

# Integración de la plataforma AIBO con ROS

Diego Muñoz Galan

17 de julio de 2014







# Resumen

El proyecto persigue el objetivo de implementar un driver para integrar la plataforma robótica AIBO en el marco de trabajo ROS (Robot Operating System). A lo largo de esta memoria se describe el procedimiento llevado a cabo para alcanzar un resultado que cumpla dicho objetivo. El trabajo realizado y documentado en ésta memoria se divide en tres partes.

Consta de una primera parte de documentación e investigación de la plataforma robótica en la que se describe tanto el hardware como los posibles software alternativos con los que operar.

Sigue una segunda parte donde se comparan varios métodos de programación con los que se podría implementar el modulo de ROS. En ésta se incluye un primer análisis cualitativo en el que se contemplan gran parte de las posibilidades de programación y se realiza una primera selección. En un segundo análisis cuantitativo terminara por decidir el método más adecuado.

En la tercera parte se explica cómo 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 su funcionamiento, y por otro el modulo que facilita la integración de un modelo de visualización 3D.

Esta memoria incluye también un desarrollo del presupuesto y el impacto ambiental intrínsecos al proyecto. Por último, se encuentran las conclusiones del proyecto, así como unas recomendaciones para los próximos proyectos que puedan realizarse sobre esta línea de investigación. Por este motivo, se ha redactado un manual de usuario en forma de anexo con la información necesaria para preparar el marco de trabajo y para poder usar el modulo implementado.

# Índice general

<b>Resumen</b>	<b>3</b>
<b>1 Prefacio</b>	<b>11</b>
1.1 Motivación . . . . .	11
1.2 Requerimientos previos . . . . .	12
<b>2 Introducción</b>	<b>13</b>
2.1 Estado del arte . . . . .	13
2.1.1 Robótica . . . . .	13
2.1.2 Robótica en la educación . . . . .	15
2.1.3 AIBO . . . . .	16
2.2 Objetivos . . . . .	17
2.3 Alcance . . . . .	17
2.4 Planificación . . . . .	18
<b>3 AIBO ERS-7</b>	<b>21</b>
3.1 Hardware . . . . .	21
3.2 El software . . . . .	23
3.2.1 OPEN-R . . . . .	23
3.2.2 Tekkotsu . . . . .	25
3.2.3 URBI . . . . .	27
<b>4 Comparación de alternativas para la programación</b>	<b>29</b>
4.1 Lectura de datos . . . . .	31
4.1.1 Método con liburbi y C++ . . . . .	32
4.1.2 Método con telnet y python . . . . .	32
4.1.3 Comparación de los resultados . . . . .	33
4.2 Envío de datos . . . . .	41
4.3 Elección del método . . . . .	48

<b>5 Implementación del paquete de ROS</b>	<b>51</b>
5.1 ROS . . . . .	51
5.1.1 Paquete aibo_server . . . . .	52
5.1.2 Estructura del programa . . . . .	53
5.1.3 Resultados del aibo_server . . . . .	55
5.1.4 Mejoras del aibo_server . . . . .	59
5.2 Paquete Aibo_description . . . . .	63
5.2.1 El modelo . . . . .	63
5.2.2 El nodo state_publisher . . . . .	65
5.2.3 Estructura de paquete . . . . .	66
<b>6 Presupuesto</b>	<b>69</b>
6.1 Coste material . . . . .	69
6.2 Coste personal . . . . .	69
<b>7 Impacto ambiental</b>	<b>71</b>
<b>Conclusiones</b>	<b>73</b>
<b>Agradecimientos</b>	<b>75</b>
<b>Referencias</b>	<b>76</b>
<b>Anexos</b>	<b>79</b>
<b>A Manual de usuario</b>	<b>79</b>
A.1 Requisitos previos . . . . .	79
A.2 Instalación . . . . .	82
A.3 aibo_server . . . . .	83
A.3.1 Tópicos . . . . .	83
A.3.2 Mensajes . . . . .	84
A.4 aibo_description . . . . .	88
<b>B Scripts y programas</b>	<b>89</b>
B.1 Adquisición del valor de una articulación con liburbi. . . . .	89
B.2 Adquisición del valor de una articulación con Python. . . . .	90
B.3 Envió de trayectoria punto a punto con liburbi. . . . .	92
B.4 Envió de trayectoria punto a punto con Python. . . . .	95

B.5 Envió de trayectoria punto a punto con ROS. . . . .	98
B.6 Modulo de imitación entre AIBOs. . . . .	100

# Índice de figuras

2.1	Robot manipulador de KUKA. . . . .	14
2.2	Robot cuadrúpedo WildCat. . . . .	15
2.3	Diagrama de Gantt . . . . .	19
3.1	AIBO ERS-7. . . . .	21
3.2	Arquitectura de AIBO con OPEN-R . . . . .	23
3.3	Comunicación entre objetos. . . . .	24
3.4	Arquitectura del AIBO con Tekkotsu. . . . .	26
3.5	Proceso de ejecución de un comportamiento en Tekkotsu. . . . .	26
3.6	Arquitectura del AIBO con URBI. . . . .	27
4.1	Estructura de la lectura de una variable. . . . .	31
4.2	Terminal URBI consultando un dato. . . . .	33
4.3	Diagrama de cajas de la frecuencia de datos para las 15 réplicas de cada método. . . . .	34
4.4	Mínimos de la frecuencia en las 15 réplicas de los dos métodos. . . . .	35
4.5	Histograma de la frecuencia en las 15 réplicas de los dos métodos. . . . .	36
4.6	Diagrama de cajas de la frecuencia de datos para las 15 réplicas de cada método con todas las articulaciones. . . . .	38
4.7	Mínimos de la frecuencia en les 15 repliques de los dos métodos. . . . .	40
4.8	Histograma de la frecuencia en les 15 repliques de los dos métodos. . . . .	40
4.9	Medias de las frecuencias de envío para los valores de una articulación y para el valor de todas las articulaciones. . . . .	41
4.10	En azul, las posiciones enviadas a 1Hz y en negro las lecturas. . . . .	44
4.11	En azul, las posiciones enviadas a 2Hz y en negro las lecturas. . . . .	45
4.12	En azul, las posiciones enviadas a 5Hz y en negro las lecturas. . . . .	46
4.13	En azul, las posiciones enviadas a 10Hz y en negro las lecturas. . . . .	47
4.14	Retrasos entre la orden enviada y la acción. . . . .	48

5.1	Nodo aibo_server. . . . .	52
5.2	Estructura básica del ejecutable aibo_server. . . . .	54
5.3	Consulta de la frecuencia de publicación en los diferentes tópicos de ai- bo_server. . . . .	56
5.4	Valores de los acelerómetros usando la herramienta rqt_plot. . . . .	56
5.5	Estructura de ROS con los nodos aibo_server y SinLeg. . . . .	57
5.6	Entrada y respuesta del sistema ante una señal sinusoidal de enviando puntos a 2Hz. . . . .	57
5.7	Entrada y respuesta del sistema ante una señal sinusoidal de enviando puntos a 7Hz. . . . .	58
5.8	Entrada y respuesta del sistema ante una señal sinusoidal de enviando puntos a 10Hz. . . . .	58
5.9	Diagrama del callback de ROS. Los flags <i>anterior</i> , <i>C1activo</i> y <i>C2activo</i> se inicializan con valor 1 y los flags <i>reiniendo1</i> y <i>reiniendo2</i> se inicializan a 0. . . . .	60
5.10	Empiezan a producirse bloqueos. . . . .	61
5.11	Se producen bloqueos que se intentan evitar. . . . .	62
5.12	Recuperación del seguimiento de la trayectoria. . . . .	62
5.13	Buen seguimiento de la trayectoria. . . . .	63
5.14	Estructura del modelo URDF. . . . .	64
5.15	Modelo visto des de el framework de rviz. . . . .	65
5.16	Estructuras de los nodos de ROS con aibo_server y el modelo visualizado con rviz. . . . .	67
A.1	Tarjeta de memoria para AIBO. . . . .	81

# Índice de tablas

3.1	Estructura del envío de mensajes en OPEN-R . . . . .	24
4.1	Comparación entre lenguajes usados sobre AIBO . . . . .	30
4.2	Esumen de los criterios de elección. . . . .	49
6.1	Coste material detallado. . . . .	69
6.2	Coste personal detallado. . . . .	69



# Capítulo 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 [1]. Desde 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 utilizar los AIBO para realizar tareas de cooperación y sincronización mediante una programación de alto nivel. Habida cuenta de la incomodidad que proporcionan los actuales lenguajes de programación para dicha plataforma, fundamentalmente como consecuencia de su falta de mantenimiento y dado que sus curvas de aprendizaje son realmente complicadas, se planteó la posibilidad de programar al AIBO mediante ROS[2].

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 de importantes restricciones presupuestarias en las universidades. Además cabe señalar que la implementación de un driver para AIBO facilitaría enormemente la programación, ya que abre la posibilidad de uso de un gran abanico de paquetes y herramientas existentes para ROS.

