ROBOCUP 2015 - MANAGER

SMAGGHE Cyril & HAGE CHEHADE Sandra

Encadrants : MERZOUKI Rochdi & COELEN Vincent

Remerciements

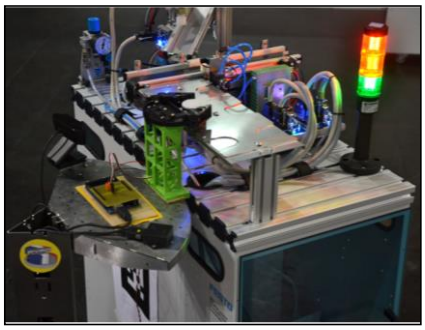
Nous tenons à remercier le département IMA pour l'investissement financier fourni afin de payer les différents frais de déplacement ainsi que ceux concernant le logement à Magdebourg en Allemagne. Nous remercions également M. Merzouki Rochdi de nous avoir épaulés dans la recherche de sponsors en particulier auprès de l'université Lille 1.

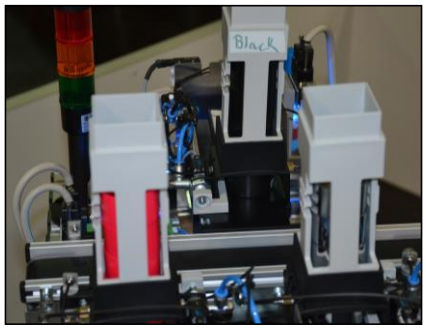
Nous remercions le centre de recherche CRIStAL de nous prêter l'un de ses locaux. Ainsi l'ensemble de l'équipe a pu réaliser de nombreux tests afin de valider les différentes conceptions matérielles et logicielles.

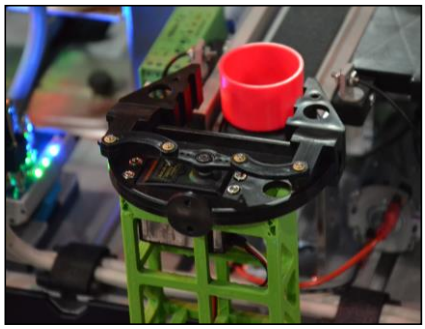
Nous tenons à remercier plus particulièrement M. Coelen Vincent pour la patience qu'il a su fournir. Nous le remercions également d'avoir partagé avec nous ses connaissances concernant le codage en C++ et l'utilisation de ROS.

Nous remercions également les autres membres de l'équipe pour leurs contributions : Vergez Valentin, Payelle Vianney, Tissot Elise, Krikorian Romain et Danel Thomas.

Sommaire





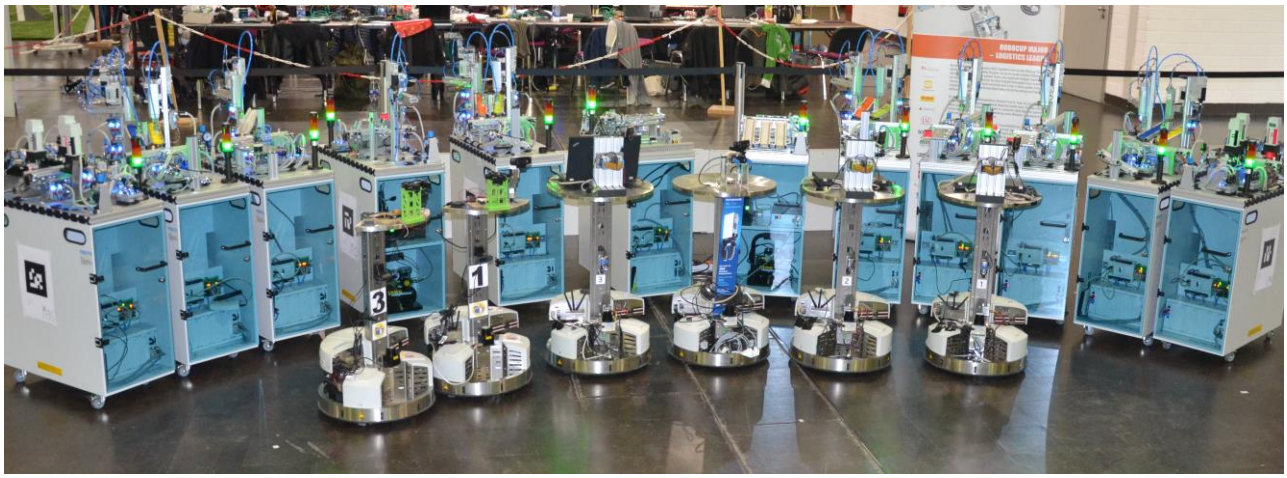


Introduction

Lors de la compétition de l'Open German, prélude de la Robocup, il fallait mettre en place un système d'autonomisation de production à l'aide de Robotinos (robots mobiles de Festo ayant un système d'exploitation Linux). Cette compétition vise à faire avancer la recherche dans la robotique mobile. Ainsi de nombreux participants à cette compétition sont soit des doctorants ou alors de futurs doctorants.

