

# Implementación de

Diego Muñoz Galan

24 de junio de 2014



## Resumen

El proyecto persigue el objetivo de implementar un driver para integrar la plataforma robótica AIBO con el marco de trabajo ROS. A lo largo de esta memoria se narra el procedimiento seguido para llegar a un resultado que cumpla con los requisitos. Se puede dividir el trabajo realizado y documentado en ésta memoria en tres partes.

Consta de una primera parte de documentación e investigación de la plataforma objeto en la que se describe tanto el hardware como los posibles softwares con los que trabajar.

Sigue una segunda parte en la que se comparan varios métodos de programación con los que se podría implementar el modulo de ROS deseado. En ésta se incluye un primer análisis cualitativo en el que se contemplan gran parte de las posibilidades de programación y un segundo análisis más exhaustivo de los métodos que han pasado la primera selección para finalmente decidir el método a usar.

Para concluir se explica como se han implementado los módulos que facilitan la integración con ROS. Por un lado el modulo de control al que se le han aplicado unos criterios de valoración y se ha puesto a prueba y por otro el modulo que facilita la integración de un modelo de visualización 3D.



# Índice

<b>Resumen</b>	<b>1</b>
<b>1. Prefacio</b>	<b>8</b>
1.1. Motivación . . . . .	8
1.2. Requerimientos previos . . . . .	8
<b>2. Introducción</b>	<b>9</b>
2.1. Estado del arte . . . . .	9
2.1.1. Robótica . . . . .	9
2.1.2. Robótica en la educación . . . . .	10
2.1.3. AIBO . . . . .	11
2.2. Objetivos . . . . .	12
2.3. Alcance . . . . .	13
<b>3. AIBO ERS-7</b>	<b>14</b>
3.1. Hardware . . . . .	14
3.2. El software . . . . .	16
3.2.1. OPEN-R . . . . .	16
3.2.2. Tekkotsu . . . . .	18
3.2.3. URBI . . . . .	19
<b>4. Comparación de alternativas para la programación</b>	<b>21</b>
4.1. Lectura de datos . . . . .	22
4.1.1. Método con liburbi y C++ . . . . .	23
4.1.2. Método con telnet y python . . . . .	24
4.1.3. Comparación de los resultados . . . . .	25
4.2. Envío de datos . . . . .	31
4.3. Elección del método . . . . .	36
<b>5. Implementación del paquete de ROS</b>	<b>38</b>
5.1. ROS . . . . .	38
5.2. Paquet aibo_server . . . . .	39
5.2.1. Estructura del programa . . . . .	40
5.2.2. Resultados . . . . .	42
5.2.3. Mejoras . . . . .	44
5.2.4. Demostración . . . . .	46
5.3. Paquete Aibo_description . . . . .	46
5.3.1. El modelo . . . . .	47

5.3.2. El nodo state_publisher . . . . .	48
5.3.3. Estructura . . . . .	49
<b>Conclusiones</b>	<b>51</b>
<b>Referencias</b>	<b>52</b>
<b>Anexos</b>	<b>53</b>
<b>A. Configuración de los marcos de trabajo</b>	<b>53</b>
A.1. OPEN-R SDK . . . . .	53
A.2. Configuración wifi . . . . .	53
A.3. Tekkotsu . . . . .	54
A.4. URBI . . . . .	54
A.5. ROS . . . . .	54
<b>B. Manual de usuario</b>	<b>55</b>
B.1. Requisitos previos . . . . .	55
B.2. Instalación . . . . .	55
B.3. aibo_server . . . . .	55
B.4. aibo_description . . . . .	56
<b>C. Scripts y programas</b>	<b>57</b>
C.1. Adquisición del valor de una articulación con liburbi. . . . .	57
C.2. Adquisición del valor de una articulación con Python. . . . .	58
C.3. Envió de trayectoria punto a punto con liburbi. . . . .	59
C.4. Envió de trayectoria punto a punto con Python. . . . .	62
C.5. Envió de trayectoria punto a punto con ROS. . . . .	65
C.6. Modulo de imitación entre AIBOs. . . . .	67

## Índice de figuras

1.	Robot manipulador de KUKA. . . . .	9
2.	Robot cuadrúpedo WildCat. . . . .	10
3.	AIBO ERS-7. . . . .	14
4.	Arquitectura de AIBO con OPEN-R . . . . .	16
5.	Comunicación entre objetos. . . . .	17
6.	Arquitectura del AIBO con Tekkotsu. . . . .	18
7.	Proceso de ejecución de un comportamiento en Tekkotsu. . . . .	19
8.	Arquitectura del AIBO con URBI. . . . .	20
9.	Estructura de la lectura de una variable. . . . .	23
10.	Terminal URBI consultando un dato. . . . .	24
11.	Diagrama de cajas de la frecuencia de datos para las 15 replicas de cada método. . . . .	25
12.	Mínimos de la frecuencia en les 15 repliques de los dos métodos. . . . .	26
13.	Histograma de la frecuencia en les 15 repliques de los dos métodos. . . . .	27
14.	Diagrama de cajas de la frecuencia de datos para las 15 replicas de cada método con todas las articulaciones. . . . .	28
15.	Medias de las frecuencias de envío, comparando una el obtener el valor de una articulación con obtener el valor de todas las articulaciones. . . . .	29
16.	Mínimos de la frecuencia en les 15 repliques de los dos métodos. . . . .	30
17.	Histograma de la frecuencia en les 15 repliques de los dos métodos. . . . .	30
18.	En azul está graficado las posiciones enviadas a 1Hz y en rojo las lecturas. En columnas se encuentran las replicas del mismo método y por filas los tres métodos usados, en orden descendente son liburbi con un cliente, liburbi con dos cliente y python. . . . .	32
19.	En azul está graficado las posiciones enviadas a 2Hz y en rojo las lecturas. En columnas se encuentran las replicas del mismo método y por filas los tres métodos usados, en orden descendente son liburbi con un cliente, liburbi con dos cliente y python.5 . . . . .	33
20.	En azul está graficado las posiciones enviadas a 5Hz y en rojo las lecturas. En columnas se encuentran las replicas del mismo método y por filas los tres métodos usados, en orden descendente son liburbi con un cliente, liburbi con dos cliente y python. . . . .	34
21.	En azul está graficado las posiciones enviadas a 10Hz y en rojo las lecturas. En columnas se encuentran las replicas del mismo método y por filas los tres métodos usados, en orden descendente son liburbi con un cliente, liburbi con dos cliente y python. . . . .	35

22.	Retrasos entre la orden enviada y la acción para los métodos liburbi con dos clientes y python. . . . .	36
23.	Nodo aibo_server. . . . .	39
24.	Estructura básica del ejecutable aibo_server. . . . .	41
25.	Estructura de ROS con los nodos aibo_server y SinLeg. . . . .	42
26.	Entrada y respuesta del sistema ante una señales sinusoidales de diferente frecuencia y muestreo. . . . .	43
27.	Diagrama del callback de ROS. Los flags <i>anterior</i> , <i>C1activo</i> y <i>C2activo</i> se inicializan con valor 1. . . . .	45
28.	Estructura de los nodos de ROS en la demostración de sincronización. . . .	46
29.	Estructura del modelo URDF. . . . .	47
30.	Modelo visto des de el framework de rviz. . . . .	48
31.	Estructuras de los nodos de ROS con aibo_server y el modelo visualizado con rviz. . . . .	50

## Índice de cuadros

1.	Estructura del paso de mensajes en OPEN-R . . . . .	17
2.	Comparación entre lenguajes usados sobre AIBO . . . . .	22
3.	Comparación de los métodos usados y sus resultados en los experimentos. .	37

## 1. Prefacio

### 1.1. Motivación

Hace ya ocho años que la empresa SONY dejó de producir la mascota robótica por excelencia, el perro robot AIBO. Des de su desaparición del mercado, en 2006, éste ha caído en una espiral de desuso hasta el punto que hoy en día se pueden encontrar gran cantidad de ellos guardados en armarios de todo el mundo.

Este proyecto partió de la idea de volver a aprovechar los AIBO para realizar tareas de cooperación y sincronización partiendo de un nivel de programación elevado. Debiido a la incomodidad que proporcionaban los actuales lenguajes de programación para dicha plataforma, dado que ya no están siendo mantenidos y sus curvas de aprendizaje son realmente duras, se planteó la posibilidad de programar al AIBO mediante ROS5.1. Siendo ROS uno de los marcos de trabajo para la robótica que actualmente está teniendo más acogida, su uso sobre AIBO haría posible que volvieran a ser una plataforma a tener en cuenta para la investigación en estos momentos en que la universidad no cuenta con demasiados fondos. Por el otro lado la implementación de un driver para AIBO facilitará enormemente la programación, dando la posibilidad de usar el gran abanico de paquetes y herramientas existentes para ROS.

Por estos motivos participar en la integración del AIBO con ROS es un proyecto que dado su posible utilidad en el futuro lo hace irresistible.

Además el propio aprendizaje intrínseco del proyecto como es la oportunidad de conocer y entender ROS implementando ciertos paquetes resulta algo realmente interesante y útil para cualquier estudiante apasionado por la robótica.

### 1.2. Requerimientos previos

Para la realización de este proyecto es necesario un cierto bagaje y experiencia en programación, concretamente en los lenguajes C++ y python orientados a objetos, además de un conocimiento básico del funcionamiento de ROS y su estructura. Son necesarios también conocimientos de estadística y el uso de software de apoyo para la realización de algunos testes estadísticos. Por ultimo para la realización del proyecto se ha hecho uso de la herramienta LATEX.

## 2. Introducción

En este apartado se definirá tanto el punto de partida del proyecto, el punto final al que se pretende llegar y el camino que se pretende realizar.

### 2.1. Estado del arte

Se pretende dar una visión general del estado actual de la robótica y el compromiso entre software y hardware que ésta conlleva para seguidamente ir concretando y acotando sobre este gran universo hacia el punto que ocupa éste proyecto.

#### 2.1.1. Robótica

Desde la primera definición de robot o de la idea que exportó Isaac Asimov ha pasado más de medio siglo y aún así se puede decir que la robótica se encuentra dando sus primeros pasos. Si bien no es sencillo ubicar el origen de esta ciencia se puede afirmar con total seguridad que esta rama hizo grandes avances a raíz de la revolución industrial de mano de otras especialidades como la electrónica, la informática o la inteligencia artificial.

En un primer momento los robots fueron máquinas automatizadas o teleoperadas con el fin de realizar tareas dentro de la industria que tuvieran un cierto riesgo, requiriesen de alta precisión o una eficiencia tal que la mano de obra humana no pudiera ofrecer. En este contexto actualmente se encuentran gran variedad de brazos manipuladores en la mayoría de las fábricas que a la vez que aumentan su grado automatización, la eficiencia y rentabilidad económica lo hacen con el<sup>1</sup>[1]. Un ejemplo de este tipo de robots son los de KUKA<sup>1</sup> mostrado en la Figura 1.



Figura 1: Robot manipulador de KUKA.

<sup>1</sup><http://www.kuka-robotics.com/en/>

Otros sectores, aparte de la industria, han impulsado la robótica dando pie a otro tipo de robots, los robots móviles. Éste es el caso de la exploración espacial que dio sus primeros pasos con el automóvil teleoperado lunokjod<sup>2</sup> y actualmente tiene su máximo exponente con el robot Curiosityenviado a Marte en la última misión de la NASA<sup>3</sup>. Las plataformas anteriormente descritas tienen la sencillez de moverse sobre ruedas, a diferencia de otro tipo de robots móviles que usan varias extremidades, bípedos, cuadrúpedos, hexápedos y otros con gran cantidad de ellas. Si bien el uso de extremidades dificulta el control dinámico, la estabilidad y los algoritmos de navegación, ofrecen la posibilidad de andar por gran variedad de terrenos y producen una mayor aceptación social debido a su morfología familiar para el hombre . Un ejemplo es el robot cuadrúpedo de DARPA<sup>4</sup> y Boston Dynamics<sup>5</sup> Wildcat mostrado a en la Figura 2.



Figura 2: Robot cuadrúpedo WildCat.

### 2.1.2. Robótica en la educación

Las plataformas anteriormente descritas forman parte de proyectos millonarios que no forman parte de el grueso de la investigación con robótica. Son las plataformas como la usada en este proyecto las que han permitido tanto a universidades como a un público mucho más general investigar y han dado la oportunidad de crear una gran variedad de aplicaciones.

Algunos de los robots que han acercado las herramientas necesarias a un mayor número de usuarios son los siguientes:

<sup>2</sup>[http://en.wikipedia.org/wiki/Lunokhod\\_1](http://en.wikipedia.org/wiki/Lunokhod_1)

<sup>3</sup>[http://www.nasa.gov/mission\\_pages/msl/](http://www.nasa.gov/mission_pages/msl/)

<sup>4</sup><http://www.darpa.mil>

<sup>5</sup><http://www.bostondynamics.com/>

- WIFIBOT: Se trata de una serie de robots movidos por cuatro ruedas, sensores de distancia , camera y punto de acceso WiFi. Permite ser controlado por varios sistemas operativos y lenguajes<sup>6</sup>.
- Lego NXT: Es un producto modular que permite una infinidad de combinaciones construccitvas partiendo siempre de un módulo de procesador. Tiene su propio marco de trabajo<sup>7</sup>.
- NAO: Es un robot humanoide de dimensiones reducidas y con prestaciones de alta gama comercializado por Aldebaran Robotics <sup>8</sup>.
- ARDrone: Se trata de un vehículo no tripulado con cuatro hélices, o quadcoptero, que se respalda tanto la navegación autónoma como el radiocontrol.

Habiendo hablado un poco del hardware se pasa a comentar los sistemas más usados para crear la inteligencia artificial.

La gran mayoría de robots, dado que son comercializados por empresas, proporcionan un marco de trabajo propio e incluso su propio lenguaje algorítmico de alto nivel. Facilitando el acceso de forma sencilla a actuadores, sensores y sistemas de comunicación, permitiendo a un amplio público la iniciación en la robótica. Por otra parte existen ciertos softwares libres que se están estableciendo como una alternativa a los propios de cada plataforma. Ésto facilita enormemente del trabajo del programador puesto que no tiene que invertir tiempo en el aprendizaje propio de cada software. Cabe destacar dos de ellos:

- URBI: Del cual se pude encontrar información en la Sección 3.2.3.
- ROS: Es el marco de trabajo más extendido actualmente y abasta una gran cantidad de plataformas. En la Sección 5.1 se puede encontrar más información.

Por lo que respecta a la implementación de drivers para pasar de un lenguaje propio a ROS Existen varios ejemplos que se pueden encontrar actualmente en la web de ROS este es el caso de ARDrone o Bioloid.

### 2.1.3. AIBO

Con la plataforma de estudio AIBO Sección 3 se han realizado diversos proyectos de índoles muy diferentes dada su versatilidad. Sobretodo se han llevado a cabo proyectos relacionados con visión [3] o comportamientos e inteligencia artificial [8]. Un punto esencial en el desarrollo del AIBO fue que durante el período comprendido entre 1999 y 2008 fue

<sup>6</sup><http://www.wifibot.com/>

<sup>7</sup><http://www.lego.com/en-us/mindstorms/?domainredir=mindstorms.lego.com>

<sup>8</sup><http://www.aldebaran.com/en>

la plataforma elegida para realizar la competición RoboCup Estandard Plataform League, posteriormente sustituido por la plataforma NAO. La RoboCup es una competición internacional destinada a promover la robótica y donde se realizan pruebas como la comentada anteriormente que forma parte de la sección de fútbol con robots autónomos. La prueba de simulacros de rescate fue otra donde el AIBO tuvo una gran acogida y fue muy usado [5]. La RoboCup fue fuente de inspiración para muchos investigaciones que se pueden clasificar en tres grupos según su finalidad:

- Visión i reconocimiento de marcas para la posterior ubicación dentro del campo [6].
- Comunicaciones y control sobre los movimientos [7].
- Definición de roles dentro del campo [4].

Actualmente el AIBO no se usa como se hacia antes, en parte porque ya no es la plataforma estándar de la RoboCup y en parte porque desde que en 2006 SONY dejó de producirlo su lenguaje base OPEN-R dejó de ser mantenido. A medida que ha caído en desuso los softwares que los soportaban ha dejado hacerlo, incluso aquellos que nacieron especialmente para el AIBO. Es el caso de URBI que desde la versión 1.5 de su librería, liburbi, no es compatible. Además la recerca de éste proyecto se ha visto dificultada por la desaparición de la red de gran parte de la documentación de estos lenguajes.

Ha habido otros intentos de recuperación de la plataforma AIBO es el caso de un proyecto que trataba de hacer compatible la versión 2.3 de liburbi para AIBO. Otro proyecto del que solo se ha conseguido encontrar un artículo que lo mencione y nada de documentación trataba de realizar la integración mediante EusLisp de a ROS varias plataformas, entre ellas AIBO.

## 2.2. Objetivos

EL objetivo de este proyecto es recuperar al AIBO como plataforma educativa aportándole una capa superior de programación que permita la integración de la plataforma con ROS. Con este fin se compararan diversos caminos con tal conseguir el mejor resultado.

Como objetivos concretos se marcan:

- Buscar la mejor opción de comunicación, entendiendo como mejor un equilibrio entre sencillez y calidad de la comunicación. Ésto conllevará por un lado un análisis cualitativo de los posibles métodos y un análisis mas exhaustivo de los posibles caminos que hayan pasado la primera selección.
- La creación de un paquete de ROS que permita extraer todos los valores de los sensores y articulaciones y a su vez permita controlar al AIBO de forma remota.

- La creación de los paquetes necesarios para demostrar el buen funcionamiento del paquete anteriormente mencionado.
- La creación de un paquete que incluya un modelo del AIBO que pueda ser integrado dentro de los simuladores y visores del marco de trabajo ROS.

### 2.3. Alcance

El proyecto incluirá la creación de un paquete que permita extraer los valores de los sensores y articulaciones así como la acción sobre las articulaciones, pero no incluirá la adaptación de la cámara, micrófonos ni altavoces. El modelo incluirá la cinemática y permitirá el uso del modelo con la herramienta rviz y implementación de un nodo que permita su comunicación con la plataforma real. No incluirá la experimentación con física aplicada al simulador ni la creación de algún mapa.

### 3. AIBO ERS-7

En 1999 SONY lanzo al mercado la mascota robótica AIBO (Artificial Intelligence roBOt, amigo en japonés). AIBO es un robot conforma de perro que fue ideado para interaccionar con su dueño como una mascota real, es capaz de percibir estímulos del exterior mediante una serie de sensores. Lleva incorporado de serie un software llamad AIBO MIND dentro de una tarjeta de memoria que le permite comportarse como un perro real, reconocer a su dueño, reconocer objetos, jugar con una pelota y aprender, entre muchas otras cosas.

