AIP PRIMECA

Manuel technique

# Solution de contrôle et de supervision du bras robot Stäubli RX60 sous ROS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 732287.
The RIO project starts in Jan 2020 and ends in Dec 2020.

RIO ROSin

Nathan MORET


RIO


ROSin

# Table des matières

# Le matériel

## 1. Bras robot Stäubli RX60

Le bras robot Stäubli RX60 est un bras 6 axes d'une masse de 41 kg produit en 2005 (Annexe 2). Il n'est actuellement plus commercialisé par son constructeur. Son contrôleur est à l'origine un contrôleur CS7 qui a par la suite été mise à jour en contrôleur CS8. Le langage de programmation est le langage Val3 propre aux robots Stäubli.

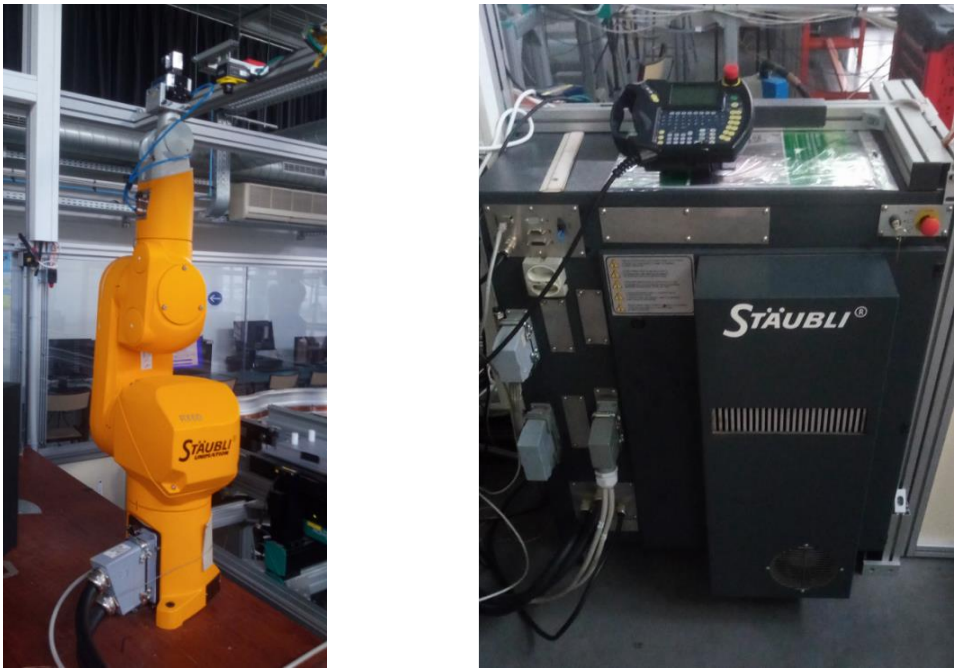



*Figure 1 - Photos du bras robot RX60 et de son contrôleur CS8*

Ces caractéristiques sont présentées dans les tableaux suivants :

| | |
|---|---|
| **Volume de travail**<br><br>Rayon de travail maxi entre axes 2 et 5<br>Rayon de travail mini entre axes 2 et 5<br>Rayon de travail entre axes 3 et 5 | <br><br>600 mm<br>233 mm<br>310 mm |
| Vitesse maxi au centre de gravité de la charge<br>Répétabilité à température constante | 8 m/s<br>+/- 0,02 mm |

*Tableau 1 - Caractéristiques générales du bras robot Stäubli RX60*

| Axe | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Amplitude (°) | 320 | 255 | 269 | 540 | 230 | 540 |
| Vitesse nominale (° /s) | 287 | 287 | 319 | 410 | 320 | 700 |
| Résolution angulaire (°.$10^{-3}$) | 0,724 | 0,724 | 0,806 | 1,177 | 0,879 | 2,747 |

*Tableau 2 - Amplitude, vitesse et résolution du bras robot Stäubli RX60*

| Charge transportable | Bras standard |
|---|---|
| A vitesse nominale | 2,5 kg |
| A vitesse réduite | 4,5 kg |

*Tableau 3 - Charge transportable du bras robot Stäubli RX60*

# 2. Logiciel

**Linux :** Ubuntu 18.04.5 LTS


**ROS :** Distribution Melodic

# Présentation globale

.

## 1. Architecture de la solution

Lorsqu'on programme un robot industriel avec la solution du constructeur, on code un programme dans un langage propre au constructeur sur une suite robotique pour le transférer sur le contrôleur. Une fois le programme téléchargé sur le contrôle, il communique à l'OS (système d'exploitation) du contrôleur les instructions qui sont alors effectuées par le robot.

Pour contrôler le robot avec ROS, la première piste explorée est donc de remplacer le programme en langage propre au constructeur par une couche ROS contenant le programme à réaliser par le robot. Cette solution impose donc de faire communiquer la couche ROS directement avec l'OS du contrôleur du robot. C'est une solution très bas niveau qui nécessite de connaître l'architecture hardware du contrôleur. Devant la charge de travail conséquente à l'élaboration d'une telle solution en comparaison avec la durée de mon stage, cette piste a vite été abandonnée ce qui nous a poussé à considérer le contrôleur du robot comme une boîte noire et atténuant l'intérêt d'utiliser certains outils qu'offre ROS comme ROS control.

La baie du contrôleur disposant de ports Ethernet et USB, l'architecture finalement retenue se base sur un programme en langage propre au robot permettant de communiquer entre l'OS du contrôleur et la couche ROS, c'est-à-dire un driver. Ce driver a pour rôle de réceptionner, d'interpréter et de transférer les informations entre l'OS du contrôleur et la couche ROS.

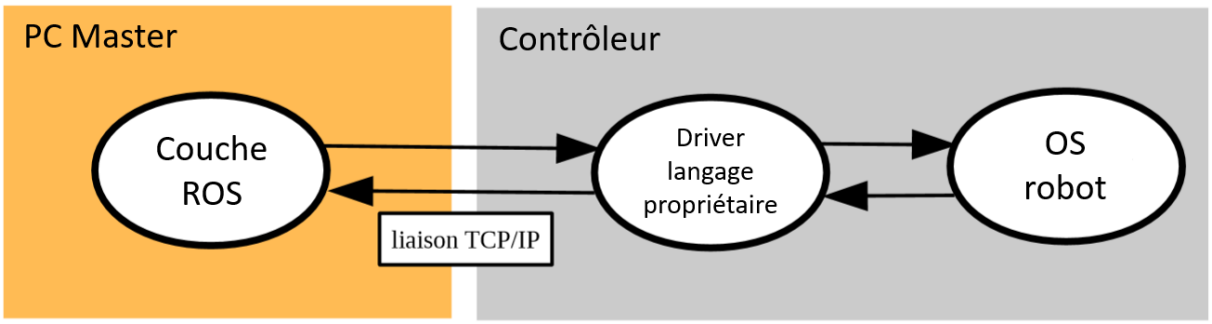



*Figure 2 - Schéma de l'architecture de la solution*

Cette architecture fait intervenir la couche ROS comme une surcouche de contrôle par l'intermédiaire d'un programme de contrôle en langage du robot classique détourné en driver. Cette solution impose donc le développement d'un driver différent pour chaque contrôleur pouvant être transfert par le biais du port USB ou Ethernet. Une fois la driver crée, cette architecture permet de ne plus dépendre des solutions d'intégration propriétaire.

# 2. Communication entre ROS et le contrôleur

Trois couches de programme doivent communiquer ensemble, la couche ROS, la couche driver et la couche OS du contrôleur. Le driver étant codé comme un programme de contrôle classique avec la solution du constructeur, la communication entre le driver et l'OS du contrôleur n'a pas besoin d'être traitée et est inclus dans la boîte noire du contrôleur. Il reste donc à définir la communication entre le driver et la couche ROS.

