# Librairie en C/Python pour communiquer entre une Raspberry Pi et un Arduino dans le projet CASPER

Ing. J. DELVAUX
Dr ir L. JOJCZYK
ISICHt - Mons

*Ce travail de fin d'études s'inscrit dans le développement d'un robot nommé CASPER à destination d'enfants hospitalisés. Le projet a été lancé récemment par plusieurs universités barcelonaises. Ce TFE fait partie d'une preuve de concept destinée à démontrer la faisabilité de l'architecture électronique imaginée. Il se focalise sur la communication entre les deux plateformes utilisées (Arduino et Raspberry Pi) ainsi que sur l'envoi des données recueillies par les capteurs vers une base de données.*

*Mots-clefs : C, Python, Raspberry Pi, Arduino, ROS, base de données.*

*This thesis is part of a project to create a robot called CASPER to help children hospitalized. At its beginning, the goal of the proof-of-concept is to highlight the pros and cons of the electronic architecture. More precisely, it focuses on making the communication between the two electronic boards (Arduino and Raspberry Pi). The other part of this thesis is to send sensors data to a distant database.*

*Keywords : C, Python, Raspberry Pi, Arduino, ROS, database.*

# 1. Introduction

## 1.1. Contexte

CASPER (*Cognitive Assistive Social PEt Robot for Hospitalized Children*) est un projet lancé en 2015 par plusieurs universités barcelonaises[1]. L'objectif est de développer un robot nommé CASPER à destination d'enfants hospitalisés afin de les divertir et de les instruire. L'architecture électronique (deux cartes remplissant des fonctions distinctes) ayant été décidée par les responsables du projet, la première étape (preuve de concept) est de la développer, vérifier si elle est fonctionnelle, quelles sont ses limitations, ses avantages …

## 1.2. Architecture du robot CASPER

L'idée initiale du projet est de combiner deux plateformes matérielles : la première (A) fournit la puissance de calcul pour l'intelligence artificielle tandis que la seconde (B) permet d'interfacer facilement capteurs et actuateurs. De plus, le robot est également capable d'envoyer des données par wifi à une base de données distante. Celles-ci, après analyse par des psychologues, permettront d'améliorer le comportement de CASPER.

Les cartes électroniques choisies sont les suivantes :

A. Raspberry Pi 2 : de la taille d'une carte de crédit, la Raspberry repose sur un processeur ARMv7 avec 4 coeurs fonctionnant chacun à 900MHz. En plus de cette puissance de calcul, elle possède 1GB de RAM. Elle utilise un système d'exploitation basé sur Debian (Linux) nommé Raspbian.
   La Raspberry Pi était initialement destinée à pouvoir être utilisée dans des pays émergents afin de donner accès à l'informatique pour un plus grand nombre. Elle s'est vite imposée comme étant une plateforme électronique attrayante vu son faible coût. Toutefois, be base, elle lui manque certaines fonctionnalités l'empêchant d'être efficace pour des applications temps-réel.
B. Arduino UNO : bien connue des étudiants et hobbyistes, cette carte permet d'interfacer capteurs et actuateurs très rapidement. Elle bénéficie d'un grand nombre de librairies grâce à son immense communauté. De plus, une librairie Arduino pour communiquer avec les moteurs Dynamixel a déjà été développée dans le cadre d'un autre projet de l'URL. Le microcontrôleur est un Atmel 324P-PU.

[1] Universitat Politècnica de Catalunya (UPC) sous la direction de Cecilio Angulo Bahón, Universitat Autonoma de Barcelona (UAB) sous la direction de Marta Díaz Boladeras et Universitat Ramon Llull (URL) sous la direction de Jordi Albó Canals.

La Figure 1 présente le projet CASPER[2]. Celui-ci est constitué :

- Robot :
  - Raspberry Pi : elle communique avec une base de données distante et l'arduino. Elle est interfacée par un écran tactile (via une programme développé pour commander les moteurs et afficher les valeurs des capteurs) ainsi qu'une caméra (pour permettre un futur traitement d'images) ;
  - Arduino : il commande des moteurs (dans ce cas-ci des Dynamixel et moteurs DC) mais récupère des informations de toute une série de capteurs (IMU, capteur de pulsation …).
- Base de données : développée sous MySql, elle permet d'améliorer le comportement du robot après analyse.

