Rapport de projet De Stijl

version 20 janvier 2017

ABDELMOUMEN Oussama (Conception, code)
GRASA Guillaume (Conception, code)
HELLO Anouk (Conception, rédaction du compte-rendu, un peu de code)
PROUVOST Chloé (Conception, rédaction du compte-rendu, un peu de code)

— Ce qu'il faut faire —

Remplacez tous les textes en bleu et supprimer les textes en rouge

Le rapport est à rendre en pdf et à envoyer par mail à votre encadrant de TP au plus tard le 20 janvier 2017.

Vous devez aussi rendre votre code (uniquement les fichiers que vous avez écrits ou modifiés) sous la forme d'une archive (zip ou tar).

Vous pouvez utiliser word ou un autre logiciel d'édition pour rédiger ce rapport, par contre vous devez **obligatoirement** respecter la structure décrite ici.

Critères d'évaluation :
- Qualité rédactionnelle,
- Exhaustivité et justesse des règles de codage,
- Qualité de la conception (clarté, respect de la syntaxe, exhaustivité, justesse),
- Qualité des explications,
- Respect des règles dans la production du code

Compétences évaluées :
- rédaction et communication sur un dossier de conception
- concevoir une application concurrente temps réel
- analyser une conception
- passer d'un modèle de conception à une implémentation
- écriture de code C et utilisation de primitives au niveau système

1 Conception

1.1 Diagramme fonctionnel général

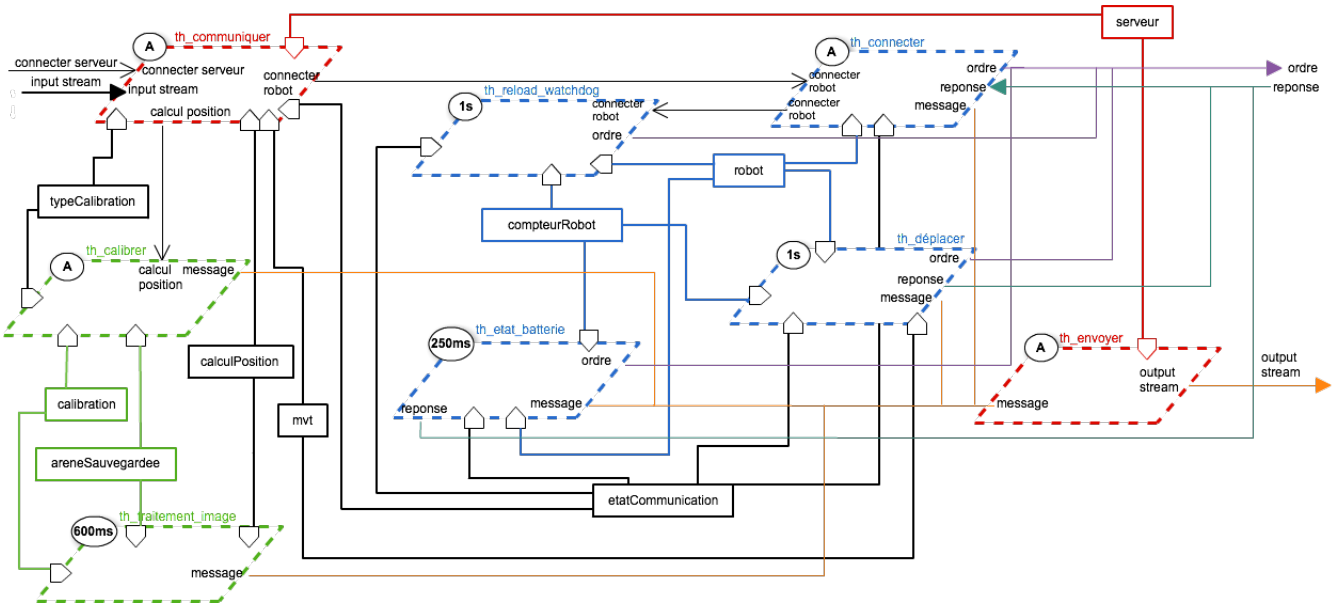

Fig. 1: Diagramme fonctionnel du système

1.2 Groupe de threads gestion du moniteur

1.2.1 Diagramme fonctionnel du groupe gestion du moniteur

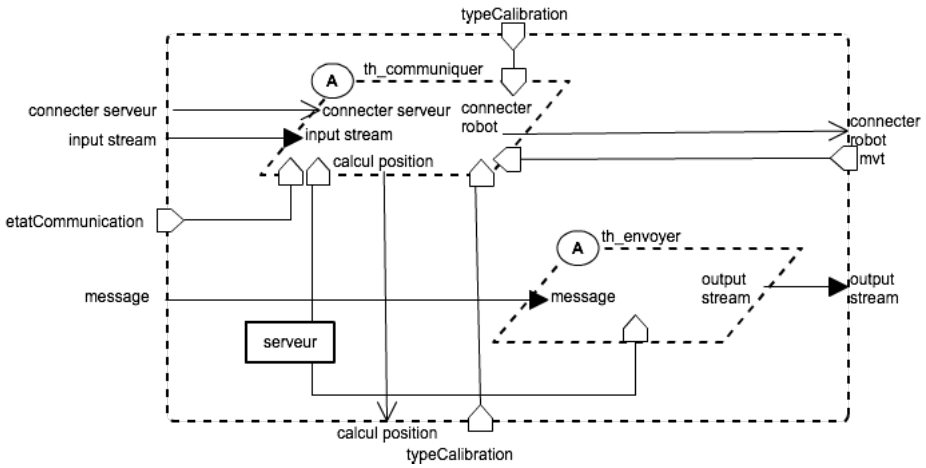

Fig. 2: Diagramme fonctionnel du groupe de threads gestion du moniteur

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
thCommuniquer	Prend en charge les messages entrants depuis le moniteur	50
thEnvoyer	Envoi l'ensemble des messages du superviseur au moniteur	55