#### 3.1. Hardware

Sony ha desarrollado varios modelos y versiones del AIBO como del software AIBO MIND, mejorando tanto actuadores y sensores como la inteligencia.

Entre los diferentes modelos existentes se trabajará con el modelo ERS-7, que se caracteriza por tener una camera de mayor resolución y un procesador más potente.



Figura 3: AIBO ERS-7.

El AIBO ERS-7 tiene las siguientes características:

- Audio:
  - Entrada de audio: Micrófono estéreo.
  - Salida de audio: Altavoces de 20.8 mm i 500 mW.
- Sensores integrados:
  - Sensores de presión:
    - 1 sobre la cabeza.
    - 3 en la espalda.
  - 1 sensor de contacto en cada pata.
  - 1 sensor booleano bajo la boca.
  - Acelerómetros en x,y i z.

- 2 sensores de proximidad de infrarrojos situados en el morro y en el pecho.
  - Sensor de vibración.
- Grados de libertad: Están controlados con motores de continua seguido de una reductora y un encoder absoluto.
- 3 grados de libertad a cada una de las 4 patas.
  - 3 grados de libertad al cuello..
  - 1 grado de libertad a cada oreja.
  - 1 grado de libertad a la boca.
  - 2 grados de libertad a la cola.
- Entrada de imagen.
- Sensor de imagen CMOS de 350000 pixels.
  - Ángulos: 56.9° horizontal i 45.2° vertical.
  - Resoluciones: 208x160, 104x80, 52x40.
  - 30 frames por segundo.
- Dimensiones:
- Altura: 278 mm.
  - Largo: 319 mm.
  - Ancho: 180 mm.
  - Peso con batería: 1.65 Kg.
- CPU:
- Procesador: MIPS R7000 @ 576 MHz.
  - RAM: 64 MB.
  - Memoria flash: 4 MB.
- Conectividad:
- LAN inalámbrico IEEE 802.11b.
  - Xifrat: WEP.

### 3.2. El software

APERIOS es el sistema operativo en tiempo real que usan los AIBO. Está destinado y diseñado para poder trabajar con grandes flujos de datos de audio e imagen simultáneamente en tiempo real. En un principio AIBO iba a ser comercializado con una finalidad puramente lúdica, pero debido a su atractivo diseño y su robustez atrajo la atención de programadores que vieron el potencial para convertirse en una herramienta de investigación y aprendizaje. Ésto llevó a SONY a desarrollar un software de desarrollo llamado OPEN-R SDK y que usaba un lenguaje propio, OPEN-R. A raíz de este modulo salieron otros lenguajes e interfaces que trabajaban en una capa superior, sobre OPEN-R y APERIOS, como son *URBI*<sup>9</sup> o *Tekkotsu*<sup>10</sup>.

#### 3.2.1. OPEN-R

OPEN-R es un API en C++ que corre sobre el sistema operativo APERIOS 4. OPEN-R diferencia entre dos niveles, la capa de sistema por donde se accede al hardware del robot y la capa de aplicación que se trata de los programas hechos por el usuario.

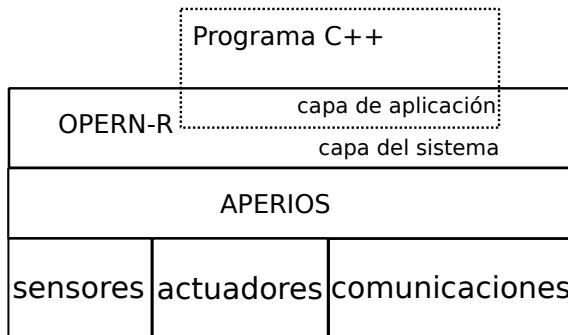


Figura 4: Arquitectura de AIBO con OPEN-R

Al tratarse de un lenguaje de bajo nivel su estructura es inherente al sistema operativo que se basa en objetos que interaccionan entre si mediante mensajes o meta-objetos. El concepto de objeto es parecido al utilizado en los sistemas *UNIX*<sup>11</sup> y *Windows*<sup>12</sup>, con la diferencia de que son monohilo y que la comunicación se realiza mediante mensajes que incluyen datos y un identificador el método en el que se ejecutará en el objeto destino. Esto implica que cada objeto tiene varios puntos de entrada y salida que son los métodos: DoInit(), DoStart(), DoStop(), DoDestroy(). En el paso de mensajes uno se define como sujeto (el que envía) y el otro como observador (el que recibe) que inicia

<sup>9</sup>[www.urbiforge.org](http://www.urbiforge.org)

<sup>10</sup>[tekkotsu.org](http://tekkotsu.org)

<sup>11</sup>[www.unix.org](http://www.unix.org)

<sup>12</sup><http://windows.microsoft.com/en-us/windows/home>

su ejecución después de que el mensaje haga saltar el evento del método correspondiente [2].

SUJETO	OBSERVADOR
envío de datos	
notificación del evento	
	recepción de datos
	evento preparado

Cuadro 1: Estructura del paso de mensajes en OPEN-R

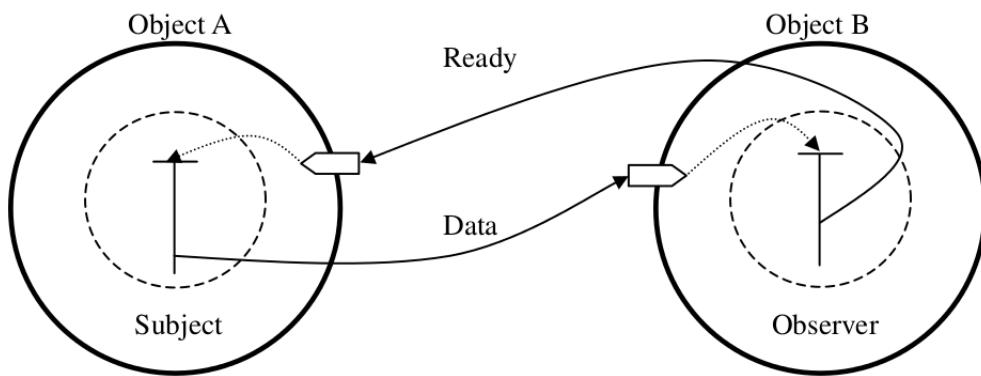


Figura 5: Comunicación entre objetos.

Una de las mayores complicaciones que acarrea trabajar con OPEN-R es que usa una gran variedad de archivos que deben ser modificados para poder configurar el programa de aplicación que se realice. Por ello es importante conocer bien los archivos con los que trabaja, los más importantes están listados a continuación:

- Archivos .h i .cc: Son los archivos de programación de C++.
- STUB.config: Son archivos de configuración donde se define como un objeto se comunica con los demás.
- Archivos .ocf: Definen el comportamiento en cuanto a tiempos de ejecución, memoria y prioridad de ejecución.
- Makefile: Archivo para la configuración global de la compilación.
- OBJECT.cfg: Listado de objetos que se ejecutarán.
- CONECT.cfg: Se determinan las conexiones entre objetos que se ejecutarán.

- WLANCONFIG.txt<sup>13</sup>: Donde se definen las características de la conexión inalámbrica del AIBO.

### 3.2.2. Tekkotsu

Tekkotsu es un software de desarrollo para robots móviles. Orginalmente fue creado para AIBO, pero actualmente permite programar otros plataformas com el Chiara<sup>14</sup>, iRobot Create<sup>15</sup>, HandEye<sup>16</sup> o Lynxmotion Arms<sup>17</sup>.

Mantiene la arquitectura semejante a OPEN-R, basada en objetos y paso de eventos y mantiene la herencia de C++. Proporciona una capa de mayor nivel que OPEN-R pero permite llamar a OPEN-R para acceder de forma rápida a sensores, actuadores y sistemas de comunicación, lo que significa que la capacidad de control sobre el robot se encuentre limitada por el propio hardware y no por el Tekkotsu. Por otro lado permite al usuario trabajar en alto nivel con herramientas de procesamiento visual básico, interfaces de monitorización, modelos de cinemática inversa y soporte para la administración de redes inalámbricas.

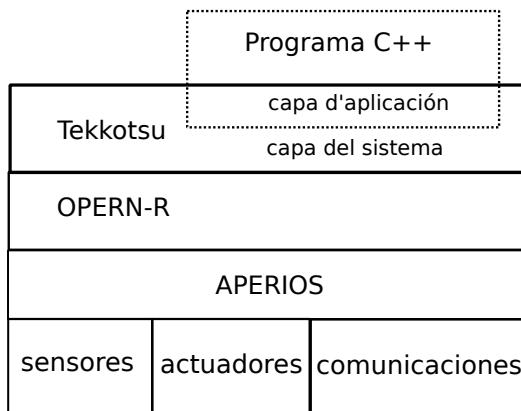


Figura 6: Arquitectura del AIBO con Tekkotsu.

Además Tekkotsu proporciona una interfaz de usuario, ControllerGUI, que permite acceder al AIBO remotamente y ejecutar los comportamientos que estén programados en la tarjeta de memoria. Estas funciones se pueden realizar mediante la interfaz gráfica basada en Java como por una conexión telnet<sup>18</sup> al puerto 10020 [9].

Tekkots se organiza como un conjunto de comportamientos o *behaviors* y clases llamadas MotionComand. Sus funciones se ejecutan en dos procesos, el *Main* y el *Motion*.

<sup>13</sup>Este ultimo archivo es necesario configurarlo para cualquier método de programación, URBI, Tekkotsu o OPEN-R. Ver APENDIX

<sup>14</sup><http://chiara-robot.org>

<sup>15</sup><http://chiara-robot.org/Create/>

<sup>16</sup><http://chiara-robot.org/HandEye/>

<sup>17</sup><http://www.lynxmotion.com/>

<sup>18</sup><http://es.wikipedia.org/wiki/Telnet>

El primero se encarga de la percepción i toma de decisiones y el segundo hace referencia al control en tiempo real de los actuadores. Existe un tercer proceso que se encarga de la salida de audio [10].

Los comportamientos, como se llama a todo programa de realizado en Tekkotsu, igual que sus análogos en OPEN-R, tienen una estructura basada en unos métodos que hay que respetar: doStart() y doStop(). Éstos simplifican los cuatro métodos de OPEN-R, pero permiten mantener la estructura y hacen más sencilla y rápida la comunicación.

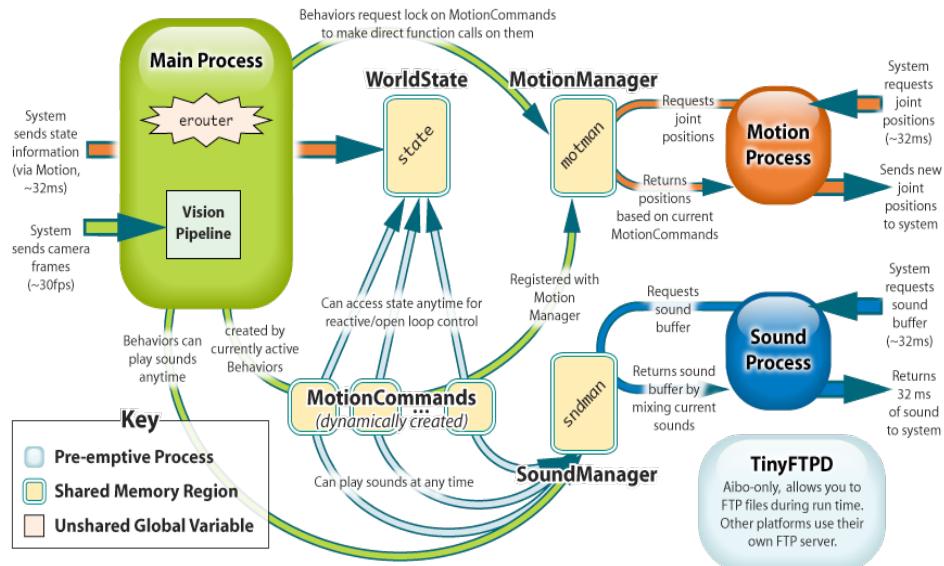


Figura 7: Proceso de ejecución de un comportamiento en Tekkotsu.

### 3.2.3. URBI

URBI es el acrónimo de Universal Real-Time Behavior Interface y se trata de una plataforma de software libre para controlar y programar robots y sistemas automatizados en general. Algunos de los robots móviles que soporta URBI son AIBO, Bioloid, Mindstorm NXT, Pioneer, Wifibot o ARDrone [11].

URBI es un lenguaje basado en scripts de alto nivel con la ventaja que permite ejecutar comandos en paralelo. Existen dos formas de trabajar con URBI: La primera se trata de usar el lenguaje de script con la intención de que este sea interpretado como un objeto de OPEN-R dentro de la tarjeta de memoria. La segunda consiste en una arquitectura cliente/servidor, donde el servidor es el AIBO, que permite enviar los scripts a través del terminal de URBI o bien enviar macros o órdenes concretas usando la librería liburbi,

Figura 8.

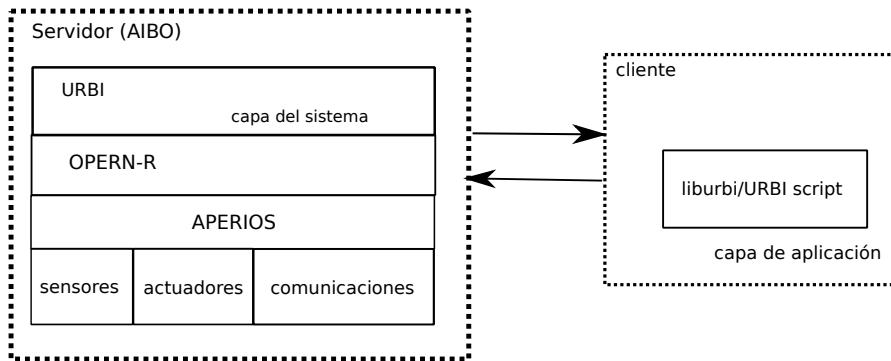


Figura 8: Arquitectura del AIBO con URBI.

La segunda opción abre varias vías de programación. Por un lado permite comunicarse por telnet y enviar comandos y por otro usar la librería liburbi que permite trabajar con C++, Java y Matlab

URBI permite y facilita el acceso a cada una de las articulaciones y sensores remotamente sin necesidad de implementar un servidor. Por ejemplo para consultar el valor de la articulación superior de la pierna izquierda se puede enviar el comando `legLF1.val;` y para asignarle un valor `legLF1.val=20;[12]`.

## 4. Comparación de alternativas para la programación

Teniendo en cuenta la pretensión y objetivo final es poder programar AIBO con ROS se ha buscado la forma de atacar el problema desde todos los flancos. Una de ellas y completamente diferente a las demás ha sido buscar un punto de acceso al hardware con la intención de poder instalar alguna distribución de linux, lo que permitiría implementar el modulo de ROS de como un sistema embebido. Pero tras desmontar gran parte de las piezas solo se encontró un puerto que no coincidía con ningún estándar y se abandonó dicho camino.

Solo queda usar una arquitectura cliente/servidor de forma que el modulo de ROS se ejecute en el cliente remoto, pero sea capaz adquirir de forma sencilla y rápida el estado del AIBO y a la vez actuar sobre el.

Se plantean tres posibilidades de programación: OPEN-R, Tekkotsu y URBI. En los tres casos es imprescindible programar el cliente en el correspondiente lenguaje. Pero respecto al servidor URBI ahorraría tener que desarrollarlo puesto que el sistema operativo nos permite interactuar de forma remota. Tekkotsu permite interactuar de forma remota pero no hay ningun comportamiento que permita enviar ordenes a los actuadores y habría que implementarlo. Por ultimo con OPEN-R habría que implementar el servidor entero tanto el envío como la recepción. De todos modos se ha probado de instalar y establecer los tres marcos de trabajo, Anexo A, y probado los tres lenguajes para comprobar su dificultad y su funcionamiento. Respecto a OPEN-R se ha encontrado muy poca documentación y el comienzo de su aprendizaje ha sido realmente duro tanto el lenguaje en si como la configuración de todos los archivos. Tekkotsu parece un lenguaje más sencillo y no necesita apenas configurar archivos pero la única documentación encontrada ha sido para la ultima versión que no se ha conseguido compilar dentro de la tarjeta de memoria. En cambio si se ha logrado con una versión más antigua, la nueva parece no ser compatible con el compilador de OPEN-R, y se han podido hacer algunas pruebas aunque sin documentación de la versión y habiendo cambiado bastante las funciones, ha implicado un gran esfuerzo. Por lo que URBI se refiere su uso es realmente sencillo des del terminal , se ha encontrado bastante documentación para el uso en modo script y algo menos para el uso de liburbi. El único problema que hay que tener en cuenta es que la libreria liburbi 1.5, la más reciente que es compatible con AIBO, no es compatible con un sistema linux de 64 bits.

	<b>OPEN-R</b>	<b>Urbi</b>	<b>Tekkotsu</b>
Necesidad de programar un servidor.	Si	No	Si
Necesidad de programar un cliente.	Si	Si	Si
Permite la consulta del estado por telnet.	No	Si	Si
Permite accionar las articulaciones por telnet.	No	Si	No
Dificultad de programación (0-5)	5	2	3
Documentación (0-5)	2	3	1
Plataforma del PC.	Windows/ Linux/ OS	Windows/ Linux 32bits/ OS	Linux

Cuadro 2: Comparación entre lenguajes usados sobre AIBO

En vista de la Tabla 2 y de lo anteriormente comentado se descarta usar Tekkotsu por haber encontrado muy poca documentación de la versión que ha podido ser compilada. Si bien seria OPEN-R sería una buena opción ya que es la capa mas baja de programación que se permite tocar su aprendizaje es realmente duro y posiblemente no habría tiempo de implementar el modulo deseado. Por lo tanto se partirá de URBI como lenguaje dado su fácil uso y que proporciona unas buenas herramientas para alcanzar el objetivo.

Partiendo del lenguaje URBI se plantean dos opciones de desarrollo:

- Usar liburbi con C++.
- Usar un script de python que use la terminal de URBI bajo una conexión telnet.

Con tal de encontrar cual es la mejor opción se realizarán una serie de experimentos antes de implementar el paquete.

#### 4.1. Lectura de datos

En el primer experimento se valorará la velocidad de lectura de una sola variable del sistema y se cuantificará su frecuencia de refresco. Se ha usado para este experimento el valor de una articulación.

El procedimiento para adquirir los datos será el mostrado en la Figura 9.