Por estos motivos participar en la integración del AIBO en ROS es un proyecto altamente irresistible.

Por último destacar que 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 investigador 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, así como un conocimiento básico del funcionamiento de ROS y su estructura. Son necesarios también conocimientos de estadística y del uso de software de apoyo para la realización de algunos testes estadísticos.

# Capítulo 2

## Introducción

### 2.1. Estado del arte

Se pretende dar una visión general del estado actual de la robótica y del 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 la mano de otras disciplinas 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, requiriéndose 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 él[3]. Un ejemplo de este tipo de robots son los de KUKA<sup>1</sup> mostrado en la Figura 2.1.

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 Lunokhod<sup>2</sup> y actualmente tiene su máximo

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

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



Figura 2.1: Robot manipulador de KUKA.

exponente con el robot Curiosity, enviado 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. Si bien el uso de extremidades dificulta el control dinámico, la estabilidad y los algoritmos de navegación, ofrecen la posibilidad de desplazarse por gran variedad de terrenos y provocan 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 2.2.

---

<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/>



Figura 2.2: Robot cuadrúpedo WildCat.

### 2.1.2. Robótica en la educación

Las plataformas anteriormente descritas forman parte de proyectos millonarios que no son el grueso de la investigación en 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 generar 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:

- WIFIBOT: Se trata de una serie de robots movidos por cuatro ruedas, sensores de distancia , camera y punto de acceso WiFi. Puede ser controlado por varios sistemas operativos y lenguajes<sup>6</sup>.
- Lego NXT: Es un producto modular que permite una infinidad de combinaciones constructivas, partiendo siempre de un módulo central de procesador. Consta de 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>.

<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>

- ARDrone: Se trata de un vehículo no tripulado con cuatro hélices, o quadcoptero, que provee tanto la navegación autónoma como el radiocontrol.

La gran mayoría de robots, 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, han permitido a un amplio público la iniciación en la robótica. Por otra parte, se han desarrollado ciertos softwares libres que son una alternativa a los propios de cada plataforma, lo que facilita enormemente el trabajo del programador ya que no se tiene que invertir tiempo en el aprendizaje propio de cada software. Cabe destacar dos de ellos:

- URBI (Universal Real-time Behavior Interface)[5]: Del cual se pude encontrar información en la Sección 3.2.3.
- ROS: Es el marco de trabajo más extendido actualmente y abarca una gran cantidad de plataformas. En la Sección 5.1 se puede encontrar más información.

Por lo que respeta 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 se han realizado diversos proyectos de índole muy diversa. Sobretodo se han llevado a cabo proyectos relacionados con visión [6] y comportamientos e inteligencia artificial [11]. Un punto esencial en el desarrollo del AIBO fue que durante el período comprendido entre 1999 y 2008 fue la plataforma elegida para realizar la competición RoboCup Estandard Platform League, posteriormente sustituido por la plataforma NAO. La RoboCup es una competición internacional destinada a promover la robótica, donde se realizan diferentes pruebas como por ejemplo la de fútbol con robots autónomos. También en la prueba de simulacros de rescate AIBO tuvo una gran acogida y fue muy usado [8]. La RoboCup fue fuente de inspiración para múltiples investigaciones que se pueden clasificar en tres grupos según su finalidad:

- Visión y reconocimiento de marcas para la posterior ubicación dentro del campo [9].
- Comunicaciones y control de los movimientos [10].
- Definición de roles dentro del campo [7].

Actualmente el AIBO no se usa como se hacía 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[4] dejó de ser mantenido. A medida que ha caído en desuso, las nuevas actualizaciones de los softwares que lo soportaban han dejado de ser compatibles, 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. Así mismo gran parte de la documentación de estos lenguajes ya no se encuentra en la red lo que ha supuesto una dificultad añadida a este proyecto.

Ha habido otros intentos de recuperar la plataforma AIBO como es el caso de un proyecto que trataba de hacer compatible la versión 2.3 de liburbi para AIBO [17]. Otro proyecto trataba de realizar la integración mediante EusLisp a ROS de varias plataformas, entre ellas AIBO [16]. Sin embargo sólo se ha conseguido encontrar un artículo que lo mencione y ninguna documentación.

## 2.2. Objetivos

El objetivo de este proyecto es recuperar al AIBO como plataforma educativa y de investigación aportándole una capa superior de programación que permita la integración de la plataforma en ROS.

Como objetivos instrumentales se marcan los siguientes:

- Buscar la mejor opción de comunicación, entendida como el mejor equilibrio entre sencillez y calidad. Ésto conlleva un análisis cualitativo de los posibles métodos y un segundo análisis cuantitativo para determinar la elección definitiva.
- Crear un package de ROS que permita extraer todos los valores de los sensores y articulaciones y a su vez permita controlar al AIBO de forma remota.
- Desarrollar los package necesarios para demostrar el buen funcionamiento del paquete anteriormente mencionado.
- Implementar un package 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 incluye los siguientes desarrollos:



- Crear un paquete de ROS que permita extraer los valores de los sensores y articulaciones así como actuar sobre las últimas y poder controlar el AIBO.
- Implementar las pruebas necesarias para fundamentar el resultado de cualquier comparación.
- Implementar las pruebas para el análisis cuantitativo con solo un lenguaje.
- Crear un modelo 3D del AIBO y su descripción en formato URDF (Unified Robots Description Format).
- Implementar el driver que permita interaccionar el modelo visualizado con rviz y el AIBO real.

No se incluyen los siguientes aspectos:

- La adquisición de los datos obtenidos por la cámara o los micrófonos del AIBO.
- El renderizado del modelo.
- La dinámica del modelo.
- La definición de los sensores para el modelo.
- La descripción del modelo adaptada completamente para el simulador Gazebo<sup>9</sup>
- La creación de un mapa físico para el uso del modelo.

## 2.4. Planificación

La planificación del proyecto implica su realización en 17 semanas distribuyéndose las tareas programadas como muestra el diagrama de Gantt de la Figura 2.3.

---

<sup>9</sup>Es el simulador de licencia libre más usado en el marco de trabajo de ROS.



Figura 2.3: Diagrama de Gantt



# Capítulo 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 si fuera mascota real. Es capaz de percibir estímulos del exterior mediante una serie de sensores. Lleva incorporado de serie un software llamado AIBO MIND dentro de una tarjeta de memoria que le permite, entre otras cosas, reconocer a su dueño, reconocer objetos, jugar con una pelota y cierta capacidad de aprendizaje.

### 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 ERS-7, que se caracteriza por tener una camera de mayor resolución y un procesador más potente.



Figura 3.1: 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 en cada una de les 4 potes.
  - 3 grados de libertad en el cuello.
  - 1 grado de libertad en cada oreja.
  - 1 grado de libertad en la boca.
- 2 grados de libertad en la cola.
- Entrada de imagen.
  - Sensor de imagen CMOS de 350000 pixels.
  - Ángulos: 56.9° horizontal y 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 convertirlo en una herramienta de investigación y aprendizaje. Ésto llevó a SONY a facilitar un software de desarrollo llamado OPEN-R SDK que usaba un lenguaje propio: OPEN-R. A raíz de este módulo salieron otros lenguajes e interfaces que trabajaban en una capa superior, sobre OPEN-R y APERIOS, como son *URBI*<sup>1</sup> o *Tekkotsu*<sup>2</sup>.

### 3.2.1. OPEN-R

OPEN-R es un API(Application programming interface) en C++ que se ejecuta sobre el sistema operativo APERIOS 3.2. OPEN-R diferencia entre dos niveles, la capa de sistema por donde se accede al hardware del robot y la capa de aplicación para los programas desarrollados por el usuario.

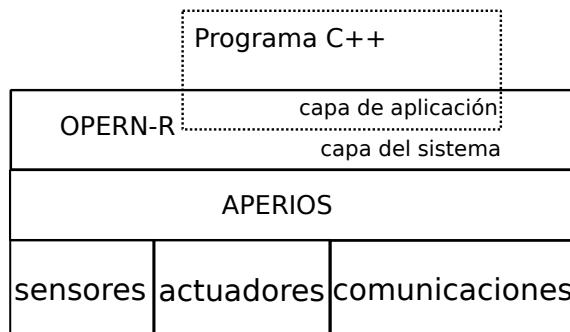


Figura 3.2: Arquitectura de AIBO con OPEN-R

Al tratarse de un lenguaje de bajo nivel su estructura es inherente al sistema operativo que se basa en objetos que interaccionan entre sí mediante mensajes o meta-objetos. El concepto de objeto es similar al utilizado en los sistemas *UNIX*<sup>3</sup> y *Windows*<sup>4</sup>, con la diferencia de que son monohilo y de que la comunicación se realiza mediante mensajes

<sup>1</sup>[www.urbiforge.org](http://www.urbiforge.org)

<sup>2</sup>[tekkotsu.org](http://tekkotsu.org)

<sup>3</sup>[www.unix.org](http://www.unix.org)

<sup>4</sup><http://windows.microsoft.com/en-us/windows/home>

que incluyen datos y un identificador del 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 envío de mensajes, uno de los objetos se define como sujeto (el origen) y el otro como observador (el destino) el cual inicia su ejecución después de que el mensaje haga indique el evento del método correspondiente.