Les éléments de la cellule flexible de la plateforme AIP-PRIMECA Occitanie sont connectés à un parc d'ordinateur par un réseau LAN (réseau local) filaire Ethernet comme le montre la figure ci-dessous :

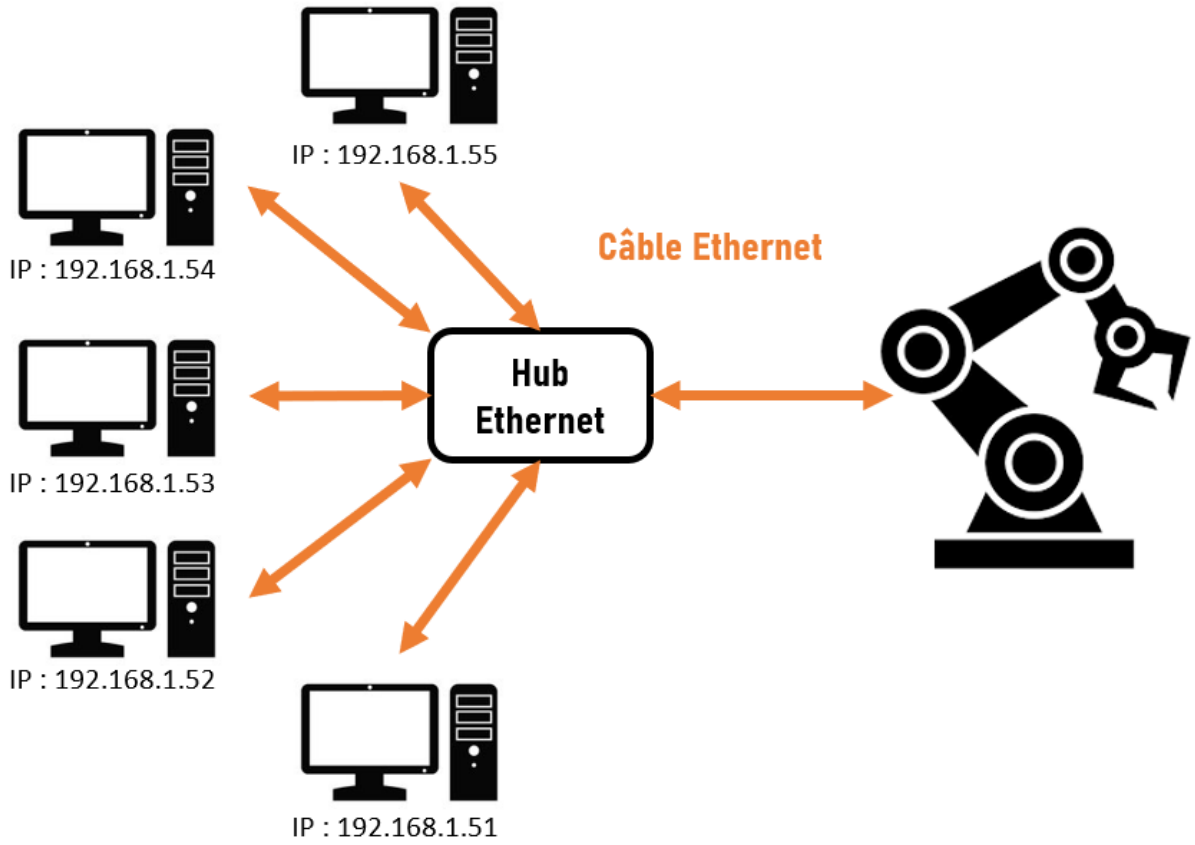



*Figure 3 - Réseau de communication*

Il est nécessaire de commencer par choisir un modèle de communication. Par soucis de standardisation et de simplicité, le choix s'est vite tourné vers le modèle TCP/IP. Le modèle TCP/IP est une norme de communication basée sur une architecture en quatre couches :

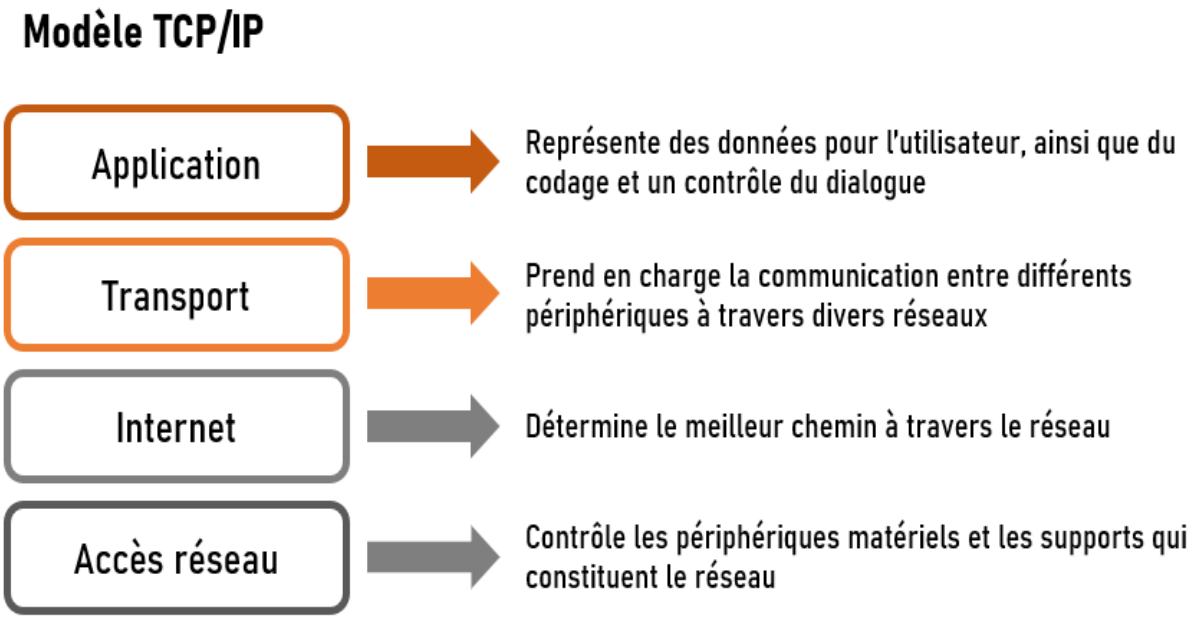



*Figure 4 - Couches du modèle TCP/IP*

Le nom de ce modèle provient du nom ses deux principaux protocoles, le protocole TCP (Transmission Control Protocol) pour la couche de transport et le protocole IP (Internet Protocol) pour la couche internet. Leur fonctionnement est le suivant :

- **Protocole TCP :**
  TCP à un premier rôle d'initialisation et de fin de communication entre deux machines par le biais de message de synchronisation et d'acquittement. Ensuite le rôle de ce protocole et de fragmenter en segment les messages de la couche application pour les transmettre à la couche internet et inversement de inversement de reconstruire en message les segments issues de la couche internet pour les transmettre à la couche application. Pour finir TCP gère le flux de données par le biais de buffer (tampon) de taille finie.

- **Protocole IP :**
  Ce protocole encapsule les données que l'on appelle alors datagramme. Le principe de cet encapsulage est d'ajouter un en-tête aux données permettant leur transport notamment à partir de l'adresse IP des machines.