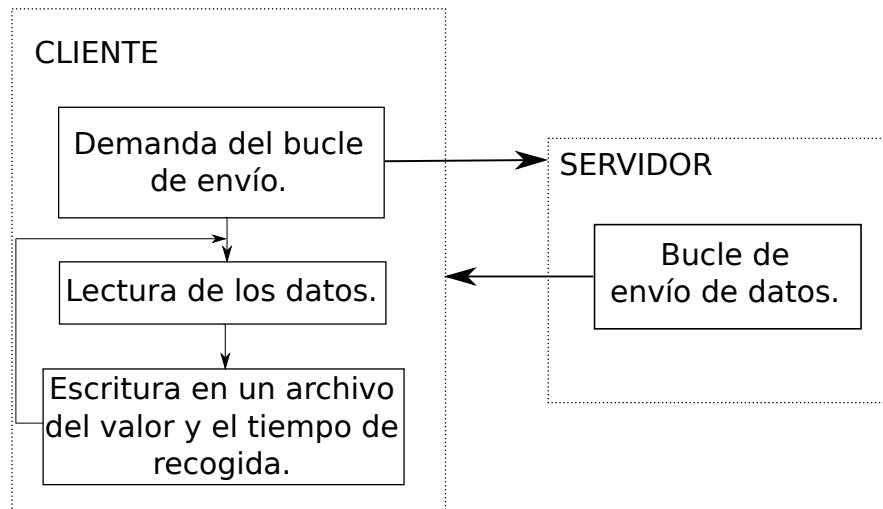


Figura 9: Estructura de la lectura de una variable.

La escritura de los valores de la articulación y el tiempo se escriben en un archivo de texto para su posterior tratamiento.

El experimento consistirá en realizar la adquisición de datos durante 10 segundos y repetir el experimento 15 veces por método. La velocidad de envío estará limitada por el servidor ya que se le pide que envíe el refresco continuo de la variable.

#### 4.1.1. Método con liburbi y C++

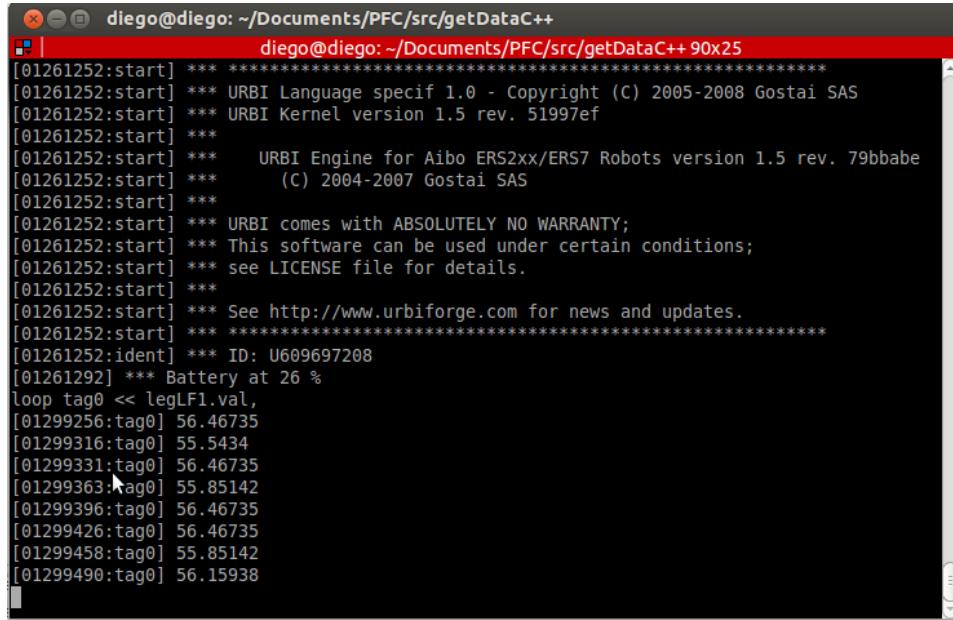
La estructura del programa es la que sigue<sup>19</sup>.

- Inicialización del cliente URBI.
- Inicialización del callback: Éste es llamado cada vez que se reciba un dato con la etiqueta asignada.
  - Se toma el valor de la variable (aunque no se necesaria para este experimento).
  - Se consulta el tiempo en que ha sido recibido el dato.
  - Se guarda el dato y el tiempo.
- Demanda del bucle y asignación de una etiqueta.
- Inicialización del archivo donde se guardarán los datos.
- Ejecución del bucle de URBI: Se trata de una función que crea un bucle de comunicación cliente/servidor.

<sup>19</sup>El script se puede encontrar en el Anexo C.1

#### 4.1.2. Método con telnet y python

La idea de este script<sup>20</sup> es trabajar con la terminal de URBI, Figura 10, mediante una conexión telnet.



```
diego@diego: ~/Documents/PFC/src/getDataC++
diego@diego: ~/Documents/PFC/src/getDataC++ 90x25
[01261252:start] *** *****
[01261252:start] *** URBI Language specif 1.0 - Copyright (C) 2005-2008 Gostai SAS
[01261252:start] *** URBI Kernel version 1.5 rev. 51997ef
[01261252:start] ***
[01261252:start] ***      URBI Engine for Aibo ERS2xx/ERS7 Robots version 1.5 rev. 79bbabe
[01261252:start] ***      (C) 2004-2007 Gostai SAS
[01261252:start] ***
[01261252:start] *** URBI comes with ABSOLUTELY NO WARRANTY;
[01261252:start] *** This software can be used under certain conditions;
[01261252:start] *** see LICENSE file for details.
[01261252:start] ***
[01261252:start] *** See http://www.urbiforge.com for news and updates.
[01261252:start] ***
[01261252:ident] *** ID: U609697208
[01261292] *** Battery at 26 %
loop tag0 << legLF1.val,
[01299256:tag0] 56.46735
[01299316:tag0] 55.5434
[01299331:tag0] 56.46735
[01299363:tag0] 55.85142
[01299396:tag0] 56.46735
[01299426:tag0] 56.46735
[01299458:tag0] 55.85142
[01299490:tag0] 56.15938
```

Figura 10: Terminal URBI consultando un dato.

La estructura del programa será parecida a la anterior:

- Inicialización del objeto telnet.
- Lectura de la terminal y eliminación de la cabecera de URBI.
- Escritura de la comanda para recibir el bucle de envío. Para facilitar la lectura de los datos y diferenciar entre uno i el anterior se hace enviar una marca después de cada dato.
- Bucle de lectura y escritura.
  - Tratamiento de la lectura:
    - Lectura hasta encontrar la marca la marca.
    - Eliminación de las cadenas cortadas por el envío de la información de la batería<sup>21</sup>
    - Segmentación de la cadena para finalmente guardar sólo el valor.
  - Adquisición del tiempo.
  - Escritura del tiempo y el valor en el archivo de texto.

<sup>20</sup>Se puede consultar el script en el Anexo C.2

<sup>21</sup>Cada vez que la batería baja la carga se escribe el porcentaje restante por la terminal de URBI.

#### 4.1.3. Comparación de los resultados

De los datos obtenidos de las 15 replicas de cada método se han hecho las diferencias de tiempos entre un dato y el siguiente y se ha echo el inverso de este intervalo con tal de obtener la frecuencia entre datos.

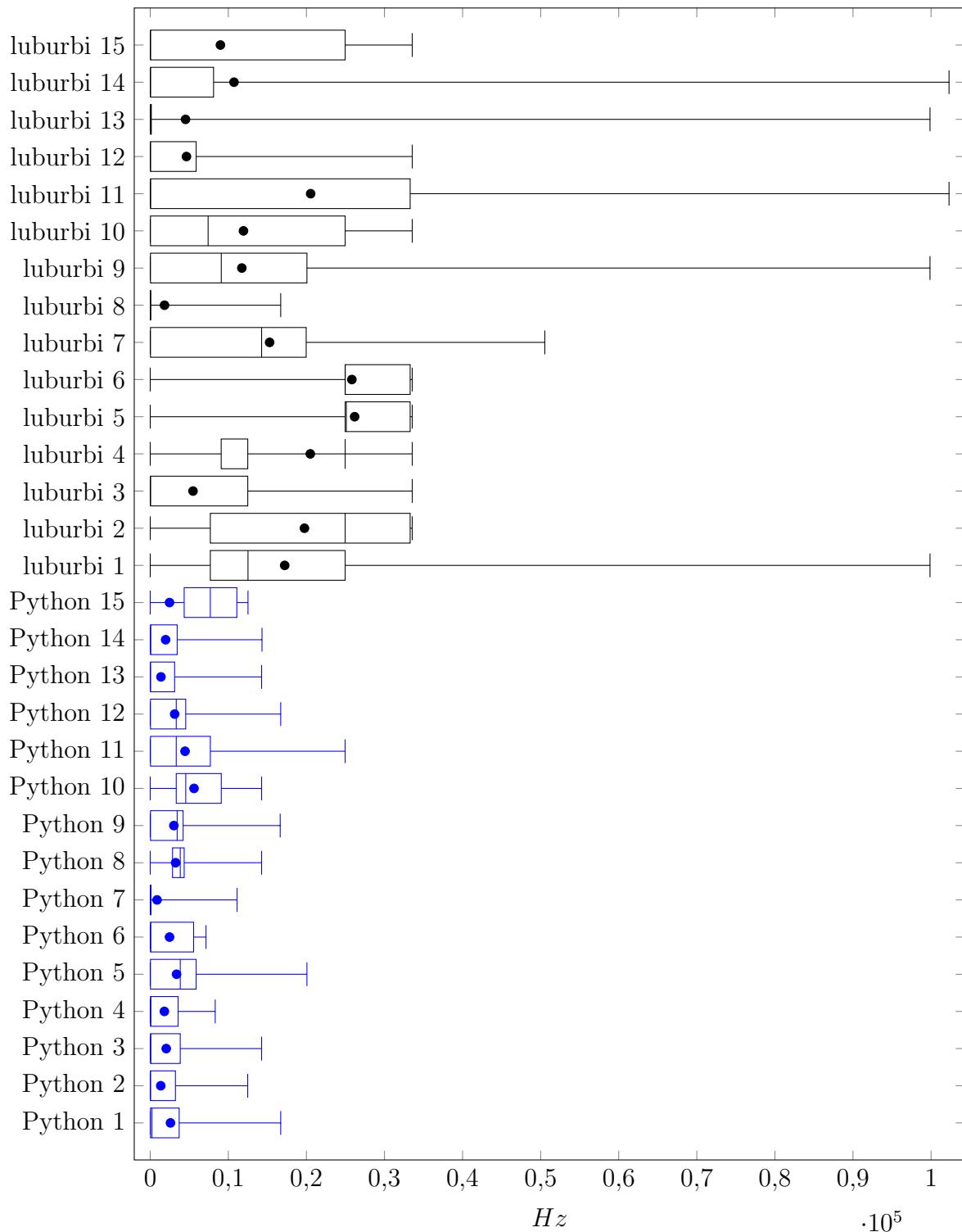


Figura 11: Diagrama de cajas de la frecuencia de datos para las 15 replicas de cada método.

El gráfico de la Figura 11 muestra como claramente al trabajar con liburbi se obtienen medias más altas en la mayoría de replicas y de la misma manera gran parte de los datos se envían más rápido. Aun así los valores mínimos parecen muy similares y la variabilidad es mucho mayor con liburbi, echo que no interesa.

Es interesante observar los mínimos de cada serie en más detalle pues nos indican cuan largos son los bloqueos que se producen y si existe alguna diferencia entre ambos programas, Figura 12. Si bien parece que liburbi tiene una tendencia a tener unos mínimos más bajos se ha realizado un análisis sobre la varianza (ANOVA) de los mínimos. Los resultados han sido que con intervalo de confianza del 95 % no se puede afirmar con un p-valor de 0.4 que los valores provengan de distintas poblaciones.

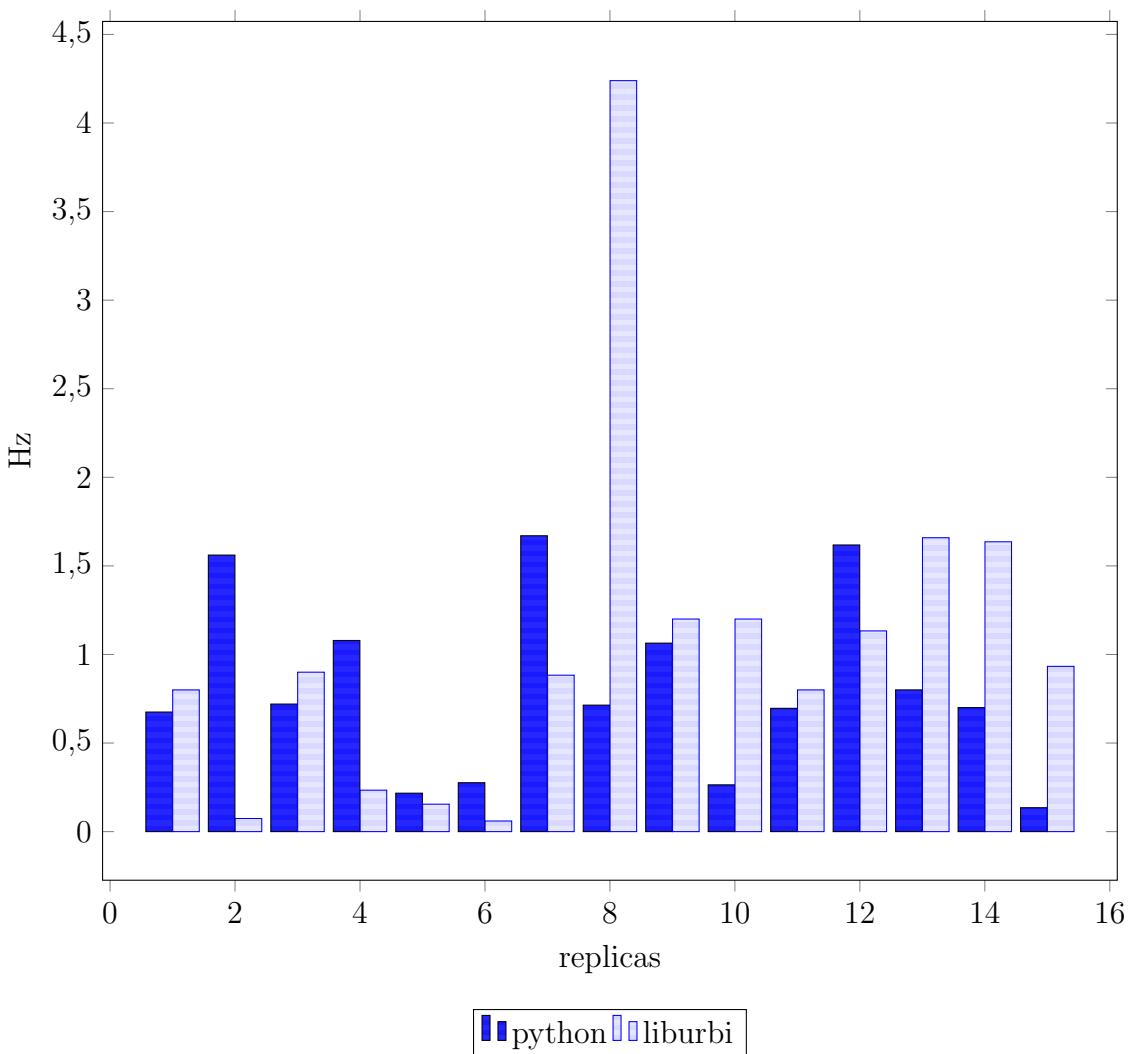
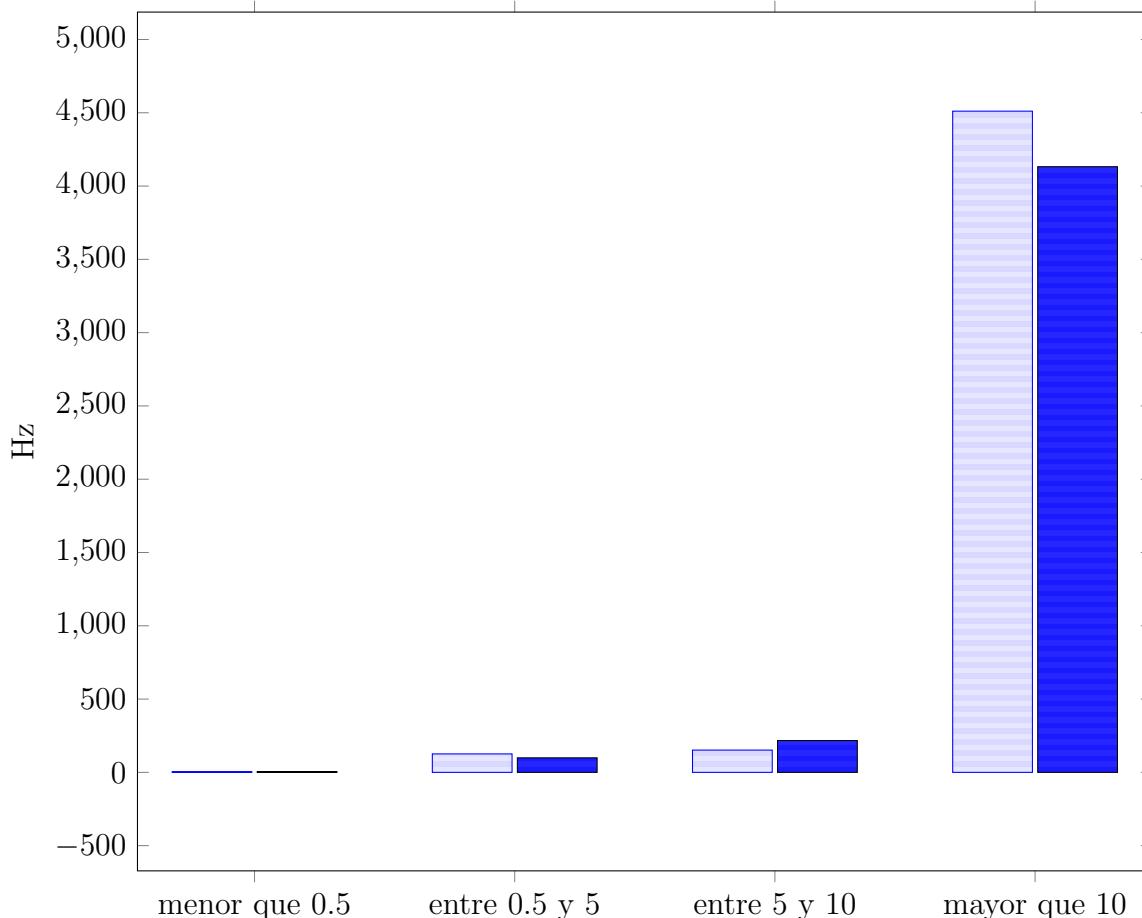


Figura 12: Mínimos de la frecuencia en les 15 repliques de los dos métodos.