SUJETO	OBSERVADOR
Envío de datos	
Notificación del evento	
	Recepción de datos
	Evento preparado

Tabla 3.1: Estructura del envío de mensajes en OPEN-R

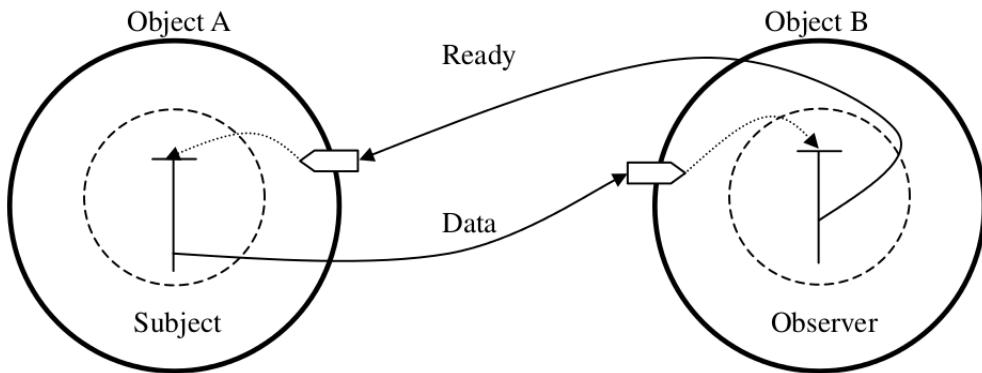


Figura 3.3: Comunicación entre objetos.

Una de las mayores complicaciones que se deriva al trabajar con OPEN-R es la gran variedad de archivos que deben ser modificados para poder configurar cualquier programa de aplicación. Por tanto es importante conocer bien los archivos con los que trabaja. A continuación se detallan los más importantes:

- Archivos .h y .cc: Son los archivos de programación de C++.
- STUB.config: Son archivos de configuración donde se define como los objetos se comunican entre sí.
- Archivos .ocf: Definen el comportamiento en lo relativo a los tiempos de ejecución, la memoria y la 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>5</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. Originalmente fue creado para AIBO, pero actualmente permite programar otros plataformas entre las que destacan: Chiara<sup>6</sup>, iRobot Create<sup>7</sup>, HandEye<sup>8</sup> o Lynxmotion Arms<sup>9</sup>.

Mantiene una arquitectura semejante a OPEN-R, basada en objetos y paso de eventos así como el lenguaje de programación 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 Tekkotsu. Así mismo facilita al usuario trabajar en alto nivel con herramientas de procesamiento visual, interfaces de monitorización, modelos de cinemática inversa y soporte para la administración de redes inalámbricas.

Además, Tekkotsu proporciona una interfaz de usuario, ControllerGUI, que proporciona acceso remoto al AIBO y ejecutar los comportamientos programados en la tarjeta de memoria. Estas funciones se pueden realizar mediante la interfaz gráfica basada en Java o por una conexión telnet<sup>10</sup> al puerto 10020 [12].

Tekkotsu 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*. El primero se encarga de la percepción y 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 [13].

---

<sup>5</sup>Este ultimo archivo es necesario configurarlo para cualquier método de programación, URBI, Tekkotsu o OPEN-R. Ver Anexo A

<sup>6</sup><http://chiara-robot.org>

<sup>7</sup><http://chiara-robot.org/Create/>

<sup>8</sup><http://chiara-robot.org/HandEye/>

<sup>9</sup><http://www.lynxmotion.com/>

<sup>10</sup><http://es.wikipedia.org/wiki/Telnet>

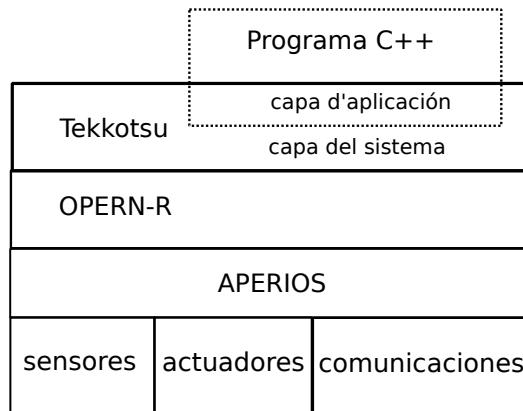


Figura 3.4: Arquitectura del AIBO con Tekkotsu.

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(). Ésto supone una simplificación respecto a los cuatro métodos de OPEN-R, lo que hace más sencilla la comunicación si perdidas de velocidad ya que mantiene la estructura del objeto.

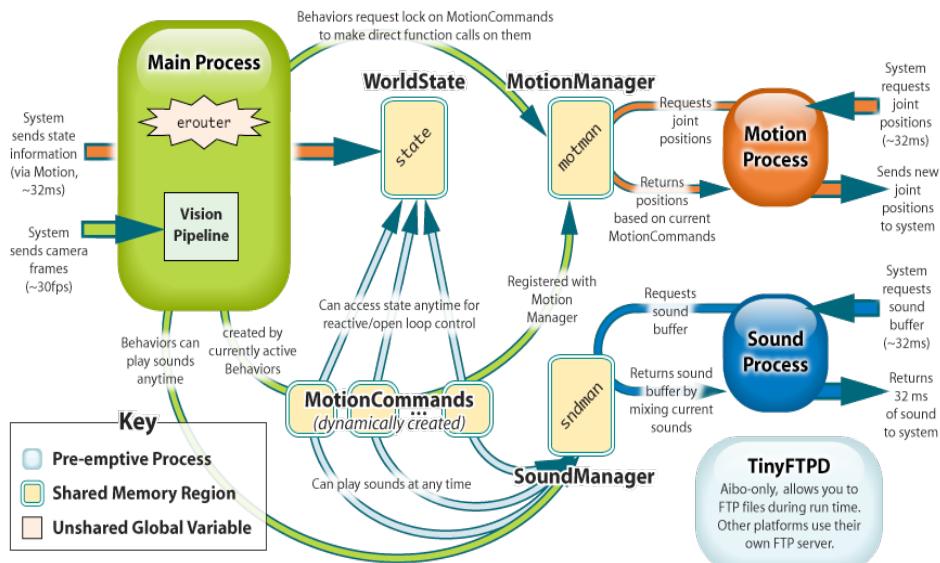


Figura 3.5: Proceso de ejecución de un comportamiento en Tekkotsu.

### 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 [14].

URBI es un lenguaje basado en scripts de alto nivel cuya mayor ventaja que permite ejecutar comandos en paralelo. Existen dos formas de trabajar con URBI: La primera se trata de usar el lenguaje de script para que 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, en la que los scripts se envían a través del terminal de URBI o alternativamente mediante el envío macros u órdenes concretas usando la librería liburbi, Figura 3.6.

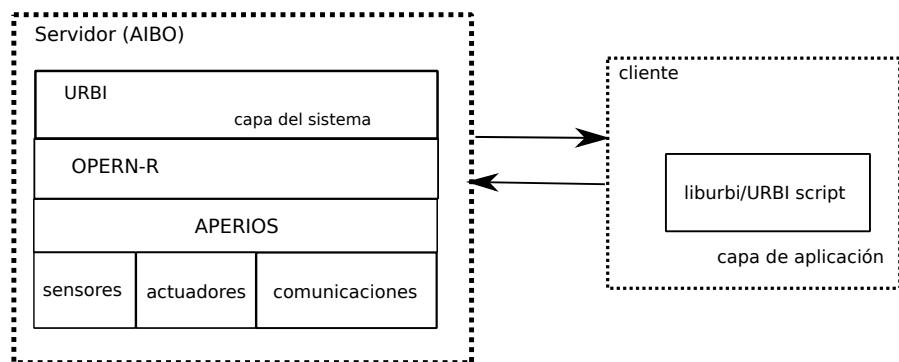


Figura 3.6: Arquitectura del AIBO con URBI.

La arquitectura cliente/servidor abre varias vías de programación. Por un lado proporciona un canal de comunicación por telnet y por otro el uso de la librería liburbi para trabajar 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;`[15].



## Capítulo 4

# Comparación de alternativas para la programación

Con la finalidad de implantar ROS en AIBO se plantea una primera elección del camino a seguir: Puede tratarse de un núcleo de ROS embebido, es decir que sea en el propio AIBO donde se ejecute el proceso. O bien que el núcleo de ROS se encuentre en un ordenador remoto y se realice una comunicación entre ambos mediante una arquitectura cliente/servidor.

Con tal de explorar la opción de un sistema embebido se ha buscado un punto de acceso al hardware con la intención de instalar una distribución linux. Sin embargo el único punto de acceso encontrado no respondía a ningún estándar, por lo que se desiste su utilización.

Por tanto sólo es posible usar una arquitectura cliente/servidor de tal forma que el modulo de ROS se ejecute en el cliente y sea capaz adquirir de forma sencilla y rápida el estado del AIBO y actuar sobre el.