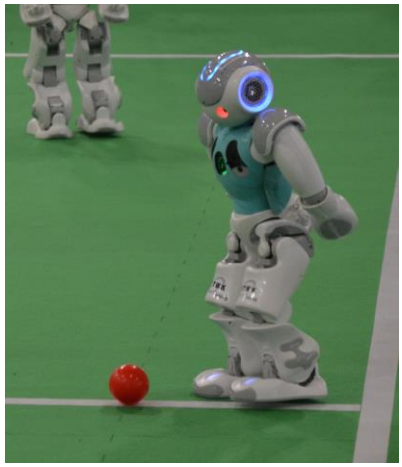
L'une des parties essentielles qu'il fallait réaliser était la partie pensante. C'est pourquoi nous avons créé un manager à l'aide de ROS. Ce manager est un ensemble de nœuds permettant de recevoir les informations envoyées par les autres nœuds, de les traiter, tout cela afin d'ordonnancer les ordres que doit exécuter chaque Robotino et de les donner auprès des nœuds correspondants.

Nous avons décidé de séparer le manager en deux parties bien distinctes : le générateur de tâches et les trois exécuteurs de tâches (en effet chaque équipe peut mettre en course trois Robotinos). Le générateur de tâches s'occupera de la mise en place des différentes tâches à réaliser, à leur ordonnancement et à leur attribution. Les exécuteurs de tâches, quant à eux, feront le lien entre les tâches demandées par le générateur de tâches et les autres nœuds.



Présentation du projet

1 - La RoboCup



La RoboCup est un tournoi international de robotique. L'origine de la création de la compétition est marquée par le but de créer une équipe de football robotisée capable de battre une équipe de football humaine championne du monde, d'ici 2050. Cette année pour la première fois lors de l'Open German qui s'est déroulé à Magdebourg en Allemagne, une équipe de robot s'est mesurée à une équipe humaine.

A présent, la RoboCup regroupe différentes ligues de robotique avec chacune des objectifs bien distincts. On peut trouver :

- La RoboCup Rescue où le principe est la simulation de sauvetage d'humains dans un environnement hostile
- La RoboCup Junior est orientée vers la robotique pour enfants comme son nom l'indique
- La RoboCup @Work consiste à remplacer un humain dans un atelier de travail
- La RoboCup @Home est une ligue d'assistance à la personne
- La RoboCup Logistic cherche à automatiser une chaîne de production. C'est cette ligue qui nous intéresse justement pour notre projet.

2 - La Logistic League

La Logistic League, sponsorisée par l'entreprise Festo, s'oriente autour de solutions flexibles pour la production industrielle utilisant des robots autonomes (Robotinos). Les participants peuvent prendre tous les capteurs qui leur semblent nécessaires afin de pouvoir de pouvoir répondre aux demandes de l'arbitre (arrêt d'urgence, production et livraison d'un produit dans un temps déterminé).

Les robots doivent envoyer périodiquement à la Referee Box des données à tout moment du jeu. Celui-ci, en retour, retourne chaque instant l'état du jeu, le nom de l'équipe, le nombre de points acquis par l'équipe ainsi que les produits demandés en priorité durant la phase de production.

Les machines avec lesquelles les robots interagissent sont de réelles machines physiques, c'est pourquoi elles sont soumises à de possibles pannes qui peuvent être tout aussi bien réelles que simulées. Le seul moyen de connaître l'état de fonctionnement des machines est le code couleur indiqué par la colonne de feu.

Lors de la phase d'exploration, chaque équipe se doit de lire l'un des ARTags de chaque machine ainsi que l'état de ses feux puis de reporter à la RefBox ce que les robots a observés. Un mauvais report de machine ou le report d'une machine adverse occasionne des malus, c'est pourquoi il est nécessaire de bien déterminer au préalable quelles sont les zones appartenant à l'équipe. Voici une photographie où l'on peut visualiser différentes machines avec leur ARTag et leur colonne de feux.

Chaque produit demandé par la RefBox durant la phase de production est constituée d'une base (rouge, noir, argent) surmontée de 0 à 3 ring(s) (bleu, vert, jaune,orange) surmonté(s) d'un cap (gris ou noir). Ci-dessous un schéma permettant de mettre en lumière différentes combinaisons possibles :

Afin de créer ces différents produits, chaque équipe dispose de plusieurs types de machines une Base Station contenant toutes les bases nécessaires, deux Ring Stations ayant chacune deux couleurs de ring, deux Cap Stations ayant chacune des étagères pouvant servir d'espace de stockage et une Delivery Station dont le but est de collecter les produits finis.