Por otro lado se ha observado, no la durada de estos pequeños bloqueos sino la cuantos se producen en cada caso. Para ello se ha realizado un gráfico donde se clasifican unos intervalos de frecuencias.

- Menor que 0.5 Hz: Este intervalos se consideran bloqueos y son los menos deseados.
- De 0.5 a 5 Hz: No se consideran bloqueos pero es una frecuencia demasiado baja para trabajar de forma remota con un robot móvil.
- De 5 a 10Hz: Se trata de una velocidad aceptable.
- Mayor que 10: Se considera la frecuencia de trabajo ideal.



python liburbi

Figura 13: Histograma de la frecuencia en las 15 repliques de los dos métodos.

La gran mayoría de los datos, según la Figura 13, se encuentran en la zona de frecuencias deseadas y ambos métodos tienen un comportamiento muy parecido. En la franja considerada como bloqueos, en las 15 replicas que equivaldrían a unos 150 segundos de funcionamiento, se han dado 4 casos en liburbi y 5 en python.

A la vista de los resultados obtenidos se querido comprobar como afecta la cantidad de datos a enviar en el tiempo de recepción. Para ello se realizó el mismo experimento pero leyendo todas las articulaciones en vez de solo una.

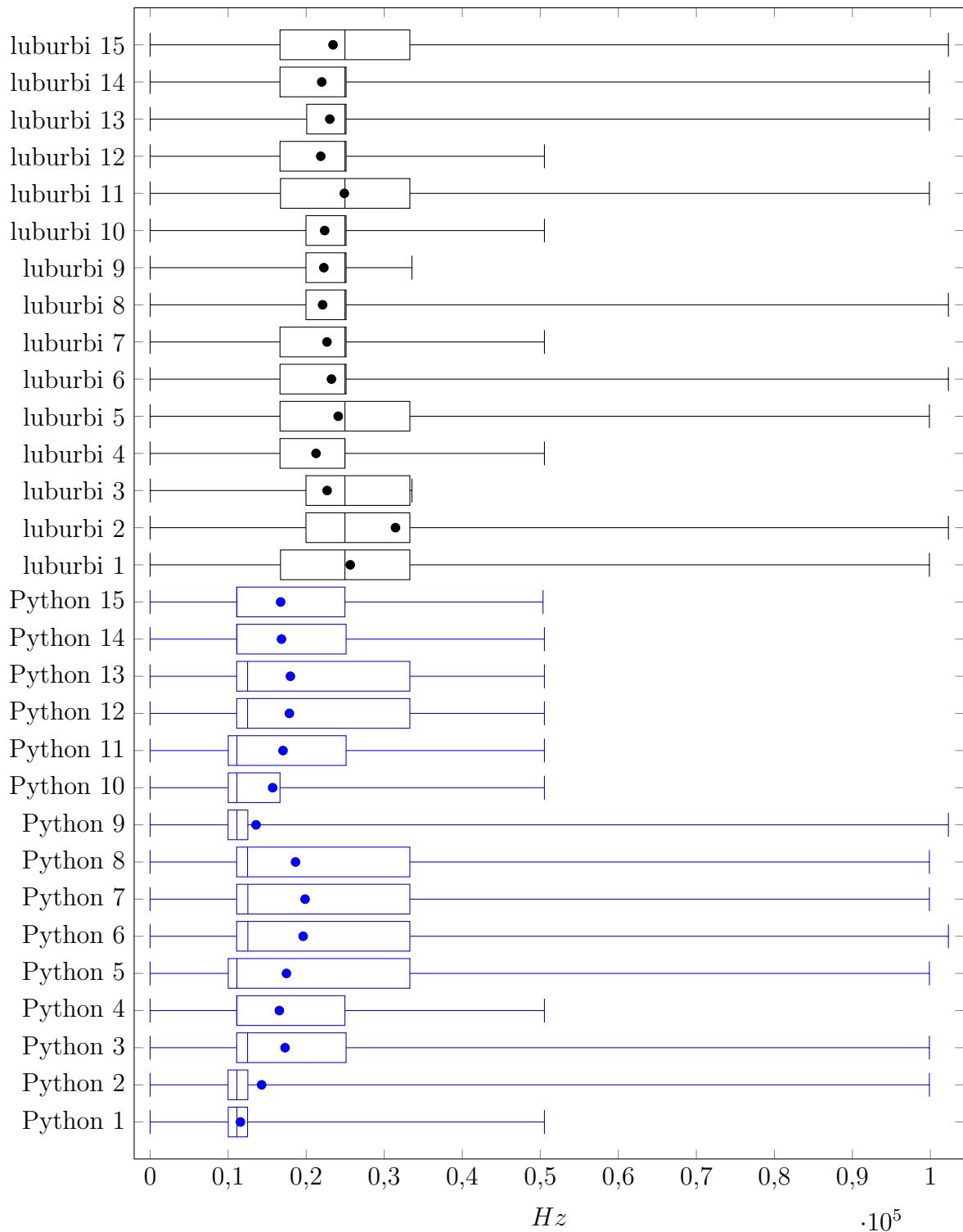


Figura 14: Diagrama de cajas de la frecuencia de datos para las 15 replicas de cada método con todas las articulaciones.

Bajo este experimento, Figura 14, ambas distribuciones son más parecidas aunque se puede distinguir la tendencia a obtener valores más altos del método con liburbi.

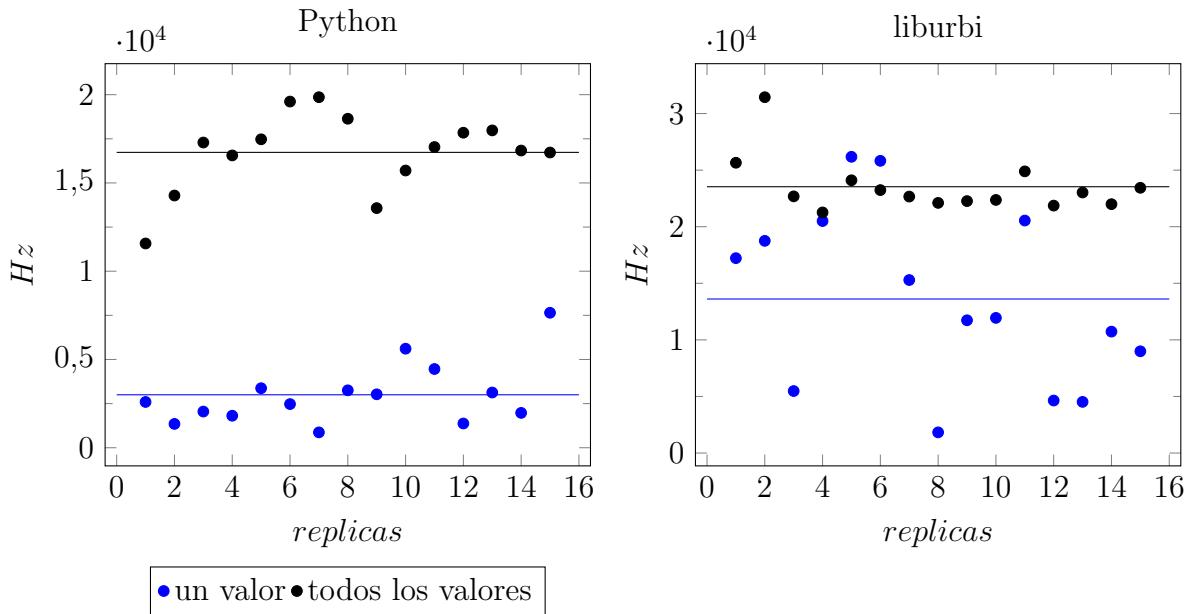


Figura 15: Medias de las frecuencias de envío, comparando una el obtener el valor de una articulación con obtener el valor de todas las articulaciones.

La velocidad de envío es significativamente más alta en ambos casos aunque en el caso de python se muestra mas evidente Figura 15. Una explicación es que el conjunto de valores se envíe como un solo paquete, en ese caso se está midiendo la frecuencia a la que las variables están disponibles, que a fin de cuentas es lo que realmente importa dado que se va a trabajar con ellas. Al parecer ante este echo responde mejor la lectura del terminal con python que no la adquisición mediante el callback de liuburbi. De todos modos se ha podido comprobar que la cantidad de variables a enviar no tiene un efecto negativo en las frecuencias medias.

Poniendo la atención en los bloqueos, Figuras 16 y 17, se puede asegurar el mismo comportamiento que en el experimento anterior.

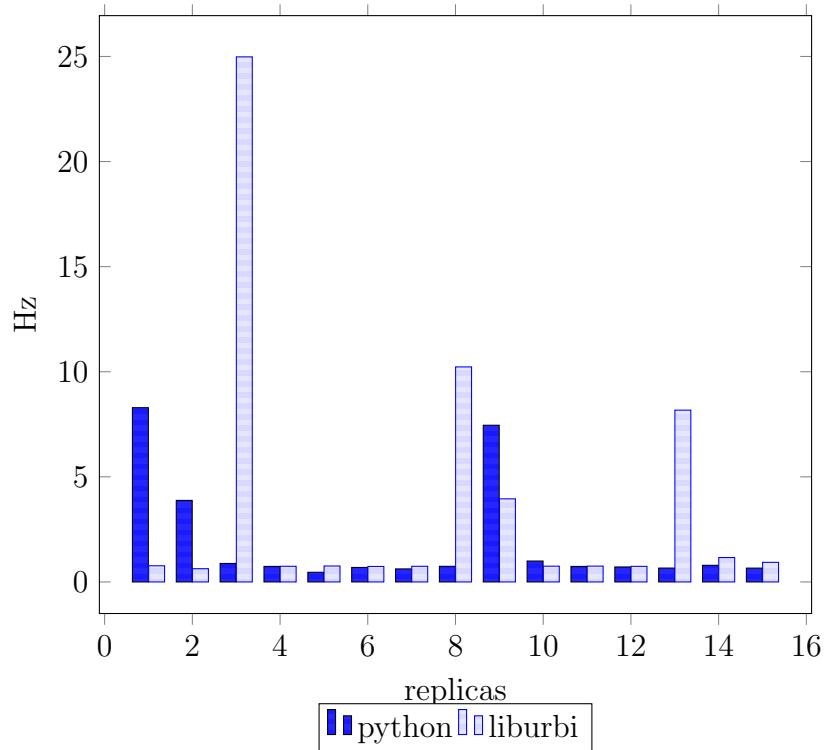


Figura 16: Mínimos de la frecuencia en les 15 repliques de los dos métodos.

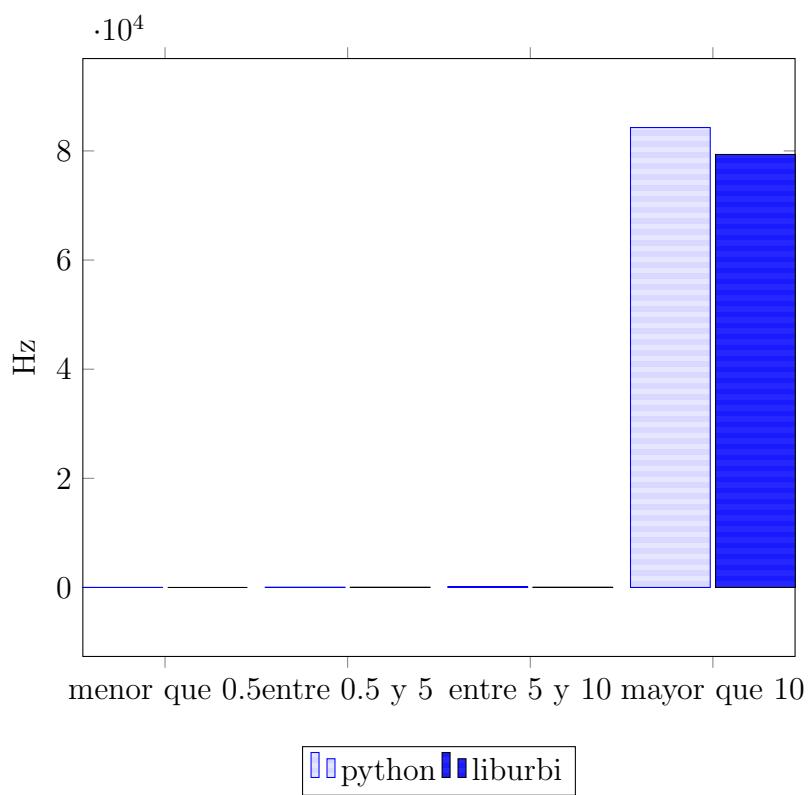


Figura 17: Histograma de la frecuencia en les 15 repliques de los dos métodos.

## 4.2. Envío de datos

Vista una de los sentidos de comunicación el siguiente experimento se va a fijar en el otro, el envío de los valores de las articulaciones del cliente al servidor con tal de controlar al AIBO. El experimento consiste en enviar una trayectoria punto a punto a una articulación y comprobar que tal responde. La trayectoria será sinusoidal sin importar el periodo. Para valorar los resultados se leerá la respuesta de la articulación usando los módulos de lectura del experimento anterior, que condicionarán el resultado por sus errores y limitaciones.

En este caso se ha contado con 3 opciones a comparar. Dado que por la terminal de URBI usando el cliente telnet no se ha permitido hacer la lectura y la escritura a la vez se ha tenido que hacer con dos clientes telnet. Esto ha planteado la duda si en liburbi tenía algún efecto el hecho de escribir y leer a la vez por el mismo cliente aunque este lo permita. Por este motivo se han planteado los siguientes programas:

- Python con un cliente telnet para lectura y otro para escritura.
- C++ con un cliente URBI tanto para lectura como para escritura.
- C++ con un cliente URBI para lectura y otro para escritura

En el desarrollo de los tres programas ha habido dos hechos a tener en cuenta y de destacada importancia. Ha sido necesario la creación de un hilo de ejecución en paralelo con tal de tener dos bucles independientes dentro del cliente, el bucle de URBI y el de envío. Por otro lado se ha investigado y experimentado con los modos de tratamiento de órdenes que tiene URBI:

- **normal**: Es el modo por defecto. En caso de conflicto la última asignación tiene prioridad, pero en caso de que la última acabe la anterior toma el relevo.
- **mix**: En caso de conflicto el valor asignado es una media de los valores en conflicto.
- **add**: El valor asignado es la suma de los valores en conflicto.
- **queue**: se forma una cola de entrada que se resuelve con un sistema FIFO (First In First Out).
- **discard**: En caso de conflicto el valor de la variable no se modifica.
- **cancel**: La última asignación toma prioridad y las anteriores son canceladas.

Tras varias pruebas se ha decidido que el modo más conveniente para la implementación del módulo es el modo **cancel**.



El experimento<sup>22</sup> se ha realizado varias veces cambiando la frecuencia a la que se envían los puntos de la trayectoria. Se ha realizado a 1, 2, 5 y 10 Hz, no se ha enviado a mas velocidad ya que el movimiento a una frecuencia de mas de 10 Hz era inconstante y poco suave. Se han descartado los experimentos en que se ha producido un bloqueo.

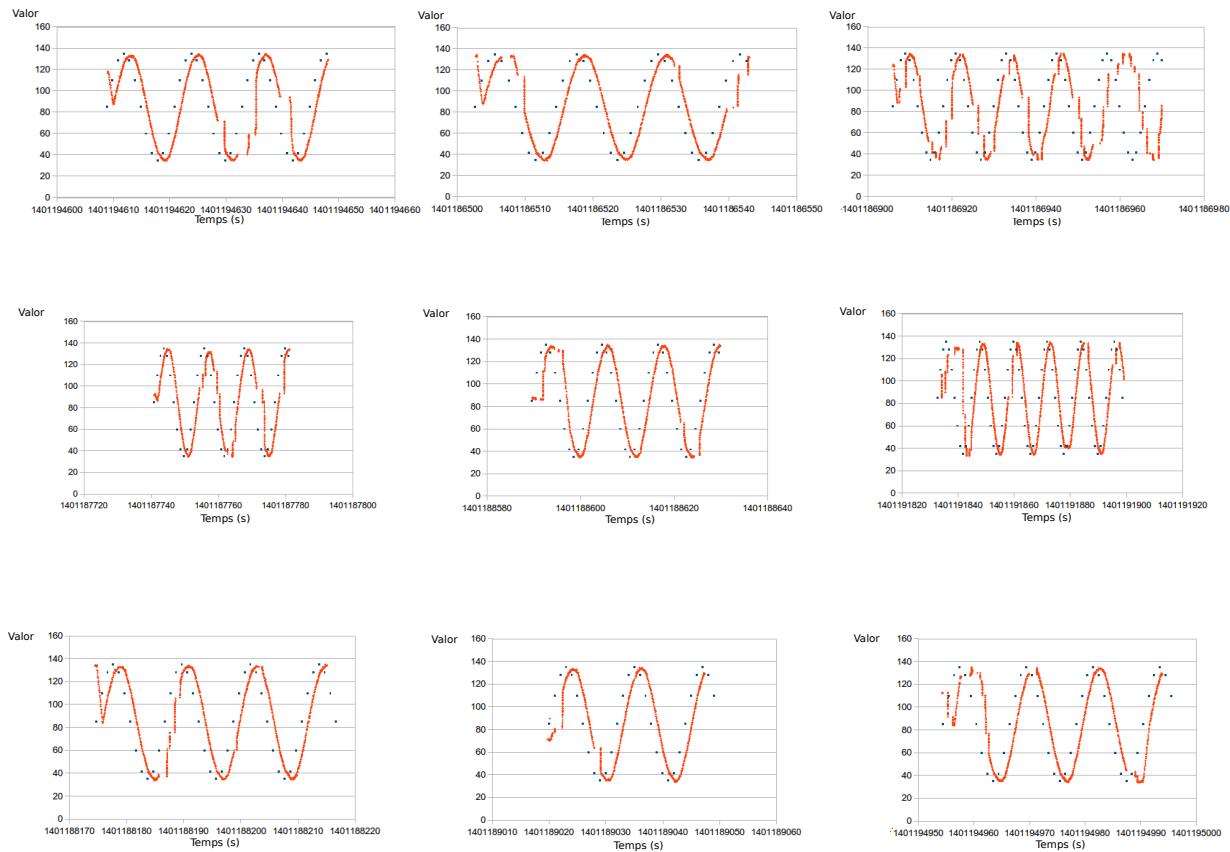


Figura 18: En azul está graficado las posiciones enviadas a 1Hz y en rojo las lecturas. En columnas se encuentran las replicas del mismo método y por filas los tres métodos usados, en orden descendente son liburbi con un cliente, liburbi con dos cliente y python.