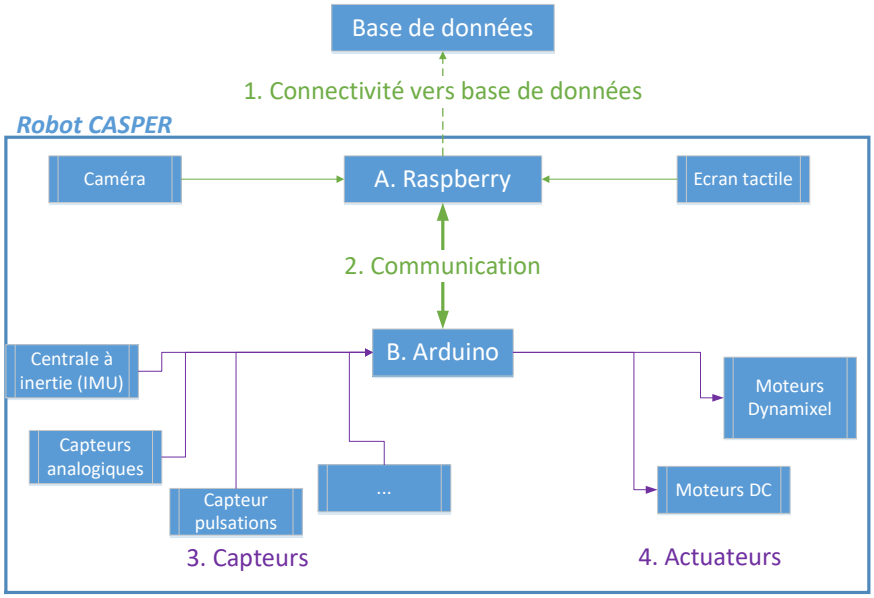


*Figure 1 : Architecture du projet CASPER*

Les algorithmes cognitifs vont être implémentés au travers de ROS sur la Raspberry Pi. ROS (*Robot Operating System*) [5] est un outil principalement destiné à la robotique. Son intérêt réside dans sa philosophie : ne pas réinventer la roue à chaque fois. Initié par le *Stanford Artificial Intelligence Laboratory* en 2007, il s'est imposé dans bon nombre de projets de par cette structure interne. Il permet de développer des algorithmes sans avoir à se soucier des composants et de la carte électronique utilisés. La portabilité du code en est son atout majeur.

[2] Tous les périphériques inclus y ont été d'une initiative personnelle afin que la prototype de preuve de concept soit le plus complet possible.

Le développement de ce projet étant relativement conséquent, le travail a été divisé en deux parties :

- La communication entre la Raspberry Pi et l'Arduino ainsi que l'envoi des informations des capteurs ont été réalisés dans ce TFE [2].
- L'interfaçage des capteurs et actuateurs avec l'Arduino a été réalisé par un autre étudiant [1].

### 1.3. Spécifications avancées

Afin que l'intégration de ces deux parties se passe bien, il était important de les rendre compatibles. Les choix suivants ont été décidés :

- L'**Arduino** est **maître** et la **Raspberry** est **esclave** car l'Arduino contrôle directement les moteurs. Afin d'éviter un comportement erratique du robot, c'est l'Arduino qui décide quand ceux-ci sont mis en marche suivant les mouvements générés par l'algorithme cognitif tournant sur la Raspberry.
- Le protocole de communication choisi entre Raspberry Pi et Arduino est l'I²C car l'adressage est facile. Toutefois, la **Raspberry** doit être **maître de la communication I²C**. En effet, le driver I²C du noyau Linux (fin 2015) ne supportait pas le mode esclave. L'Arduino interroge également en I²C des capteurs qui lui sont reliés. C'est donc une situation de multi-maîtres.

## 2. Développement de la librairie

### 2.1. Vue d'ensemble

Python est le langage de programmation choisi sur la Raspberry pour des raisons de rapidité de développement et de librairies déjà existantes[3]. Le C/C++ a été choisi pour l'Arduino car à l'heure actuelle, les outils de compilations sont bien optimisés.

La librairie présente différentes couches d'abstraction dans une optique de *modularité* : l'utilisateur/développeur peut facilement remplacer un des modules au besoin ou réutiliser l'ensemble pour un autre projet. Par exemple, deux interfaces peuvent être utilisées : l'I²C ou l'UART[4].

[3] Pypi est le gestionnaire de paquets pour python. Il permet d'accéder à pas moins de 87000 librairies développées par la communauté. Cependant, celles-ci sont portées sur l'informatique et non l'électronique. Lien : *https://pypi.python.org/pypi*.

[4] L'utilisateur peut même choisir entre l'HardwareSerial ou SoftwareSerial sur l'Arduino.

Voici une liste des principales fonctionnalités :

- Compatible avec ROS ;
- Choix de l'interface : UART ou I²C ;
- Communication bidirectionnelle : la Raspberry transmet les commandes à l'Arduino et l'Arduino envoie les données des capteurs à la Raspberry ;
- Communication avec une base de données : la Raspberry envoie des données par wifi ou Ethernet à celle-ci;
- Protection contre les erreurs de transmission : CRC[5] de 8 bits pour la longueur des données et CRC de 16 bits pour les données ;
- Gestion des erreurs ;
- Options permettant de débugger très facilement ;
- Facile à installer ;
- Modulaire et adaptable à des développements futurs.

La Figure 2 présente une vue d'ensemble de la librairie qui est composée de deux implémentations chacune composée de classes en C++ et de modules en Python :

A. Raspberry Pi(Python) :
    1. Module *Interfaces* : il gère le bas-niveau en s'occupant de mettre les chaque byte de trame sur le bus choisi et de récupérer les données reçues dans le buffer de l'interface ;
    2. Module *Framing* : il récupère les données à transmettre et les incorpore dans des trames ;
    3. Module *CRC* : il calcule le CRC de 8bits et 16bits qui vont être incorporés dans chaque trame ;
    4. Module *Communications* : il est l'interface haut-niveau avec le développeur. Ce dernier peut choisir aisément l'interface sur laquelle il veut envoyer ses données et transmettre celles-ci ;
    5. Module *Database* : il permet de facilement interagir avec la base de données en donnant des fonctions simples d'utilisation.