3 - Le manager

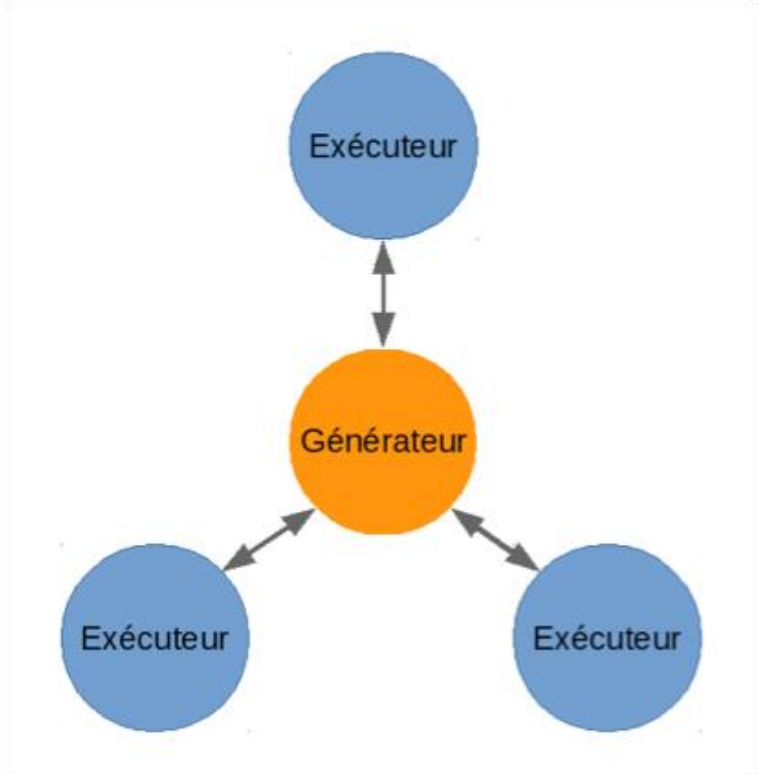
Le manager est l'unité pensante des trois robotinos. Ce manger, réalisé à l'aide ROS (Robot Operating System), est un ensemble de noeuds permettant de créer et de donner des ordres seulement à partir de l'état du jeu et des commandes envoyés par la Referee Box. Ce manager a pour mission d'optimiser le nombre de points afin de permettre à l'équipe de gagner un maximum de points car il ne faut oublier que la RoboCup est avant tout une compétition. L'ensemble du travail fourni autour du manager est essentiellement théorique et ne peut être testé seulement au moment où les autres noeuds sont fonctionnels.

Certaines contraintes intrinsèques au projet ont été mises en évidence dès le début. Il faut assurer une communication continue avec la Referee Box, de plus, il faut faire coopérer et communiquer les trois robots ensembles. Par ailleurs, il fallait assurer certaines contraintes de robustesse (interblocage entre robots, panne de robot, …). Enfin, étant donné que chaque année des candidats de Polytech Lille y participent, il faut permettre la pérennité du projet.

Le manager pour fonctionner devra interagir avec les différents nœuds suivants ayant chacun leurs rôles spécifiques :

- RefereeBox Comm : recevoir les données envoyées par la RefereeBox et envoyer des données suivant l'état du jeu à la RefereeBox
- Robots Comm : communiquer aux autres robots ce qu'il est en train de faire
- Traitement des TAGs : lire les TAG
- Traitement des feux : lire l'état des colonnes de feu
- Pince de préhension : ouvrir et fermer la pince et renvoyer l'état de la pince
- Approche finale : s'adapter au banc de charge ou de décharge de produits
- Génération trajectoire : détermine une trajectoire pour aller au lieu (X,Y)
- Réalisation trajectoire : parcourir la trajectoire
- Localisation : savoir périodiquement (faible période) où se situe le robot
- Détection d'obstacle : détecter les obstacles fixes et mobiles

Le manager se découpe en deux parties : le générateur de tâches et les exécuteurs de tâches (un pour chaque robot). Le générateur traduira les commandes de la Referee Box, ordonnancera les tâches à réaliser et les communiquera à l'exécuteur de tâches correspondant. L'exécuteur de tâches, quant à lui, utilisera les autres nœuds afin de mener à bien la tâche demandée par le générateur de tâches.



Afin de pouvoir travailler en parallèle mais de quand même pouvoir avoir accès au travail de chacun, nous avons mis l'ensemble du code sur le github de l'équipe. Il est disponible et régulièrement actualisé à l'adresse : https://github.com/PyroTeam/robocup-pkg/tree/manager-devel/manager.

Présentation des outils de développements

1 – ROS

Robot Operating System (ROS) est une collection d'outils, de librairies et de conventions permettant l'écriture de logiciels de robots. ROS vise à simplifier la création d'un comportement de robot complexe et robuste et cela à travers une très grande variété de plates-formes robotiques.

ROS a été conçu dans le but de permettre à de petits groupes de programmeurs de collaborer et de s'appuyer sur le travail de chacun afin d'obtenir un logiciel qui englobe tous les divers travaux.

2 - Distribution utilisée

Actuellement, il existe trois distributions disponibles sur lesquelles il est possible de travailler : Indigo Igloo, Hydro Medusa et Groovy Galapagas. Pour le projet, nous avons travaillé sur Hydro. En effet, cette distribution est la plus récente utilisable sur Ubuntu 12.04 et les robotinos étaient sur Ubuntu 12.04.

Depuis peu, l'entreprise Festo a fourni une version Ubuntu 14.04 disponible et utilisable sur les robotinos. C'est pourquoi, nous avons terminé le développement avec Indigo qui est la distribution la plus récente et sommes passé sous Ubuntu 14.04.

3 - Vocabulaire utilisé dans ROS

Un certain vocabulaire est à connaître afin de comprendre le développement sous ROS tels que les mots "package", "noeud", "message", "service","action",etc.