1.2.3 Diagrammes d'activité du groupe gestion du moniteur

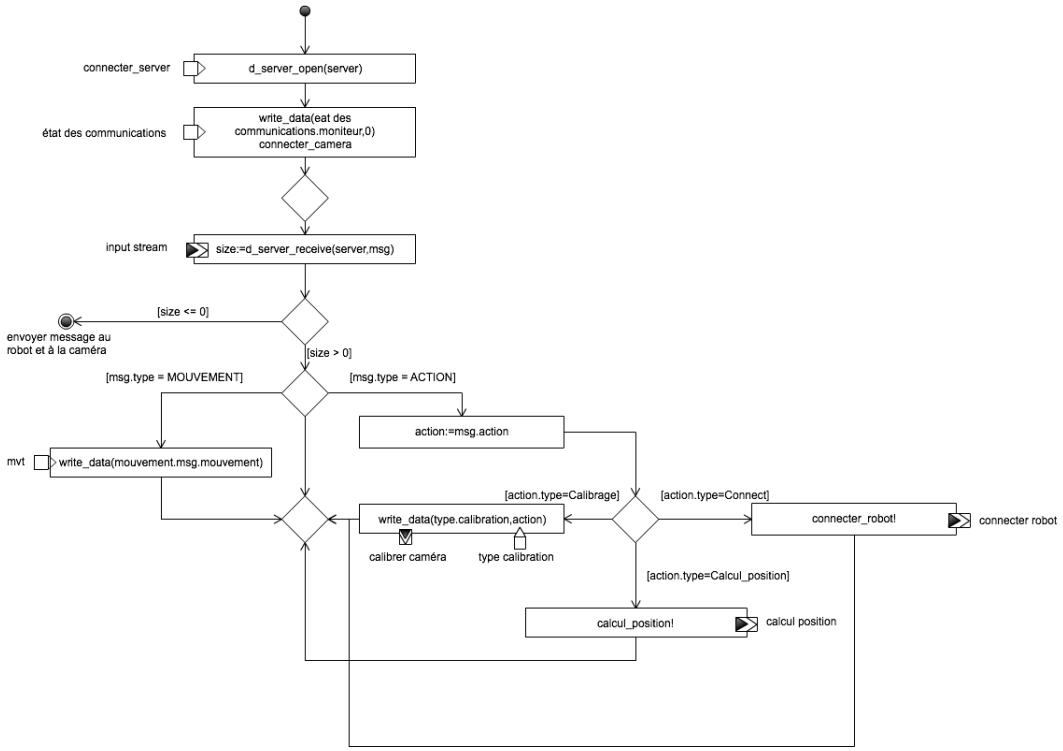

Fig. 3: Diagramme d'activité du thread `th_communiquer`

thCommuniquer : On attend que le robot soit connecté. Lorsque c'est fait si l'état de communication avec le robot n'est pas bon, on se remet en attente. Sinon on envoie l'ordre de rechargement du watchdog.
Comme il y a une communication avec le robot on doit recevoir une réponse de celui-ci. Si l'on ne reçoit rien un compteur global s'incrémente, lorsque ce compteur est à 3 on considère la connexion perdue.

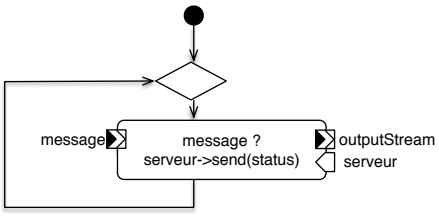

Fig. 4: Diagramme d'activité du thread `th_envoyer`

1.3 Groupe de threads gestion du robot

1.3.1 Diagramme fonctionnel du groupe gestion robot

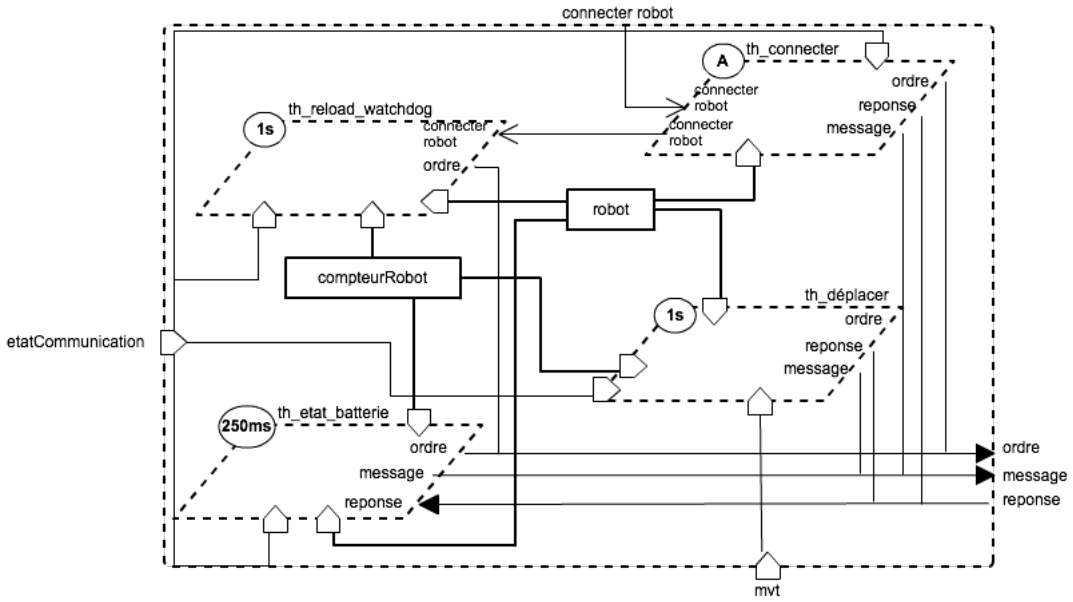

Fig. 5: Diagramme fonctionnel du groupe de threads gestion du robot

1.3.2 Description des threads du groupe gestion robot

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

Nom du thread	Rôle	Priorité
thConnecter	Ouvre la communication avec le robot	50
thReloadWatchdog	Surveille la perte de communication entre le robot et le superviseur	45
thDeplacer	Envoi des ordres de déplacement au robot	40
thEtatBatterie	Surveille l'état de la batterie du robot et l'envoi au moniteur via le thread thEnvoyer	10