Ce modèle est basé sur la communication d'un serveur et de clients. La place du serveur doit donc être déterminer. Ici le critère de simplifier l'utilisation de la solution est déterminant. Le serveur doit ainsi être intégré dans la couche ROS pour ne pas que le fonctionnement des processus de cette dernière ne soit affecté lorsque le robot n'a plus la possibilité de communiquer. L'architecture de communication utilisée entre la couche ROS et le contrôleur du robot est alors sous la forme suivante :

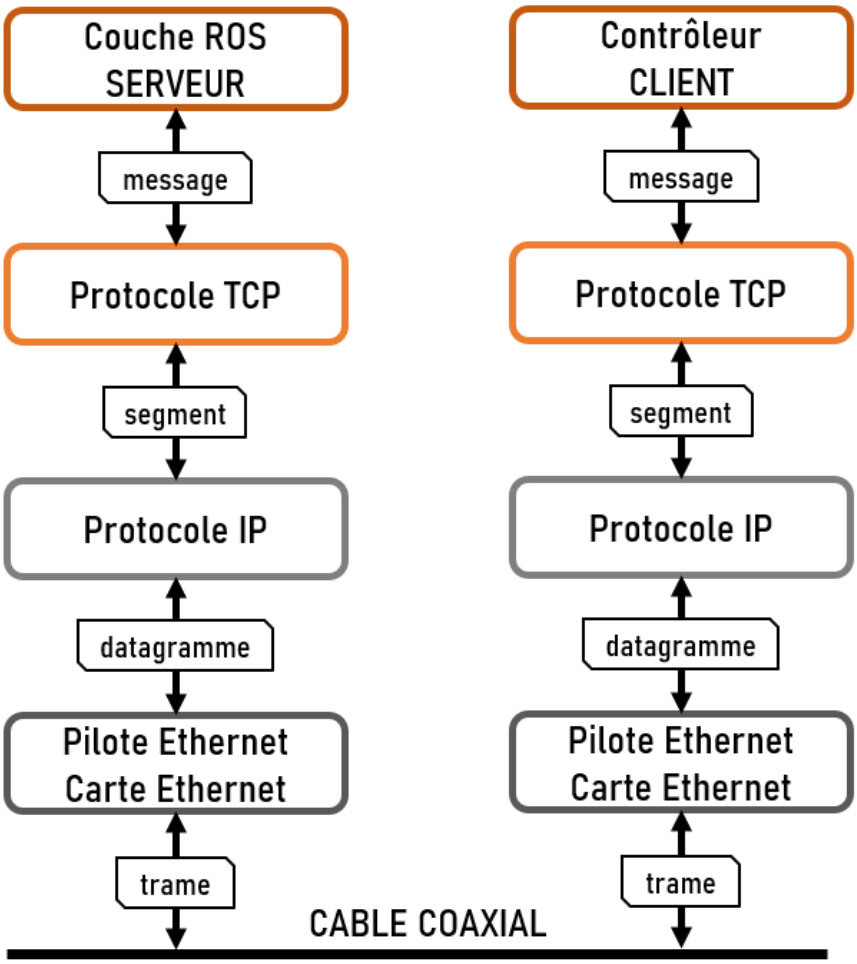



*Figure 5 - Architecture de communication de la solution*

# Fonctionnement de la solution

## 1. Protocole de communication

Les deux couches applicatives communiquent entre elles via des messages de différents types contenant plusieurs données. Il est donc nécessaire d'identifier les différents types de message, de marquer la fin d'un message et de marquer la fin des différentes données qui composent un message. La structure des messages utilisés pour la communication est la suivante :

Id X donnée X donnée X … donnée X F

Avec :

- Id : l'identifiant du message décrivant son contenu
- X : le marqueur de fin de valeur
- F : le marqueur de fin de message

Les messages utilisés ont été déterminé par l'étude des informations que peut recevoir et communiquer le contrôleur du robot. Ils peuvent ainsi être regroupés en six types différent :

- **Message de mouvement :** contient la consigne pour les différents types de mouvement (point à point, linéaire et circulaire)
- **Message de paramètres de mouvement :** contient les paramètres de vitesse et de lissage
- **Message dédié à l'outil :** contient la consigne de contrôle de l'outil ainsi que ces paramètres
- **Message de configuration du robot :** contient la consigne de configuration (Annexe 3) et de la vitesse moniteur
- **Message d'état du robot :** contient les informations de l'état du robot réel
- **Message stop :** averti la fermeture d'une application

Les informations de ces différents messages sont détaillées dans les tableaux suivants :

| | ID | Description |
|---|---|---|
| Mouvement | 11 | Coordonnée articulaire pour mouvement point à point |
| | 121 | Coordonnée cartésienne pour mouvement point à point |
| | 122 | Coordonnée cartésienne pour mouvement linéaire |
| | 1231 | Coordonnée cartésienne du point d'arrivée d'un mouvement circulaire |
| | 1232 | Coordonnée cartésienne du point intermédiaire d'un mouvement circulaire |
| Paramètres mouvement | 2 | Consigne de vitesse et de lissage du mouvement |
| Outil | 31 | Consigne des paramètres de l'outil |
| | 32 | Commande de l'action de l'outil |
| Configuration robot | 41 | Consigne de la configuration des articulations du robot |
| | 42 | Consigne de la vitesse moniteur du robot |
| Etat | 5 | Retour de l'état du robot |
| Stop | 99 | Consigne pour vider la pile de consigne du contrôleur |

*Tableau 4 - Identifiants des différents messages*

| ID | Message |
|---|---|
| 11 | 11 X j1 X j2 X j3 X j4 X j5 X j6 X F |
| 121 | 121 X x X y X z X rx X ry X rz X F |
| 122 | 122 X x X y X z X rx X ry X rz X F |
| 1231 | 1231 X x X y X z X rx X ry X rz X F |
| 1232 | 1232 X x X y X z X rx X ry X rz X F |
| 2 | 2 X accel X vel X decel X tvel X rvel X blend X leave X reach X F |
| 31 | 31 X x X y X z X rx X ry X rz X otime X ctime X F |
| 32 | 32 X action X F |
| 41 | 41 X shoulder X elbow X wrist X F |
| 42 | 42 X speed X F |
| 5 | 5 X isPowered X isCalibrated X workingMode X esStatus X speed X shoulder X elbow X wrist X isSettled X F |
| 99 | 99 X F |

*Tableau 5 - Architectures des différents messages*

| Donnée | Définition | Unité ou valeurs |
|---|---|---|
| **j1 / j2 / j3 / j4 / j5 / j6** | Position Joint axes | degré |
| **x / y / z** | Translation sur axes | millimètre |
| **rx / ry / rz** | Rotation autour axes | degré |
| **accel** | Accélération articulaire maximale autorisée | % de l'accélération nominale du robot |
| **vel** | Vitesse articulaire maximale autorisée | % de la vitesse nominale du robot |
| **decel** | Décélération articulaire maximale autorisée | % de la décélération nominale du robot |
| **tvel** | Vitesse maximale autorisée de translation du centre outil | millimètre par seconde |
| **rvel** | Vitesse maximale autorisée de rotation de l'outil | degré par seconde |
| **blend** | Mode de lissage | 0 → pas de lissage<br>1 → lissage articulaire<br>2 → lissage cartésien |
| **leave** | Distance où commence le lissage | millimètre |
| **reach** | Distance où finit le lissage en | millimètre |
| **otime** | Délai d'ouverture de l'outil | seconde |
| **ctime** | Délai de fermeture de l'outil | seconde |
| **action** | Contrôle de l'outil | 0 → ouverture de l'outil<br>1 → fermeture de l'outil<br>2 → attente de fin mouvement de l'outil |
| **shoulder** | Configuration de l'épaule | 0 → configuration de l'épaule droite imposée<br>1 → configuration de l'épaule gauche imposée<br>2 → changement configuration épaule interdit<br>3 → configuration de l'épaule libre |
| **elbow** | Configuration du coude | 0 → configuration du coude positive imposée<br>1 → configuration du coude négative imposée<br>2 → changement configuration coude interdit |

| | | |
|---|---|---|
| | | 3 → configuration du coude libre |
| **wrist** | Configuration du poignet | 0 →configuration du poignet positive imposée<br>1 →configuration du poignet négative imposée<br>2 →changement configuration poignet interdit<br>3 →configuration du poignet libre |
| **speed** | Vitesse moniteur | % de la vitesse nominale du robot |
| **isPowered** | État d'alimentation du bras | 0 → hors tension<br>1 → sous puissance |
| **isCalibrated** | État du calibrage du robot | 0 → au moins un axe du robot n'est pas calibré<br>1 → tous les axes du robot sont calibré |
| **workingMode** | Mode de fonctionnement courant du robot | 0 → invalide ou en transition<br>1 → manuel<br>2 → test<br>3 → local<br>4 → déporté |
| **esStatus** | État du circuit d'arrêt d'urgence | 0 → pas d'arrêt d'urgence<br>1 → fin arrêt urgence, attente de validation<br>2 → arrêt d'urgence ouverture |
| **isSettled** | État du mouvement du robot | 0 → position du robot non stabilisée<br>1 → le robot est arrêté |

