## Liaison Série RS232 TTL (RPi 🡺 AX12A)

Les Servomoteurs AX12A se contrôle à l’aide de trame série (type RS485) de 1Mbit/s à des niveaux TTL (0 – 5V).

Le format des trames, qui a été spécifié précédemment, reste standard pour la liaison RS485 :

* 1 bit de start
* 8 bits de données
* 1 bit de stop

La principale propriété de la liaison RS485, est qu’elle repose sur le principe du maitre-esclave. Le maitre parle, et les esclaves ne parle que lorsqu’ils y sont autorisés.

Voici le schéma d’une telle relation :

Figure : Topologie d'une liaison RS485 unidirectionnelle

Liaison 1 MHz

Esclave

( Servomoteur )

1 Maître

( Raspberry )

Esclave

( Servomoteur )

Dans un fonctionnement normal, le Raspberry Pi aurait du envoyer des ordres de commandes, et recevoir une réponse du servomoteur, avec des informations telle que la température, le couple,…

Mais par un souci d’électronique (la liaison étant unidirectionnelle électroniquement), nous ne pouvons qu’envoyer des ordres aux servomoteurs, sans avoir de réponse de leur part comme le représente la Figure 1.

Le Raspberry Pi est donc une solution viable pour contrôler les servomoteurs. Néanmoins, nous avons eu quelques problèmes pour la mise en place de cette liaison.

Le premier a été avec la librairie Pi4j utilisé. En effet, cette librairie ne permet pas d’initialiser le port série à une vitesse aussi grande.

Il a donc fallu modifier cette librairie pour intégrer cette vitesse.

Pour la modification, cette librairie s’appui sur la bibliothèque système termios.h. Cette bibliothèque est une librairie généraliste qui permet de contrôler tout élément série.

Sur le Raspberry Pi, cette librairie possède la configuration à 1MHz.

Il a donc suffit d’ajouter la gestion de 1MHz à la bibliothèque Pi4j en l’ajoutant en plus des autres vitesses par défaut.

La modification de la librairie n’a pas été très lourde, mais la compilation ensuite n’a pas été des plus facile, car cette librairie est compilée avec des scripts Maven.

Il faut savoir que pour la gestion de l’interface série avec la librairie Pi4j, deux fichier sont nécessaires.

* libsetuid-linux.so, qui est une librairie issue d’une autre bibliothèque (WiringPi), qui elle est en langage C.
* pi4j-core.jar, qui est une sorte d’interface pour interconnecter la bibliothèque C libsetuid.so en java. (le fichier .jar contient la librairie C)

Il a donc fallu modifier les scripts Maven, pour qu’ils prennent en compte la compilation de ces deux librairies (ce qui n’est pas le cas par défaut).

Une fois la compilation terminée, il n’a resté que l’ajout du fichier .jar au programme Hexapod. Ce fichier .jar se situe dans le dossier lib/pi4jow-core.jar.

Ayant passé du temps à recoder la libraire, j’ai contacté personnellement son créateur.

Il m’a répondu que beaucoup de personne lui avait demandé d’intégrer cette fonctionnalité, et qu’il le ferrait dans la prochaine release de sa librairie pi4j.

Une fois cette manipulation terminé, le port série s’initialisais bien à la bonne vitesse sur le programme, mais pas selon l’oscilloscope dans la réalité.

En effet, voici une capture réalisé à l’aide d’un oscilloscope de la liaison série en sortie de l’ArbotiX, et du Raspberry Pi.

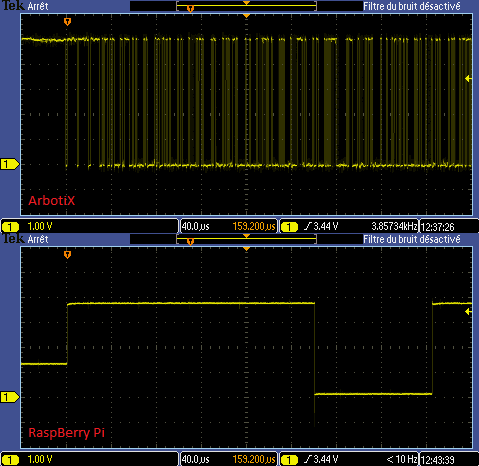


Figure : Capture Signaux sortie RS232

On peut voir que les signaux ne sont pas du tout les même. Le signal de l’ArbotiX est bien à 1Mbit/s, mais celui du Raspberry Pi est plus proche de 10kbit/s, alors que le programme spécifiait que la liaison était de 1Mbit/s.