B. Arduino(C++) :
    1. Classe *Interfaces* : identique au module Python ;
    2. Classe CRC : cas particulier du module Python afin d'économiser de la mémoire Flash ;
    3. Classe *Communications* : regroupe les modules *Communications* et *Framing* pour des raisons d'optimisation et de gain de mémoire.

[5] Cyclic Redundancy Check.

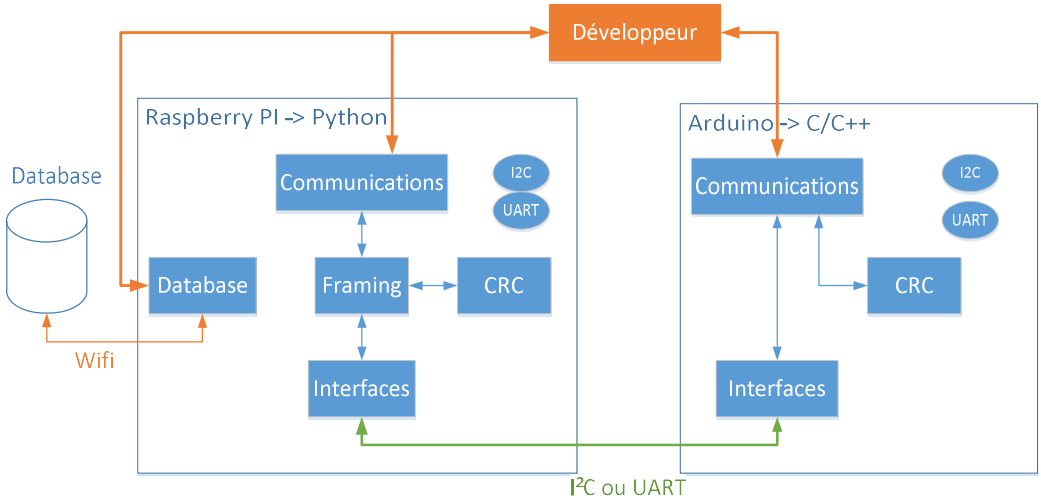


*Figure 2 : Vue d'ensemble de la librairie*

## 2.2. Formats des trames de communication

La structure de la trame est l'élément crucial dans la communication. Pour ce projet, elle est basée sur un *character-oriented protocol*[3] : des caractères de contrôles (DLE, STX, ETX[6]) permettent de savoir où sont les données. Le CRC facilite la détection des erreurs. Le CRC sur 8bits permet de protéger le nombre de bytes de données ou la commande tandis que celui sur 16bits permet de protéger les bytes de données. Deux types de messages s'échangent :

### 1. *Arduino → Raspberry : données*

| DLE | STX | Nombre de bytes de données | CRC 8 bits | Bytes de données | CRC 16 bits | DLE | ETX |
|---|---|---|---|---|---|---|---|
| 0x10 | 0x02 | 1 byte | 1 byte | x bytes | 2 bytes | 0x10 | 0x03 |

### 2. *Raspberry → Arduino : commandes*

| DLE | STX | Commande | CRC 8 bits | DLE | ETX |
|---|---|---|---|---|---|
| 0x10 | 0x02 | 1 byte | 1 byte | 0x10 | 0x03 |

[6] DLE: Data Link Escape, STX: Start of Text, ETX: End of Text

## 2.3. Envoi de données de l'Arduino à la Raspberry en UART[7]

L'envoi de données de l'Arduino à la Raspberry Pi est simple et efficace, car la Raspberry écoute en permanence sur le port de réception UART via un thread dédié. La méthode *send_datas* (de la classe *Communications*) permet d'envoyer un tableau de données. Celles-ci sont directement récupérées sur la Raspberry comme l'illustre la Figure 3.

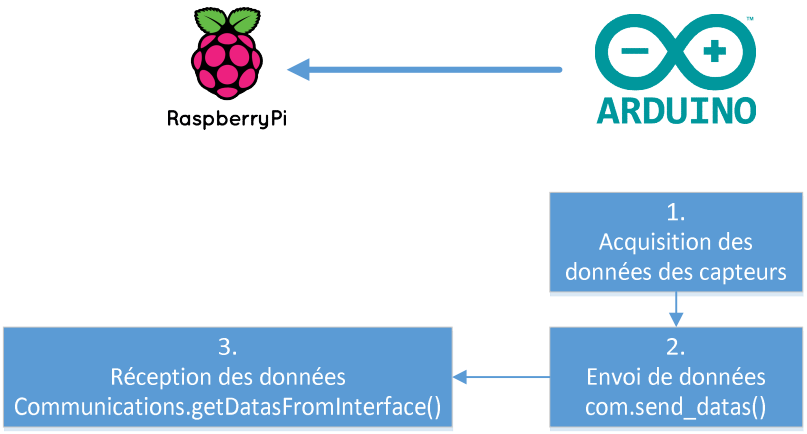


*Figure 3 : Transmission de l'Arduino à la Raspberry en UART*

## 2.4. Envoi de données de la Raspberry à l'Arduino en UART

La Figure 4 montre l'envoi de données de la Raspberry à l'Arduino. Celui-ci est plus complexe car l'Arduino est le maître de la Raspberry : l'Arduino doit autoriser la Raspberry à transmettre. Dans le cas où la Raspberry n'a rien à dire, un message spécial est envoyé afin que l'Arduino puisse effectuer d'autres tâches. Dans l'autre cas, l'arduino vient lire son buffer jusqu'au moment où elle décide qu'elle a assez de données à traiter. Dans ce cas, il fait appel à la fonction *send_raspberryStop*.