*Tableau 6 - Données contenues dans les différents messages*

# 2. Driver Val3

Le driver Val3 est le driver dédié à l'utilisation du robot Stäubli RX60 piloté par le contrôleur CS8 dont le langage de programmation est le Val3 qui est un langage de haut niveau. Ce langage est propre à la solution propriétaire proposée par le constructeur Stäubli. Il repose sur des applications composées de fonctions de bases propre à ce langage, de programmes qui sont une séquence d'instructions VAL3 à exécuter, de variables et de tâches. Une tâche est un programme en cours d'exécution qui peut fonctionner en parallèle d'autres tâches de manière synchrone ou asynchrone [ 4 ]. Dans le cas du projet, l'application Val3 crée (le driver Val3) est une application dont les tâches sont dédiées à la communication avec la couche ROS en utilisant le protocole de message précédemment présenté. L'architecture du driver est donc la suivante :

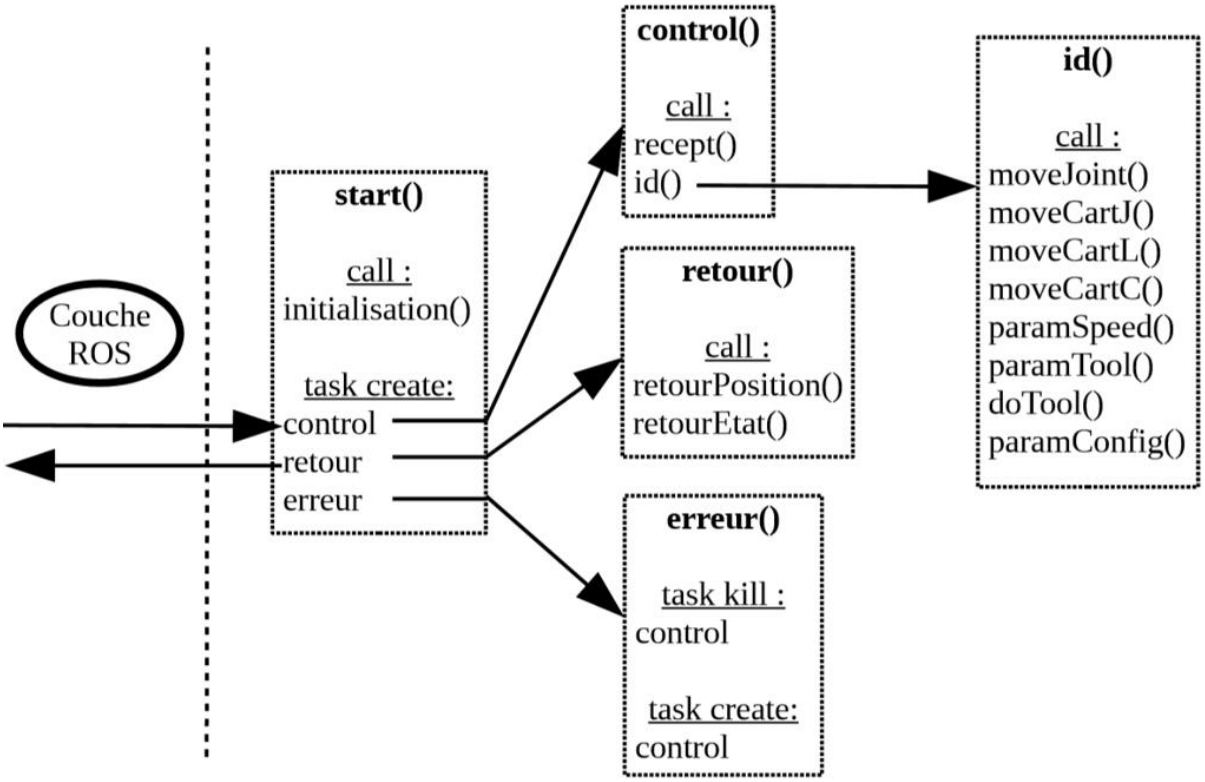



*Figure 6 - Architecture du driver Val3*

Cette application est composée de trois tâches, control, retour et erreur respectivement composées d'un programme du même nom. Ces tâches sont asynchrones et n'ont pas la même priorité. La priorité est argument d'une tâche correspondant au nombre de ligne exécutée de la tâche avant de passer à une autre tâche dans le cas d'instructions non bloquantes.

- **Tâche control :** Réceptionne les messages envoyés par la couche ROS et les traite
  → Priorité = 20

- **Tâche retour :** Lis les informations du robot réel et les envois à la couche ROS
  → Priorité = 50
- **Tâche erreur :** Traite les erreurs bloquantes mineures pour éviter une intervention de l'opérateur
  → Priorité = 1

La tâche retour possède la priorité la plus grande car le retour de l'état du robot est primordial à la couche ROS pour pouvoir adapter son contrôle sur le robot. Inversement la tâche erreur possède une priorité très faible car elle n'intervient que lorsque qu'une instruction est bloquante, il est donc inutile de l'appeler volontairement.

Ces tâches s'appuient sur différents programmes et différentes variables présentées ci-dessous :

| Type | Définition du type | Variables | Définition de la variable |
|---|---|---|---|
| sio | Entrées-sorties sur liaison série et socket Ethernet | siCo | Entrée-sortie socket Ethernet pour la communication avec la couche ROS |
| dio | Entrées-sorties numériques | diOutil | Entrée-sortie pour le contrôle de l'outil |
| config | Configurations du robot | cConfig | |
| tool | Outils montés sur un robot | tTool | |
| mdesc | Paramètres de déplacement du robot | mSpeed | |
| Joint | Positions des axes du robot | jJoint | Consigne position articulaire |
| | | jRetour | Mesure position articulaire réel |
| point | Positions cartésiennes d'un outil | pCart | Consigne position cartésienne |
| | | pInterm | Positon cortésienne du point intermédiaire pour mouvement circulaire |
| | | pRetour | Mesure position cartésienne réel |

*Tableau 7 - Variables du driver Val3*

Le but est de rendre l'application de le modulaire possible afin qu'elle puisse traiter la majorité des cas de contrôle. L'intérêt de se limiter au maximum dans le nombre de variables crées et de ne pas chercher à créer une variable pour chaque cas de figure (qui serait beaucoup trop long) mais de mettre à jour le plus souvent possible une même variable traitant la majorité des cas.