1.3.3 Diagrammes d'activité du groupe robot

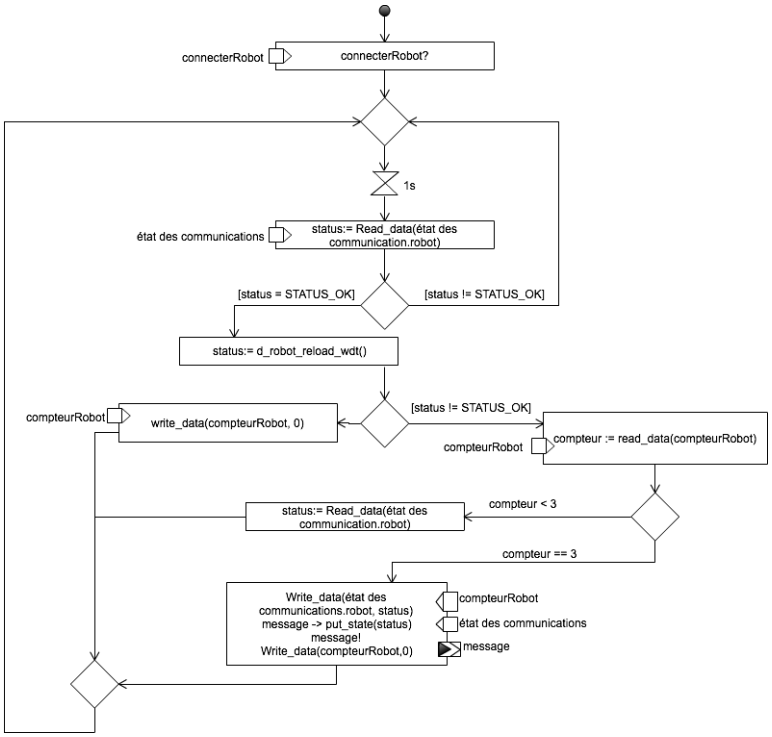

Fig. 6: Diagramme d'activité du thread thReloadWatchdog

thReloadWatchdog : On attend que le robot soit connecté. Lorsque c'est fait si l'état de communication avec le robot n'est pas bon, on se remet en attente. Sinon on envoie l'ordre de rechargement du watchdog.
Comme il y a une communication avec le robot on doit recevoir une réponse de celui-ci. Si

l'on ne reçoit rien un compteur global s'incrémente, lorsque ce compteur est à 3 on considère la connexion perdue.

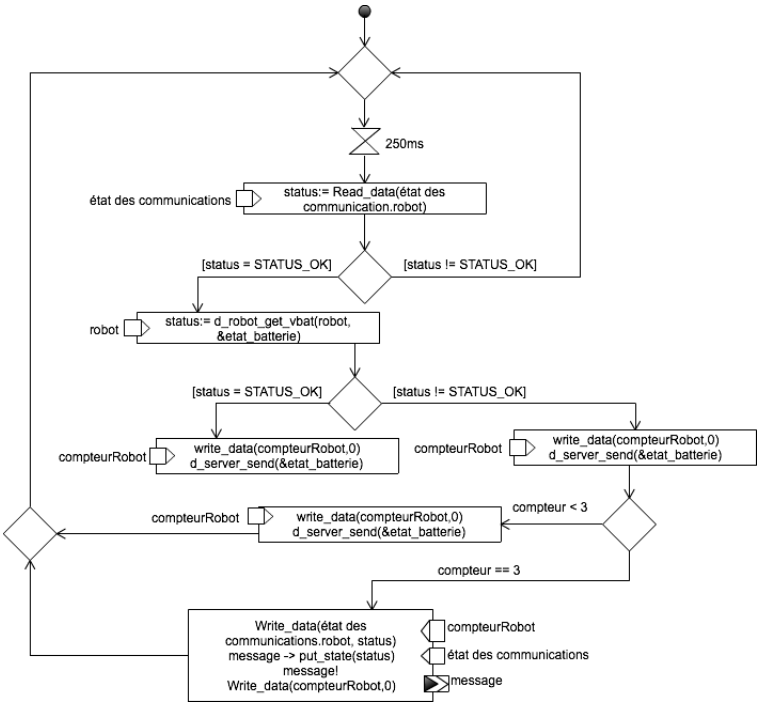

Fig. 7: Diagramme d'activité du thread thEtatBatterie

thEtatBatterie : Toutes les 250 ms on regarde l'etat de la batterie. Si l'etat des communication n'est pas bon on se remet en attente. Sinon on va aller récupérer l'information sur l'etat de la batterie du robot. De même que dans le thread précédent il y a une communication avec le robot. Si celui ci ne répond le compteur global s'incrémente. Sinon on envoie au moniteur l'état de la batterie.