Como se ha comentado anteriormente en la sección 3.2 se dispone de tres posibilidades de programación: OPEN-R, Tekkotsu y URBI. En los tres casos es imprescindible programar el cliente en el correspondiente lenguaje. URBI tiene la ventaja de que no sería necesario desarrollar el servidor ya que el sistema operativo interactúa de forma remota. Tekkotsu lee de forma remota, sin embargo no existe ningún comportamiento para el envío de órdenes a los actuadores y en consecuencia habría que implementarlo. Con OPEN-R habría que implementar el servidor completo, tanto el envío como la recepción.

No obstante para evaluar su funcionamiento se han instalado los tres marcos de trabajo y probado los tres lenguajes. Con respecto a OPEN-R se ha encontrado muy poca documentación y ha presentado un elevado coste de aprendizaje, tanto en lo relativo al

lenguaje de programación como en la configuración de los archivos. Tekkotsu ha resultado ser un lenguaje más sencillo y no necesita apenas configurar archivos no obstante la única documentación encontrada corresponde a la última versión, que no se puede compilar dentro de la tarjeta de memoria al ser incompatible con el compilador que usa OPEN-R. En cambio, sí se ha logrado compilar una versión más antigua y se han podido hacer algunas pruebas, aunque sin la documentación de la versión. Debido a las diferencias entre ambas versiones los desarrollos han sido muy limitados. Por lo que a URBI se refiere, su uso es realmente sencillo desde el terminal y con el uso de la librería de C++. También ha encontrado bastante documentación. El único problema que se ha detectado es que la librería 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>Tekkotsu</b>	<b>URBI</b>
Necesidad de programar un servidor.	Si	Si	No
Necesidad de programar un cliente.	Si	Si	Si
Consultar del estado por telnet.	No	Si	Si
Accionar las articulaciones por telnet.	No	No	Si
Dificultad de programación (0-5)	5	3	2
Documentación (0-5)	2	1	3
Plataforma del PC.	Windows/ Linux/ Mac OS	Linux	Windows/ Linux 32bits/ Mac OS

Tabla 4.1: Comparación entre lenguajes usados sobre AIBO

En la Tabla 4.1 se muestra un resumen de la comparativa de los tres lenguajes. Tekkotsu se ha descartado fundamentalmente por la ausencia de documentación de la versión que ha podido ser compilada. Si bien OPEN-R sería una buena opción ya que es la capa más baja de programación a la que se puede acceder. No obstante el elevado coste de

aprendizaje no permitiría respetar la planificación programada para la implementación del módulo. En consecuencia la opción elegida es URBI, habida cuenta de que es un lenguaje de fácil uso y que proporciona las herramientas necesarias, lo hacen el mejor candidato para alcanzar el objetivo del proyecto.

Des de 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.

En orden a determinar la opción de desarrollo elegida se realizarán una serie de experimentos comparativos que proporcionen unos resultados sobre la eficacia de las comunicaciones de ambos métodos. Los experimentos se centrarán en ambas direcciones de la comunicación, por un lado la lectura y por otro el envío.

#### 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. La ejecución del experimento requiere el desarrollo de un programa para cada método. La estructura de los programas para adquirir los datos será el mostrado en la Figura 4.1.

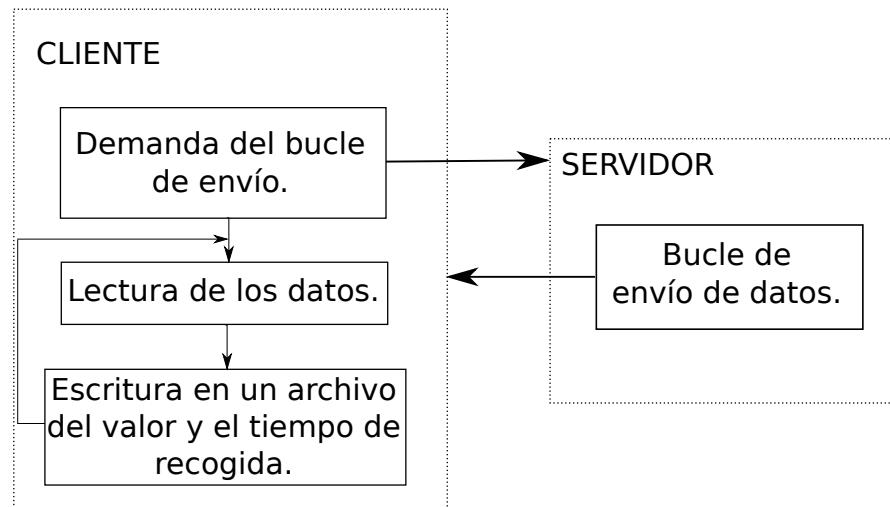


Figura 4.1: Estructura de la lectura de una variable.

La escritura de los valores de la articulación y su momento de lectura 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 cada uno de los métodos. Se pide al servidor el refresco continuo del valor de la articulación de modo que el sistema pueda trabajar a su máxima capacidad.

#### **4.1.1. Método con liburbi y C++**

La estructura del programa es la que sigue<sup>1</sup>.

- Iniciar el cliente URBI.
- Iniciar el callback: Éste es llamado cada vez que se reciba un dato con la etiqueta asignada.
  - Tomar el valor de la variable (aunque no sea necesaria para este experimento).
  - Consultar el tiempo en que ha sido recibido el dato.
  - Guardar el dato y el tiempo.
- Demandar el bucle y asignar de una etiqueta.
- Inicializar del archivo donde se guardarán los datos.
- Ejecutar del bucle de URBI: Se trata de una función que crea un bucle de comunicación cliente/servidor.

#### **4.1.2. Método con telnet y python**

La idea de este script<sup>2</sup> es trabajar con la terminal de URBI mediante una conexión telnet, Figura 4.2.

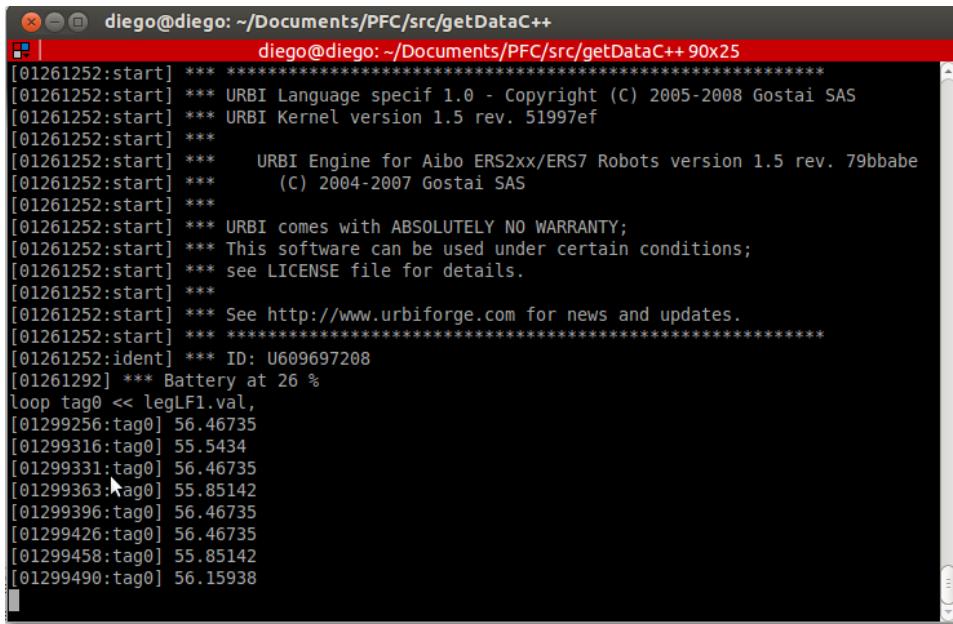
La estructura del programa será parecida a la anterior:

- Iniciar el objeto telnet.
- Leer la terminal y eliminar la cabecera de URBI.
- Escribir de el comando para recibir el bucle de envío. Para facilitar la lectura de los datos y diferenciar entre uno y el anterior se ha de enviar una marca después de cada dato.