<sup>22</sup>Los codigos se puede encontrar en los Anexos C.3 y C.4

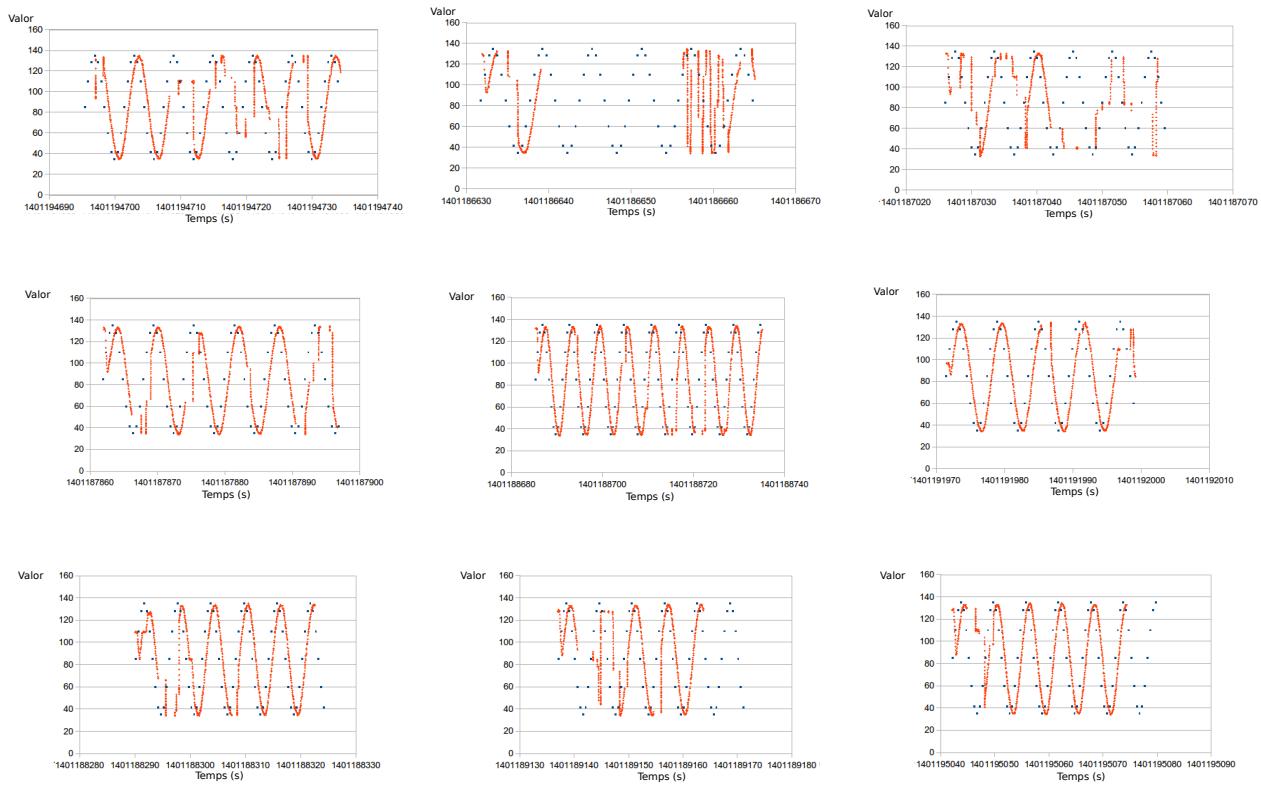


Figura 19: En azul está graficado las posiciones enviadas a 2Hz y en rojo las lecturas. En columnas se encuentran las replicas del mismo método y por filas los tres métodos usados, en orden descendente son liburbi con un cliente, liburbi con dos cliente y python.5

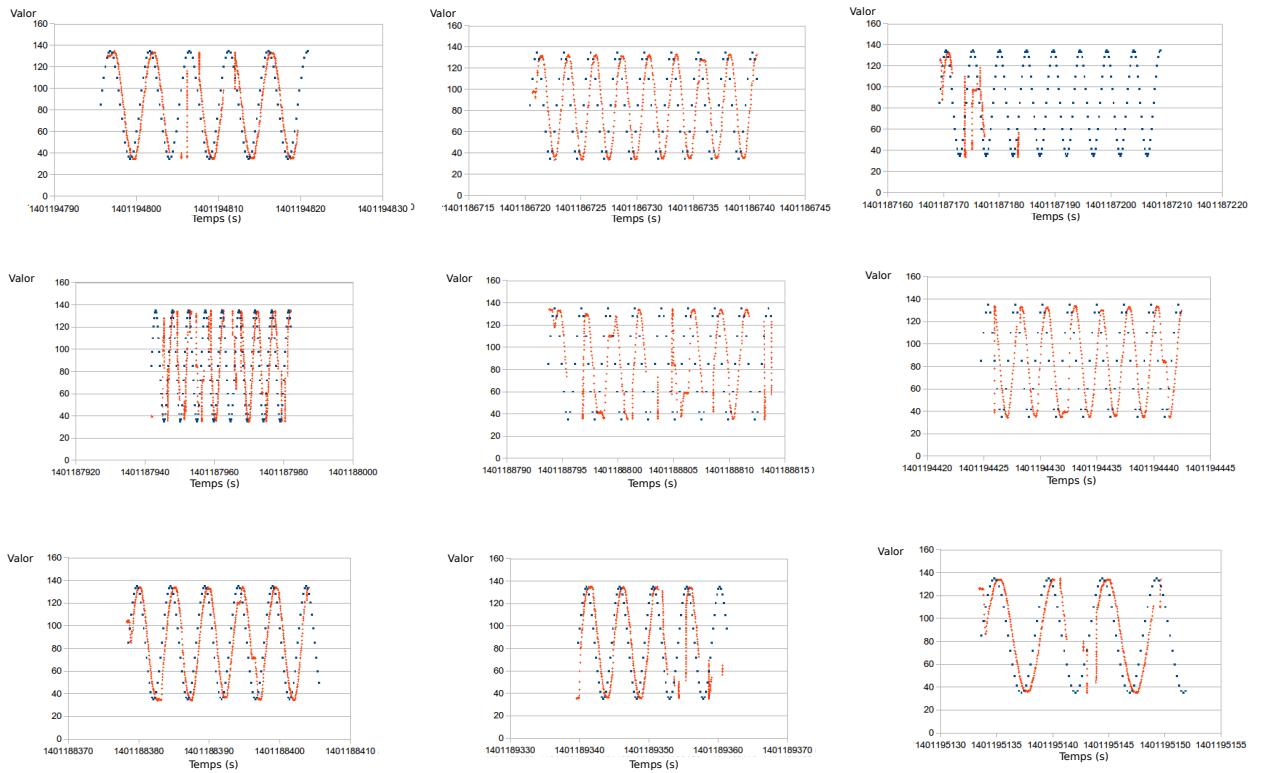


Figura 20: En azul está graficado las posiciones enviadas a 5Hz y en rojo las lecturas. En columnas se encuentran las replicas del mismo método y por filas los tres métodos usados, en orden descendente son liburbi con un cliente, liburbi con dos cliente y python.

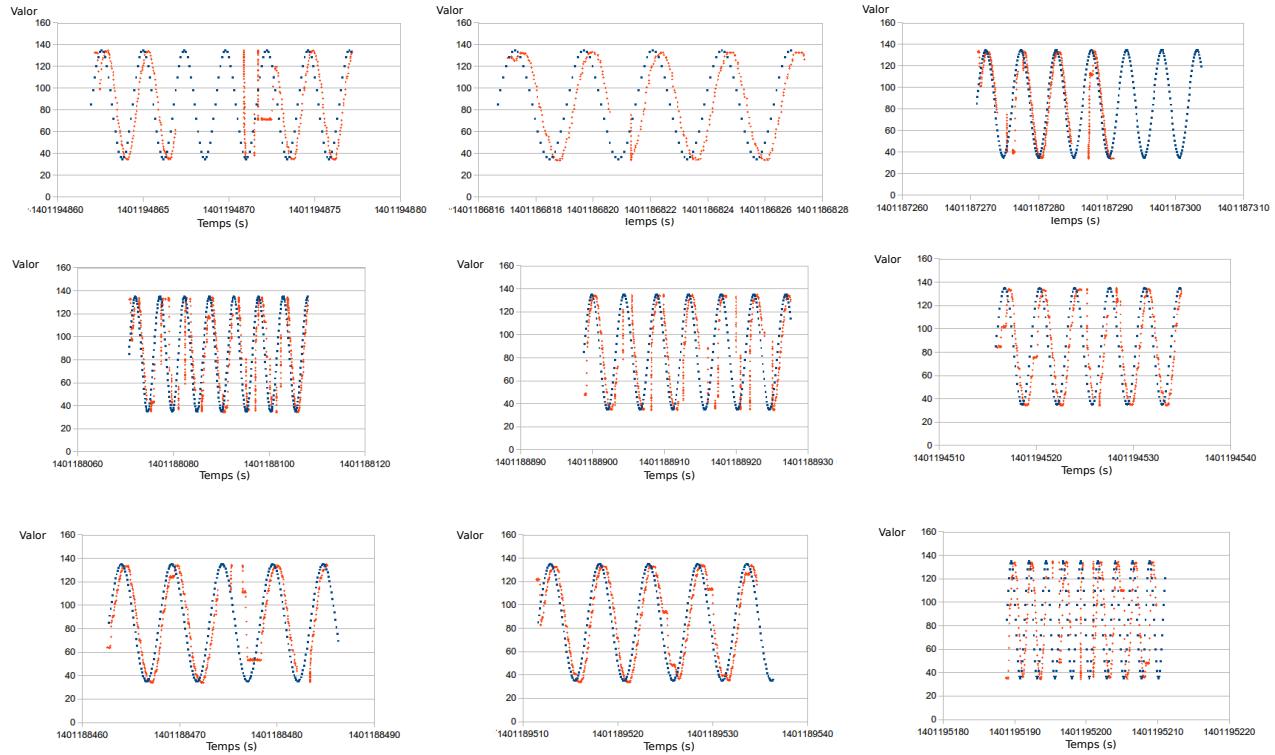


Figura 21: En azul está graficado las posiciones enviadas a 10Hz y en rojo las lecturas. En columnas se encuentran las replicas del mismo método y por filas los tres métodos usados, en orden descendente son liburbi con un cliente, liburbi con dos cliente y python.

En los gráficos de las Figuras 18, 19, 20 y 21 se pueden distinguir dos tipos de errores por los cuales no se sigue bien la trayectoria:

- La articulación no se mueve: Queda reflejado como una recta roja horizontal y el modulo responsable es el de escritura.
- No se recibe la posición actual: Queda reflejado como un periodo sin datos. en este caso no se puede saber si la articulación se movía o no<sup>23</sup>.

El caso en el que sólo se usa un solo cliente con liburbi tiene grandes bloqueos en la recepción de y muestra un movimiento más brusco e inconstante. Entre los dos otros no se puede determinar cual de los dos tiene un mejor seguimiento. Con el fin de marcar la diferencia se ha evaluado bajo otro criterio. Se han hecho las diferencias de entre el tiempo en que se envía la orden de los picos de la sinusode y el momento en que se recibe que

<sup>23</sup>Por inspección visual se ha comprobado que ambos errores son independientes.

ha llegado a tal posición. Para comprobar si existe una diferencia se realiza una ANOVA sobre los datos de la Figura 22 y resuelve con un intervalo de confianza del 95 % y un p-valor del 0.043 no se acepta la hipótesis nula y que por tanto el retraso con C++ es significativamente mayor.

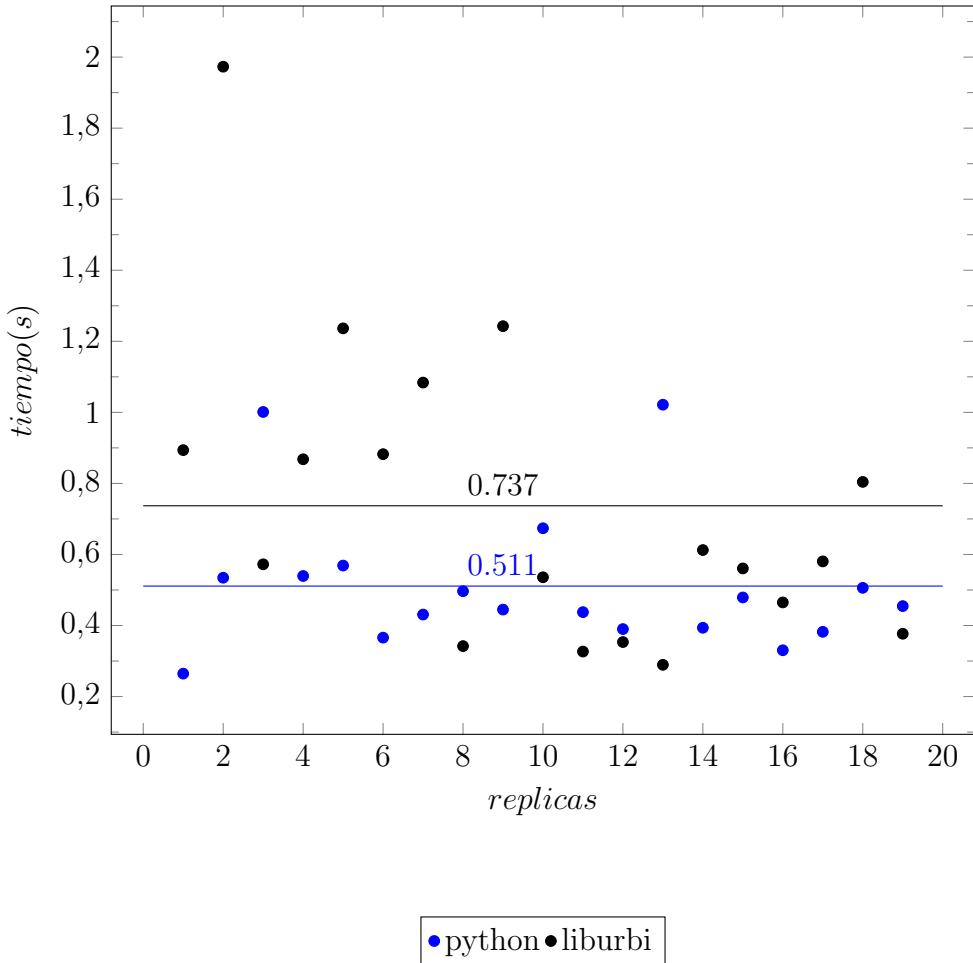


Figura 22: Retrasos entre la orden enviada y la acción para los métodos liburbi con dos clientes y python.

### 4.3. Elección del método

A partir de los resultados anteriores se procede a elegir el caso que garantizará un mejor funcionamiento del paquete a implementar. En la Tabla 3 se encuentran resumidos los criterios de evaluación.

	<b>Python con telnet</b>	<b>C++ con liburbi y 1 cliente</b>	<b>C++ con liburbi y 2 clientes</b>
Frecuencia máxima de lectura	menor	Mayor	Mayor
Frecuencia media de lectura	Menor	Mayor	Mayor
Congelaciones en la lectura	Si	Si	Si
Frecuencia de envío	=	=	=
Seguimiento de trayectorias	Bueno	Malo	Bueno
Efecto negativo del envío en la lectura	No	Si	No
Retraso de la respuesta	Menor	Mayor	
Bloqueos de en el envío	Si	Si	Si

Cuadro 3: Comparación de los métodos usados y sus resultados en los experimentos.

En cuanto a recepción de datos se refiere es mas rápido el usando liburbi aunque sufra las mismas congelaciones que usando python y a efectos prácticos para la implementación del paquete es lo mismo que transmita a 10KHz que a 20KHz. Y respecto al envío el seguimiento de la trayectoria es parecido en ambos métodos, python y liburbi con dos clientes, pero parece que tenga un retraso superior el segundo. De todos modos no se considera concluyente el experimento echo pues dada la variabilidad de los resultados es posible que se haya trabajado con pocos datos el análisis sobre el retraso. Finalmente se elige usar la librería liburbi dado que se cree más convincente la ventaja en la adquisición de los datos que el retraso en la respuesta. Por otra parte se ha tenido en consideración que el paquete será más complejo que los experimentos y puede ser útil que la velocidad de ejecución de un programa en C++ sea mayor que en python.

## 5. Implementación del paquete de ROS

### 5.1. ROS

ROS es el acrónimo de Robot Operating System que lejos de ser un sistema operativo es más bien un marco de trabajo que proporciona unas herramientas y bibliotecas para ayudar a desarrollar software para aplicaciones en robótica. Proporciona entre otros abstractaciones de hardware, controladores para dispositivos, herramientas de visualización, comunicación por mensajes, administración de paquetes y todo bajo licencia de software libre. La principal característica sobre la que se basa ROS es el sistema de comunicación que proporciona. A bajo nivel está basada en el paso de mensajes que pueden ser leídos por diversos procesos simultáneamente. Dichos mensajes se escriben sobre unos canales de comunicación llamados *tópicos* a los que se puede acceder mediante métodos de publicación y suscripción. Sobre los tópicos se puede publicar o se puede suscribir desde un terminal o bien cualquier objeto de ROS a los que se les llama *nodos*. Todos los nodos que se pretendan tener comunicados entre sí deben estar creados sobre el mismo núcleo o *roscore*.

Todos los programas que se pretendan ejecutar sobre ROS son creados dentro de un paquete i todo paquete de ROS contiene una serie de archivos necesarios para su compilación:

- manifest.xml: Se incluye información sobre el nombre del paquete, el autor, tipo de licencia y paquetes externos necesarios.
- CMakeLists.txt: Se indica el uso de bibliotecas, el uso de mensajes propios del paquete y se declaran los ejecutables.
- mainpage.dox: Permite hacer un resumen explicativo del paquete.
- archivos ejecutables: ROS permite usar su API con C++ i python.
- carpeta msg: Contiene los archivos \*.msg donde se definen los mensajes propios del paquete.
- carpeta srv: contiene los archivos \*.srv donde se definen los servicios propios del paquete.
- Carpeta launch: Contiene los archivos \*.launch que permiten lanzar varios nodos de diferentes paquetes y tipos con los parámetros convenientes.

## 5.2. Paquet aibo\_server

Se pretende implementar una paquete que en lanzarse se conecte al AIBO indicado por la dirección IP. Una vez conectado debe recoger los datos enviados por el servidor URBI del AIBO y publicarlos sobre una serie de tópicos. De forma inversa debe tratar los valores de las articulaciones publicados en un tópico al que esta suscrito y enviarlos al servidor con el fin de actuar sobre la plataforma.