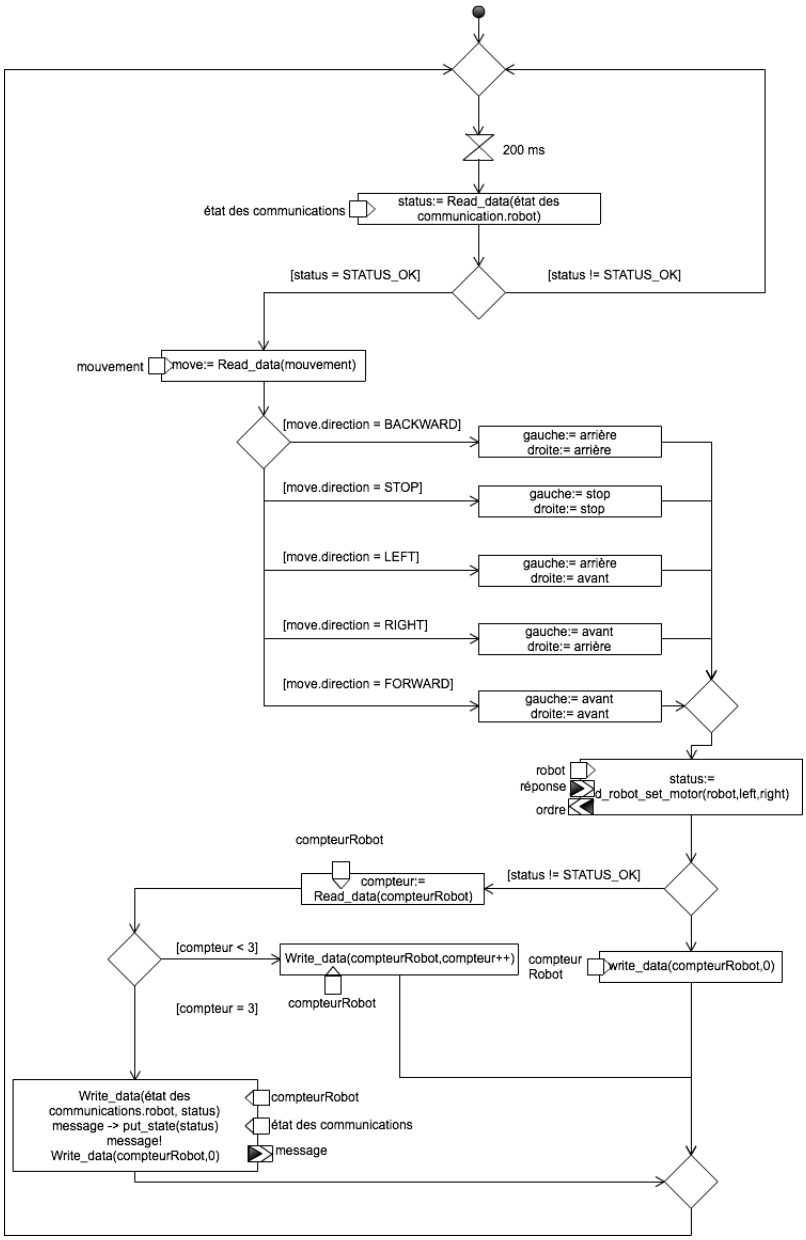

Fig. 8: Diagramme d'activité du thread thDeplacer

thDeplacer : Lorsque l'état de la communication avec le robot est ok, le superviseur récupère du moniteur l'ordre de mouvement. Il traduit ensuite ces ordres et les envoie au robot. De même que dans le thread précédent il y a une communication avec le robot. Si celui ci ne répond le compteur global s'incrémente. Lorsque ce compteur atteint 3 la connexion est considérée perdue.

1.4 Groupe de threads vision

1.4.1 Diagramme fonctionnel du groupe vision

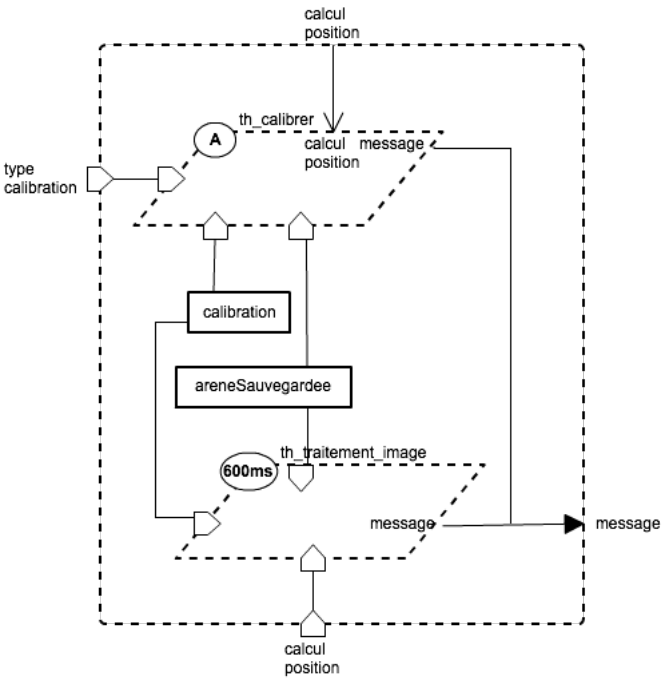

Fig. 9: Diagramme fonctionnel du groupe de threads gestion de vision

1.4.2 Description des threads du groupe vision

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

Nom du thread	Rôle	Priorité
thTraitementImage	Capture une image, si l'utilisateur souhaite la position du robot, il la dessine sur cette image, compresse l'image et l'envoi en message. Si il ne veux pas la position il compresse l'image puis l'envoi en message. Il ne peut pas s'exécuter si il y a une calibration.	30
thCalibration	Détecte l'arène depuis la webcam. L'image de l'arène apparait sur le moniteur. Si cette calibration ne convient pas à l'utilisateur, il peut redemander une détection. Si elle lui convient, il la valide sur le moniteur.	35