Les logiciels sous ROS sont organisés par packages. Un package est un répertoire contenant un fichier "package.xml" et pouvant contenir des noeuds ROS, des librairies indépendantes, des fichiers de configuration et toute autre chose permettant de constituer un module utile.

Un noeud n'est ni plus ni moins qu'un exécutable dans un package ROS. Les noeuds ROS utilisent des librairires de ROS pour communiquer entre eux. Pour réaliser cela, les noeuds peuvent publier des messages sur un topic ou s'abonner à un topic pour recevoir des messages ainsi que fournir ou utiliser un service.

Les topics sont des bus nommés par lesquels les noeuds s'échangent des messages. Un message, quant à lui, est un type de données ROS utilisé lorsqu'un noeud publie ou s'abonne à un topic.

L'interaction requête/réponse est réalisée grâce à un service, qui est défini comme étant une paire de messages : l'un pour la requête et l'autre pour la réponse. Ainsi le générateur de tâche enverra comme service celui de réaliser une tâche et l'exécuteur lui répondra s'il veut bien la faire.

L'action est une forme avancée de service permettant de répondre aux besoins de réaliser un service pouvant être long et susceptible d'être annulé à tout moment par le serveur ou le client.

Générateur de tâches

1 - Phase d'exploration

Concernant la phase d'exploration, le générateur de tâches se contente de demander aux différents robots d'explorer les zones de l'équipe à l'aide du service **order.srv** (voir annexe 1). La classe **CorrespondanceZE** permet de récupèrer les informations concernant les zones de chaque équipe en s'abonnant au topic **refBoxComm/ExplorationInfo** contenant le message **ExplorationInfo.msg** (voir annexe 2). Lorsque que toutes les zones ont été explorées, le générateur de tâches attend le début de phase de production.

Afin de déterminer quels sont les robots disponibles, le générateur de tâches s'abonne au topic **task_exec_state** contenant le message **activity.msg** (voir annexe 3). Pour parvenir à réaliser ceci, la classe Action collectera les informations et modifiera la valeur de ses attributs

suivant le contenu du topic **task_exec_state**. Lorsqu'un exécuteur de tâches publie comme **state END**, cela signifie que le robot est prêt à recevoir un nouvel ordre puisqu'il a terminé le dernier qui lui a été demandé de faire.

Le générateur de tâches, par le biais du service, demande de réaliser comme type d'ordre **DISCOVER** au robot actif et non occupé avec pour paramètre **NONE**. Cela est géré grâce à la classe **Srvorder**. Puis le générateur de tâches attend de pouvoir redonner un ordre au prochain robot libre.

Lors de l'Open German, nous n'avons pu que tester une phase d'exploration avec un unique robot à cause de problèmes avec d'autres noeuds. Cependant, on a pu constater que le générateur de tâches récupérait les bonnes zones et envoyait seulement à l'exécuteur de tâche valide des ordres. Il faudra refaire d'autres tests avec plusieurs robots en course mais pour l'instant le travail réalisé autour du générateur de tâches durant la phase d'exploration est validé et fonctionnel.

2 - Phase de production

La phase de production a demandé bien plus de travail en ce qui concerne le générateur de tâches. Nous développerons ici uniquement le choix de la structure de travail ainsi que les fonctions permettant de rendre "intelligents" les différents robotinos. Tout le travail se base sur le principe de "*Diviser pour mieux régner*", c'est-à-dire la production d'un produit sera découpée en sous-tâches. Ces sous-tâches seront découpées en sous-sous-tâches et ainsi de suite. Ainsi les ordres demandés par le générateur de tâches seront des tâches les plus élémentaires qui soient.

Choix de la structure de travail

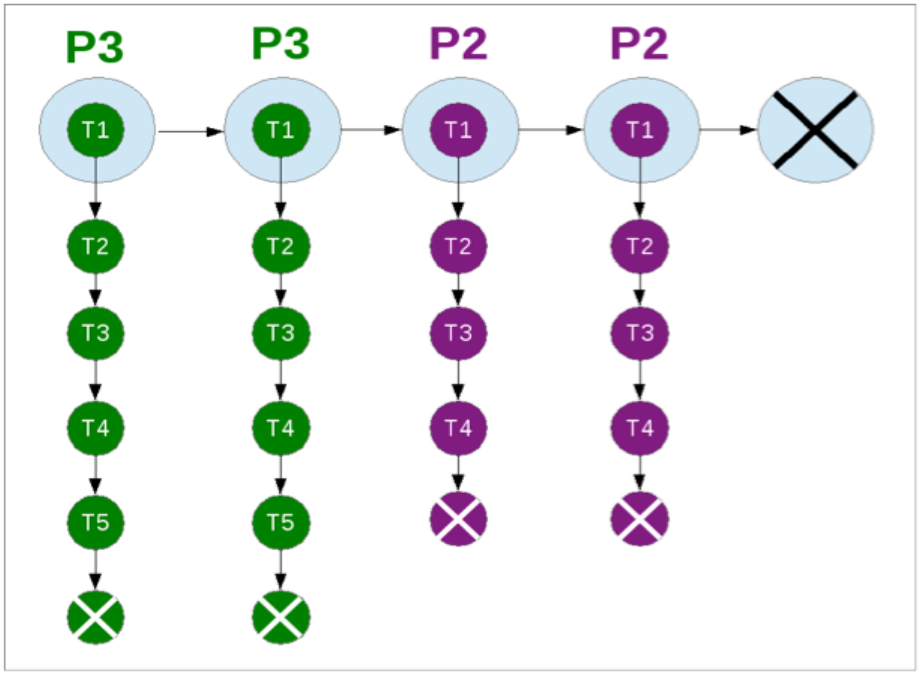
Avant d'envoyer un quelconque ordre à l'exécuteur de tâches, il était nécessaire de définir une classe permettant d'englober l'ensemble des paramètres que pouvait contenir une tâche.
Cette classe nommée ***Task*** a les attributs suivants :