---

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

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



```

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 4.2: Terminal URBI consultando un dato.

- Bucle de lectura y escritura.
  - Tratar la lectura:
    - Lectura hasta encontrar la marca la marca.
    - Eliminar las cadenas cortadas por el envío de la información de la batería<sup>3</sup>
    - Segmentar la cadena para finalmente guardar sólo el valor.
  - Adquirir el tiempo.
  - Escribir el tiempo y el valor en el archivo de texto.

#### 4.1.3. Comparación de los resultados

De los datos obtenidos de las 15 réplicas de cada método se han obtenido las diferencias de tiempos entre un dato y el siguiente y se traduce a frecuencia calculando el inverso.

---

<sup>3</sup>Cada vez que la batería baja la carga se escribe el porcentaje restante por la terminal de URBI.

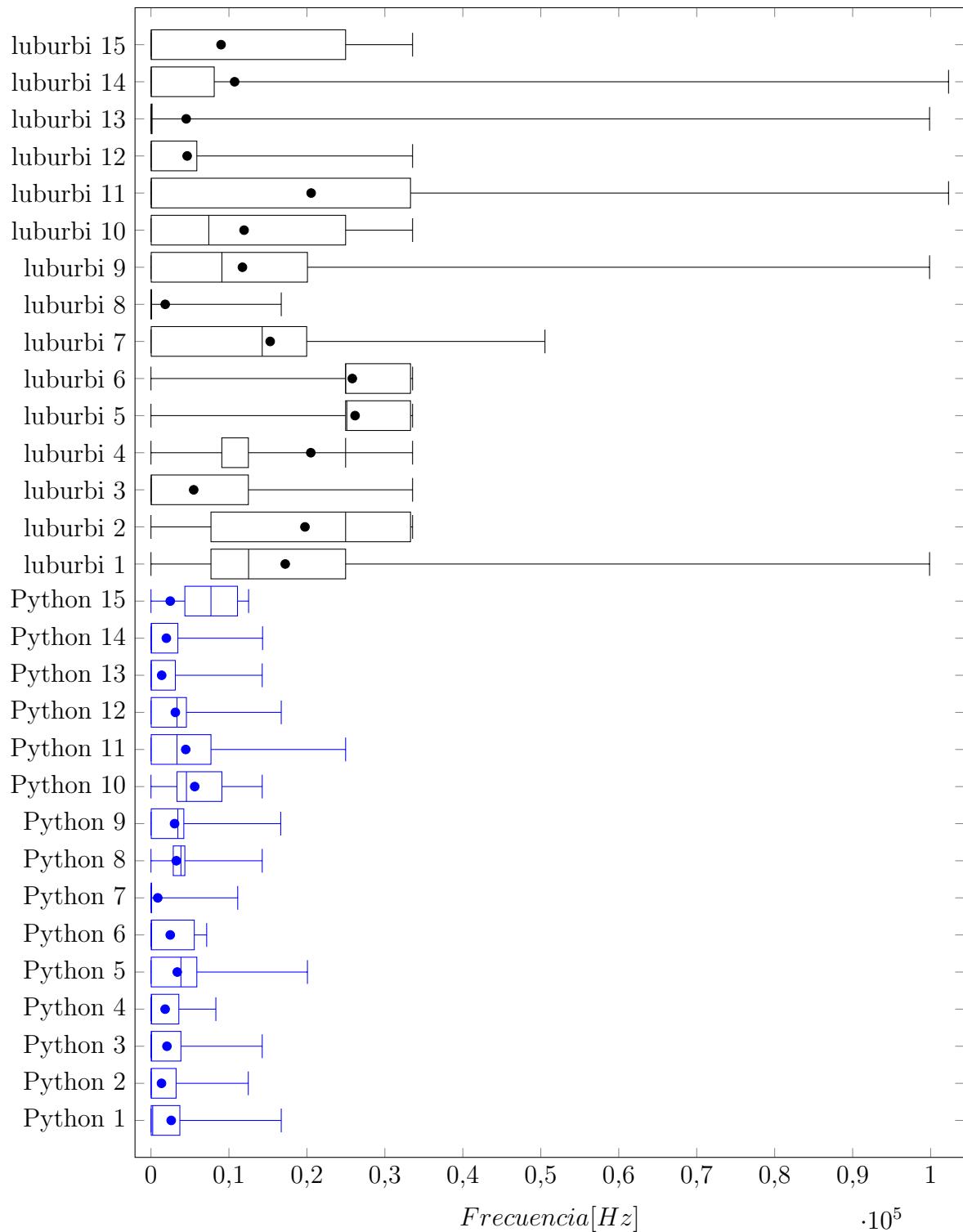


Figura 4.3: Diagrama de cajas de la frecuencia de datos para las 15 réplicas de cada método.

En el gráfico de la Figura 4.3 se muestra claramente que los resultados de liburbi

alcanzan medias más altas ( $\bar{x} = 13,611 Hz$  y  $\sigma = 13,271$  para liburbi y  $\bar{x} = 3,000 Hz$  y  $\sigma = 2,763$  para python) en la mayoría de réplicas, así mismo la mayoría de datos se envía más rápido como se refleja en los valores de las medianas de liburbi que son mayores que los valores del tercer quartil del método python. Por otro lado, los valores mínimos de cada método son muy similares y la variabilidad de los resultados de liburbi es mayor.

Tan importante como los valores medios de las frecuencias es comprobar que si se producen bloqueos, es decir si durante un cierto periodo de tiempo no se actualiza la variable. Para verificar si se producen bloqueos es necesario atender a los valores mínimos de las series, Figura 4.4.