1.4.3 Diagrammes d'activité du groupe vision

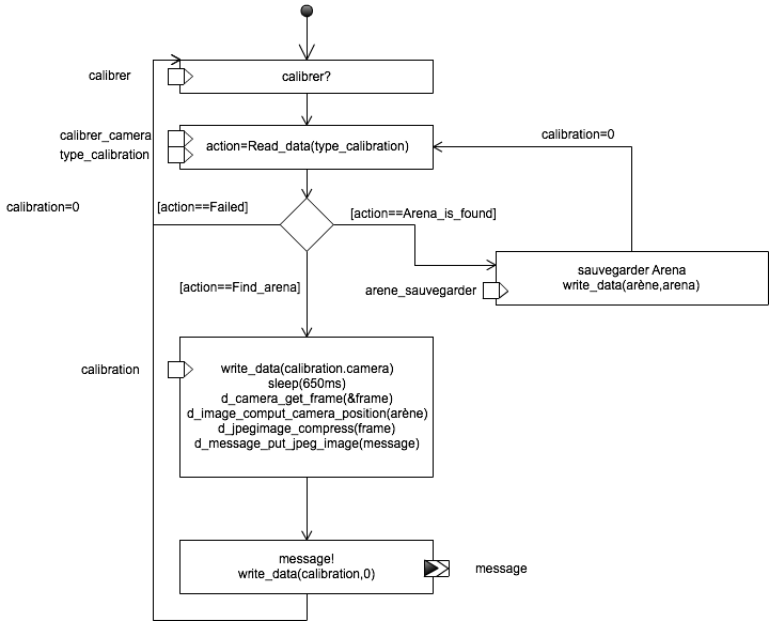

Fig. 10: Diagramme d'activité du thread `thCalibration`

thCalibration : On attend que la calibration de l'arène soit réalisée. Si la calibration n'a pas marché on se remet en attente d'une nouvelle calibration. Si l'arène est trouvée on sauvegarde cette calibration sinon si la calibration a fonctionné mais que l'arène n'est pas encore détectée, on cherche l'arène et on envoi au moniteur la calibration.

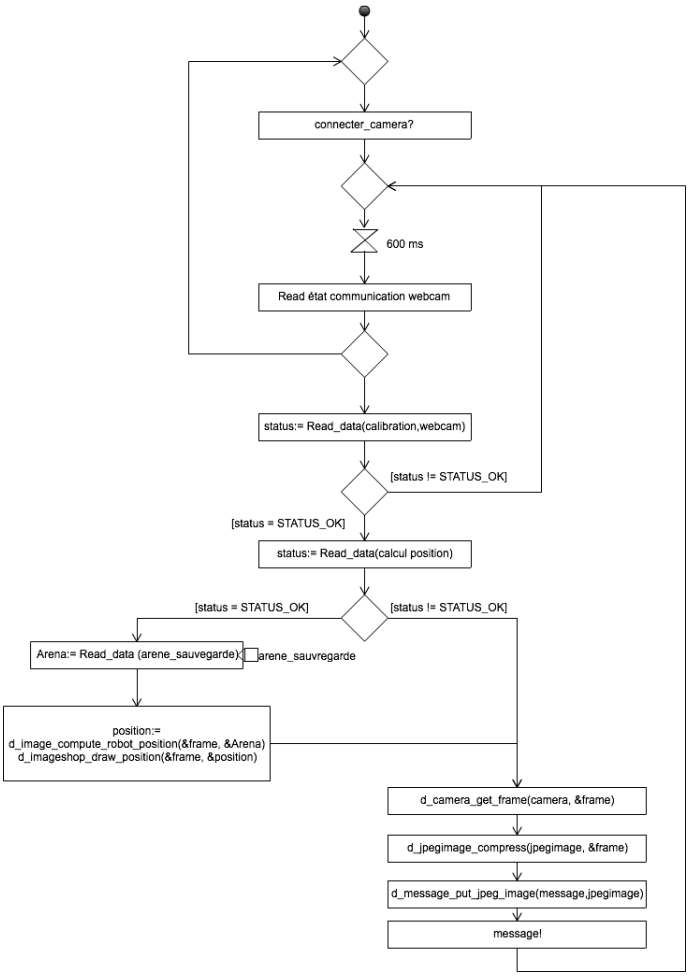

Fig. 11: Diagramme d'activité du thread `thTraitementImage`

thTraitementImage : Lorsque la camera est connectée on regarde l'état de la communication avec elle. On récupère les données de la calibration et les données de la webcam. On regarde ensuite si l'utilisateur demande à trouver la position du robot dans l'arène. Si oui on cherche la position du robot dans l'arène que l'on a sauvegardée puis envoie la position du robot au moniteur et on l'affiche sur l'image.

2 Analyse et validation de la conception

Pour les trois exigences suivantes montrer en quoi votre conception permet d'y répondre.

Exemple de ce qui est attendu

Voici un exemple avec pour exigence : « une fois la communication établie avec le robot, les ordre de mouvement sélectionnés par l'utilisateur sur le moniteur sont transmis au robot. »

Il faut justifier à travers la conception que cette exigence est bien prise en considération et estimer le temps maximum que peut prendre la transmission d'un ordre.

Pour illustrer cela, je m'appuie sur la première conception du système qui est faite dans le document « Dossier de conception ».

Justification : une fois la communication établie, les ordres de l'utilisateur sont reçus par le superviseur via le port `connecter_serveur` du thread `th_communiquer` (attente de la fonction `d_server_receive` sur la figure 4).

Le diagramme d'activité du thread (fig. 4) montre qu'à la réception d'un message de type MOUVEMENT la valeur qu'il porte est écrite dans la donnée partagée `mouvement`.

Cette donnée est lue périodiquement par le thread `th_deplacer`. Son diagramme d'activité (fig. 7) montre qu'en fonction de la valeur de la donnée `mouvement` les variables internes à `th_deplacer` nommées `gauche` et `droite` sont mises à jour pour ensuite être transmises au robot via l'appel à la fonction `d_robot_set_motor`.

Estimation du temps de traitement : la figure 13 montre la séquence d'exécution la plus désavantageuse pour la prise en compte d'un ordre de mouvement. L'utilisateur commence par sélectionner son ordre sur l'interface graphique qui ajoute un délai de 200 ms avant de le transmettre au superviseur (on ignore ici les délais réseaux). Le thread `th_communiquer` étant le plus prioritaire, il traite immédiatement le message (on remarque par exemple sur la figure la préemption du thread `th_deplacer`). Le temps de traitement de `th_communiquer` est estimer à 1 ms d'après le document « Dossier de conception ».

La pire des situations est une exécution du thread `th_deplacer` juste avant la réception du message de mouvement qui devra attendre 1 seconde avant de lire la nouvelle valeur de `mouvement`. Le temps de transmission de l'ordre au robot a été estimé à 41 ms (voir p. 12 du document « Dossier de conception »).