[7] À des fins de meilleure compréhension, l'envoi et la réception sont expliqués pour l'UART. Se référer au 2.5 pour plus d'informations sur les différences UART-I²C.

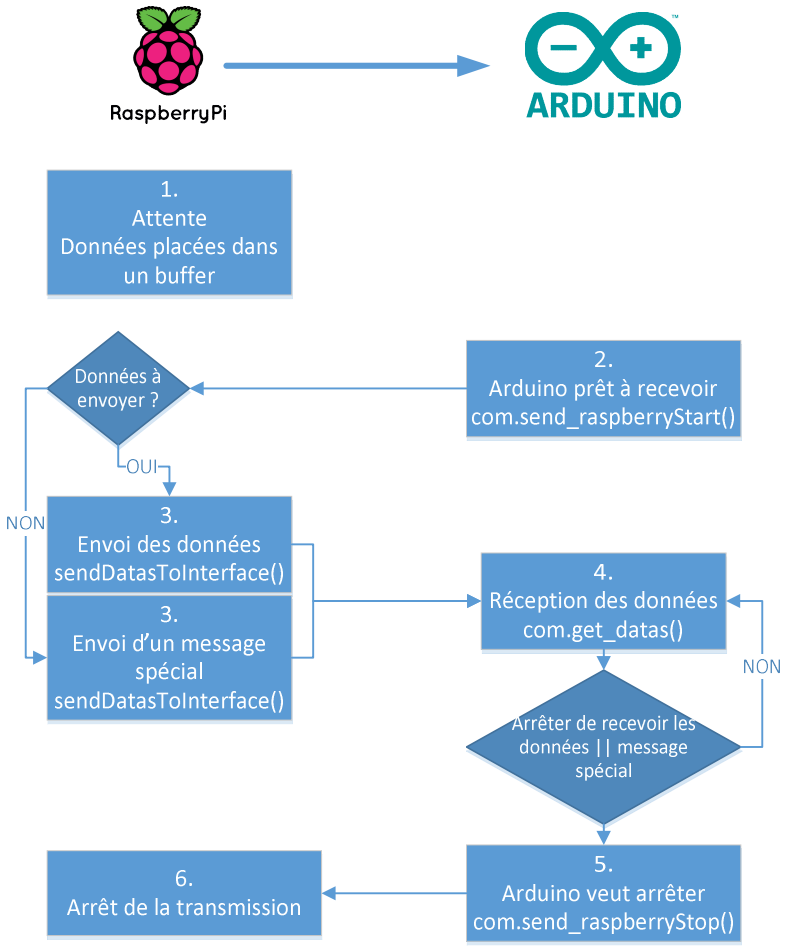


*Figure 4 : Transmission de la Raspberry à l'Arduino en UART*

## 2.5. Spécificités UART et I²C

Le schéma de la Figure 5 illustre les différences entre la manière de communiquer en I²C et en UART. La complexité inhabituelle avec l'I²C est due à une limitation de la Raspberry avec son driver linux I²C. Celui-ci ne supporte pas d'être esclave. La librairie développée tient donc compte de cette particularité.

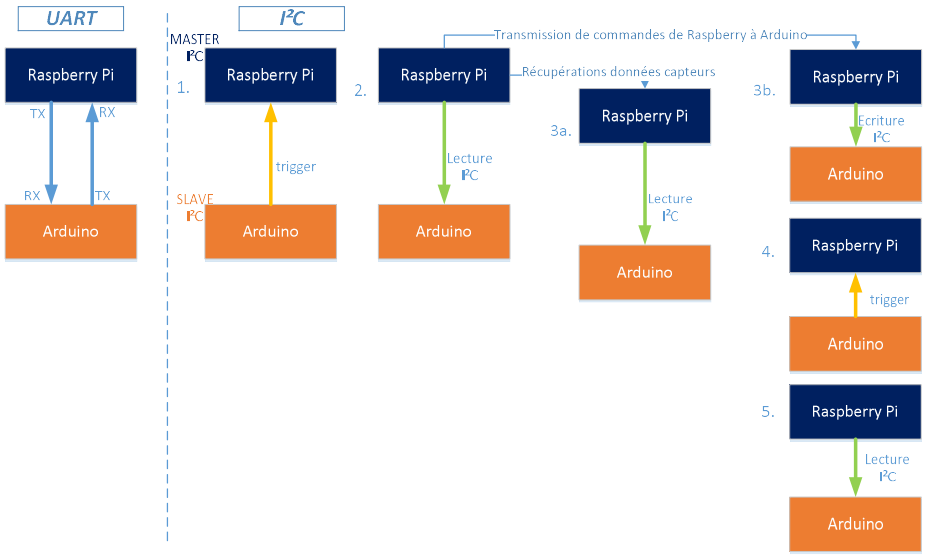


*Figure 5 : Différences de procédure de communication entre UART et I²C*

### *UART*

La procédure de communication en UART est simple et rapide. Les données sont mises sur le port TX et reçues sur le port RX. Aucune particularité n'est à noter.