Aparentemente liburbi presenta unos valores mínimos más bajos, sin embargo el análisis de la varianza (ANOVA) de los mínimos indica que no se corresponden con poblaciones distintas, se acepta la hipótesis nula ( $p\_valor = 0,4$  y  $I.C. = 95\%$ ).

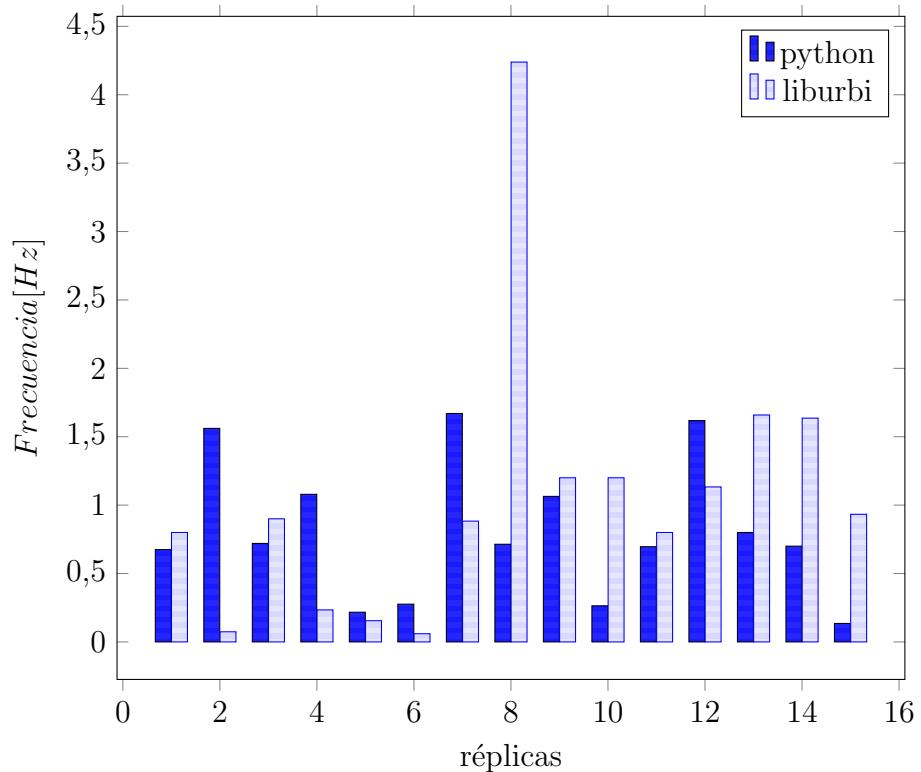


Figura 4.4: Mínimos de la frecuencia en las 15 réplicas de los dos métodos.

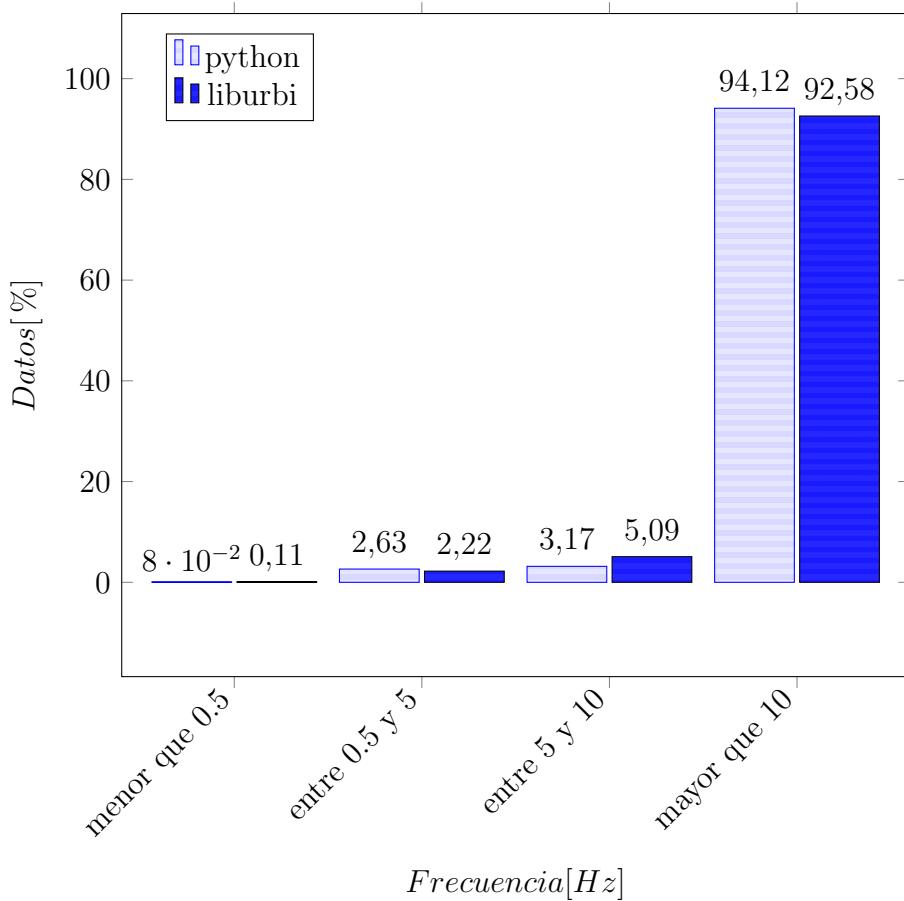


Figura 4.5: Histograma de la frecuencia en las 15 réplicas de los dos métodos.

Por otro lado, también es interesante conocer el volumen de bloqueos que se producen en cada caso con independencia de su duración. En este sentido es necesario agrupar los datos en intervalos de frecuencia. En la Figura 4.5 se clasifican los datos en los siguientes grupos:

- Menor que 0.5 Hz: Este intervalo refleja propiamente el concepto de bloqueos.
- 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.

El 94,12% de los datos en el caso de python y el 92,58% en el caso de liburbi están dentro del rango de trabajo ideal mientras que los bloqueos suponen un 0.08% y un 0.11% respectivamente. Esto representa en un periodo de funcionamiento de 150 segundos 4

bloqueos usando python y 5 al usar liburbi. En conclusión en este aspecto parece ser que los resultados de python son levemente mejores tanto en referencia al numero de bloqueos como en el rango de trabajo ideal.

A continuación se quiere verificar si la cantidad de datos a enviar es una variable que afecte al tiempo de recepción. Para ello se ha realizado el mismo experimento pero leyendo todas las articulaciones, lo que supone multiplicar el numero de datos a enviar.

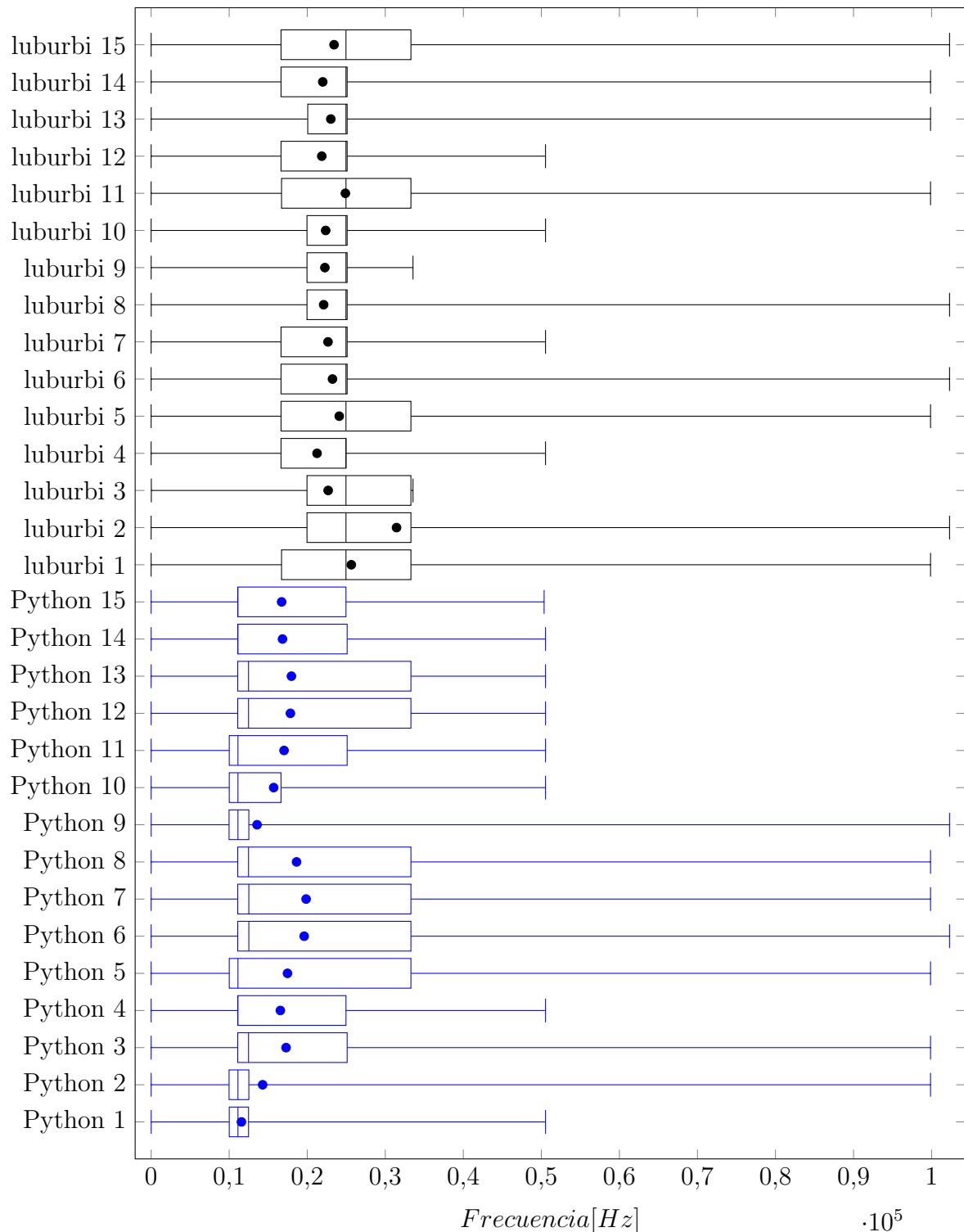


Figura 4.6: Diagrama de cajas de la frecuencia de datos para las 15 réplicas de cada método con todas las articulaciones.

Los resultados del experimento se muestran en la Figura 4.6 donde se muestra que los

valores medios de las frecuencias de liburbi son notablemente superiores a las de python ( $\bar{x} = 23,532,36Hz$  y  $\sigma = 11,603,63$  para liburbi y  $\bar{x} = 16,731,41Hz$  y  $\sigma = 11,530,49$  para python).