Nous avons donc un délai possible de l'ordre de $0.2+0.001+1+0.041 = 1.242$ s entre la sélection d'un mouvement par l'utilisateur et sa réception par le robot.

2.1 Exigence 1 : En cas de perte de communication entre le robot et le superviseur, le robot est stoppé et le nouvel état est signalé à l'utilisateur via le moniteur.

Toutes les secondes, le superviseur envoie un ordre de rechargement du watchdog au robot pour éviter qu'il expire. Car si il expire, un compteur local au robot s'incrémente et lorsqu'il atteint 7, le robot s'arrête et doit être redémarré manuellement.

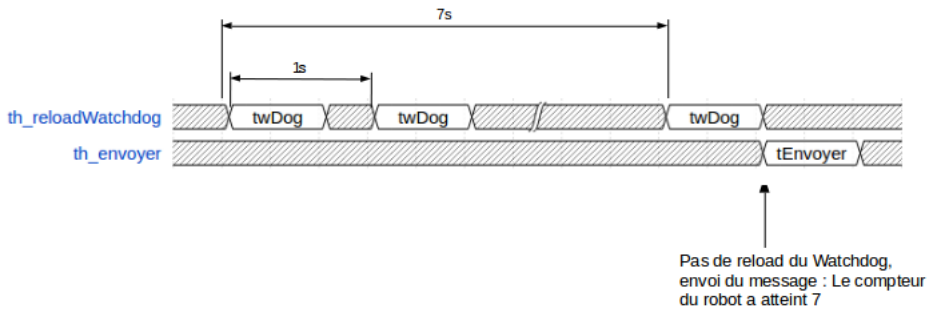

Fig. 12: Estimation de la plus grande durée entre l'instant où la communication est perdue et l'instant où l'utilisateur en est informé.

La plus grande durée entre l'instant où la communication est perdue et l'instant où l'utilisateur en est informé est d'à peu près 8 secondes. Le reloadWatchdog s'execute toutes les 1s et sa priorité est très forte. Donc si l'on perd la connexion à t=1.1s et que le watchdog s'était exécuté à t=1s on commencera à compter à partir de 2s se fera donc 7,9s

2.2 Exigence 2 : La communication entre le robot et le superviseur est déclarée perdue si et seulement si trois échecs successifs de communication entre le robot et le superviseur surviennent.

Quand le superviseur donne un ordre au robot, il répond en envoyant son status. Si le superviseur ne reçoit aucune réponse, un compteur global s'incrémente. S'il atteint 3, on déclare que la connexion est perdue et on en informe le moniteur via la file de messages. De même, si le robot ne répond pas à l'ordre de rechargement du watchdog ou à l'ordre de réception du nouveau niveau de la batterie, le compteur global, qui est partagé entre tous les threads qui communiquent avec le robot, s'incrémente. Si le robot répond, le compteur se remet à 0.

2.3 Exigence 3 : L'image qui est affichée sur le moniteur ne doit pas être plus vieille que 600 ms (différence de temps entre la capture de l'image et son affichage sur le moniteur).

Le thread traitement image a une période de 600ms. L'exécution du thread traitement image prend au maximum 250ms. De plus, la faible priorité du thread traitementImage, d'autres threads peuvent prendre la main et s'exécuter, augmentant le delai d'execution maximal de traitementImage. En revanche, comme chacun des threads à la priorité supérieure à traitementImage ont une période supérieure à 600ms, donc ne pourront pas s'exécuter plus d'une fois entre chaque exécution du thread traitementImage. Donc dans le pire des cas le délai d'attente entre 2 exécutions est de : 4*40ms + 250ms $<$ 600 ms

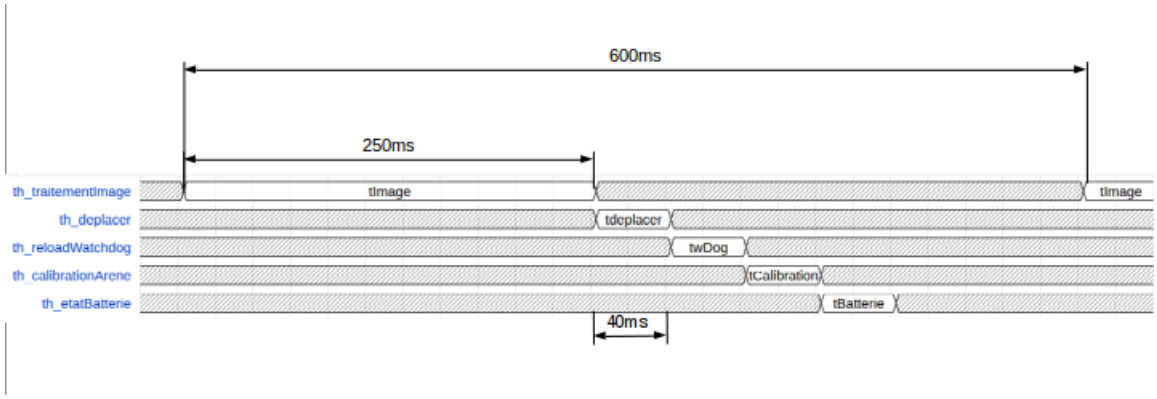

Fig. 13: Chronogramme de l'ordonnacement du système.

3 Transformation AADL2XENO

3.1 Thread

3.1.1 Instanciation et démarrage

Un thread AADL est instancié en C par une tâche (`RT_TASK`) Xenomai. Pour cela, une structure `RT_TASK` est déclarée comme variable globale pour chaque tâche.

Le service `rt_task_create` est utilisé pour créer la tâche, c'est-à-dire réserver son space mémoire et la déclarer au noyau, et `rt_task_start` pour lancer l'exécution de la tâche.