Nous avons donc essayé de changer le programme en modifiant la valeur de la vitesse de la liaison, mais la communication n’a pas dépassé le débit ci-dessus. Elle restait saturé à 10kbit/s.

Nous avons trouvé la solution en comparant deux relevés à l’oscilloscope de la même trame :

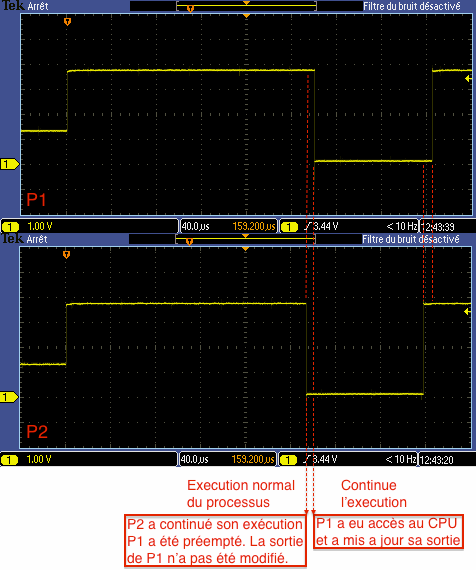


Figure : Trames sortie Raspberry Pi

Les deux trames sont différente sur l’oscilloscope, mais se sont pourtant deux trames qui doivent être identique, alors comment se fait-ce.

Et bien comme indiqué sur la Figure 2, l’OS utilisé sur le Raspberry Pi est Raspbian, qui n’est pas un OS temps réel. Le processus qui gère le RS485 peu (est) préempté.

Nous avons donc une différence de temps qui peu être non négligeable.

Ceci explique cela, car l’OS limite donc la vitesse du bus série, pour pouvoir donner priorité à d’autre processus.

Pour résoudre ce souci, il faut rajouter ceci dans le fichier */boot/config.txt* :

*init\_uart\_clock=16000000*

Ces deux lignes vont forcer l’OS à autoriser la vitesse de 1Mbit/s, et utiliser 16MHz pour l’UART.

16MHz, car à défaut d’utiliser un OS temps réel, si on va plus vite, on peu admettre finir avant la terminaison.

En utilisant cette solution, nous auront toujours des écarts comme sur la Figure 2, mais cela n’est pas dérangeant car les moteurs ne verront pas cette différence.

De plus, la liaison étant une communication RS485 asynchrone, l’horloge sera resynchronisé à chaque émission de la trame de 9 bits.

# Utilisation des WebSockets

Le but premier de notre projet étant d’être une base pour de futur projet, le choix du type de contrôle est très important.

Deux choix sont donc possibles :

* Créer un protocole de contrôle sur mesure qui entraine de devoir spécifier le protocole pour qu’il soit le plus accessible possible
* Utiliser un protocole déjà existant et répandu

Du fait que nous allions utiliser des programmes fait en JavaScript par la suite pour les interfaces de contrôle, le choix des WebSockets c’est imposé de lui même.

En effet, si l’on veut faire communiquer un programme JavaScript, nous sommes obligé d’utiliser les WebSockets, car les sockets normales TCP/UDP, n’y sont pas implémenté.

Les WebSockets vont donc permettre d’utiliser le JavaScript comme client pour se connecter au Raspberry Pi qui sera le serveur (la connexion sera bidirectionnel).

Mais plus encore, car les WebSocket en plus d’être implémenté en JavaScript, sont aussi utilisable en C/C++, java, LabView,…

L’avantage donc d’utiliser ces WebSockets, est donc de fournir un moyen de communication universelle, pour que n’importe quel matériel puisse contrôler l’hexapode.