- **m_title** : intitulé de la tâche cela peut être chercher une base, décapsuler, mettre un ring sur le produit intermédiaire, …

- **m_parameter** : un paramètre quelconque mais souvent cela concerne de la pièce souhaitée
- **m_beginnningDelivery** : date à laquelle on peut commencer à délivrer le produit fini
- **m_endDelivery** : date limite à laquelle on peut ramener le produit fini
- **m_creation** : temps de création estimé, en effet, les aléas du jeu peuvent provoquer certains retards
- **m_ratio** : ratio du nombre de points que vaut un produit sur son temps de production
- **m_inProcess** : booléen indiquant si la tâche est en cours de traitement ou non
- **m_robot** : numéro du robot qui est exécuté la tâche (0 par défaut)
- **m_endCarryingOut** : date de fin de réalisation de la tâche actuelle

La structure avec laquelle le générateur de tâche travaillera, en particulier durant la phase de production, est une liste de liste de tâches. En effet, chaque liste de tâches correspondra à la création d'un produit et la liste de liste de tâches correspondra à l'ensemble de la production restante encore à faire.

L'exemple suivant simule le cas où la RefBox a demandé deux fois le produit P3 et deux fois le produit P2. Le produit P3 est réalisable en cinq tâches. Ce produit mettra plus de temps à être fabriqué mais rapportera plus de points. Le produit P2 est réalisable en quatre tâches.



Cette structure de travail est continuellement mise à jour dès qu'au minimum un robot est prêt à recevoir un ordre.

La fonction RatioCalculus

La fonction **ratioCalculus** (voir annexe 4) est la fonction permettant de déterminer les ratios de chaque liste de tâches. Les calculs sont effectués seulement sur le premier élément de chaque liste.

Certaines tâches sont bien plus prioritaires que d'autres, c'est pourquoi il est nécessaire d'imposer des ratios en dur. Les tâches les plus importantes sont celles dont l'intitulé est **PUT_CAP**, **PUT_RING** ou **DESTOCK** à condition qu'il y ait eu juste auparavant **TAKE_RING**, **TAKE_BASE** ou **TAKE_CAP** (voir annexe 1). En effet, il est nécessaire de mettre une priorité très importante sur ces tâches car sinon un autre robot pourrait susceptible de vouloir prendre un produit qui serait dans la pince d'un autre robot.

Vient ensuite la tâche dont l'intitulé est **DESTOCK** même si la tâche précédente n'était ni **TAKE_RING**, ni **TAKE_BASE**, ni **TAKE_CAP**. Par contre, il faut vérifier que c'est le bon moment pour ramener un produit avant de le délivrer le produit car sinon cela ne rapporte pas beaucoup de points le fait de le ramener à une date non comprise dans l'intervalle de temps exigé par la RefBox.

Les autres tâches sont par contre classées suivant le nombre de points que rapporte un produit et le temps de production restant. Ainsi les tâches rapportant le plus de points qui mettront le moins de points pour obtenir un produit fini seront les plus prioritaires. La seule tâche faisant exception est celle dont l'intitulé est **UNCAP** si celle-ci n'est suivie d'aucune autre tâche car dans ce cas on forcera le ratio à être le plus faible possible. Cela permet d'éviter qu'une simple décapsulation soit prioritaire par rapport à la production d'un produit.

La fonction addInWork

La fonction **addInWork** (voir annexe 5) permet de rajouter un ordre de la RefBox dans la structure de travail à condition évidemment qu'il n'est pas encore été pris en compte. Il vérifie le nombre de cap disponible et s'il y en a alors il rajoute directement la liste de tâches nécessaires à la production du produit exigé autant de fois que la RefBox l'a demandé.