| Nom | Définition | Variable(s) utilisée(s) | Id message(s) utilisé(s) |
|---|---|---|---|
| **start** | Appelle dans un premier temps le programme initialisation et dans un second temps lance les 3 tâches parallèles control, retour et erreur | | |
| **initialisation** | Test l'écriture et la lecture de message sur la liaison de communication avec la couche ROS puis créé cette liaison | siCo | |
| **control** | Appelle successivement les programmes recept et id dans une boucle infinie | | |
| **recept** | Traduit les données reçues en format ASCII en message de format string | siCo | |
| **id** | Répartit les messages transmis par le programme recept() dans les différents programmes de contrôle en fonction de l'identifiant | | 42<br>99 |
| **moveJoint** | Commande un mouvement articulaire à partir d'une position articulaire reçues | jJoint<br>tTool<br>mSpeed | 11 |
| **moveCartJ** | Commande un mouvement articulaire à partir d'une position cartésienne reçues | pCart<br>tTool<br>mSpeed | 121 |
| **moveCartL** | Commande un mouvement linéaire à partir d'une position cartésienne reçues | pCart<br>tTool<br>mSpeed | 122 |
| **moveCartC** | Commande un mouvement circulaire à partir d'une position cartésienne reçues et d'une position cartésienne intermédiaire | pCart<br>pInterm<br>tTool<br>mSpeed | 1231<br>1232 |
| **paramSpeed** | Actualise les paramètres de mouvement | mSpeed | 2 |
| **paramTool** | Actualise les paramètres de l'outil | tTool | 31 |
| **doTool** | Commande l'outil à partir des données reçues ou ordonne l'attente de fin de mouvement de l'outil | diOutil | 32 |
| **paramConfig** | Actualise les paramètres de configuration de l'épaule, du coude et du poignet | cConfig | 41 |
| | | | |

| | | | |
|---|---|---|---|
| **retour** | Appelle successivement les programmes retourPosition et retourEtat dans une boucle infinie | | |
| **retourPosition** | Mesure la position cartésienne de l'outil et articulaire du robot, créé en format string et envoie le message contenant ces informations à la couche ROS | pRetour<br>jRetour<br>siCO | 11<br>121 |
| **retourEtat** | Lis l'état du robot, crée en format string et envoie le message à la couche ROS | siCO | 5 |
| **erreur** | Tue et recréé la tâche control quand l'erreur de position inaccessible pour le robot apparaît | | |

*Tableau 8 - Données contenues dans les différents messages*

Dans la tâche control le driver ne traite qu'un message à la fois, c'est-à-dire que lors de la réception d'un nouveau message, il le traduit, l'interprète et réceptionne le message suivant que lorsque l'instruction a été envoyé à l'OS du contrôleur. Le robot mettant un temps non négligeable pour réaliser une action en comparaison avec le temps de traitement d'un message, les instructions d'actions s'accumule dans une pile de consigne de l'OS permettant au contrôleur de faire exécuter les consignes au robot tout en tenant compte des consignes futures.

# 3. Couche ROS

C'est au niveau de la couche ROS que le contrôle et la supervision du robot industriel sont réalisés. Cette couche doit donc avoir un caractère générique pour pouvoir être utilisable sans modification pour différents robots. Son rôle est de communiquer avec le driver préalablement programmé du robot visé par l'intermédiaire d'un serveur qu'elle héberge, récupérer les informations du robot et le contrôler en fonction de ces informations. L'architecture de cette couche s'appuis ainsi sur trois nœuds, un premier nœud dédié au serveur, un autre nœud servant à décrire l'état du robot réel et un dernier nœud utilisé pour réaliser des applications de contrôle du robot. Cette architecture est schématisée ci-dessous :

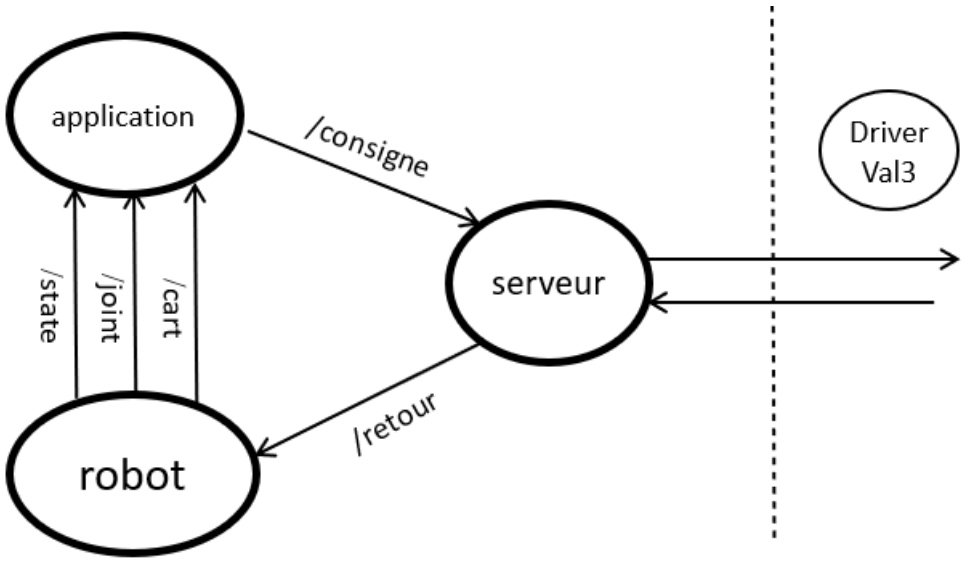



*Figure 7 - Architecture de la couche ROS*

Ces nœuds sont contenus dans la package *ros_arm* crée pour le développement de la couche ROS de la solution et communiquent entre eux via les topics suivants :

| Topic | Type | Message |
|---|---|---|
| **/retour** | std_msgs/String | string data |
| **/cart** | ros_arm/Cart | double x<br>double y<br>double z<br>double rx<br>double ry<br>double rz |
| **/joint** | ros_arm/Joint | double j1<br>double j2<br>double j3<br>double j4 |

| | | |
|---|---|---|
| | | double j5<br>double j6 |
| **/state** | ros_arm/State | int32 isPowered<br>int32 isCalibrated<br>int32 workingMode<br>int32 esStatus<br>int32 moniteurSpeed<br>int32 shoulder<br>int32 elbow<br>int32 wrist<br>int32 isSettled |
| **/consigne** | std_msgs/String | string data |

*Tableau 9 - Topics de la couche ROS*

Les messages des topics /retour et /consigne possédant une structure simple, le type utilisé provient de la libraire standard des messages de ROS. Les autres messages possédant une structure plus complexe et étant adaptée à la solution développée, il a été nécessaire de les déclarer dans le package *ros_arm.*

## 3.1. Nœud serveur

Le nœud serveur est un nœud programmé en langage python (par soucis de simplicité de codage) qui n'est dédié qu'à l'initialisation et la création du serveur de communication et au transfert de message entre les autres nœuds et le driver.

Le serveur doit donc gérer deux types de communication différent, une communication publisher/suscriber avec les autres nœuds de la couche ROS et une communication TCP/IP avec le driver du contrôleur du robot. Pour se faire il fonctionne par l'intermédiaire de deux threads qui sont des processus s'exécutant en parallèle. Ces deux threads sont lancés dans la fonction main() et fonctionne de la façon suivante :

- Le premier thread exécute la fonction reception() qui souscrit au topic /consigne et réceptionne les messages de ce topic dans une fonction callback()
- Le deuxième thread exécute la fonction serveur() qui attends la connexion d'un client (ici le driver du contrôleur) avant de communiquer avec lui par l'envoie des messages réceptionnés dans le premier thread et par la réception des messages envoyés par le client. Les messages du client sont ensuite publiés sur le topic /retour.

Le nœud serveur est donc un nœud publisher et suscriber.

Le serveur est mono client, c'est-à-dire qu'une fois que la connexion est établie un client, d'autres client ne peuvent pas s'y connecter ce qui empêche la réception de donnée parasite. Le serveur gère aussi une perte de connexion proprement en fermant et en relançant la connexion pour attendre la reconnexion du client.

## 3.2. Nœud robot

Le nœud robot a pour but de d'extraire et de traduire les informations sur le robot des messages envoyés par le driver et de mettre ces informations à disposition du nœud application de façon structuré et exploitable. Ce nœud est programmé avec une vision objet lui permettant de structurer l'exécution de ses deux rôles. Ayant appris la programmation objet avec le langage C++, j'ai utilisé ce dernier pour programmer ce nœud.