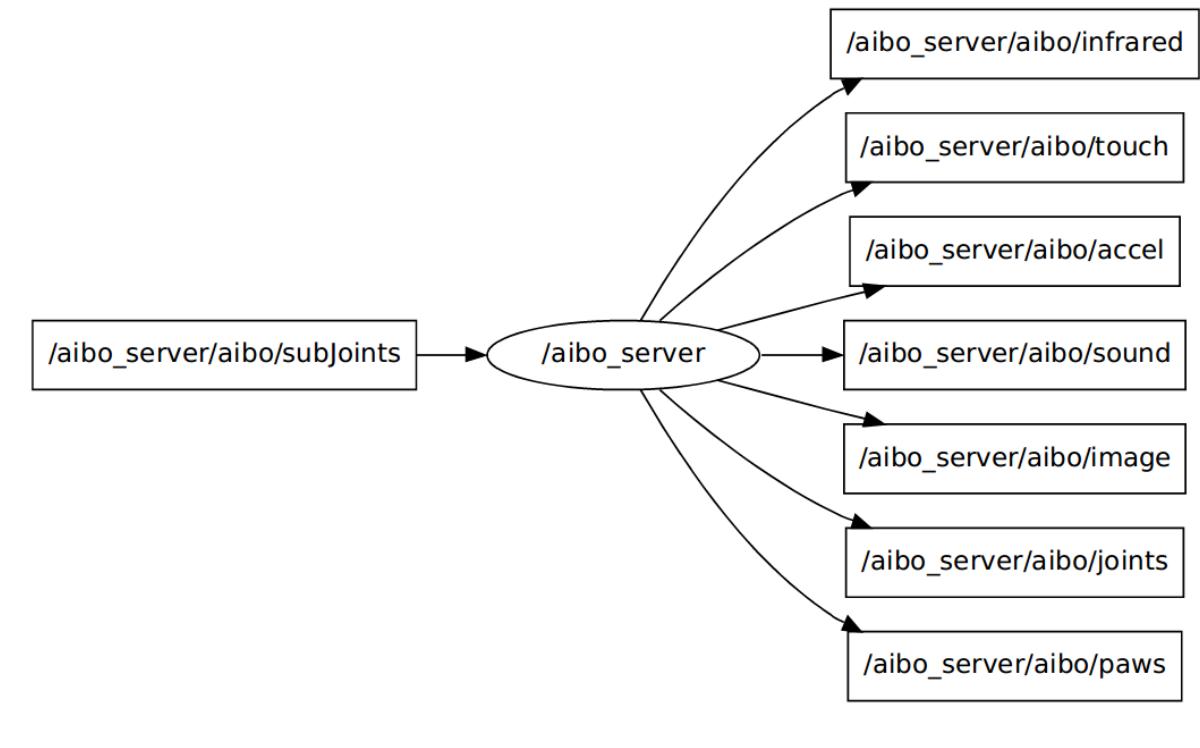


Figura 23: Nodo aibo\_server.

Para la realización de este paquete de ROS se han tenido en cuenta las siguientes consideraciones:

- Incluir en el manifest.xml los paquetes de ROS necesarios para su implementación:
  - roscpp: permite utilizar la API ROS para C++.
  - std\_msgs: permite el uso de unos mensajes estándar.
  - sensor\_msgs: permite el uso de mensajes especialmente destinados a ciertos tipos de sensores.
- Implementación de los archivos necesarios para el ejecutable:

- AiboNode.cpp: Archivo a partir del que se crea el ejecutable del paquete. En él se encuentra toda la estructura del programa.
  - AiboServer.cpp y AiboServer.h: Se define la clase aibo en la que se basa Aibo-Node.cpp.
  - AiboParams.h: se definen las constantes.
- 
- Definición de los mensajes propios:
    - Accel.msg: Mensaje destinado al acelerómetro.
    - Bumper.msg y BumperArray.msg: Mensaje destinado a los sensores de contacto de las patas.
    - IRArray.msg: Mensaje destinado a los tres infrarrojos.
    - Jointes.msg: Mensaje destinado a las articulaciones.
    - Sound.msg: Mensaje destinado al envío de sonido.
    - TouchArray.msg: mensaje destinado a los sensores de tacto de la espalda y la cabeza.
  - Modificación del CMakeList.txt para incluir los mensajes y los archivos C++ descritos además de la librería liburbi.

### 5.2.1. Estructura del programa

Basándose en los resultados de la sección anterior, Sección 4, se ha procedido a la implementación del paquete de ros usando liburbi i dos clientes, uno para la recepción y otro para el envío.

La estructura del programa que se ha implementado se puede ver en la Figura 24.



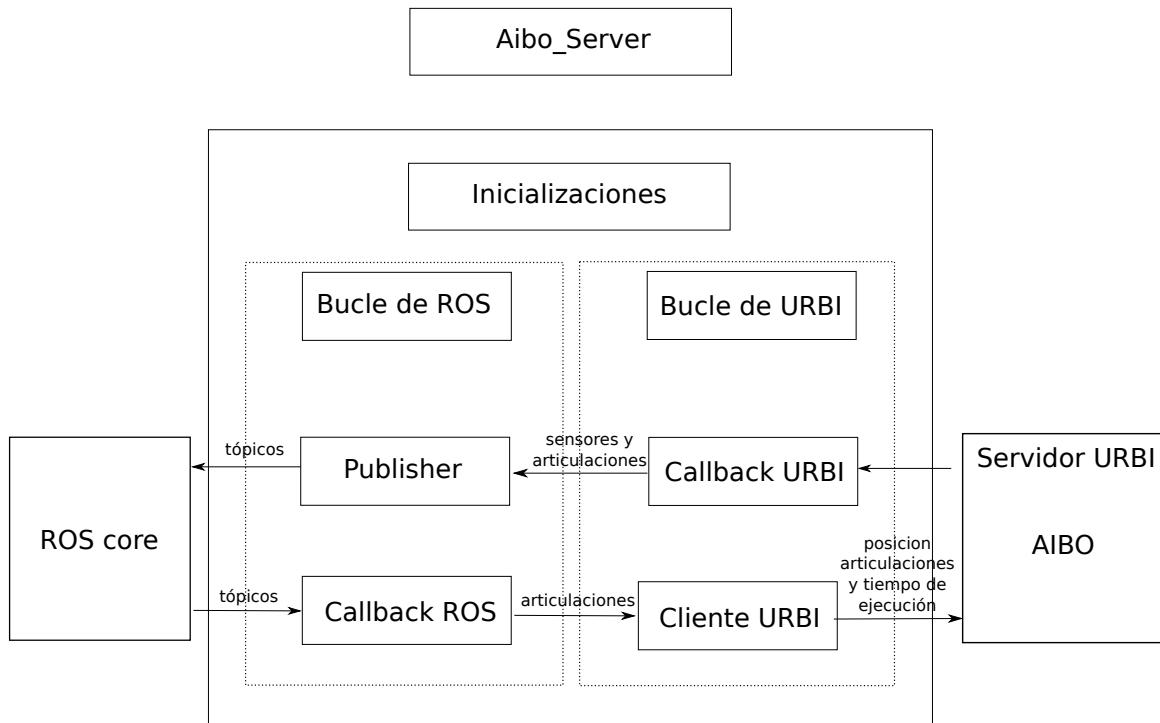


Figura 24: Estructura básica del ejecutable aibo\_server.

- Inicializaciones:
  - Inicialización del nodo de ROS:
    - Creación del nodo: Se le otorga un identificador al nodo, en este caso aibo\_server.
    - Definición de la frecuencia de ejecución del bucle de ROS.
    - Se inicializa el *Subscriber* que permitirá llamar a llamara al Callback de ROS.
  - Inicialización de la instancia de la clase **aibo**:
    - Inicialización de los clientes URBI de lectura y escritura.
    - Creación de los tópicos e inicialización de los *Publishers* que publicaran en ellos.
    - Definición del Callback de URBI para cada sensor y articulación.
    - Demanda del envío de datos desde URBI.
    - Definición del método de tratamiento de órdenes que usará el servidor URBI.

- Bucle de URBI:
  - Se crea un hilo de ejecución en paralelo donde se lleva acabo el bucle de URBI.
  - Llamada a los callbacks de URBI: En los callbacks se guardan los valores obtenidos en las variables de clase correspondientes.
  
- Bucle de ROS:
  - Publicación de las variables de los sensores y articulaciones en los tópicos correspondientes.
  - Llamada al callback de ROS: Éste envía la orden al servidor URBI mediante el cliente de envío.

### 5.2.2. Resultados

Con tal de valorar el trabajo realizado se procede a realizar un paquete de ROS que permita cuantificar los resultados.

El primer test realizado es el mismo que con el que se ha valorado las experimentaciones de la sección 4. Se trata de aplicar una sinusoida al movimiento de una articulación.

Para la realización del test se ha implementado un sencillo programa<sup>24</sup> que crea un nodo que publica sobre el tópico `/aibo_server/aibo/subJoints/jointRF1` valores que varían de forma sinusoidal. La estructura entre los nodos se muestra en la figura 25.

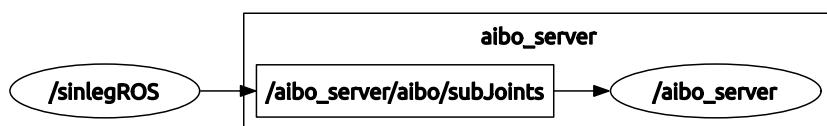


Figura 25: Estructura de ROS con los nodos aibo\_server y SinLeg.

De lo extraído en varias replicas a diversas frecuencias se puede reportar que el seguimiento de la trayectoria lleva un retraso mínimo de unos 0.4 segundos que era el retraso propio obtenido en las experimentaciones de la sección 4.2 aunque por lo general como se puede observar en la figura 26 dicho retraso no es la norma general estando el retraso medio muy por encima de éste.

<sup>24</sup>El código se puede encontrar en el Anexo C.5

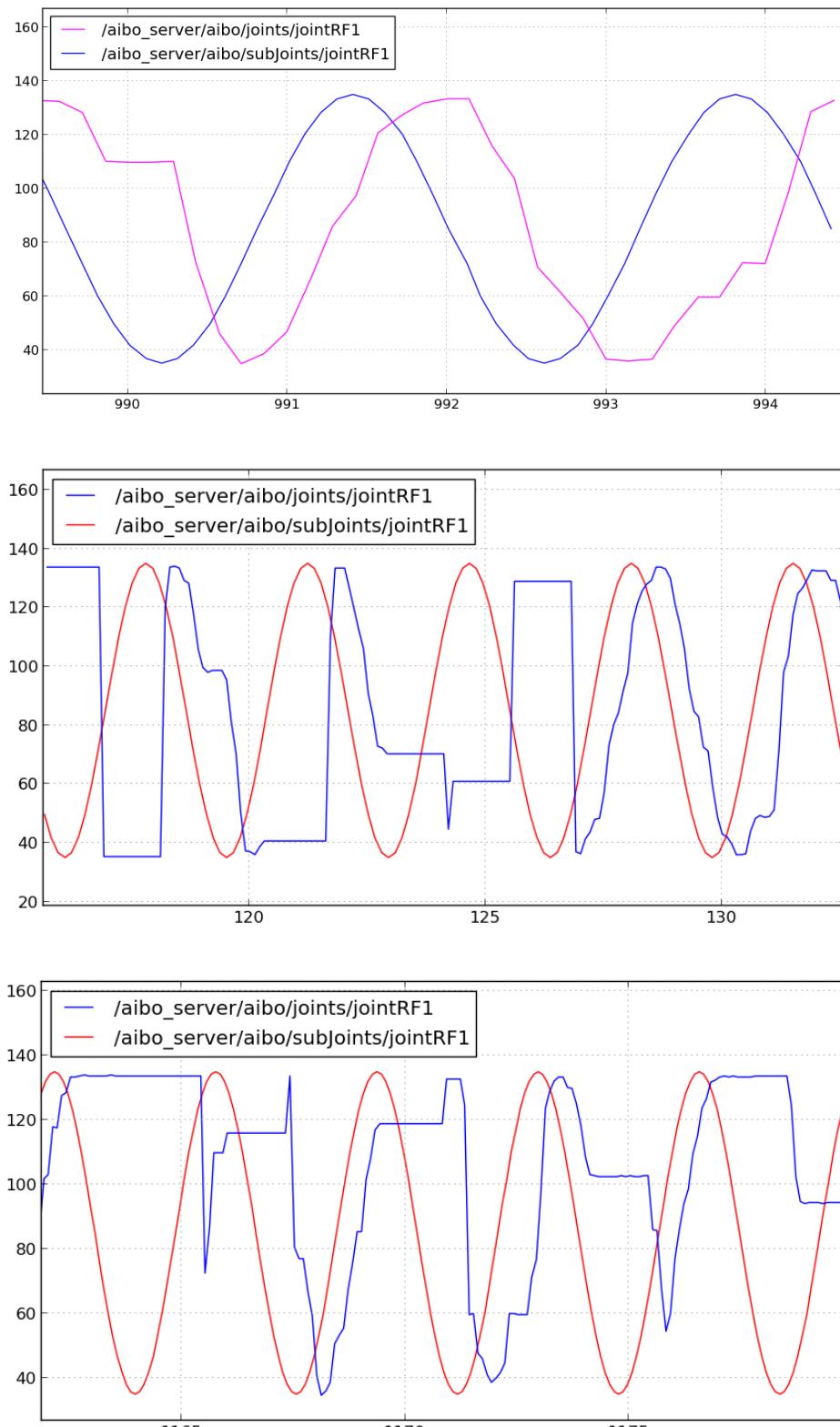


Figura 26: Entrada y respuesta del sistema ante una señales sinusoidales de diferente frecuencia y muestreo.

Valorando el modulo se puede concluir que no funciona de la forma deseada ni requerida para varias aplicaciones en las que se require una lectura y respuesta rápida del

sistema.

### 5.2.3. Mejoras

Los problemas encontrados, tras un proceso de depuración se pueden clasificar de la siguiente forma:

- Cortas congelaciones en la recepción de los valores de los sensores y articulaciones: respecto a ello ya se ha probado que en las experimentaciones anteriores que los resultados no podían mejorar usando liburbi.
- Envío de la orden correctamente pero tarda en realizar el movimiento: Este es un problema interno del tratamiento de las órdenes del servidor URBI y que por lo tanto no se puede hacer nada al respecto desde el punto de vista del cliente.
- Bloqueo en enviar la orden de movimiento: Como en las experimentaciones de la sección anterior se producen bloqueos debidos a que la función de liburbi que permite enviar órdenes se queda bloqueada. Estos bloqueos hacen que se bloquee todo el nodo durante segundos e incluso minutos.

A raíz del último problema comentado se ha implementado una solución que evita dos de las consecuencias que conlleva éste.

En primer lugar en caso de que se produzca un bloqueo en el envío de una posición del cliente al servidor, éste no debe afectar a la recepción de las demás variables. Como solución se propone que el envío y la recepción se produzcan en dos threads distintos e independientes. En segundo lugar se desea evitar los bloqueos en el envío. Esto se pretende hacer inicializando un tercer cliente URBI de manera que cuando el cliente de envío se quede bloqueado en alguna orden el nuevo cliente lo reemplazara en el envío mientras éste se reinicializa. Si en caso de bloqueo del nuevo cliente se espera que el primero, ya reinicializado tome el relevo.

Bajo estas ideas se ha reescrito el callback de ROS como muestra el diagrama de la figura 27.



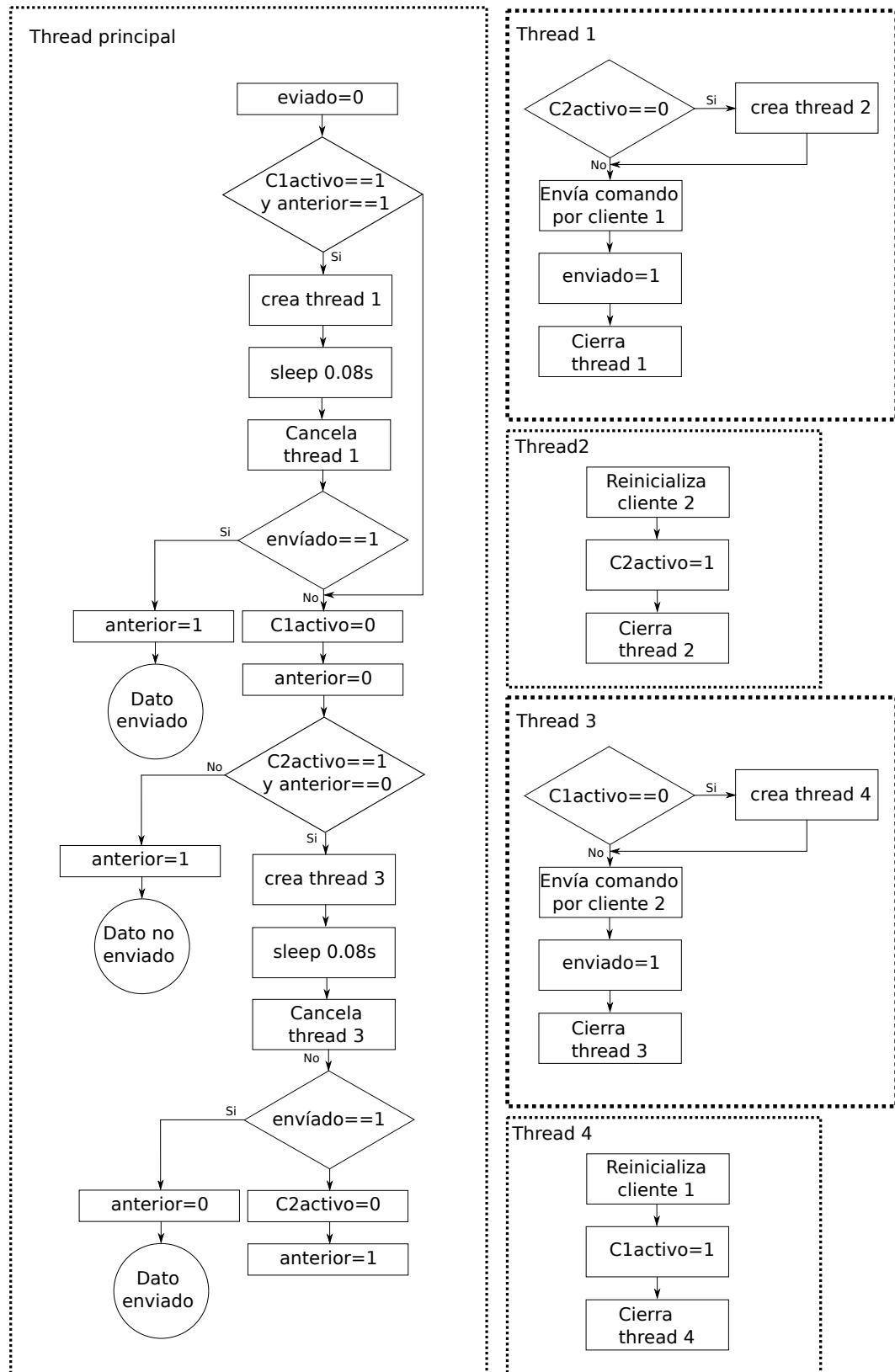


Figura 27: Diagrama del callback de ROS. Los flags `anterior`, `C1activo` y `C2activo` se inicializan con valor 1.