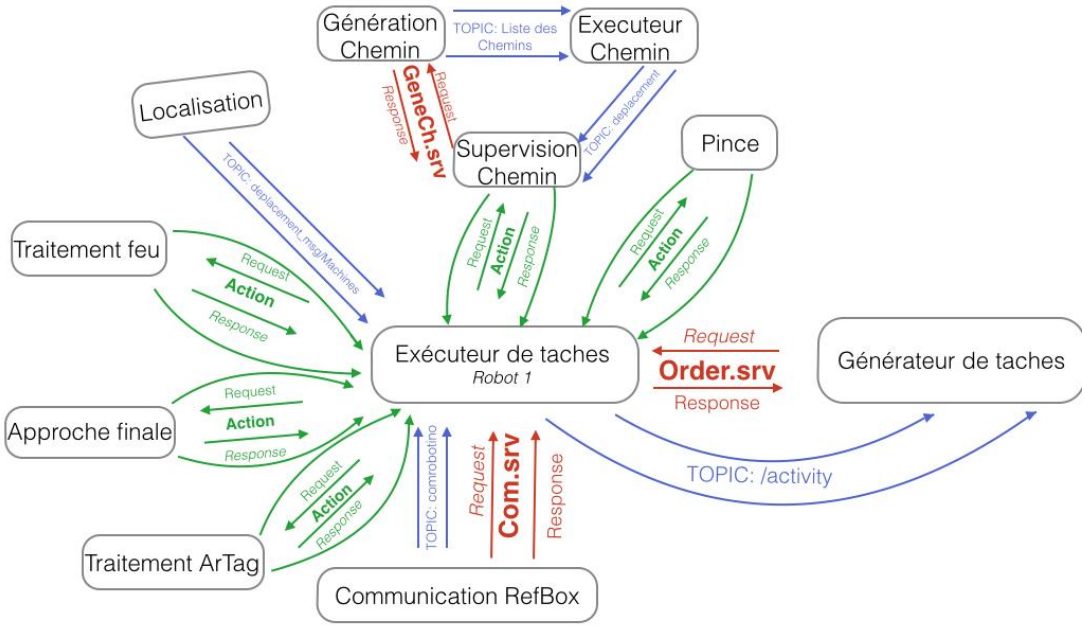
Dans le cas contraire, si aucun cap n'est disponible, la liste de tâches nécessaires à la création du produit sera concaténée avec la liste contenant uniquement la tâche dont l'intitulé est **UNCAP** pour faire en sorte que dès le début un cap soit disponible pour pouvoir créer un produit fini.

Ainsi, à l'aide de ces deux uniques fonctions, on peut facilement créer les différentes tâches qui seront demandées à l'exécuteur de tâches. D'autres fonctions permettent d'utiliser ces deux fonctions ou alors d'actualiser correctement la structure de travail mais elles ne constituent pas le centre des travail réalisé autour du générateur de tâches.

Exécuteur de tâches

1 - Communication entre nœuds

L'exécuteur de tâches est un noeud qui recevra des ordres de la part du générateur de tâches et qui, à son tour, va envoyer des ordres aux autres noeuds.
Voici le schéma de communication entre les différents noeuds, en rouge on a les services, en vert les action et en bleu les topics :



Générateur de tâches

Le générateur envoie une requête grâce au service " Order.srv ". En fonction de l'ordre demandé, l'exécuteur saura s'il va devoir explorer une machine (pour la phase d'exploration), ou demander une base, un ring etc..

L'état d'avancement de l'ordre sera publié sur le topic "activity" en continu. Ce topic sera utilisé pour la communication entre les robotinos puisqu'il précise les machines utilisées par chaque robotino.

Communication RefBox

Ce noeud permet de communiquer avec les machines par le biais de la RefereeBox. Malheureusement, cette dernière n'était pas encore terminée pour la phase de production, nous n'avons donc pas pu réaliser cette partie dans l'exécuteur pour l'OpenGerman.

Traitement ARTag

Durant la phase d'exploration, il sera nécessaire de lire l'id d'un ARTag qui sera juste en face d'une caméra. Ceci sera réalisé grâce à une action qui aura comme réponse l'id de l'ARTag correspondant. L'interprétation de l'id permettra de savoir de quelle machine il s'agit, et si elle correspond à notre équipe ou à l'équipe adverse ainsi que le coté (entrée ou sortie de la machine). Les id de tous les ARTags sont écrits en dur dans l'exécuteur de tâches.

Approche Finale

L'approche finale a pour but de finaliser et de préciser l'accostage d'une machine par un robotino. En effet, quand le robotino doit s'approcher d'une machine pour lire l'ARTag, déposer/prendre un produit, sa position doit être la plus précise possible. Ça sera donc à l'exécuteur de tâches d'envoyer une requête grâce à une action qui précisera le type de la machine, l'endroit vers lequel le robotino va s'approcher (colonne de feu, convoyeur, ARTag).

La condition essentielle pour que ce noeud fonctionne est que le robotino doit être en face de la machine. Parce que, le principe de l'approche finale est de visualiser, grâce au Scan Laser, le segment de la machine. Donc l'exécuteur de tâches doit s'assurer que cette condition soit bien assurée avant d'appeler ce noeud.

Pince

La demande de fermeture/d'ouverture de la pince se fait grâce à une action. L'intérêt d'utiliser une action est de pouvoir savoir l'avancement de la requête. Parce qu'il sera nécessaire de fermer ou d'ouvrir à 100% la pince afin d'éviter la perte du produit.

Traitement feu

Ce noeud permet de lire les états des trois couleurs de la colonne de feu Festo. Le feu peut être allumé, éteint ou clignotant.

L'exécuteur de tâches appelle ce noeud grâce à une action qui retournera l'état de chaque feu. L'interprétation des feux se fait dans l'exécuteur de tâches.

