permet d’attendre jusqu’à la période suivante.

## 2.2 Donnée partagé

### 2.2.1 Instanciation Les données partagées ont été implémentés par l’utilisation de mutex afin de garantir leur bon partage entre processus.

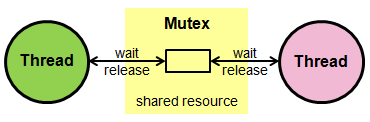


Figure 9: Schéma d’une ressource partagée entre deux threads via l’utilisation d’un mutex

Nous avons donc créé pour chaque variable partagée un mutex associé dans le fichier tasks.h. Cette variable est de type *RT\_MUTEX*. Par exemple, la variable robotStarted est créée à la ligne 77 *inti robotStarted = 0* puis son mutex est créé à la ligne 101 *RT\_MUTEX mutex\_robotStarted*.  
Si nous considérons la création d’un mutex pour une variable partagée ayant pour nom ‘*exemple*’, il faudrait alors instancier son mutex associé, grâce à l’instruction : *rt\_mutex\_create(&mutex\_exemple, NULL)* dans le fichier tasks.cpp*; p*our notre variable robotStarted, cela donne donc : *rt\_mutex\_create(&mutex\_robotStarted, NULL)* à la ligne 101.

### 2.2.2 Accès en lecture et écriture

Afin de garantir la protection des données, les accès en lecture et en écriture se font à l’aide des fonctions *rt\_mutex\_acquire* à l’acquisition de la ressource et *rt\_mutex\_release* au rendu de la variable. Il est possible de spécifier combien de temps on souhaite attendre la libération du mutex, et dans notre code, la plupart du temps, ce temps est considéré comme infini grâce à l’attribut TM\_INFINITE.  
En considérant que notre variable *exemple* (voir 2.2.1), son accès en écriture serait effectué grâce aux instructions suivantes :  
  
*rt\_mutex\_acquire(&mutex\_exemple, TM\_INFINITE);  
copie = exemple;  
exemple = 3;  
rt\_mutex\_release(&mutex\_exemple);*

Plus concrètement, si nous continuons notre exemple avec la variable partagée RobotStarted :

*rt\_mutex\_acquire(&mutex\_robotStarted, TM\_INFINITE);  
robotStarted=0;  
rt\_mutex\_release(&mutex\_robotStarted);*

Ce morceau de code signifie que la connexion avec le robot a été perdu (robotStarted=0).

## 2.3 Port d’événement

### 2.3.1 Instanciation

Les ports d’évènements sont gérés par le biais de sémaphores. Sous Xenomai, les sémaphores peuvent être instanciés grâce à l’instruction :   
rt\_sem\_create(&sem\_barrier, NULL, 0, S\_FIFO);  
sem\_barrier est une variable de type RT\_SEM.

### 2.3.2 Envoi d’un événement

Pour envoyer un évènement (par exemple la libération d’un sémaphore), on utilise l’instruction :  
rt\_sem\_broadcast(&sem\_barrier);

2.3.3 Réception d’un événement   
  
Grâce à l’instruction présentée en 2.3.2, tout thread qui attend après la libération de ce sémaphore est notifié que le sémaphore est libéré (évènement), et peut continuer de s’exécuter.

## 2.4 Ports d’événement-données

### 2.4.1 Instanciation

Les événement-données sont ici représentés sous forme de pile. Elles sont de type RT\_QUEUE et ont une taille maximale, représentée par la constante MSG\_QUEUE\_SIZE.  
Pour instancier une pile, on utilise l’instruction :   
rt\_queue\_create(&q\_example, ‘’example\_queue’’, sizeof(Message\*) \*MSG\_QUEUE\_SIZE, Q\_UNLIMITED, Q\_FIFO) ;

2.4.2 Envoi d’une donnée   
  
Pour envoyer une donnée par la queue, il faut utiliser l’instruction :

rt\_queue\_write(&q\_example, &msg, sizeof(&msg), Q\_NORMAL) ;

Il est à noté que des wrappers pour les fonctions d’envoie et de réception ont été fournis avec le sujet. Ainsi, les fonctions Tasks::WriteInQueue et Tasks::ReadInQueue sont utilisés directement dans notre code, gérant les erreurs à notre place et fournissant un interface simplifié avec la pile.

### 2.4.3 Réception d’une donnée

Pour lire une donnée dans la queue, on utilise l’instruction :  
rt\_queue\_read(queue, &msg, sizeof (&msg), TM\_INFINITE))

Les port d’évènements sont gérés par le biais de sémaphores. Sous Xenomai, les sémaphores peuvent être instanciés grâce à l’instruction :   
rt\_sem\_create(&sem\_barrier, NULL, 0, S\_FIFO);  
sem\_barrier est une variable de type RT\_SEM.

# 3 Analyse et validation de la conception

Afin démontrer que les fonctionnalités que nous avons implémentées s’exécutent correctement, nous avons réalisé une vidéo récapitulative. Tous les membres du groupe se sont exprimés dans cette vidéo.

Pour envoyer un évènement (par exemple la libération d’un sémaphore), on utilise l’instruction :  
rt\_sem\_broadcast(&sem\_barrier);

# Bibliographie

Figure 9 : <https://lh3.googleusercontent.com/proxy/3O9h9fuoDOOx_h4BX1ViVSXFShKGs6Hdxkst0-QQhG60m-r9J5oy9HOejxLPQmpmGL6HxezNjiCESkARgJa7K6IUlWbf2N_LZ-YNsd5f1MIRaDRqJLr_m5uDWOCE>