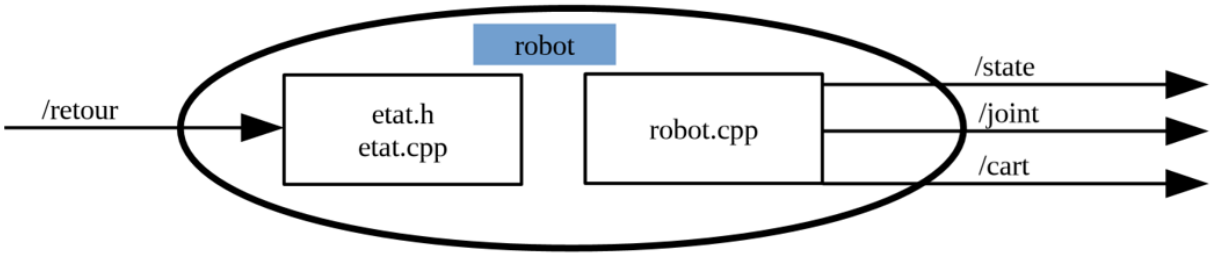



*Figure 8 - Architecture du nœud robot*

Ce nœud est composé d'une classe Etat et d'un programme principal robot dont leur rôle est le suivant :

- **Classe Etat :** souscrit au topic /retour puis réceptionne, traduit et met à jour les informations du robot dans des variables publiques via son unique fonction callback()
- **Programme robot :** initialise et crée le nœud robot puis publie sur le topic correspondant les valeurs des variables de l'objet *etat* de la classe Etat.

Ainsi lors de la création de l'objet *etat* par le programme robot, les informations du robot sont mises à jour et mis à la disposition des autres nœuds de la couche ROS.

## 3.3. Nœud application

Pour les mêmes raisons que le nœud robot, le nœud application est programmé en langage C++. Ce nœud est le siège de la programmation d'une application pour un utilisateur de la solution. Il est composé d'une classe Recept, d'une classe Control et d'un programme principale application.

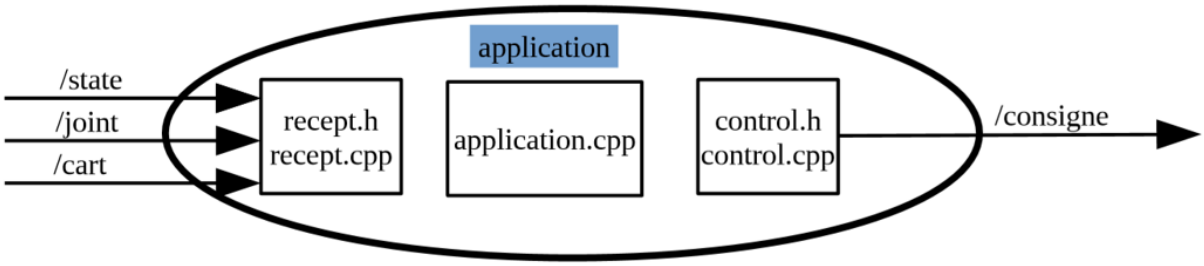



*Figure 9 - Architecture du nœud application*

Le fonctionnement de ces différents éléments est le suivant :

- **Classe Recept :** permet de souscrire à l'un des trois topics publiés par le nœud robot et met à jour ses variables publiques (qui sont les même que la classe Etat) avec les informations reçues.
- **Classe Control :** contient toutes les fonctions de contrôle du robot utilisables dans le programme application. Ces fonctions prennent en argument les valeurs d'une consigne qu'elles organisent en message de type string pour ensuite le publier sur le topic /consigne. Chaque fonction correspond à un message différent :

| Fonction | Message publié |
|---|---|
| moveJoint | 11 X j1 X j2 X j3 X j4 X j5 X j6 X F |
| moveCartJ | 121 X x X y X z X rx X ry X rz X F |
| moveCartL | 122 X x X y X z X rx X ry X rz X F |
| moveCartC | 1231 X x X y X z X rx X ry X rz X F<br>1232 X x X y X z X rx X ry X rz X F |
| paramSpeed | 2 X accel X vel X decel X tvel X rvel X blend X leave X reach X F |
| paramTool | 31 X x X y X z X rx X ry X rz X otime X ctime X F |
| doTool | 32 X action X F |
| paramConfig | 41 X shoulder X elbow X wrist X F |
| setMoniteurSpeed | 42 X speed X F |
| stop | 99 X F |

*Tableau 10 - Fonctions de la classe Control*

- **Programme application :** initialise et crée le nœud robot puis crée les objets nécessaires à la programmation de l'application pour le robot (objet *state* de la classe Recept, objet *joint* de la classe Recept, objet *cart* de la classe Recept, objet *control* de la classe Control). Un espace de ce programme est dédié à l'utilisateur de la solution pour déclarer les paramètres de l'application et programmer l'application par appel des fonctions des objets.

## 3.4. Planificateur de mouvement MoveIt

Afin d'intégrer à la solution l'outil de planification de mouvement MoveIt que propose ROS, j'ai utilisé le package **rx60_moveit_config** développé par l'université de

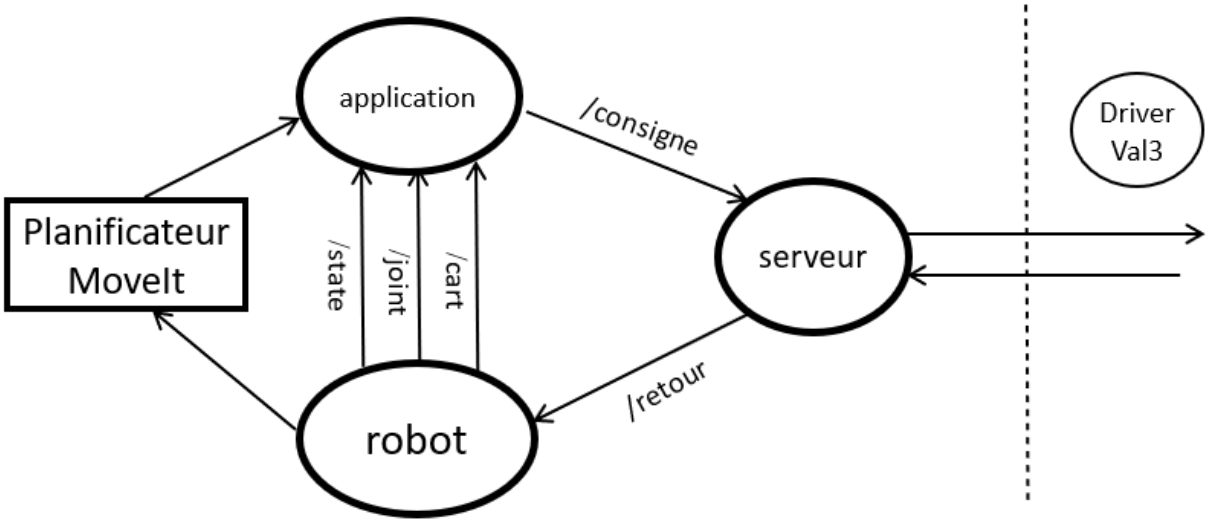



*Figure 10 - Place du planificateur dans la solution*

Kaiserslautern pour la configuration de MoveIt au robot Stäubli RX60. Le planificateur a été intégré à la solution de la manière suivante :

Aux vues de la complexité du fonctionnement du planificateur de mouvement, ce dernier a été considéré comme une boîte noire. L'objectif de son intégration dans la solution a donc été de trouver les bons topics d'entrée et de sortie afin de les connecter aux bons nœuds de la solution.

Le planificateur MoveIt fonctionne avec les coordonnées articulaires du robot. Quand il planifie un mouvement, il discrétise la trajectoire entre la position articulaire de départ du robot et la position articulaire de consigne en un nombre de position articulaire de passage variant selon la complexité du mouvement.

Suite à l'étude des topics utilisés par le planificateur grâce aux outils rqt_topic et rqt_graph de ROS qui permettent respectivement d'analyser les topics actifs en temps réel et l'architecture graphique des nœuds et des topics actifs, il est apparue que le planificateur récupère la position articulaire du robot réel via le topic */joint_states* du type sensor_msgs /JointState et que la trajectoire composée des positions articulaire de passage est publiée lors de l'exécution sur le topic */execute_trajectory/goal* du type moveit_msgs/JointTrajectory.