Localisation

Ce noeud permet de localiser les machines grâce aux données Scan Laser. En fait, il publie en continu les coordonnées du centre des machines ainsi que leurs orientations. L'exécuteur de tâches récupère ces données et les enregistre au fur et à mesure.

Ces données sont très importantes notamment lors de l'appel du noeud de l'approche finale, puisqu'il sera important d'arriver en face de la machine concernée.

Navigation

L'exécuteur de tâches demandera, grâce à une requête d'action, au noeud "supervision de chemin", d'aller vers un point(x,y,theta). Si ce noeud estime que le point d'arriver est un point interdit, l'exécuteur de tâches demandera d'aller vers un point juste à côte, et ainsi de suite.

2 - Traduction de l'ordre (DISCOVER)

Durant la phase d'exploration, l'exécuteur de tâches va recevoir un ordre du générateur de tâches pour explorer une zone de la map via le service **order.srv** (voir annexe 1) :

order = DISCOVER

id = zone à explorer

Etape 1 : navigation

L'ET demande au noeud navigation de se déplacer vers le point en bas à gauche de la zone via l'action "**MoveToPose.action**" (voir annexe 6).

position_finale = point en bas à gauche de la zone (x,y,theta)

Etape 2 : approche finale

L'ET va demander au noeud "approche finale" de s'approcher de la machine afin de lire son ARTag via l'action "**finalApproaching.action**" (voir annexe 7)

type = DS

side = OUT

parameter = NONE

Notons que le "type" n'a pas d'importance pour la phase d'exploration vu qu'on ne connaît pas, au début, de quelle machine on s'approche. Cependant, il est nécessaire de préciser le "side" puisque la colonne de feu se situe du côté sortie de la machine.

Etape 3 : ARTag

Grâce à ce dernier, l'ET pourra connaître le type de la machine et si le Robotino est du côté de l'entrée ou de la sortie. Parce que, en fait, s'il est en entrée l'exécuteur de tâches va demander à la navigation puis l'approche finale d'aller au côté sortie de la machine puisque la colonne de feu se trouve du côté sortie.

Etape 4 : Lecture Feu

Une fois arrivé du côté sortie de la machine, l'ET va demander au noeud du traitement feu de lire l'état des trois feux de la colonne FESTO grâce à l'action "**processLightSignal.action**" (voir annexe 8).

Le noeud va uniquement envoyer une table de **LightSpec** (voir annexe 9)

Etape 5 : Envoie RefereeBox

L'interprétation des feux se fait grâce à la RefereeBox. En effet, la RefereeBox publie sur un topic un tableau avec tous les états possibles de feux avec une chaîne de caractères pour chaque état. L'exécuteur va donc pouvoir récupérer la chaîne de caractères correspondante à l'état actuel des feux de la machine explorée. L'exécuteur peut donc maintenant identifier une machine en envoyant : la zone explorée, l'id de l'ARTag trouvé, ainsi que la chaîne de caractères qui correspond aux feux.

Etape 6 : ET a terminé

Une fois ces données envoyées à la RefereeBox, l'ET publiera sur le topic "**activity.msg**" (annexe 3)

state = END

Ainsi, le générateur de tâches saura que l'ordre est terminé.

La phase de production est similaire à la phase d'exploration concernant l'exécuteur de tâches. La différence principale consiste de l'utilisation du noeud « Pince » ainsi celui de la communication avec les machines MPS.

Conclusion

En conclusion, notre projet était divisé en deux parties. La première est la réalisation du générateur de tâches, et la deuxième, l'exécuteur de tâches. Ces deux parties forment le cerveau et la colonne vertébrale des trois robotinos. Ils dirigeront les autres noeuds afin de permettre à l'équipe PYRO de gagner le maximum de points.

Ce projet de 4ème année était à la fois très enrichissant et très intéressant. En effet, nous avons découvert ROS, un système d'exploitation consacré à la robotique ainsi qu'à utiliser un langage non étudié en cours (C++). Ce projet nous a en outre permis de réaliser un réel travail d'équipe notamment au niveau des branches de développement sur GitHub que chacun avait créées pour sa partie. La principale difficulté de ce projet est la dépendance avec les autres noeuds. Il faut donc s'assurer que ces noeuds fonctionnent afin de pouvoir tester, c'est pourquoi une partie de nos codes n'a malheureusement pas pu être mise à l'épreuve de la réalité.

Nous avons donc eu la chance de participer à l'OpenGerman 2015 à Magdebourg et ainsi tester nos codes dans les conditions réelles. Durant cette expérience, étant donné que la phase de production n'était pas entièrement codée du côté des personnes organisant la compétition, nous n'avons que pu faire une phase d'exploration.

Cependant, nous avons tout de même réussi à mettre des points lors de la phase d'exploration et ainsi obtenir la seconde place. Tout cela est loin d'être terminé, en effet, nous essayerons de participer à la RoboCup 2015 en Chine en juillet et l'année prochaine, nous participerons à l'OpenFrance ainsi qu'à la RoboCup 2016.