Les WebSockets ont été normalisé dans la RFC 6455 par le W3C[[1]](#footnote-1).

Au niveau structure, les WebSockets sont très simple, car ce n’est qu’une surcouche d’une socket TCP.

En effet, une WebSocket utilise une socket TCP dans laquelle des en-têtes de longueurs et des systèmes de contrôle d’erreur sont ajouté.

Les WebSockets se sont très largement inspiré du protocole http, car elles utilise les même message (GET, POST,…) pour réaliser l’hanshake[[2]](#footnote-2) avec le serveur.

Dans notre application les WebSockets seront utilisé comme sur la Figure 4.

Navigateurs Web

(WebSockets client)

Serveur WebSockets Raspberry Pi

Socket TCP

Port 3842

Figure : Principe d'une liaison WebSocket

Le navigateur Web représente le PC ou la tablette sur lequel tournera la page HTML contenant le code JavaScript. Ce navigateur comme évoqué précédemment pourra être remplacé par n’importe quel matériel pour contrôler le robot.

Le serveur WebSocket lui sera situé sur le Raspberry Pi. Il servira d’intermédiaire entre le client et le programme de gestion du mouvement du robot.

Vu que nous avons choisie de réaliser notre programme en java, de nos jours, la meilleur implémentation des WebSockets dans ce langage est faite dans le produit Jetty, qui est un produit d’Eclipse.



Figure : Logo produit open source Jetty

Ce produit Jetty est open source et possède une communauté très active.

Comme indiqué sur la Figure 4, le port TCP utilisé pour la liaison WebSocket sera le 3842.

Pour ce qui est du choix des trames circulant sur le réseau, Jetty est généralement utilisé en faisant transité des données sous forme XML ou JSON.

Nous avons fait le choix pour rester au plus simple d’utiliser un protocole texte.

De ce fait, les trames sont de la forme :

<code capteur>:<valeur>

Voici donc un tableau récapitulatif des différentes valeurs possibles :

|  |  |  |
| --- | --- | --- |
| Description | Code capteur | Intervalle Valeurs |
| Axe X joystick gauche | XL | - 255 à 255 |
| Axe Y joystick gauche | YL | - 255 à 255 |
| Axe X joystick gauche | XR | - 255 à 255 |
| Axe Y joystick gauche | YR | - 255 à 255 |
| Code boutons | BT | Codes binaires des boutons |

Figure : Tableau des valeurs des éléments de contrôles renvoyés

Grace à ces commandes et aux WebSockets, il est donc parfaitement possible de contrôler le robot avec n’importe quel type de programme.

Avec notre page HTML chargé par la tablette ou le PC, les WebSockets sont utilisé pour envoyé les informations des éléments de commande toutes les 200 millisecondes. Cette valeur a été définie pour être la plus grande possible mais sans que le robot n’ai trop de latence entre le changement d’ordre, et la commande effective sur le robot.

# Gestion de la commande avec une tablette

Pour notre projet, nous avons du trouver une solution pour contrôler le robot. Au début, nous voulions contrôler le robot avec un iPad. Il aurait donc fallu développer une application native sur l’iPad, et seul un iPad aurait pu contrôler le robot.

L’application native étant totalement contraire pour laisser accessible notre robot, il nous a fallu trouver une autre solution.

L’idée de la solution universelle nous est venue avec la librairie virtualjoystick.js que nous utilisons.

Cette librairie fournie une API sous licence MIT, qui permet d’utiliser les capacités du multitouch.

En effet, sur l’iPad, ou d’autre tablette, les écrans sont multitouch, et un script JavaScript peut récupérer les points de contacts.

L’intérêt principale de cette libraire est qu’elle est compatible avec énormément d’équipement tel que les tablettes, les Smartphones multitouch, ou encore avec la table surface de Microsoft.