3.1.2 Code à exécuter

Sous Xenomai, le lien entre le thread et le traitement à exécuter se fait grâce à l'appel de la fonction `rt_task_start` qui prend en argument une structure `RT_TASK` et le thread à exécuter. Par exemple pour lancer le thread connecter, on appelle la fonction de la manière suivante : `rt_task_start`(&tconnect, &connecter, NULL).

3.1.3 Niveau de priorités

On déclare d'abord en variable globale le niveau de priorité que l'on souhaite attribuer au thread, puis on donne ce niveau de priorité en argument lors de la création de ce thread avec le service `rt_task_create`.

3.1.4 Activation périodique

On rend périodique l'activation d'un thread AADL sous Xenomai grâce à deux fonctions : `rt_task_set_periodic` qui permet de fixer la période de la tâche et `rt_task_wait_period` qui permet de libérer le processeur quand la tâche a terminé son traitement et doit attendre la prochaine période.

3.1.5 Activation événementielle

Pour gérer les activations événementielles d'un thread AADL, nous avons utilisé des sémaphores pour permettre la synchronisation des différents threads.

3.2 Port d'événement

3.2.1 Instanciation

On instancie les sémaphores à la valeur 0 dans le main.c grâce à la fonction rt_sem_create(&semEvent..,0,.. et en déclarant en extern dans le global.h la sémaphore : extern RT_SEM semEvent et dans le global.c RT_SEM semEvent.

3.2.2 Envoi d'un événement

Pour envoyer un événement on libère le sémaphore associé à un l'événement en incrémentant sa valeur de 1 grâce à la fonction rt_sem_v(&semEvent, INFINITE). En terme de synchronisation cela est équivalent à event !

3.2.3 Réception d'un événement

La synchronisation sur un événement se fait avec l'acquisition de la sémaphore de l'événement qui décrémente sa valeur de 1 grâce à la fonction rt_sem_p(&semEvent). En terme de synchronisation cela est équivalent à event ?

3.3 Donnée partagée

3.3.1 Instanciation

Pour instancier une donnée partagée, on déclare la donnée en global puis on crée un mutex associé à cette donnée grâce à la fonction `rt_mutex_create` qui prend en argument une structure RT_MUTEX. Cela permettra d'éviter l'accès simultané à la donnée partagée par plusieurs processus.

3.3.2 Accès en lecture et écriture

Pour accéder en lecture ou en écriture à une donnée partagée, il faut verrouiller le mutex associé à la donnée. Pour cela on appelle la fonction `rt_mutex_acquire`. Pour relâcher la donnée, il faut déverrouiller le mutex associé grâce à la fonction `rt_mutex_acquire`.

3.4 Ports d'événement-données

3.4.1 Instanciation

L'instanciation des ports d'événement-données avec Xenomai est faite grâce à l'instantciation de la structure serveur, qui permet l'envoi et la réception de données à travers une file de messages.

3.4.2 Envoi d'une donnée

Pour envoyer des données on appelle la fonction appartenant à la structure serveur qui a déjà été implémentée : server->send(serveur,msg). Le message étant obtenu à partir de la file grâce à la fonction rt_queue_read.

3.4.3 Réception d'une donnée

Quels services avez-vous employé pour recevoir des données ?
Pareil grâce à la fonction receive de la structure server : server->receive(server,msg). Le message étant écrit dans la file grâce à la fonction rt_queue_send.

4 Application de la transformation AADL2Xenomai

— **fonctions.c**

```
1
2  void th_controler (void *arg)
3  {
4
5    int new_osd;
6    int nouvelle_commande;
7
8    rt_printf ("th_controler :  initialisation du systeme\n");
9    initialiser_le_systeme();
10
11   t_printf("th_controler : Demarrage des threads\n");
12
13   /* On donne deux ressources pour liberer les deux threads en attente */
14   rt_sem_v(&semDemarrage, TM_INFINITE);
15   rt_sem_v(&semDemarrage, TM_INFINITE);
16   rt_printf("th_controler : Debut de l'execution de periodique a 80ms\n");
17   rt_task_set_periodic(NULL, TM_NOW, 80000000);
18
19
20   while (1)
21     {
22
23     rt_mutex_acquire (&mutexCommande, TM_INFINITE); //lecture de la
       nouvelle commande
24     nouvelle_commande = commande;
25     rt_mutex_release (&mutexCommande);
26
27     new_osd = calcul_osd(nouvelle_commande); //calcul du nouvel OSD
28
29     rt_mutex_acquire (&mutexOSDBuff, TM_INFINITE); //modification de OSD
       Buffer
30     osdBuffer = new_osd;
31     rt_mutex_release (&mutexOSDBuff);
32
33     rt_mutex_acquire (&mutexOSD, TM_INFINITE); //modification de OSD
34     osd = new_osd;
35     rt_mutex_release (&mutexOSD);
36
37     rt_task_wait_period(NULL);
38     rt_printf("th_controler  : Activation periodique\n");
39
40    }
41 }
42
43
44 void th_capture (void *arg)
45 {
```

```
47    DImage *image_capturee;
      DMessage *message;
49    DJpegimage *jpegimage;

51
      rt_printf ("th_capture :  en attente de demarrage\n");
53    rt_sem_p(&semDemarrage, TM_INFINITE);
      rt_printf("th_capture : Demarrage Succes\n");
55

57    while (1)
        {
59          image_capturee = d_new_image();
            image_capturee = acquisition_image(image);
61          rt_sem_v(&semImageCapturee, TM_INFINITE); //synchronisation avec le
       thread th_display, image_capturee !

63
            jpegimage = d_new_jpegimage();
65          d_jpegimage_compress(jpegimage,frame);
            message = d_new_message();
67          d_message_put_jpeg_image(message,jpegimage);

69          rt_printf("th_capture  : Envoie de l'image capturee \n");
            if (write_in_queue(&queueMsgGUI, message, sizeof (DMessage)) < 0) {
      message->free(message);}
71
        }
73  }

75
    void th_display (void *arg)
77  {

79    DImage *image_capturee;
      DMessage *message;
81    DJpegimage *jpegimage;

83
      rt_printf ("th_display :  en attente de demarrage\n");
85    rt_sem_p(&semDemarrage, TM_INFINITE);

87    rt_printf("th_display : Demarrage Succes\n");

89
      while (1)
91      {
            rt_sem_p(&semImageCapturee, TM_INFINITE); //synchronisation avec le
       thread th_capture, image_capturee ?
93
            if ((err = rt_queue_read (&queueMsgGUI, &image_capturee, sizeof (
      DMessage), TM_INFINITE)) >= 0) {
95
                rt_mutex_acquire (&mutexOSD, TM_INFINITE); //osd_var :=
      Read_data(osd)
97              osd_var = osd;
```

```
            rt_mutex_release (&mutexOSD);
99
            img = encode(image_capturee, osd_var);
101
            rt_mutex_acquire (&mutexBufferEcran, TM_INFINITE); //osd_var :=
    Read_data(osd)
103         buffer_ecran = img;
            rt_mutex_release (&mutexBufferEcran);
105       }
        else  {rt_printf ("th_display : Error msg queue write: %s\n",
    strerror (-err));}
107
    }
109 }
```