Annexes

Annexe 1 : Order.srv

```
time game_time
int8 number_order
int8 number_robot

int8 TAKE_BASE          = 0
int8 PUT_CAP            = 1
int8 TAKE_CAP           = 2
int8 PUT_RING           = 3
int8 TAKE_RING          = 4
int8 BRING_BASE_RS      = 5
int8 DELIVER            = 6
int8 UNCAP              = 7
int8 DESTOCK            = 8
int8 DISCOVER           = 9

int8 type

int8 BLACK          = 10
int8 SILVER         = 11
int8 RED            = 12
int8 ORANGE         = 13
int8 YELLOW         = 14
int8 BLUE           = 15
int8 GREEN          = 16
int8 GREY           = 17
int8 DS             = 18
int8 STOCK          = 19
int8 NONE           = 20

int8 parameter
int8 id

---

bool accepted
int8 number_order
int8 number_robot
int8 id
```

Annexe 2 : ExplorationInfo.msg

```
ExplorationSignal[]  signals

ExplorationZone[]    zones
```

Annexe 3 : activity.msg

```
int8    nb_robot

int8    IN_PROGRESS     = 10
int8    END             = 11
int8    ERROR           = 12

int8    state

int8    BS          = 0
int8    RS1         = 1
int8    RS2         = 2
int8    CS1         = 3
int8    CS2         = 4
int8    DS          = 5
int8    NONE        = 6

int8    machine_used

int8    nb_order
```

Annexe 4 : Fonction ratioCalculus

```
//calcule le ratio de chaque première tâche de chaque liste de tâche

void ratioCalculus(list<list<Task> > &work,double time,int robot,bool
take[]){
	list<list<Task> >::iterator t_it;
	for(t_it = work.begin(); t_it != work.end(); t_it++)
	{
		t_it->begin()->setRatio(0);
		if(t_it->begin()->getTitle() == int(orderRequest::DESTOCK))
		{
			if(t_it->begin()->inTime(time))
			{
				t_it->begin()->setRatio(100);
			}
		}
		else
		{
			if((t_it->size() == 1) && (t_it->begin()->getTitle() ==
				int(orderRequest::UNCAP)))
			{
				t_it->begin()->setRatio(0.1);
			}
			else
			{
				t_it->begin()->setRatio(
					t_it->begin()->pointPerProduct());
			}
		}
		if(take[robot]==true)
		{
			if(t_it->begin()->getTitle() ==
int(orderRequest::PUT_CAP) ||
			   t_it->begin()->getTitle() ==
int(orderRequest::PUT_RING)||
			   t_it->begin()->getTitle() ==
int(orderRequest::DELIVER))
			{
				t_it->begin()->setRatio(300);
			}
		}
		if(t_it->begin()->getInProcess())
		{
			t_it->begin()->setRatio(0);
		}
		t_it->begin()->setRatio(
			t_it->begin()->getRatio() / t_it->begin()-
>getCreation());
	}
}
```

Annexe 5 : Fonction addInWork

```
// rajoute les nouvelles listes de tâches à faire éxecuter
void addInWork(list<list<Task> > &work, Order &order,int
&availableCap){
	if(!alreadyInWork(work,order) && !order.getProcessed())
	{
		order.setProcessed(true);
		for(int i=0; i< order.getQuantity(); i++)
		{
			if(availableCap > 0)
			{


	work.push_back(creationListTasksProduct(
					order.getProduct(),
					order.getBeginningDelivery(),
					order.getEndDelivery()));

					availableCap --;
			}
		else
		{

			list<list<Task> >::iterator workIterator;
			workIterator = work.begin();
			while(workIterator != work.end() && !
					uncapInWork(*workIterator))
			{
				workIterator++;
			}
			if(workIterator != work.end())
			{
				list<Task> ltmp = creationListTasksProduct(
							order.getProduct(),
							order.getBeginningDelivery(),
							order.getEndDelivery());
				int tmpCreation = ltmp.begin()->getCreation();
				workIterator->splice(workIterator->end(),
ltmp);
				workIterator->begin()->setCreation
								(tmpCreation + 30);
			}
		}
	}
}
}
```

Annexe 6 : MoveToPose.action

```
# Define the goal

geometry_msgs/Pose2D position_finale

---

# Define the result

uint8 result

uint8 FINISHED       = 1
uint8 ERROR          = 2

---

# Define a feedback message

uint8 percent_complete
```

Annexe 7 : FinalApproaching.action

```
#goal definition

int8 BS = 0
int8 RS = 1
int8 CS = 2
int8 DS = 3

int8 type

int8 IN      = 100
int8 OUT  = 101

int32 side

int8 S1                 = 10
int8 S2                 = 20
int8 S3                 = 30
int8 CONVEYOR    = 40
int8 LIGHT           = 50
int8 LANE_RS       = 60

int8 parameter

---

#result definition

bool success

—

#feedback

int32 percent_complete
```

Annexe 8 : processLightSignal.action

```
# Define the goal

---

# Define the result

comm_msg/LightSpec[] light_signal

—

# Define a feedback message

uint32 percent_complete

uint32 images_processed
```

Annexe 9 : LightSpec.msg

```
        uint8 RED           = 0
        uint8 YELLOW        = 1
        uint8 GREEN         = 2

        uint8 OFF           = 0
        uint8 ON            = 1
        uint8 BLINK         = 2

        uint8 color

        uint8 state
```