Cette librairie est très puissante car elle regroupe des éléments qui permettent de connaître par exemple l’emplacement de dix doigts sur l’écran,…

Nous n’utiliserons donc pas toutes ses possibilités. Ce que nous utilisons est une spécification, qui permet d’utiliser des joysticks virtuels comme sur la Figure 3 ci-dessous :

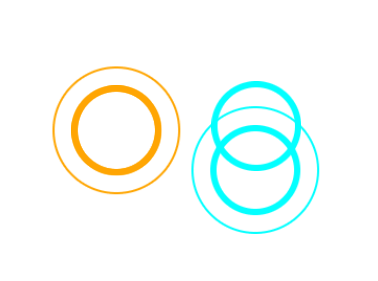


Figure : Joystick virtuel sur écran multitouch

Cette interface s’utilise en posant un doigt sur la partie gauche de l’écran (faisant apparaître un joystick orange), et un autre sur la partie droite de l’écran (faisant apparaître un joystick bleu). Le déplacement du doigt sur l’écran entraine le déplacement du joystick par rapport a son point de référence (la où le doigt a été posé).

Avec cette bibliothèque le degré de liberté du joystick est paramétrable. Nous avons choisi de prendre en standard 255 de rayon pour que sa valeur évolue entre -255 et 255, qui est assez commun avec les valeurs des joysticks sur le marché.

Cet interfaçage nous permet donc d’obtenir des coordonnées d’un joystick gauche et droit, mais pas d’information de boutons. Mais pour un élément portable tel qu’un iPad, c’est suffisant pour contrôler le robot.

# Gestion de la commande avec une manette

Pour le contrôle du robot avec une manette, nous avons choisi d’utiliser une manette de PS4, qui a l’avantage d’être géré nativement sous Mac OS X.



Figure : Contrôleur PS4 utilisé pour commander l'hexapode

Ayant déjà implémenté la solution d’une page JavaScript pour la tablette, nous avons cherché une librairie dans ce même langage pour pouvoir connecter notre manette au Macintosh.

Notre choix c’est porté sur une librairie sortie avec les dernières versions de Firefox, la bibliothèque GamePadAPI.js.

Cette librairie est elle aussi open source, et est développé par Mozilla. Elle est donc compatible avec le navigateur Firefox, ainsi que Google Chrome, dans leurs dernières versions (Firefox 29, et Chrome 21)

Cette bibliothèque est très intéressante, car elle fourni des commandes « universelles » pour n’importe quel type de contrôleur. C’est à dire que les boutons situés aux mêmes emplacements physiques que tous les autres contrôleurs ont le même mappage logiciel.

Les unités sont elles aussi uniformes sur tous les contrôleurs possédant des joysticks.

Beaucoup de contrôleurs sont pris en charge, dont les manettes des consoles traditionnelles (manette Xbox et Playstation), à défaut d’avoir les pilotes compatibles pour l’OS de l’hôte.

Pour notre projet cette librairie s’ajoute parfaitement bien.

En effet, lorsque la page html était lancée sur un périphérique sans interface tactile, l’élément ne pouvait pas utiliser les joysticks virtuels évoqués précédemment, alors que maintenant si un matériel ne prend pas en charge le multitouch, on attend la connexion d’un éventuel controleur.

# Choix de l’OS

Etant habitué à des environnements Debian, nous avons choisis Raspbian qui est un OS spécialement réalisé pour le Raspberry Pi.