### *I²C*

La solution trouvée pour faire fonctionner la Raspberry Pi en maître de la communication I²C tout en gardant l'Arduino maître est l'utilisation d'un trigger. Ce trigger correspond à une pin que vient changer l'Arduino en la passant de BAS à HAUT avant de la repasser, quelques millisecondes plus tard, à l'état BAS. La procédure est la suivante :

1. Arduino envoie un trigger à la Raspberry ;
2. Celui-ci est détecté par la Raspberry qui interroge l'Arduino en I²C;
3. Dans le cas d'une récupération d'informations de capteurs (3a), la Raspberry continue la lecture I²C de l'Arduino pour un nombre d'octets donné. Dans celui d'une transmission de commandes (3b), la Raspberry vient écrire sur le bus I²C ;
4. Lorsque l'Arduino souhaite arrêter de recevoir des commandes, il envoie à nouveau un trigger;
5. La Raspberry vient lire le bus I²C dans lequel un message de stop apparaîtra.

### 2.6. Tests et validation

Afin de valider le comportement de la librairie, quatre tests ont été développés :

1. CRC : ce test a notamment pour but de vérifier le CRC en affichant le résultat de l'opération. Celui-ci est ensuite comparé avec un outil en ligne. Il peut être également utilisé pour générer une lookup-table qui peut être comparée à un livre de référence (afin de vérifier les résultats de 0 à 255) ou copiée et implémentée dans un micro-contrôleur par exemple.
2. Robustesse : ce test s'attache à chercher des failles dans le codage/décodage des trames en s'attaquant au module *Framing*.
3. Communication : par l'exécution d'un sketch[8] spécifique sur l'Arduino, ce test permet de s'assurer que le décodage/encodage des données s'effectue correctement aussi bien sur l'Arduino que sur la Raspberry.
4. Base de données : ce test permet de vérifier la bonne configuration de la Raspberry en tentant d'accéder à la base de données.

Les tests 1, 2 et 4 sont spécifiques à la Raspberry.

## 3. Intégration de la librairie dans CASPER

### 3.1. Vue d'ensemble

Après le développement de la librairie Python, l'étape suivante était d'intégrer celle-ci dans ROS sur la Raspberry. La communication avec l'Arduino étant bidirectionnelle et l'Arduino pouvant transmettre quand il le souhaite, la Raspberry doit être à l'écoute en permanence.

La Figure 6 illustre l'intégration de cette librairie dans CASPER. Deux scripts sont dédiés à la communication avec l'Arduino. Ceux-ci utilisent des fonctions « hauts-niveaux » de la classe *Communications*.

- Le premier, *casper_down*, a un double rôle : stocker les messages à transmettre venant de l'algorithme principal et démarrer/arrêter la communication avec l'Arduino.
- Le second, *casper_up*, écoute en permanence soit le port RX de l'UART soit la pin de trigger pour l'I²C. En fonction du message reçu de l'Arduino, il l'envoie au travers de ROS au bon destinataire. Dans le cas d'une commande pour démarrer ou arrêter la transmission de la Raspberry à l'Arduino ce sera *casper_down* ; dans le cas de données ce sera l'algorithme principal.

[8] Nom donné à un programme Arduino.

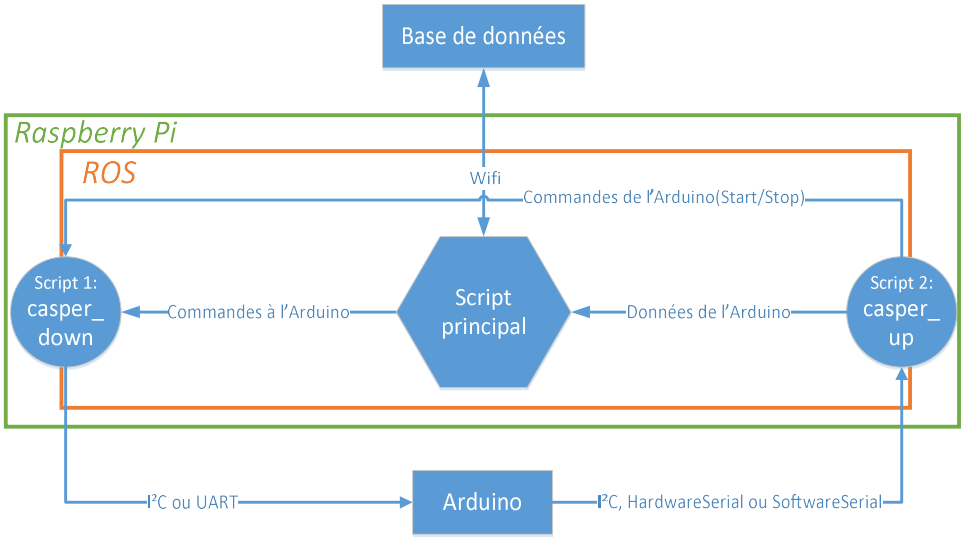


*Figure 6 : Relations entre les différents scripts dans ROS*

## 3.2. Démonstration technique

L'envie de partager ce travail et de donner un aperçu des résultats atteignables avec cette librairie s'est matérialisée par la réalisation d'une démonstration technique. Celle-ci est composée des éléments suivants (repris dans la Figure 7) :

- Une base de données qui recueille toutes les informations des capteurs. Celle-ci a été désignée expressément pour la démonstration ;
- Une caméra reliée à la Raspberry Pi ;
- Un écran tactile relié à la Raspberry. Avec cet écran et grâce à une interface graphique développée sous PyQt, l'utilisateur peut commander les moteurs et gérer l'acquisition de mesures par les capteurs. Il a également la possibilité de voir ce que filme la caméra ;
- Des capteurs et moteurs Dynamixel reliés à l'Arduino.