Por lo que respecta al análisis de los valores mínimos de las frecuencias y la distribución de los intervalos de trabajo, Figuras 4.7 y 4.8 respectivamente, los resultados obtenidos son prácticamente idénticos se puede asegurar el mismo comportamiento que en el experimento anterior.

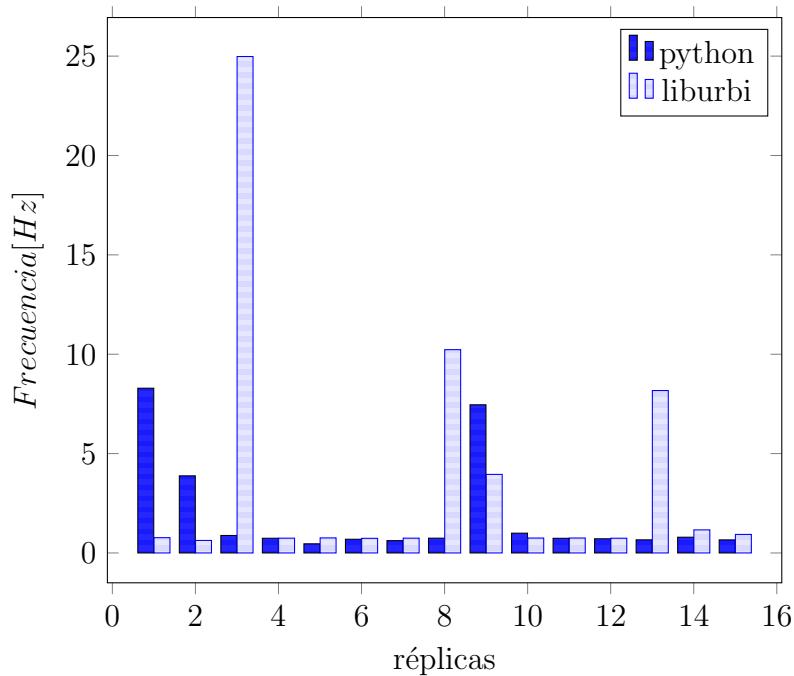


Figura 4.7: Mínimos de la frecuencia en les 15 repliques de los dos métodos.

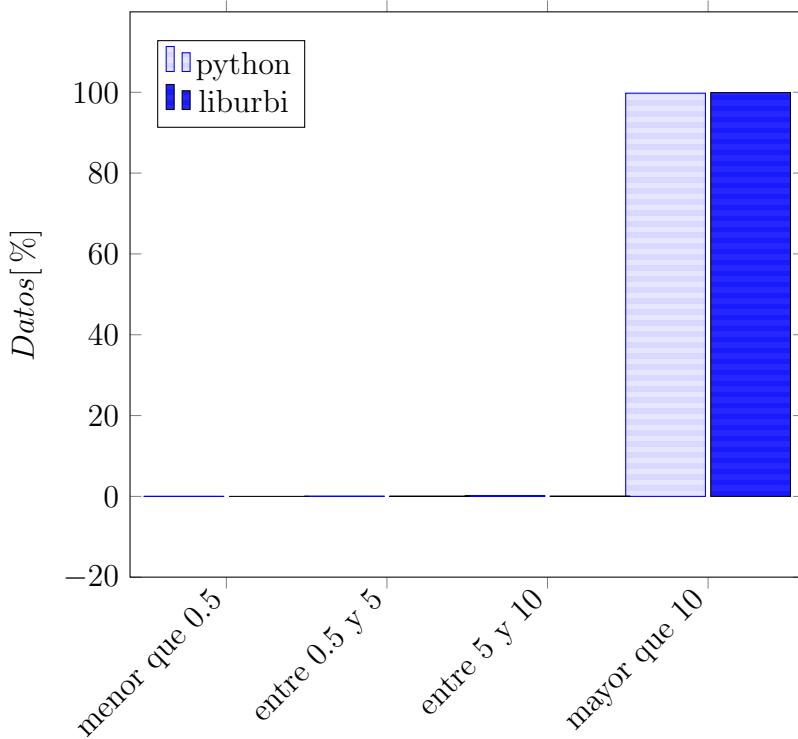


Figura 4.8: Histograma de la frecuencia en les 15 repliques de los dos métodos.

Se trata de comparar los resultados obtenidos para un valor y para todas las articula-

ciones. La velocidad de envío es significativamente más alta para todos los articulaciones en ambos casos aunque más acentuada en el caso de python, Figura 4.9. La explicación de este hecho es que el conjunto de valores se envía como un solo paquete y no individualmente, en consecuencia la frecuencia a la que las variables están disponibles es mayor. Cabe concluir que se ha podido comprobar que la cantidad de variables a enviar no tiene un efecto negativo en las frecuencias medias.

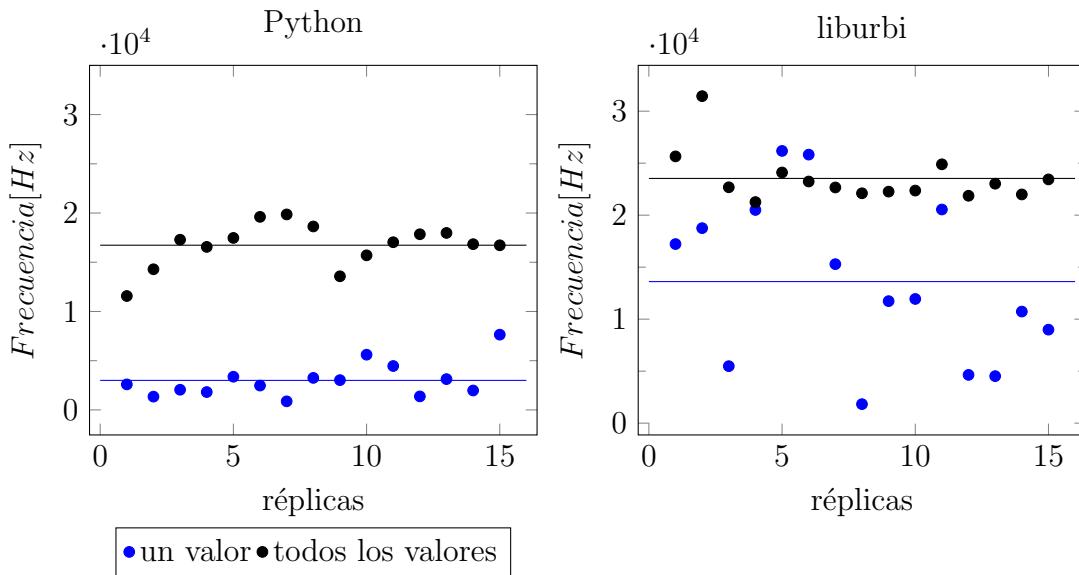


Figura 4.9: Medias de las frecuencias de envío para los valores de una articulación y para el valor de todas las articulaciones.

## 4.2. Envío de datos

El segundo experimento consiste en enviar una trayectoria punto a punto a una articulación y comprobar la respuesta. Se utilizará una trayectoria sinusoidal. Para capturar los resultados se leerá la respuesta de la articulación usando los módulos de lectura del experimento anterior.

En este experimento se ha contado con 3 posibles opciones de programación. Esto es así debido a que la terminal de URBI usando un cliente telnet con python no permite la lectura y escritura simultánea, y en consecuencia se han tenido que desarrollar dos clientes telnet. Aunque sí bien liburbi permite la lectura y escritura por un solo cliente, se ha considerado oportuno añadir un programa con dos clientes, equivalente al método utilizado con python.