— **global.h**

```
extern RT_TASK th_controler;
2 extern RT_TASK th_display;
extern RT_TASK th_capture;
4


6 /* @descripteurs des mutex */
 extern RT_MUTEX mutexCommande;
8 extern RT_MUTEX mutexOSDBuff;
 extern RT_MUTEX mutexOSD;
10 extern RT_MUTEX mutexBufferEcran;

12

 /* @descripteurs des sempahore */
14 extern RT_SEM semDemarrage;
 extern RT_SEM semImageCapturee;
16


18 /* @descripteurs des files de messages */
 extern RT_QUEUE queueMsgGUI;
20


22 /* @variables partagees */
 extern int osdbuffer;
24 extern int osd;
 extern int commande;
26 extern int buffer_ecran;


28

 /* @constantes */
30 extern int MSG_QUEUE_SIZE;
 extern int PRIORITY_THCONTROLER;
32 extern int PRIORITY_THCAPTURE;
 extern int PRIORITY_THDISPLAY;
```

— **global.c**

```
1 RT_TASK th_controler;
 RT_TASK th_display;
```

```
RT_TASK th_capture;

RT_MUTEX mutexCommande;
RT_MUTEX mutexOSDBuff;
RT_MUTEX mutexOSD;
RT_MUTEX mutexBufferEcran;

RT_SEM semDemarrage;
RT_SEM semImageCapturee;

RT_QUEUE queueMsgGUI;


 int MSG_QUEUE_SIZE = 10;

 int PRIORITY_THCONTROLER = 60;
 int PRIORITY_THDISPLAY = 70;
 int PRIORITY_THCAPTURE = 80
```

— **main.c**

```
void initStruct(void) {
    int err;
    /* Creation des mutex */
    if (err = rt_mutex_create(&mutexOSDBuff, NULL)) {
        rt_printf("Error mutex create: %s\n", strerror(-err));
        exit(EXIT_FAILURE);
    }
    if (err = rt_mutex_create(&mutexOSD, NULL)) {
        rt_printf("Error mutex create: %s\n", strerror(-err));
        exit(EXIT_FAILURE);
    }

    if (err = rt_mutex_create(&mutexCommande, NULL)) {
        rt_printf("Error mutex create: %s\n", strerror(-err));
        exit(EXIT_FAILURE);
    }

    if (err = rt_mutex_create(&mutexCommande, NULL)) {
        rt_printf("Error mutex create: %s\n", strerror(-err));
        exit(EXIT_FAILURE);
    }


    /* Creation du semaphore */
    if (err = rt_sem_create(&semDemarrage, NULL, 0, S_FIFO)) {
        rt_printf("Error semaphore create: %s\n", strerror(-err));
        exit(EXIT_FAILURE);
    }

    if (err = rt_sem_create(&semImageCapturee, NULL, 0, S_FIFO)) {
        rt_printf("Error semaphore create: %s\n", strerror(-err));
        exit(EXIT_FAILURE);
    }

    /* Creation des taches */
```

```
36      if (err = rt_task_create(&tth_controler, NULL, 0, PRIORITY_THCONTROLER,
         0)) {
            rt_printf("Error task create: %s\n", strerror(-err));
38          exit(EXIT_FAILURE);
        }
40      if (err = rt_task_create(&tth_capture, NULL, 0, PRIORITY_THCAPTURE, 0))
         {
            rt_printf("Error task create: %s\n", strerror(-err));
42          exit(EXIT_FAILURE);
        }
44      if (err = rt_task_create(&tth_display, NULL, 0, PRIORITY_THDISPLAY, 0))
         {
            rt_printf("Error task create: %s\n", strerror(-err));
46          exit(EXIT_FAILURE);
        }
48


50      /* Creation des files de messages */
        if (err = rt_queue_create(&queueMsgGUI, "toto", MSG_QUEUE_SIZE*sizeof(
        DMessage), MSG_QUEUE_SIZE, Q_FIFO)){
52          rt_printf("Error msg queue create: %s\n", strerror(-err));
            exit(EXIT_FAILURE);
54      }

56  }

58  void startTasks() {
        int err;
60      if (err = rt_task_start(&tth_controler, &th_controler, NULL)) {
            rt_printf("Error task start: %s\n", strerror(-err));
62          exit(EXIT_FAILURE);
        }
64      if (err = rt_task_start(&tth_capture, &th_capture, NULL)) {
            rt_printf("Error task start: %s\n", strerror(-err));
66          exit(EXIT_FAILURE);
        }
68      if (err = rt_task_start(&tth_display, &th_display, NULL)) {
            rt_printf("Error task start: %s\n", strerror(-err));
70          exit(EXIT_FAILURE);
        }
72

    }
74

    void deleteTasks() {
76      rt_task_delete(&tth_controler);
        rt_task_delete(&tth_capture);
78      rt_task_delete(&tth_display);
    }
```