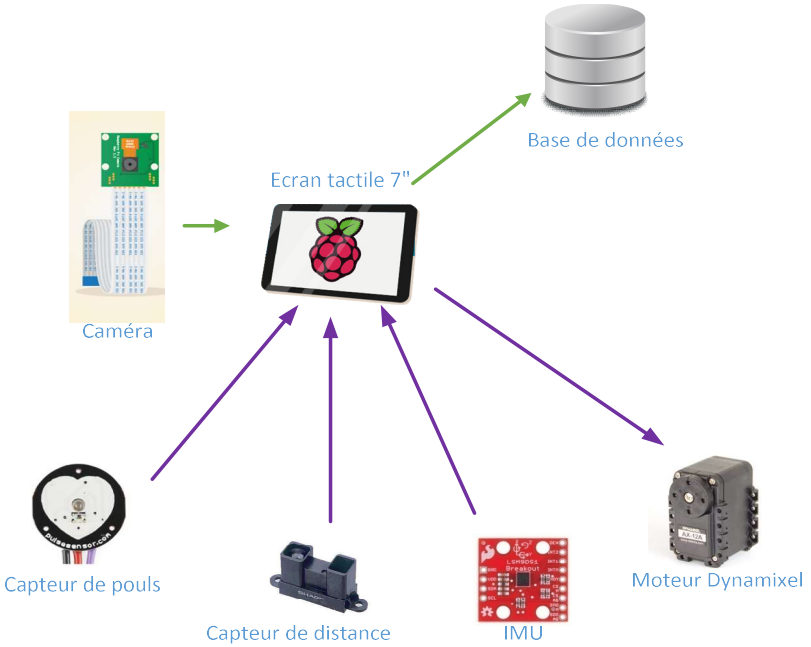


*Figure 7 : Démonstration technique*

L'Arduino, en plus de traiter les demandes effectuées par l'interface graphique (commande de moteur(s) et/ou demande d'acquisition d'un ou plusieurs capteurs), possède une fonction de feedback automatique[9]. À une fréquence définie, il interroge l'ensemble des capteurs et envoie ces informations à la Raspberry. L'algorithme utilisera ces données afin de contrôler le robot. Mais celles-ci seront également envoyées à la base de données pour permettre d'améliorer les algorithmes de CASPER.

## 3.3. Configuration et mise en place

L'installation et la mise en place de la librairie Python ainsi que la démonstration technique peuvent poser des soucis dans le cas où le développeur n'est pas familiarisé avec Linux. En effet, certaines difficultés peuvent se poser :

- Problèmes de permissions ;
- Paquets à installer ;
- Compilation de ROS et de son workspace ;
- Utilisation de l'UART, I²C ;
- …

[9] Cette fonctionnalité est décrite dans [1].

Dans ce cadre, il est important de faciliter le déploiement ainsi que la compréhension de la librairie.
Deux scripts shells ont été développés pour répondre au premier besoin. Ceux-ci permettent notamment d'installer les paquets Linux nécessaires, configurer l'accès à la base de données, créer un dossier où sont stockés les logs, changer les permissions des scripts ou encore compiler le workspace ROS.
La compréhension de la librairie est possible grâce aux schémas fournis et aux commentaires dans le code. Chaque fonction est ainsi détaillée avec son but, ses arguments et ce qu'elle renvoie.
De plus, deux fichiers textes sont présents. Le premier détaille la procédure d'installation avec les scripts ainsi que certains problèmes couramment rencontrés et leur résolution. Le second détaille chaque point de la configuration avec, par exemple, la manière de configurer l'UART, l'I²C, le wifi …

# 4. Résultats

## 4.1. Première phase de tests

La première phase de tests a concerné la validation de la librairie développée pour la communication entre la Raspberry et l'Arduino. Ces tests ont été lancés via l'environnement py.test[10] présent sur la Raspberry (et plus spécifiquement celui de communication). Voici la procédure utilisée :

1. Lancement du test sur la Raspberry ;
2. La Raspberry envoie *x* messages avec des données aléatoires à l'Arduino ;
3. L'Arduino attend ces *x* messages puis les décode ;
4. L'Arduino encode ceux-ci, les copie dans son buffer avant de les réenvoyer vers la Raspberry ;
5. Réception des messages par la Raspberry, décodage puis comparaison entre les données envoyées et celles reçues. Le test est réussi si celles-ci concordent.
6. Répétition des étapes 1 à 5 autant de fois que voulu.

L'ensemble de ces tests (aussi bien avec l'UART que l'I²C) étant concluants, l'étape suivante a été la validation du prototype grâce à la démonstration technique. Celle-ci combine le travail réalisé autour de l'Arduino ainsi que celui de ce TFE.

[10] Py.test est un module de python qui permet de réaliser des tests unitaires. Son point fort est de fournir beaucoup d'informations sur le debug bien que, dans certains cas très particuliers, il ralentit voire fait planter le script testé.