Con esta modificación se ha conseguido que el envío de datos, que de forma natural con liburbi es de forma síncrona, puesto que la función de envío espera una respuesta del servidor, se trate de forma asíncrona. Este modo no garantiza la llegada de todos los paquetes de datos pero gana en velocidad de transmisión y evita los bloqueos.

Volviendo de nuevo a comprobar los resultados tras la modificación se vuelve a aplicar un movimiento sinusoidal al movimiento de una articulación.

#### 5.2.4. Demostración

Con tal de asegurar el funcionamiento del paquete aibo\_server se ha realizado un programa demuestra como se pueden sincronizar dos o varios AIBO compartiendo variables de estado los unos con los otros. En esta demostración en particular se pretende que un AIBO imite los movimientos que realice otro. Es uno de las aplicaciones en que el flujo de datos es realmente grande, dado que las posiciones se modificarán dato a dato, y pondrá realmente a prueba la capacidad del modulo implementado. Se ha creado un paquete el que incluye un programa<sup>25</sup> con un nodo que está suscrito a el tópico de lectura de las articulaciones de un nodo aibo\_server y los publica sobre el tópico de escritura de otro nodo aibo\_server. Para crear estos tres nodos se ha hecho un archivo .launch que crea dos nodos del tipo aibo\_server con los nombres /ai1 y /ai2 y un tercer nodo como el descrito. La estructura resultante es la de la Figura 28.

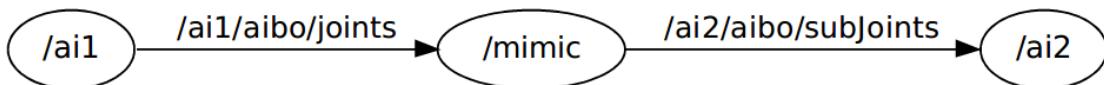


Figura 28: Estructura de los nodos de ROS en la demostración de sincronización.

En las siguientes Figuras se puede observar como reacciona una articulación de ambos AIBOs al mover manualmente el de uno de ellos.

#### 5.3. Paquete Aibo\_description

Este paquete pretende ser el vínculo entre el AIBO y una de las herramientas más usadas de ROS para robots móviles, rviz<sup>26</sup>. Rviz es un visualizador 3D para aplicaciones de robótica que permite visualizar un modelo, el entorno sensado o un mapa virtual.

El paquete consta de dos partes diferenciadas, la creación de el modelo de AIBO para que pueda ser representada la imagen 3D y la implementación de un nodo que haga responder el modelo en tiempo real a lo que sucede en el robot real.

<sup>25</sup>El código se puede encontrar en el Anexo C.6.

<sup>26</sup><http://wiki.ros.org/rviz>

### 5.3.1. El modelo

Para generar un modelo, se debe editar un xml con formato URDF<sup>27</sup>(Unified Robots Description Format) donde se especifican las dimensiones del robot, las articulaciones y sus movimientos, la geometría de las extremidades y parámetros físicos como la masa o la inercia. Para el modelo del AIBO se han creado 22 partes y 21 articulaciones se han descartado para el modelo la boca y las orejas. En la Figura 29 Se puede ver un esquema de las articulaciones, las partes y la disposición de los ejes que conforman el modelo implementado en el archivo URDF.

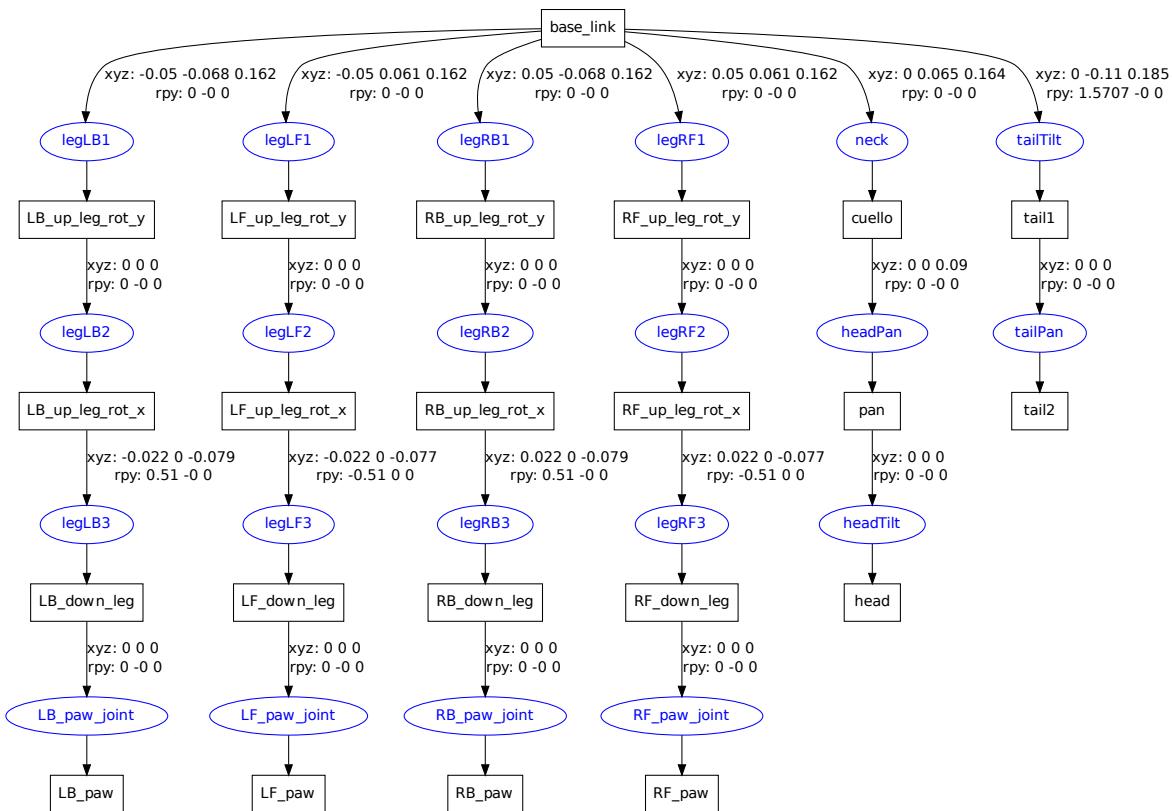


Figura 29: Estructura del modelo URDF.

En cada una de las partes se debe definir la geometría y sus dimensiones. Con el fin de darle un aspecto visual atractivo y mas semejante a al AIBO real la geometría ha sido definida mediante archivos .stl que contienen las figuras de cada una de las partes. Los archivos stl se han obtenido de modificar un archivo stl que incluía un modelo sólido de todo el AIBO. Se ha dividido, modificado las piezas usando los softwares de apoyo FreeCAD<sup>28</sup> y Netfabb<sup>29</sup>. Finalmente el resultado es el mostrado en la Figura 30 donde se

<sup>27</sup><http://wiki.ros.org/urdf>

<sup>28</sup><http://freecadweb.org/>

<sup>29</sup><http://www.netfabb.com/>

ve el modelo desde la ventana de rviz.

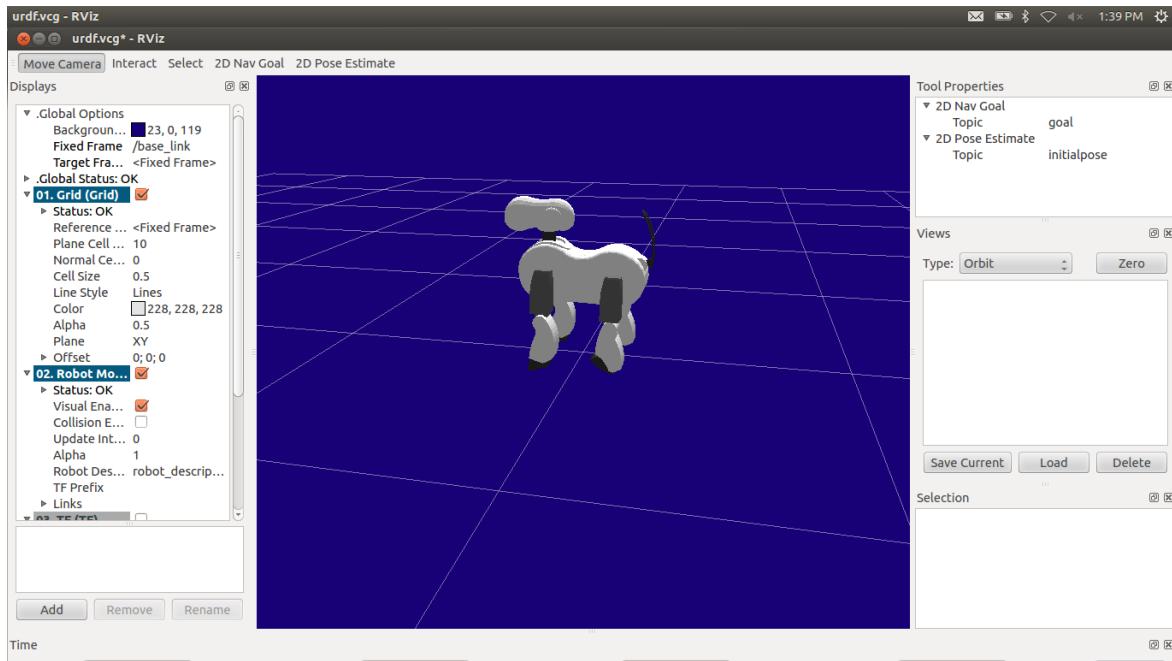


Figura 30: Modelo visto des de el framework de rviz.

### 5.3.2. El nodo state\_publisher

El nodo state\_publisher esta creado por un programa escrito en C++ (state\_publisher.cpp) y su principal función es estar suscrito al tópico /aibo\_server/ aibo/joints, donde se están publicando las posiciones las articulaciones del robot real, y las publica en un tópico llamado joint\_states al que ya se podrá comunicar rviz. Además envía la transformada del modelo cinemático. La estructura del programa es la siguiente:

- Inicialización:
  - Inicialización del nodo state\_publisher.
  - Inicialización del *publisher*.
  - Inicialización del *subscriber*.
  - Inicialización del *broadcaster*: Canal de envío de la transformada.
- Declaración de mensajes:
  - joint\_state: a través del que se publica el estado de las articulaciones.
  - odom\_trans: a través del que se enviará la transformada.
- Bucle de ROS:
  - Actualización del estado de las articulaciones.

- Actualización de la transformada.
  - Envío de las articulaciones.
  - Envío de la transformada.
  - Revisión del callback.
- Callback del *subscriber*:
- Adquisición de los valores de las articulaciones.

### 5.3.3. Estructura

Para la realización del paquete se han implementado o modificado los siguientes archivos:

- Incluir en el manifest.xml los paquetes de ROS necesarios:
  - roscpp: Permite usar el API de ROS para C++.
  - urdf: Permite leer y parsear archivos urdf.
  - std\_msg: Permite importar mensajes estándar.
  - tf: Permite la crear el modelo cinemático inverso.
  - aibo\_server: Permite importar los mensajes propios.
- Implementación del archivo state\_publisher.cpp, Sección 5.3.2.
- Carpeta urdf: se incluye el archivo Aibo.urdf donde está descrito el modelo.
- Carpeta meshes: se incluyen los archivo stl con los modelos 3D de las partes del modelo.
- Carpeta launch: incluye el archivo aibomod.launch. En el se definen los nodos y los parámetros a partir de los que será posible la visualización del modelo con rviz.
  - Se asigna al parametro robot\_description el archivo Aibo.urdf.
  - Se lanza un nodo del tipo robot\_state\_publisher del paquete con el mismo nombre. Dicho nodo esta suscrito al tópico joint\_states sobre el que publica el nodo implementado, state\_publisher, y se encarga de parsear el modelo que haya bajo el parámetro robot\_state\_publisher y publica sobre el tópico /tf el cual usa rviz para generar el movimiento del modelo.
  - Se lanza un nodo del tipo state\_publisher.
  - Se lanza un nodo del tipo rviz con la configuración deseada.

En la Figura 31 se puede observar como queda la estructura de los nodos con el visualizador rviz y el nodo aibo\_server en funcionamiento.



Figura 31: Estructuras de los nodos de ROS con aibo\_server y el modelo visualizado con rviz.

## Conclusiones

Cuando se trabaja con sistemas reales uno se da cuenta que no todo funciona de forma ideal y que un algoritmo que debería funcionar conlleva a múltiples errores o respuestas no esperadas. En éste caso concreto se ha visto como las comunicaciones que establecía el lenguaje base elegido no eran tan buenas como se esperaba. De todas formas se ha alcanzado el objetivo siguiendo la metodología establecida en primer momento, se ha implementado un modulo de ROS que permite integrar la plataforma AIBO, formado por los paquetes **aibo\_server** y **aibo\_description**. Estos paquetes se pueden encontrar en el repositorio (url del github) y están disponibles para su uso o modificación con tal de complementarlos o mejorar los resultados. Además se pretende colgar, fuera del alcance del proyecto, los paquetes y la documentación necesaria para su uso en la pagina de ROS con tal de contribuir y aportar un grano de arena al desarrollo de software libre.

Este proyecto deja un camino abierto a las posibles mejoras y complementos. Entre otros la integración del modulo de visión y audio usando URBI como se ha echo. Por otra parte dado que los resultados han demostrado no ser un modulo demasiado robusto cabria la posibilidad de implementar el mismo modulo partiendo OPEN-R, eliminando de esta manera una capa de programación intermedia, URBI, que posiblemente ralentice y empeore la comunicación entre cliente y servidor. En cuanto al modelo y el nodo que permite su integración con rviz podría completarse añadiendo los sensores del AIBO como los acelerómetros, los infrarrojos, los sensores de presión y contacto y la camera.

A nivel personal y educativo este proyecto ha servido para poner a prueba la capacidad de análisis y de resolución de problemas. Ha ayudado a tener un amplio conocimiento de uno de los frameworks relacionados con la robótica mas usados del momento. Además ha ayudado a mejorar las habilidades de desarrollo de software con los lenguajes C++ y python.

## Referencias

- [1] CEA, Comité Español de Automàtica, *El libro blanco de la robótica en España*. 2011.
- [2] Sony Corporation, *OPEN-R Programmer's Guide*. 2004.
- [3] Xavier Perez, *Vision-based Navigation and Reinforcement Learning Path Finding for Social Robots*. 2010.
- [4] Ángel Montero Mora, Gustavo Méndez Muñoz, José Ramon Dominguez Rodriguez, *Metodologías de diseño de comportamientos para AIBO ERS7*. 2009.
- [5] RoboCup, <http://www.robocup2014.org/>
- [6] Jesus Morales, *Localización de objetos y posicionamiento en el escenario de RoboCup Four-Legged con un robot AIBO*. 2007.
- [7] Jesús Martinez Gómez, *Diseño de un teleoperador con detección de colisiones para robots AIBO*. 2006.
- [8] Ricardo A. Tellez, *Aibo Programming*. <http://www.ouroboros.org/aibo.html>
- [9] Ethan Tira-Thompson, *Tekkotsu Quick Reference*, ERS-7, Tekkotsu 3.0.
- [10] David S. Touretzky and Ethan J. Tira-Thompson, *Exploring Tekkotsu Programming on Mobile Robots*. Carnegie Mellon University, 2010. <http://www.cs.cmu.edu/~dst/Tekkotsu/Tutorial/contents.shtml>
- [11] *URBI*, <http://www.urbiforge.org/index.php/Main/Robots>
- [12] Jean-Christophe Baillie, *URBI Doc for Aibo ERS2xx ERS7 Devices documentation*. 2005.

# Anexos

## A. Configuración de los marcos de trabajo

### A.1. OPEN-R SDK

#### A.2. Configuración wifi

Para configurar la conexión wireless del AIBO hay que acceder al archivo WLANS-FLT.txt que se encuentra en el directorio /OPEN-R/SYSTEM/CONF de la tarjeta de memoria del AIBO.

El archivo contiene los siguientes variables:

- HOSTNAME: Especifica el nombre de AIBO que se está usando.
- ETHER\_IP: La IP que usará el AIBO.
- ETHER\_NETMASK: La mascara de subred.
- IP\_GATEWAY: La IP a través de la que el AIBO accederá a la red.
- ESSID: Nombre de la red inalámbrica.
- WEPENABLE: con valor 1 si existe contraseña en la red o 0 en caso contrario.
- WEPKEY: contraseña de la red.
- AP MODE: Método de conexión del AIBO.
  - 0 para conexión punto a punto.
  - 1 para conexión con ruter.
  - 2 para conexión automática en el método que primero se lo permita.
- CHANNEL: especificar el canal para el modo punto a punto.

La configuración por defecto es:

HOSTNAME = AIBO

ETHER\_IP = 10.0.1.100

ETHER\_NETMASK = 255.255.255.0

IP\_GATEWAY = 10.0.1.1

ESSID = AIBONET

WEPENABLE = 1

WEPKEY = AIBO2

AP MODE = 2

CHANNEL = 3

**A.3. Tekkotsu****A.4. URBI****A.5. ROS**

## B. Manual de usuario

### B.1. Requisitos previos

Es necesario antes de instalar los paquetes seguir los siguientes pasos.

- Instalar linux de 32bits.
- Copiar la Tarjeta de memoria para el AIBO con URBI(url).
- Descargar librería liburbi 1.5 para C++(url).
- Instalar de ROS(url).
- Instalar los paquetes de ROS:
  - rviz
  - urdf
  - robot\_model
  - tf
- Descargar el paquete aibo\_server y aibo\_description(url).