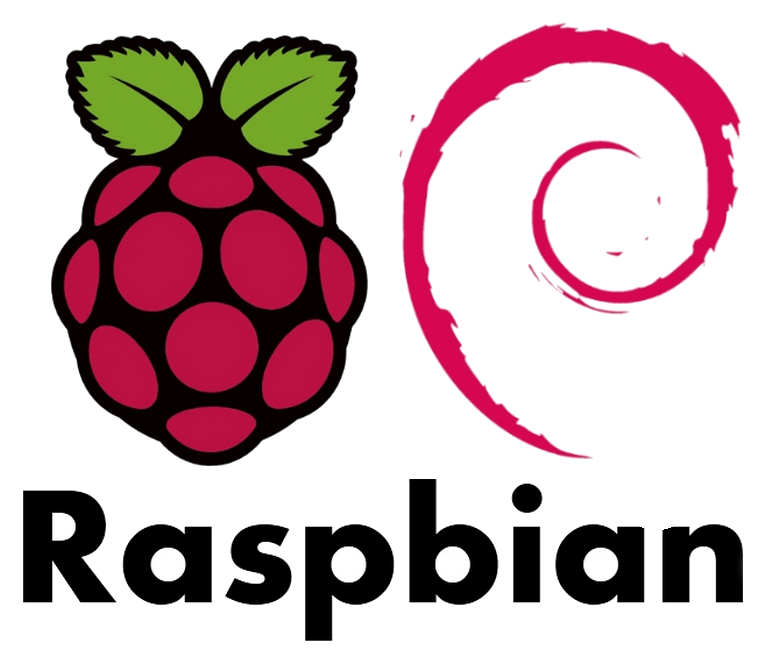


Figure : logo OS Raspbian pour Raspberry Pi

Malgré tout cet OS est fournie avec des éléments qui nous sont inutiles, et qui ralentissent le Raspberry Pi. On concède que plus le Raspberry Pi sera optimisé, plus notre programme s’exécutera efficacement.

Cette OS étant livré avec des outils mathématiques, un environnement graphique,… nous avons supprimé tous les paquets de ces produits.

De plus, nous avons effectué un ménage sur le Raspberry Pi. Nous avons donc retirer 6 des 7 terminaux disponible physiquement car ils étaient inutiles pour nous.

Par manque de temps, nous n’avons pas pu placer de dissipateurs thermiques sur les puces du Raspberry Pi, mais nous avons quand même procédé à un overclockage de sa fréquence d’exécution pour qu’elle atteigne 900MHz (ce qui correspond à un overclockage medium sur l’outil raspi-config).

Après l’overclockage, et les optimisations voici les relevés effectués sur le Raspberry Pi en fonctionnement :

Figure : Graphique mémoire vive libre Raspbian (en Mo)

Comme le montre la Figure 10, l’optimisation a permis de gagner de l’espace et donc de pouvoir éviter que si un nouveau processus veut plus de RAM, que l’OS passe par la carte SD pour swapper les données.

On voit quand même que sans l’optimisation, les programmes auraient tenu sur la RAM, car il y a une bonne marge de sécurité.

Figure : Graphique utilisation du processeur Raspbian (en %)

Sur la Figure 11, en revanche, on peut voir que l’enjeu était plus important. En effet, pour garder une exécution normal du programme, il faut garder le plus possible du CPU libre. Comme sa les exécutions se passent sans d’incessantes interruption du Scheduler.

Certes, l’optimisation n’a pas libéré beaucoup de CPU, mais tout petite amélioration est bonne à prendre pour que notre produit s’exécute le mieux possible. On voit quand même que l’utilisation du processeur est de 70% quand les programmes sont lancés, ce qui n’est pas négligeable.

Le système Debian nous étant connue, nous avons créé un script pour qu’il s’exécute au démarrage, et qu’il lance les programmes de l’hexapode.

Notre robot est donc « plug and play ». En effet, au démarrage, il lance tous les programmes, et l’utilisateur n’a plus qu’a si connecter.

C’est l’avantage de tous les scripts créé spécialement pour que l’expérience utilisateur soit la plus accessible possible.

Pour réaliser le chargement de tout les programmes au démarrage, nous avons du utiliser le produit screen, qui est un émulateur de terminaux virtuels.

Pourquoi utiliser ce type de produit, et bien parce que nos programmes (et autres produits utiles) ne sont pas programmé pour tourner comme des daemons.

C’est à dire que nos programmes restent attaché au terminal qui les a lancé, quoi qu’il arrive. Pour les convertir en daemons, il aurait fallu détacher le thread principal, pour qu’il soit totalement asynchrone pour le terminal. Mais ce n’est pas le comportement que nous voulions, car nous voulions pouvoir avoir un visuel, et interagir avec nos programmes si besoin.