## 4.2. Test de la démonstration technique

Dès le premier essai, des problèmes sont apparus. Après une quinzaine de secondes, la Raspberry n'a plus reçu les valeurs des capteurs par la fonction de feedback. De plus, la réactivité de la Raspberry après le trigger de l'Arduino s'est avérée lente (8msec) et l'Arduino ne répondait qu'après, en moyenne, 10 tentatives de lecture I²C. Comme la source du problème était difficile à déterminer, l'investigation a été scindée en deux points :

1. Réexaminer le code de l'Arduino et comprendre pourquoi l'Arduino n'a pas répondu tout de suite à la commande de lecture I²C;
2. Tenter de comprendre pourquoi l'un des triggers n'est pas détecté par la Raspberry et d'où vient le temps ce réaction important aux triggers de l'Arduino.

## 4.3. Résultats de l'investigation avec l'Arduino

L'idée dans cette investigation a été de remplacer la Raspberry par un Arduino et d'écrire un sketch afin de simuler le comportement de la Raspberry. Le détail complet de ces résultats est consultable dans [1]. Voici néanmoins une liste succincte de ceux-ci :

- La librairie de communication pour l'Arduino fonctionne correctement ;
- L'Arduino répond immédiatement à une lecture I²C ;
- La cohabitation avec le code Arduino pour contrôler les moteurs et effectuer la lecture des capteurs ne pose aucun souci ;
- Aucun problème de communication dans le temps.

Les échecs de lecture sur le bus I²C apparaissent uniquement avec la Raspberry. Après des recherches poussées, il est apparu que le phénomène est connu sous le nom de *clock stretching*[6].
La librairie en python effectue, dans le cadre d'une lecture I²C, une écriture puis une lecture. Ce qui correspond en réalité au protocole SMBus[7] qui est un dérivé de l'I²C. L'Arduino maintient la ligne SCL à l'état bas, le temps de traiter l'instruction I²C (dans ce cas une écriture).
Cependant, la Raspberry n'est pas capable de ralentir la génération de l'horloge I²C de manière adaptée. Elle considère donc ça comme un échec de lecture, ce qui résulte par un NACK[11]. Afin de ne pas avoir ces problèmes, il ne faut donc pas utiliser la librairie standard I²C mais une autre librairie Python : Pigpio. À l'essai de cette dernière, il est apparu que les lectures I²C étaient des lectures et non des écritures/lectures.

[11] Negative acknowledgment ou not acknowledged.

## 4.4. Résultats de l'investigation avec la Raspberry

Afin de déterminer avec certitude pourquoi un des triggers n'était pas détecté par la Raspberry, l'idée a été d'écrire un script spécifique à cette tâche. Dans un premier temps, celui-ci ne faisait que compter le nombre de flancs montants. Il est apparu que dans le cas d'un grand nombre de triggers (lancés à 1-2 ms d'intervalle), la librairie la plus connue et utilisée qu'est *RPi.GPIO* s'est avérée inefficace : certains triggers n'étaient pas détectés.

Il existe pourtant deux méthodes pour réagir aux triggers :

1. La fonction bloquante qui attend le flanc correspondant pour continuer le programme : *wait_for_edge()*. Elle prend en argument la pin et le type de flanc (montant, descendant ou les deux) ;
2. La fonction de callback : une fonction est appelée à chaque fois qu'un flanc est détecté.

Aucune de ces méthodes ne fonctionne à 100% dans la détection de triggers. Toutefois, il existe, dans la librairie Pigpio[12], des fonctions du même acabit. Après quelques expérimentations, 100% des triggers envoyés ont été détectés.

L'étape suivante a été d'effectuer une lecture I²C de l'Arduino après chaque trigger. Bien que parfaitement fonctionnel avec un délai de réaction de la Raspberry aux triggers inférieurs à 500 µs, la Raspberry ratait, après une trentaine de secondes, un des triggers.

Le problème ne pouvait venir que de l'ajout de cette lecture I²C. Une des solutions aurait pu être l'utilisation d'un thread dédicacé à la détection de triggers. Hélas, la communication se bloquait à nouveau après peu de temps.

Au final, l'origine du problème doit venir de la Raspberry et de son inhabilité à agir en temps réel. L'Arduino, lui, avec son microcontrôleur à 16MHz, est capable de réagir à un trigger et d'effectuer une lecture I²C. Raspbian, l'OS de la Raspberry, n'est pas un OS temps réel. Ce qui implique que le système d'exploitation effectue ses opérations propres et que, lorsqu'il a fini, il effectue les tâches demandées par l'utilisateur.

De plus, la Raspberry n'est pas conçue pour fonctionner comme système temps-réel dû à son absence d'une horloge temps réel (RTC[13]). Il existe pourtant certaines distributions comme ChibiOS/RT[5] , qui par l'ajout de cette RTC, le transforme en OS

[12] Pigpio est une librairie développée par *joan2937* permettant de contrôler le GPIO de la Raspberry Pi. Son implémentation est différente des librairies communément utilisées (*RPi.GPIO et SMBus*) ce qui dans certains cas est utile (par exemple le READ I²C).
[13] Real-time Clock.

temps-réel. Mais celle-ci n'est pas basée sur Linux. Or, ROS nécessite un système basé sur Linux pour pouvoir fonctionner.