### B.2. Instalación

- Guardar los paquetes en el workspace de ROS.
- Ejecutar en un terminal:
  - `rosmake --pre-clean aibo_server`
  - `rosmake --pre-clean aibo_description`

### B.3. aibo\_server

Ejecutar en terminal:

```
rosrun aibo_server aibo_server arg[1] arg[2]
```

Donde:

- arg[1] es la IP del AIBO.
- arg[2] es la frecuencia en numero entero a la que se desea que vaya el bucle de ROS (no se recomienda a más de 10Hz).

#### B.4. aibo\_description

Ejecutar en la terminal: `roslaunch aibo_description aibomod.launch`

Con tal de vincular el modelo con un AIBO se debe lanzar también un nodo `aibo_server`.

## C. Scripts y programas

### C.1. Adquisición del valor de una articulación con liburbi.

```

1 #include <urbi/uobject.hh>
3 #include <urbi/uclient.hh>
5 #include <stdio.h>
7 #include <time.h>
# include <iostream>
7 #include <fstream>

9 std :: ofstream myfile;

11 //adquisicion del tiempo
12 void getTime(double& tim)
13 {
14     struct timespec tsp;
15     clock_gettime(CLOCK_REALTIME, &tsp);
16     tim = (double)(tsp.tv_sec+ tsp.tv_nsec*0.000000001);
17     return;
18 }
19 //escritura en archivo
20 void saveData( double tim ,double value){
21     myfile << tim;
22     myfile << "\t";
23     myfile << value;
24     myfile << "\n";
25     return;
26 }
27
//callback de URBI
28 urbi::UCallbackAction onJointSensor( const urbi::UMessage &msg)
29 {
30     double value = (double)msg.value->val;
31     if (msg.tag=="legRF1")
32         double tim;
33         getTime(tim);
34         saveData(tim ,value);
35         return urbi::URBI_CONTINUE;
36     }
37
38
39 int main(int argc , char** argv)
40 {
41     //inicializacion del cliente URBI

```

```

43 urbi::UClient* client = new urbi::UClient(argv[1], 54000);
44 //inicializacion del callback
45 client->setCallback(urbi::callback(onJointSensor), "legRF1");
46 //demanda del bucle de envio
47 client->send("loop legRF1 << legRF1.val,");
48 //abre el archivo de escritura
49 myfile.open("Data1.txt");
50 myfile.precision(15);
51 //bucle de URBI
52 urbi::execute();
53 //cierra el archivo de escritura
54 myfile.close();
55 return(0);
}

```

src/getDataOneLeg/getDataC++/getData.cpp

## C.2. Adquisición del valor de una articulación con Python.

```

import telnetlib
2 import time

4 def tractarData(data):
    datos=[]
    # ectura hasta la marca
    while (data.find("555555")<0):
        data=data+te.read_some()
    # descarta la informacion de la bateria y divide la cadena
    if (data.find("Bat")<0):
        dataNow=data.split("555555")[0]
        data=data.split("555555")[1]
        dataArray=dataNow.split("\n")
        del dataArray[0]
        leg=0
        for i in range(len(dataArray)):
            if (dataArray[i].find("tag0")>0):
                leg=float(dataArray[i].split("]")[1])
        return leg, data
    else:
        data=""
        return 0, data

24 # scritura en archivo
25 def writeData(f, text):
26     f.write(text)

```



```

28 # inicializacion cliente telnet
  te=telnetlib.Telnet("192.168.0.125",54000)
30 time.sleep(1)
  data=1
32 # eliminacion de la cabezera URBI
  while data:
34   data=te.read_eager()
    print data
36 # demanda del bucle de envio
  te.write("motors on;\r\n")
38 te.write("motors.load=0;\r\n")
  te.write("loop tag0 << legLF1.val , loop tag19 << 5*111111;")
40 # primera lectura y tratamiento de la cadena
  dataArray, data=tractarData(data)
42 # abre el archivo de escritura
  f = open("Data.txt", "w")
44 #bucle de tratamiento y escritura
  while True:
46   leg, data =tractarData(data)
    t= str(leg) + "\t"
48   writeData(f,t)
    sec = "{0:.15f}".format(time.time()) + "\n"
50   writeData(f,sec)
    print sec
52 # cierra el archibo de escritura
  f.close()

```

src/getDataOneLeg/getDataPython/getData.py

### C.3. Envió de trayectoria punto a punto con liburbi.

```

1
#include <urbi/uobject.hh>
3 #include <urbi/uclient.hh>
# include <urbi/usyncclient.hh>
5 #include <stdio.h>
# include <time.h>
7 #include <iostream>
# include <fstream>
9 #include <pthread.h>
# include <unistd.h>
11 #include <math.h>

13
15 std::ofstream f;

```

```

float angle=0;
17
urbi::UClient* client;
19 urbi::UClient* client2;

//guarda el tiempo
21 void getTime(double& tim)
23 {
    struct timespec tsp;
25    clock_gettime(CLOCK_REALTIME, &tsp);
    tim = (double)(tsp.tv_sec+ tsp.tv_nsec*0.000000001);
27    return;
}
29 //escribe los datos de salida
void saveData( double tim, double value, std::string tag){
31
    char command[100];
33    sprintf (command,"%f \t %f",tim , value);
    f << command << std::endl;
35
    return;
}
37 //escribe los datos de entrada
39 void saveData2( double tim, double value, std::string tag){

41    char command[100];
43    sprintf (command,"%f \t %f",tim , value);
    f2 << command << std::endl;

45    return;
}
47 //Bucle de envio
void *move(void *)
{
49
    while(true){

51        float l=50*sin(angle*2*M_PI/360) + 85;
53        char command[100];
55        sprintf (command,"legLF1.val=%f time: 100,%f",l);
57        client2->send(command);
        angle+=10;
59        double tim;
        getTime(tim);

61        saveData2(tim,l , "LF1");
        usleep(100000);
}

```



```

63     }
64     return NULL;
65 }
//Callback de URBI
66 urbi::UCallbackAction onJointSensor( const urbi::UMessage &msg)
67 {
68     double value = (double)msg.value->val;
69     std::cout.precision(18);
70     if (msg.tag=="legLF1"){
71         double tim;
72         getTime(tim);
73         saveData(tim, value, msg.tag);
74     }
75 }
76
77     return urbi::URBI_CONTINUE;
78 }
79
80 int main(int argc, char** argv)
81 {
82     //inicia clientes URBI
83     client = new urbi::UClient(argv[1], 54000);
84     client2 = new urbi::UClient(argv[1], 54000);
85     //Inicia callback de URBI para cada tag
86     client->setCallback(urbi::callback(onJointSensor), "legRF1");
87     client->setCallback(urbi::callback(onJointSensor), "legRF2");
88     client->setCallback(urbi::callback(onJointSensor), "legRF3");
89     client->setCallback(urbi::callback(onJointSensor), "legRH1");
90     client->setCallback(urbi::callback(onJointSensor), "legRH2");
91     client->setCallback(urbi::callback(onJointSensor), "legRH3");
92     client->setCallback(urbi::callback(onJointSensor), "legLF1");
93     client->setCallback(urbi::callback(onJointSensor), "legLF2");
94     client->setCallback(urbi::callback(onJointSensor), "legLF3");
95     client->setCallback(urbi::callback(onJointSensor), "legLH1");
96     client->setCallback(urbi::callback(onJointSensor), "legLH2");
97     client->setCallback(urbi::callback(onJointSensor), "legLH3");
98     client->setCallback(urbi::callback(onJointSensor), "mouth");
99     client->setCallback(urbi::callback(onJointSensor), "headNeck");
100    client->setCallback(urbi::callback(onJointSensor), "headPan");
101    client->setCallback(urbi::callback(onJointSensor), "headTilt");
102    client->setCallback(urbi::callback(onJointSensor), "tailTilt");
103    client->setCallback(urbi::callback(onJointSensor), "tailPan");
104    client2->send("motors on;");
105    //define el modo de resolver conflictos
106    client2->send("legLF1.val->blend = cancel;");
107    //demanda del loop

```

```

client->send("loop legRF1 << legRF1.val & loop legRF2 << legRF2.val &
loop legRF3 << legRF3.val & loop legRH1 << legRH1.val & loop legRH2 <<
legRH2.val & loop legRH3 << legRH3.val & loop legLF1 << legLF1.val &
loop legLF2 << legLF2.val & loop legLF3 << legLF3.val & loop legLH1 <<
legLH1.val & loop legLH2 << legLH2.val & loop legLH3 << legLH3.val &
loop neck << neck.val & loop headTilt << headTilt.val & loop headPan <<
headPan.val & loop tailPan << tailPan.val & loop tailTilt << tailTilt.
val & loop mouth << mouth.val;");

109 //Abre el archivo de datos de salida
f.open ("DataOut.txt");
f.precision(15);
111 //Abre el archivo de datos de entrada
f2.open ("DataIn.txt");
f2.precision(15);
113 // crea thread para el envio
pthread_t t1;
115 pthread_create(&t1, NULL, &move,NULL);
pthread_join(t1,NULL);
117 //Bucle de URBI
urbi::execute();
119 //cierra archivos
f.close();
f2.close();

121
123
125 return(0);
}

```

src/sinLegRead/sinLegC++/2Client/sinLeg.cpp

#### C.4. Envió de trayectoria punto a punto con Python.

```

1 import sys
2 import telnetlib
3 import time
4 import math
5 import threading

7 global Joint
angle=0

9
#funcion de envio
11 def move():
    global te1
    global angle
    global f2
13
15 A=50

```



```

offset1=85
17  paso=15
    legLF1=A*math.sin(math.radians(angle)) + offset1
19  te1.write("legLF1.val= %1.6f time: 500,\r\n" % legLF1)
angle+=paso
21  sec = "{0:.15f}".format(time.time()) + "\t"
writeData(f2, sec)
23  t2= str(legLF1) + "\n"
writeData(f2, t2)
25  threading.Timer(0.2,move).start()
#define la clase joints
27  class Joints(object):
    _slots_=['val', 'tag']

29 #funcion de tratamiento de la cadena
31 def tractarData(data):
    while (data.find("555555")<0):
33
        data=data+te.read_some()
35    if (data.find("Bat")<0):
        dataNow=data.split("555555")[0]
37    data=data.split("555555")[1]
        dataArray=dataNow.split("\n")
39    del dataArray[0]
        Joint=Joints()
41    Joint.val=[]
        Joint.tag=["legLF1", "legLF2", "legLF3", "legLH1", "legLH2", "legLH3", "
legRF1", "legRF2", "legRF3", "legRH1", "legRH2", "legRH3", "neck", "mouth", "
headPan", "headTilt", "tailTilt", "tailPan"]
43    for i in range(len(dataArray)):
        if (dataArray[i].find("tag0")>0):
45        Joint.val.append(float(dataArray[i].split("]")[1]))
        if (dataArray[i].find("tag2")>0):
47        Joint.val.append(float(dataArray[i].split("]")[1]))
        if (dataArray[i].find("tag3")>0):
49        Joint.val.append(float(dataArray[i].split("]")[1]))
        if (dataArray[i].find("tag4")>0):
51        Joint.val.append(float(dataArray[i].split("]")[1]))
        if (dataArray[i].find("tag5")>0):
53        Joint.val.append(float(dataArray[i].split("]")[1]))
        if (dataArray[i].find("tag6")>0):
55        Joint.val.append(float(dataArray[i].split("]")[1]))
        if (dataArray[i].find("tag7")>0):
57        Joint.val.append(float(dataArray[i].split("]")[1]))
        if (dataArray[i].find("tag8")>0):
59        Joint.val.append(float(dataArray[i].split("]")[1]))

```

```

    if (dataArray[ i ].find("tag9")>0):
        Joint.val.append(float(dataArray[ i ].split("]")[1]))
    if (dataArray[ i ].find("tag10")>0):
        Joint.val.append(float(dataArray[ i ].split("]")[1]))
    if (dataArray[ i ].find("tag11")>0):
        Joint.val.append(float(dataArray[ i ].split("]")[1]))
    if (dataArray[ i ].find("tag12")>0):
        Joint.val.append(float(dataArray[ i ].split("]")[1]))
    if (dataArray[ i ].find("tag13")>0):
        Joint.val.append(float(dataArray[ i ].split("]")[1]))
    if (dataArray[ i ].find("tag14")>0):
        Joint.val.append(float(dataArray[ i ].split("]")[1]))
    if (dataArray[ i ].find("tag15")>0):
        Joint.val.append(float(dataArray[ i ].split("]")[1]))
    if (dataArray[ i ].find("tag16")>0):
        Joint.val.append(float(dataArray[ i ].split("]")[1]))
    if (dataArray[ i ].find("tag17")>0):
        Joint.val.append(float(dataArray[ i ].split("]")[1]))
    if (dataArray[ i ].find("tag18")>0):
        Joint.val.append(float(dataArray[ i ].split("]")[1]))

    return Joint, data
else:
    data=""
    return 0, data

#escribe en el archivo
def writeData(f, text):

    f.write(text)

#inicia cliente telnet de envio
tel=telnetlib.Telnet("192.168.0.124",54000)
tel.write("motors on;\r\n")
#selecciona modo de resolver conflictos
tel.write("legLF1.val->blend = cancel;\r\n")
#inicia cliente telnet de recepcion
te=telnetlib.Telnet("192.168.0.124",54000)
te.write("motors on;\r\n")
time.sleep(3)
data=1
#elimina la cabecera de URBI de la cadena
while data:
    data=te.read_eager()
    print data
#demanda del bucle de envio de datos

```

```

te.write("loop tag0 << legLF1.val , loop tag2 << legLF2.val , loop tag3 <<
legLF3.val , loop tag4 << legLH1.val , loop tag5 << legLH2.val , loop tag6
<< legLH3.val , loop tag7 << legRF1.val , loop tag8 << legRF2.val , loop
tag9 << legRF3.val , loop tag10 << legRH1.val , loop tag11 << legRH2.val ,
loop tag12 << legRH3.val , loop tag13 << mouth.val , loop tag14 << neck.
val , loop tag15 << headPan.val , loop tag16 << headTilt.val , loop tag17
<< tailTilt.val , loop tag18 << tailPan.val , loop tag19 << 5*111111;" )

107 #primera cadena leida y tratada
dataArray , data=tractarData(data)
109 #abre los archivos de escritura
f = open("Data.txt" , "w")
111 f2 = open("DataIn.txt" , "w")
#llama a la funcion de movimiento a los 200ms
113 threading.Timer(0.2 ,move) . start()
#bucle de lectura
115 while True:
    joint , data =tractarData(data)
    117 if (joint!=0) :
        sec = "{0:.15f}" .format(time.time()) + "\t"
        119 writeData(f ,sec)
        t= str(joint.tag[0]) + "\t"
        21 writeData(f ,t)
        t= str(joint.val[0]) + "\n"
        23 writeData(f ,t)
#cierra archivos
125 f .close()
f2 .close()

```

src//sinLegRead/sinLegPython/sinlegLF1.py

## C.5. Envió de trayectoria punto a punto con ROS.

```

#include <ros/ros.h>
2 #include <aibo_server/Joints.h>
# include <sensor_msgs/JointState.h>
4 #include "std_msgs/String.h"

6 ros :: Publisher pub;
ros :: Subscriber sub;
8 aibo_server :: Joints joi;
float angle=0;

10 //publica el el mensaje de tipo Joints
12 void publishJoint(){
    pub.publish(joi);
14 }

```

```

16
17     int main( int argc ,  char** argv )
18 {
19     //inicializa el objeto joints
20     joi.jointRF2=73.74736;
21     joi.jointRF3=32.0;
22     joi.jointRH1=0;
23     joi.jointRH2=0;
24     joi.jointRH3=0;
25     joi.jointLF1=0;
26     joi.jointLF2=0;
27     joi.jointLF3=0;
28     joi.jointLH1=0;
29     joi.jointLH2=0;
30     joi.jointLH3=0;
31     joi.headPan=0;
32     joi.headNeck=0;
33     joi.headTilt=0;
34     joi.mouth=0;
35     joi.tailTilt=0;
36     joi.tailPan=0;

37
38     //inicia el nodo de ROS
39     ros::init(argc ,  argv , "sinlegROS");
40     ros::NodeHandle n;
41     //define la frecuencia del loop
42     ros::Rate r(10);

43
44     std::string input , output;
45     //define el publisher
46     pub = n.advertise<aibo_server::Joints>("/aibo_server/aibo/subJoints" , 1 ,
47         false);

48
49     //loop de ROS
50     while (ros::ok())
51     {
52         float l=50*sin( angle*2*M_PI/360) + 85;
53         angle+=10;
54         joi.jointRF1=l;
55         publishJoint();
56         r.sleep();
57     }
58     return(0);
59 }
```

---

src/ROStests/sinlegROS.cpp

## C.6. Modulo de imitación entre AIBOs.

```
1 #include <ros/ros.h>
3 #include <aibo_server/Joints.h>
# include <sensor_msgs/JointState.h>
5 #include "std_msgs/String.h"

7 ros::Publisher pub;
ros::Subscriber sub;
9 aibo_server::Joints joi;

11 //define el callback de ROS
void callback(const aibo_server::Joints::ConstPtr& msg)
13 {
    joi.jointRF1=msg->jointRF1;
    joi.jointRF2=msg->jointRF2;
    joi.jointRF3=msg->jointRF3;
    joi.jointRH1=msg->jointRH1;
    joi.jointRH2=msg->jointRH2;
    joi.jointRH3=msg->jointRH3;
    joi.jointLF1=msg->jointLF1;
    joi.jointLF2=msg->jointLF2;
    joi.jointLF3=msg->jointLF3;
    joi.jointLH1=msg->jointLH1;
    joi.jointLH2=msg->jointLH2;
    joi.jointLH3=msg->jointLH3;
    joi.headPan=msg->headPan;
    joi.headNeck=msg->headNeck;
    joi.headTilt=msg->headTilt;
    joi.mouth=msg->mouth;
    joi.tailTilt=msg->tailTilt;
    joi.tailPan=msg->tailPan;

33 }
//publica en el topico
35 void publishJoint(){
    pub.publish(joi);
37 }

39 int main(int argc, char** argv)
{
```

```
41 //inicia nodo de ROS
42 ros::init(argc, argv, "aibo_tutorials");
43 ros::NodeHandle n;
44 //define frecuencia del loop de ROS
45 ros::Rate r(10);

47 std::string input, output;
48 //define el publisher y el suscriber
49 pub = n.advertise<aibo_server::Joints>("/ai2/aibo/subJoints", 1, false);
50 sub = n.subscribe<aibo_server::Joints>("/ai1/aibo/joints", 1, callback);
51 //bucle de ROS
52 while (ros::ok())
53 {
54     publishJoint();
55     //revisa los callbacks
56     ros::spinOnce();
57     r.sleep();
58 }
59 return(0);
}
```

src/ROStests/mimic.cpp