Les nœuds robot et application doivent donc être adaptés pour communiquer avec le planificateur. Le nœud robot publie alors les coordonnées articulaires du robot à la fois sur le topic */joint* et */joint_states* via le programme robot. Pour le nœud application, la classe Recept peut également souscrire au topic

*/execute_trajectory/goal* et une application dédiée à l'exécution du mouvement planifié est programmé. Cette application appelle successivement pour chaque position de passage la fonction de mouvement point à point en consigne articulaire *moveJoint* en paramétrant un lissage actif.

# 4. Synchronisation des données

Lors des premiers essais de la solution un problème de synchronisation des données a été mis en évidence. Lorsqu'une application était lancée, le contrôle accumulait de plus en plus de latence jusqu'à la saturation du buffer de la liaison TCP/IP faisant interrompre la connexion. Après la mesure du temps de traitement d'un message par le driver et la fréquence d'envoi de message par la couche ROS, il s'est avéré que la couche ROS envoyait les messages beaucoup plus rapidement que le temps de traitement du driver, faisant accumuler les consignes dans le buffer de la liaison TCP/IP jusqu'à sa saturation. Il a donc été nécessaire de synchroniser les échanges de messages.

Les mesures réalisées ont révélé que le temps de traitement des messages par le driver Val3 été compris en 30ms et 100ms selon les différents messages. La première idée a été de fixer en dur la fréquence d'envois des messages de la couches ROS à 33Hz correspondant donc à la fréquence maximale de traitement des messages par le driver. Cependant cette procédure a pour effet de saccader le mouvement de robot car l'accumulation de consignes dans la pile du contrôleur ne se fait plus. De plus lorsque plusieurs consignes d'actualisation de variable sont insérées entre deux consignes d'action, le robot se retrouve en attente de consigne d'action en plein mouvement.

La méthode retenue impose de fixer en dur une fréquence d'envoi des messages de la couche ROS supérieure à la fréquence de traitement des messages du driver (choix de 50hz) afin de permettre l'accumulations des consignes d'action dans la pile du contrôleur tout en retardant l'effet de saturation du buffer de la connexion. A cela s'ajoute l'implémentation d'un « flag », c'est-à-dire d'un signal d'autorisation d'envoi par le driver à la couche ROS, permettant d'exclure tout risque de saturation. L'effet de cette méthode est l'envoie rapide des messages par la couche ROS par paquet dans des intervalles de temps définit par le driver permettant à ce dernier de traiter des messages correspondant à l'état réel du robot. Cette synchronisation entre la couche ROS et le driver est détaillée dans la figure 11.