Nous lançons donc nos programmes dans le produit screen pour qu’ils soient lancés dans un terminal, qui sera détaché des autres. Il vivra donc sa vie de programme normalement sauf si l’on vient le modifier en passant pas screen pour par exemple arrêter son exécution.

# Communication sans-fil

Au début du projet, nous utilisions notre Raspberry Pi en Ethernet. Mais au fur et à mesure, il nous a fallu trouver une solution pour le connecter en wifi, car le produit se veut sans-fil, pour qu’il puisse être libre de ses mouvements.

Dans un premier temps par manque de temps nous avons commencé à le connecter à une box internet en wifi.

Cette solution était confortable, mais le problème, c’est qu’on devait à chaque fois paramétré le réseau wifi auquel se connecter. Ce n’était donc pas très pratique pour des personnes qui ne sont pas familiers avec des lignes de commandes.

De plus vu que nous utilisions des protocoles pseudo temps-réel, le fait que le routeur n’envoyait pas instantanément les trames (mise en cache), faisait apparaître des latences sur le contrôle du robot.

Nous nous sommes donc tournée sur des solutions de hotspot wifi, pour que le Raspberry Pi créer lui même son réseau wifi.

Cette solution est très pratique, car on connecte directement notre équipement de contrôle, et on peut demander la page HTML au Raspberry Pi.

Nous avons donc utilisé les produits hostAPD, et udhcpd.

Le premier hostAPD, sert à créer un point d’accès wifi sur le Raspberry Pi, pour que les client s’y connecte directement.

Au début, le réseau créé était d’une vitesse de 54 Mbit/s ce qui est la vitesse moyenne pour la norme g du wifi. Mais après quelque test, nous nous sommes rendu compte que cette vitesse ne convenait pas du tout, car nous avions de trop gros flux à faire passer avec la PiCam et les WebSockets, que nous verrons par la suite.

Nous avons donc paramétré cette liaison pour qu’elle fonctionne en wifi n à 150Mbit/s, qui est la limite du périphérique Wi-Pi que nous utilisons.



Figure : Connecteur Wi-Pi 150Mbit/s wifi n

Le deuxième produit est udhcpd, qui est un mini serveur dhcp permettant de fournir des adresses aux clients qui se connectent au wifi.

Dans notre philosophie du « facile à utiliser », le serveur dhcp était obligatoire pour garder cette notion de « plug and play ».

Par défaut, nous avons configuré cette adresse sur 10.137.1.1 pour le Raspberry Pi, avec un masque de 255.255.255.0, pour les autres matériaux se connectant.

# Serveur Web

Nous avons choisi d’installer un serveur Web pour que les clients utilisent directement le Raspberry Pi pour obtenir la page HTML qui leur sert d’interface pour contrôler le robot, pour rester dans la philosophie du « plug and play »

Au commencement nous avions installé un serveur Apache qui est très répandu comme serveur http.

Mais nous nous sommes rendu compte que ce serveur Web était beaucoup trop lourd pour le Raspberry Pi. En effet, il lançais 6 processus et consommais 25% de la mémoire vive du Raspberry Pi (sur 256 au totale).

Cela était inacceptable, car le programme java aura besoin de ces ressources, pour commander au mieux le robot.

Nous sommes donc passé sur un serveur web consommant beaucoup moins de ressources.

Ce serveur est lighttpd. Il a été spécialement conçu, pour ne pouvoir répondre qu’a quelques requêtes par seconde à l’opposé du serveur apache.



Figure : Logo produit lighttpd

Grace au changement de serveur, nous avons gagné de précieux cycle CPU, et l’espace RAM libre en plus.

Ce faisant, cela est largement suffisant pour ne fournir que quelque page html, contenant du JavaScript, qui sera exécuté par le client, et non pas par le Raspberry Pi.

# Camera PiCam

Pour permettre d’avoir un retour sur la position du robot, ainsi que de pouvoir permettre son contrôle à distance, nous avons choisi d’ajouter une PiCam sur le robot

Images PiCam dur robot

Au début, nous voulions ajouter le streaming de la camera sur la page html de l’iPad. Mais nous avons du renoncer, car ce n’était pas possible.