L'I²C n'est donc pas un moyen de communication efficace et sûr pour ce projet et dans la façon dont il a été désigné. C'est pourquoi il a été décidé de basculer sur l'UART qui lui, ne nécessite pas de triggers.

## 4.5. UART

L'UART avait été mis de côté car les moteurs Dynamixel fonctionnent en UART. Mais, dorénavant, toutes les trames vers les Dynamixel sont reçues par la Raspberry. La librairie doit donc être capable de distinguer ce qui est pour la Raspberry de ce qui ne l'est pas. Après modification et amélioration de celle-ci, ce fut chose faite.

Outre ce problème, il y a également le fait que le Raspberry travaille en 0-3.3V alors que l'Arduino et les moteurs Dynamixel sont en 0-5V. Afin d'éviter tout problème sur le long terme, le circuit électronique a été modifié et les tensions séparées[14]. Voici un bref aperçu/récapitulatif de celui-ci :

- Arduino ↔ Raspberry Pi : bus à 3.3V, les Dynamixel ne reçoivent rien ;
- Arduino ↔ Dynamixel : bus à 5V, la Raspberry reçoit les commandes en 3.3V.

Le baudrate de l'UART a été fixé à 115 200 bps. Lors des tests, il est apparu qu'en passant à 1Mbps, un phénomène étrange se passait lorsque la Raspberry transmettait. Bien que les trames soient correctement détectées et décodées par l'Arduino, un délai de 26µs à 129 µs apparaît entre chaque envoi de caractère. Cette particularité n'apparaissant pas avec l'Arduino, la théorie de l'OS non temps-réel se renforce encore.

En restant à 115 200 bps, le projet fonctionne correctement. La fonction feedback fixée à 1Hz a amené plus de 13 000 enregistrements dans la base de données. De plus, à aucun moment il n'y a eu de perte de communication. Enfin, il était possible de contrôler les moteurs sans aucun souci.

[14] Dans ce cas c'était à base de transistors mais des convertisseurs de niveaux de tensions (*level shifter*) auraient pu faire l'affaire.

# 5. Conclusion

Ce travail a pour premier objectif d'établir un protocole de communication entre la Raspberry Pi et l'Arduino dans le cas d'une preuve de concept pour le robot CASPER. Il est également important de communiquer avec une base de données distante afin de recueillir les données des capteurs à des fins d'analyse et d'amélioration.

La première grande étape a été de développer la librairie de communication. Ensuite, celle-ci a été intégrée dans le projet afin de réaliser une démonstration technique. Il est donc possible de commander des moteurs et de recevoir des informations des capteurs ainsi que d'afficher les images prises par la caméra connectée à la Raspberry.

Lors des tests d'intégration, il est apparu que l'I²C n'est pas une solution viable à cause des triggers qui ne sont pas détectés à 100% par la Raspberry. La cause évoquée est que l'OS et la Raspberry n'ont pas vocation à être temps-réel. Après changement vers l'UART, tout fonctionne comme espéré : aucune perte de communication n'est à signaler.

Comme ce travail n'est qu'une preuve de concept, beaucoup de documentations et scripts ont été écrits afin d'en faciliter la compréhension et la reproductibilité. En effet, d'autres personnes vont reprendre le projet où je l'ai laissé afin d'aller vers une solution commerciale.

Bien sûr, certains choix techniques peuvent être discutés. Notamment, l'Arduino maître n'est pas la meilleure solution car l'I²C ne peut être utilisé. De plus, les capteurs I²C peuvent être directement reliés à la Raspberry, ce qui fera économiser du temps et des bytes de communications inutiles.

Enfin, l'aspect consommation doit être pris en compte car le robot est destiné à être autonome. Dès que la Raspberry est utilisée avec un module wifi, un écran … sa consommation grimpe en flèche jusqu'à atteindre entre 1 et 2A. Les concurrents de la Raspberry se situent également dans cette fourchette. Le poids du robot influencera également la consommation en courant des moteurs et donc la consommation totale.

# 6. Sources

[1] LHOIR, Matthieu, *Using an Arduino board to centralize actuators & sensors for robotic applications,* Mons, Travail de fin d'études, HELHa, 2016.

[2] DELVAUX, Julien, *C/Python library to communicate between a raspberry PI and an Arduino in the casper project,* Mons, Travail de fin d'études, HELHa, 2016.

[3] TANENBAUM A. & WETHERALL D., *Computer Networks Fifth edition* 2011, 962 p.

[4] MARTINEZ A. & FERNANDEZ E., *Learning ROS for Robotics Programming* 2013.

[5] BATE, Steve, (consulté le 20 novembre 2015), *ChibiOS/RT on the Raspberry Pi.*
Adresse URL : http://www.stevebate.net/chibios-rpi/GettingStarted.html

[6] HOBBYTRONICS, (consulté le 20 novembre 2015), *Raspberry Pi I2C Slave Read (clock stretching) Problem.*
Adresse URL :http://www.hobbytronics.co.uk/raspberry-pi-i2c-clock-stretching

[7] SYSTEM MANAGEMENT INTERFACE FORUM, (consulté le 21 novembre 2015), *System Management Bus (SMBus) Specifi cation.*
Adresse URL : http://www.smbus.org/specs/