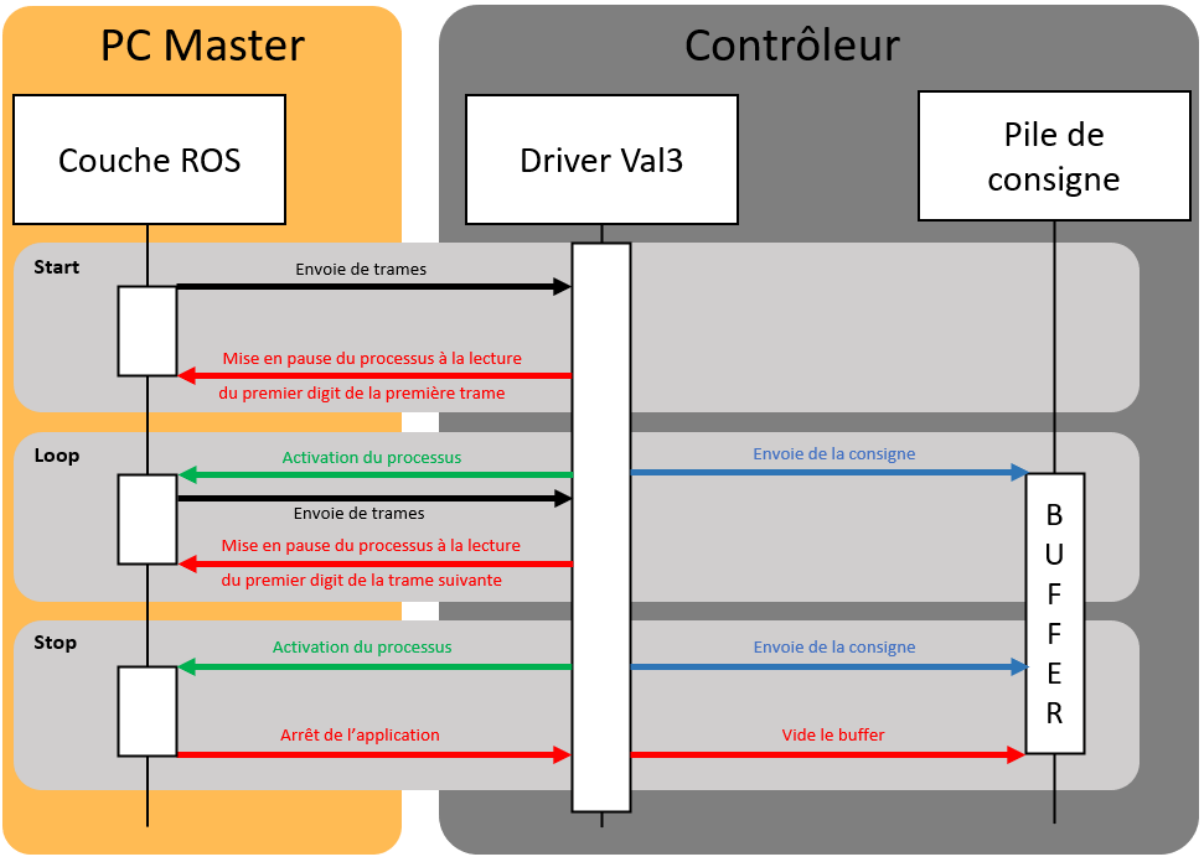



*Figure 11 - Diagramme de séquence du contrôle du robot*

# Références bibliographiques

[ 1 ] Page du projet RIO : https://www.rosin-project.eu/ftp/ros-in-occitanie-rio

[ 2 ] Site du projet européen ROS'in : https://www.rosin-project.eu/

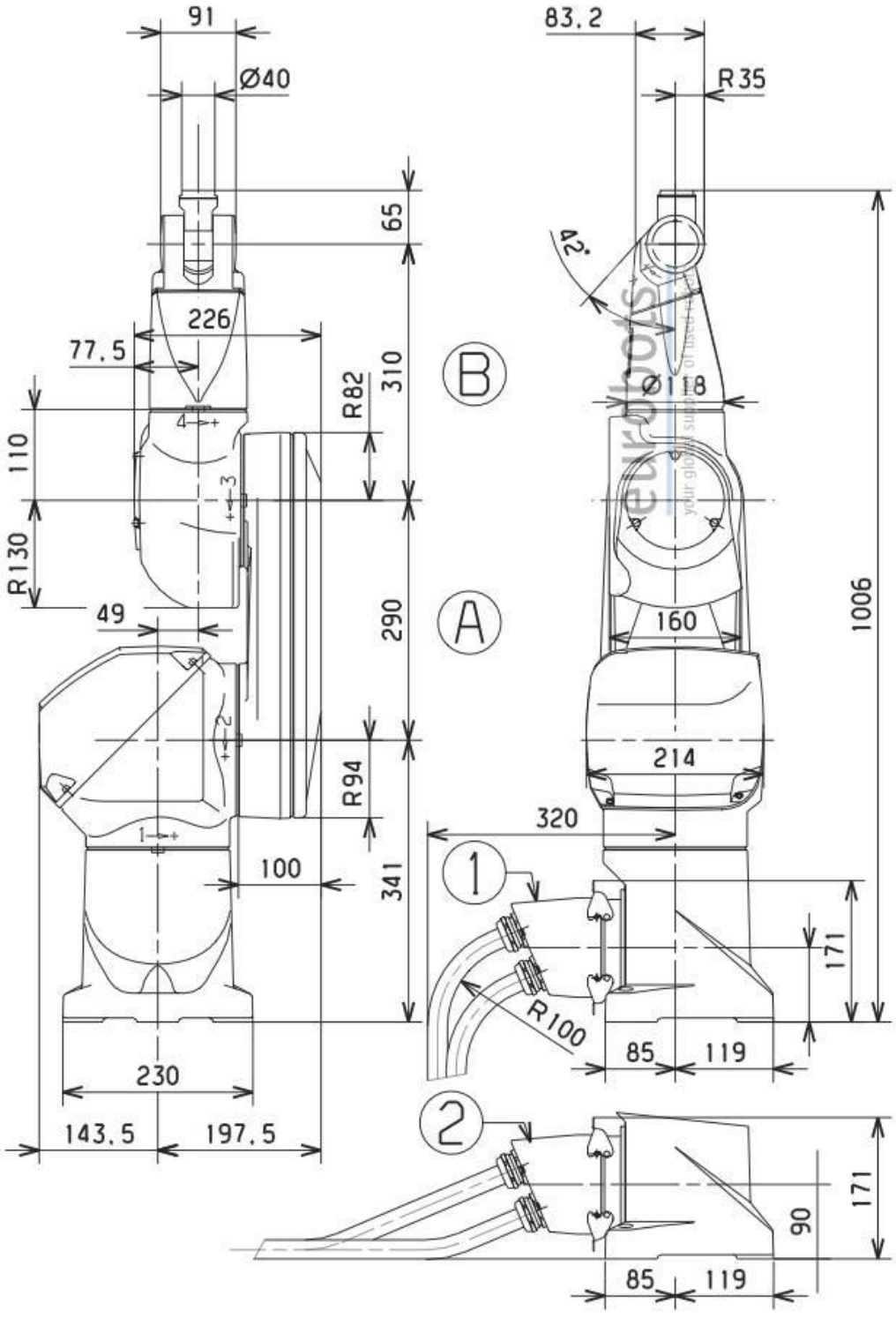
[ 3 ] *Robot Operating System*, Olivier STASSE, 2016

[ 4 ] *Manuel de Référence VAL3*, Stäubli, ver. 7, 2010

# Annexes

Annexe 1 : Module de travail de la cellule flexible

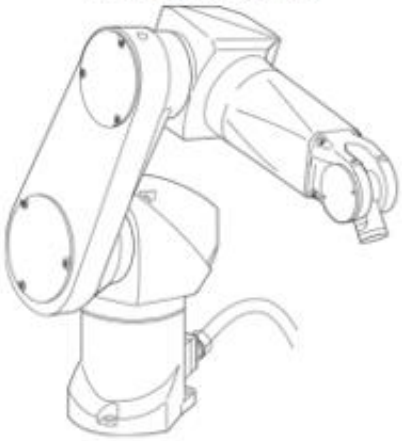
Annexe 2 : Dimensions du bras robot Stäubli RX60



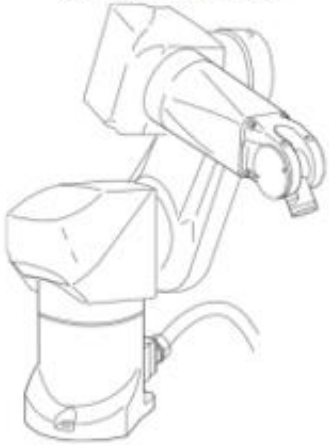
Annexe 3 : Configuration robot 6 axes

**SHOULDER CONFIGURATION :**

Configuration: righty



Configuration: lefty



**ELBOW CONFIGURATION :**

Configuration: enegative

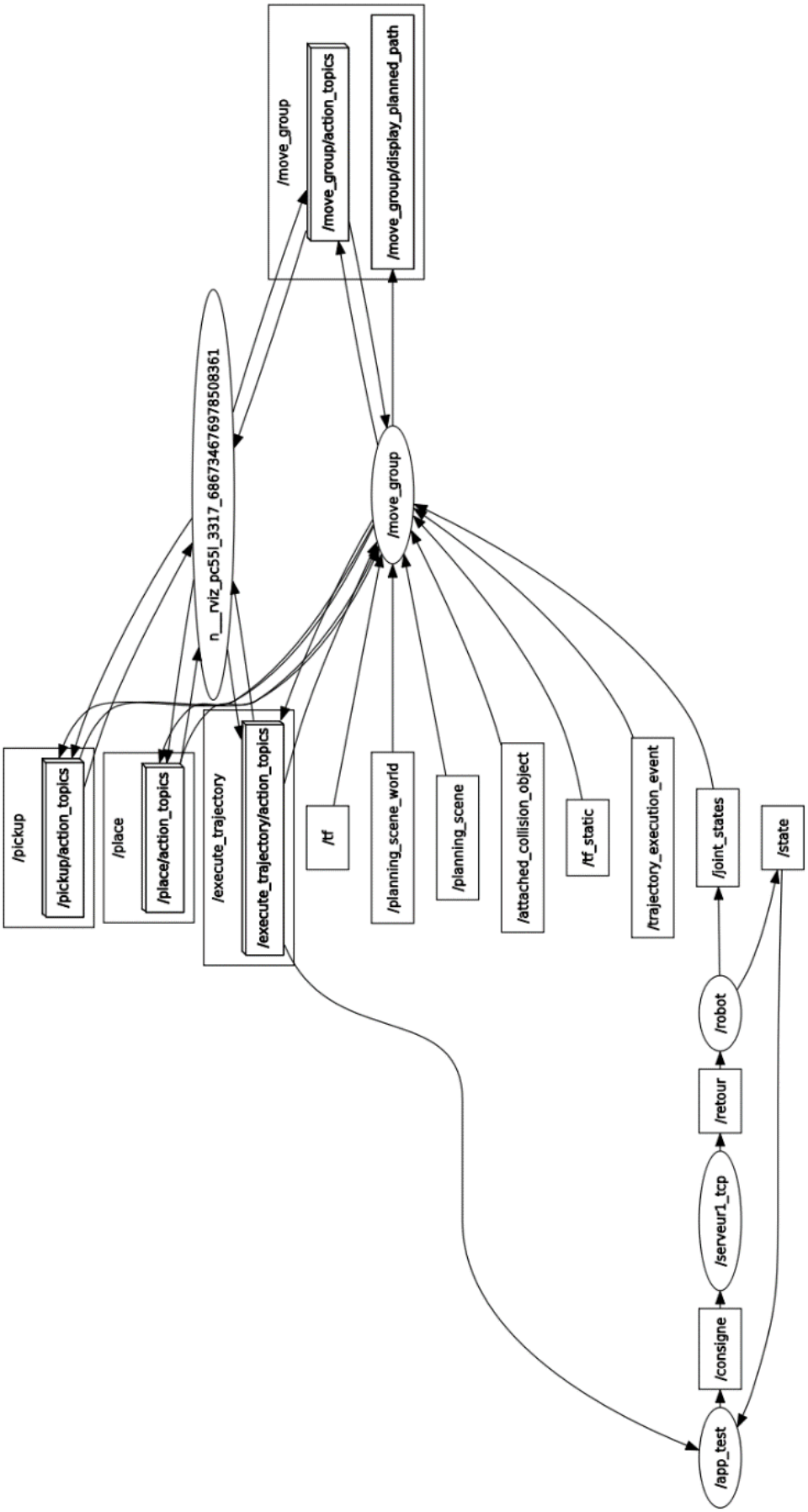
Configuration: epositive

**WRIST CONFIGURATION :**

Configuration: wnegative

Configuration: wpositive

Annexe 4 : rqt_graph de du planificateur de mouvement




RIO


ROSin