En effet, tout flux vidéo ne peut être lue qu’avec le lecteur vidéo intégré à l’iPad.

Mais nous avons tout de même ajouter la fonction caméra en s’y connectant depuis un client séparé.

Au début, nous avions utilisé le produit gstreamer, qui récupéré le flux de la webcam, le convertissait, et l’envoyé sur une page HTML.

Le principal problème de cette solution était qu’il y avait un décalage de 20 secondes qui rendait inutile le renvoi du flux vidéo pour contrôler l’hexapode.

Nous sommes donc passer par une autre solution celle du streaming par la libraire VLC.

Grace à cette librairie, il nous a été possible de streamer le flux avec plusieurs protocoles. Après des tests, il c’est avéré que le protocole le plus efficace du point de vue de la latence était le protocole RTSP[[3]](#footnote-3).

Nous sommes arrivé à une latence de 3 secondes entre la réalité et l’image retransmise en passant par une liaison wifi, ce qui est toujours mieux que les 20 secondes de la première solution.

La deuxième solution nécessite d’ouvrir le flux rtsp, avec vlc, ou un autre lecteur compatible, alors que la première ne nécessitait qu’un navigateur web.

Néanmoins même avec cet aspect, nous ne pouvons définitivement pas utiliser la première solution pour des raisons de latences.

# Choix du langage et du la librairie de programmation

Pour le choix du langage, plusieurs choix étaient possibles :

* Python
* Java
* C/C++

Pour des raisons de compétences, et de choix de produits nous avons choisie de réaliser notre programme hexapode en java.

En effet, sachant ce que nous voulions utiliser comme produit, le plus simple d’implémentation rester le java.

Pour le choix de la librairie permettant de contrôler le GPIO, du Raspberry Pi, nous avons utilisé la librairie Pi4j.



Figure : Logo bibliothèque Pi4j

Cette librairie est une bibliothèque java libre s’appuyant sur la librairie C WiringPi.

# Diagramme UML

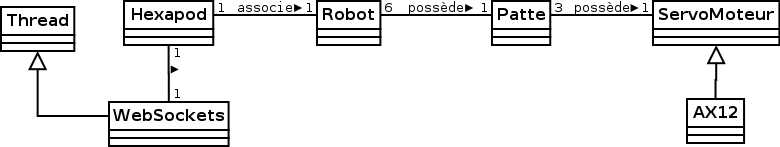


Figure : Diagramme UML projet Hexapod

Comme on peut le voir sur le diagramme UML Figure 15, deux principales classes sont présentes.

La classe **Hexapod**, qui instancie les objets permettant de contrôler le robot.

La classe **WebSockets**, qui instancie la classe Thread permettant de ne pas perturber le contrôle du robot, et d’avoir un fonctionnement totalement parallèle des WebSockets, et du contrôle du robot.

Ensuite, on peut voir que l’objet Hexapod créer un objet de type robot, qui servira de base, pour à partir des données des joysticks fournis par les WebSockets, contrôler les 6 pattes.

Les objets patte vont donc ainsi suivre le mouvement que l’objet robot leur aura donné, et vont ensuite envoyer les bons angles des servomoteurs aux 3 objets servomoteur qu’elles auront instancier.

On voit bien que notre modèle UML colle au plus proche de la réalité, en utilisant des objets existant physiquement.

Ce diagramme était donc pour nous très représentatif du rôle de toutes les parties physique du robot.

# Conclusion

En conclusion, on peut dire que nous avons utilisé le Raspberry Pi jusqu'à ses limites de fonctionnement.

Nous avons fait l’expérience de ces fonctionnalités, et on peut dire que l’on arrive à ses limites,…

1. World Wide Web [↑](#footnote-ref-1)
2. Notion définissant l’envoi d’un message d’authentification d’un client littéralement poigné de main [↑](#footnote-ref-2)
3. Real Time Streaming Protocole [↑](#footnote-ref-3)