En síntesis los programas que se van a utilizar son los siguientes:

- 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. En primer lugar 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. Y en segundo lugar ha sido necesario investigar y experimentar 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, y sólo cuando termina 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 un 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.

Se ha decidido que el modo más conveniente para la implementación del modulo es el modo **cancel**, ya que se va a trabajar con una forma de envío de ordenes de forma asíncrona de modo que sea el último dato el que tenga prioridad.

El experimento<sup>4</sup> se ha repetido cambiando la frecuencia a la que se envían los puntos de la trayectoria. Los valores de las frecuencias utilizados son: 1, 2, 5 y 10 Hz. Se han omitido mayores velocidades ya que el movimiento a una frecuencia de mas de 10 Hz era inconstante y discontinuo. Con la finalidad de obtener una mayor fiabilidad en los resultados se han realizado 3 réplicas por métodos y frecuencia. Se han descartado los experimentos en los que se ha producido un bloqueo. En los gráficos de las Figuras 4.10, 4.11, 4.12 y 4.13 representan los puntos de la trayectoria enviada y las posiciones leídas de la articulación

---

<sup>4</sup>Los códigos se puede encontrar en los Anexos B.3 y B.4

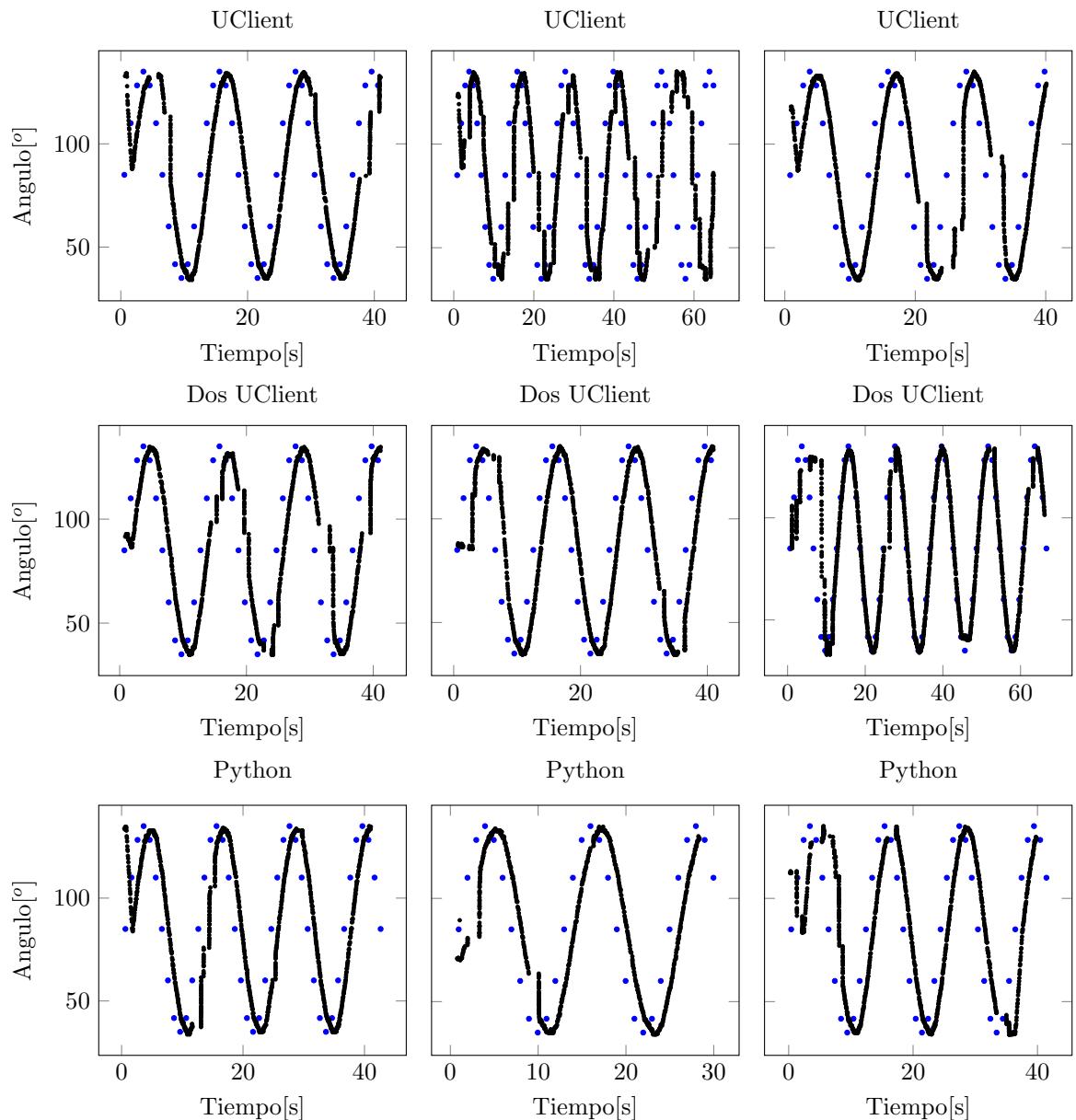


Figura 4.10: En azul, las posiciones enviadas a 1Hz y en negro las lecturas.

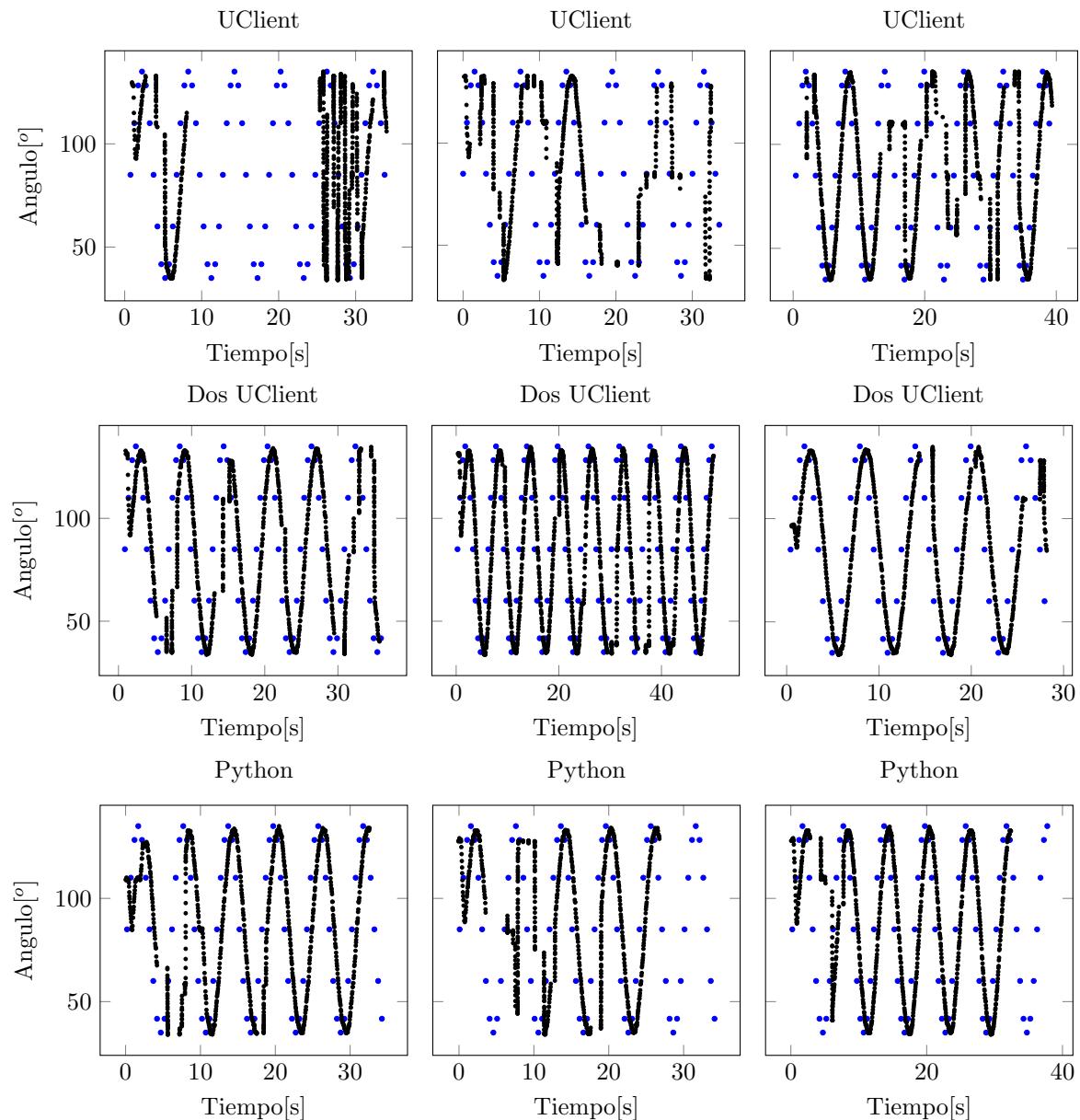


Figura 4.11: En azul, las posiciones enviadas a 2Hz y en negro las lecturas.

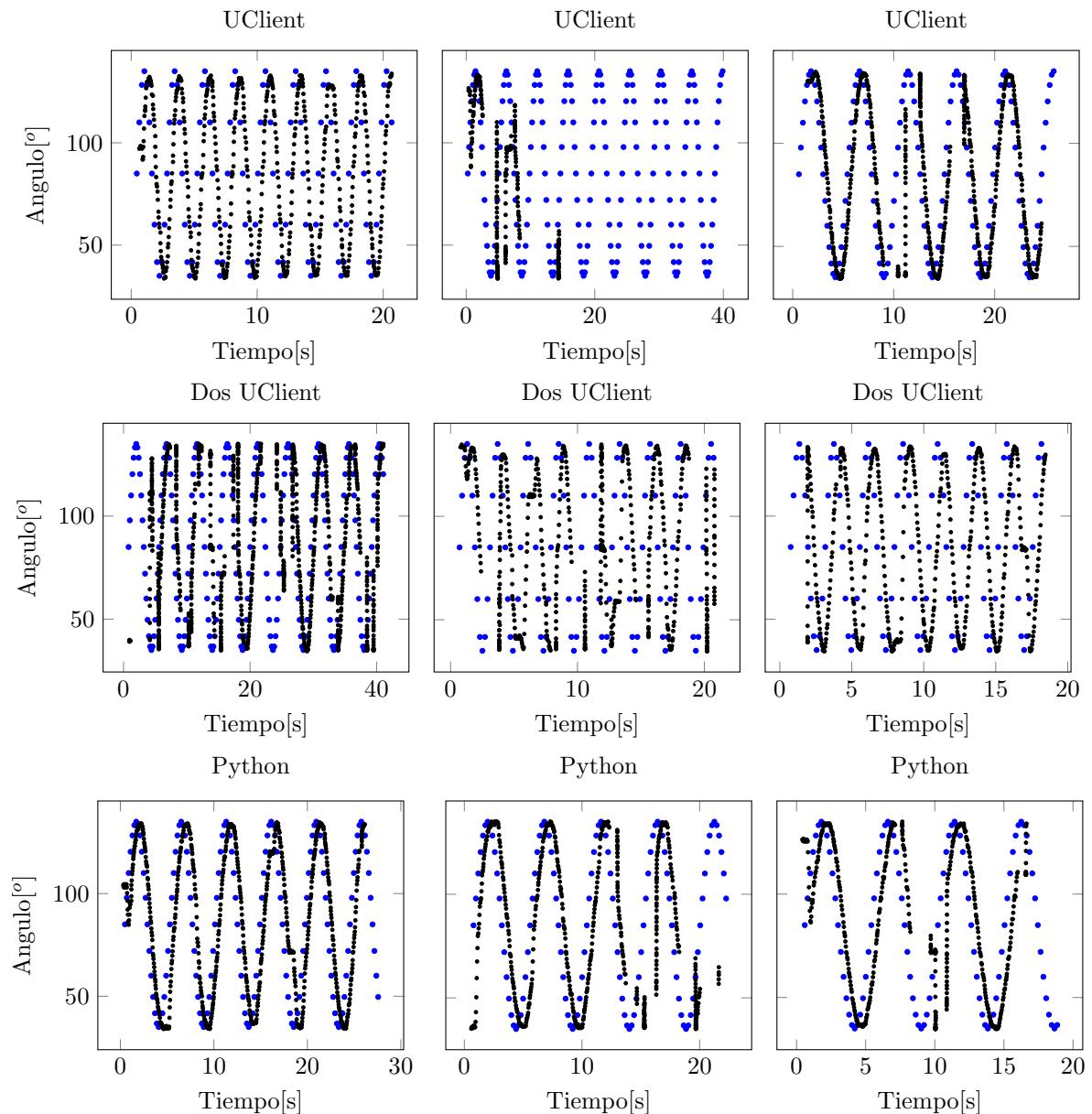


Figura 4.12: En azul, las posiciones enviadas a 5Hz y en negro las lecturas.

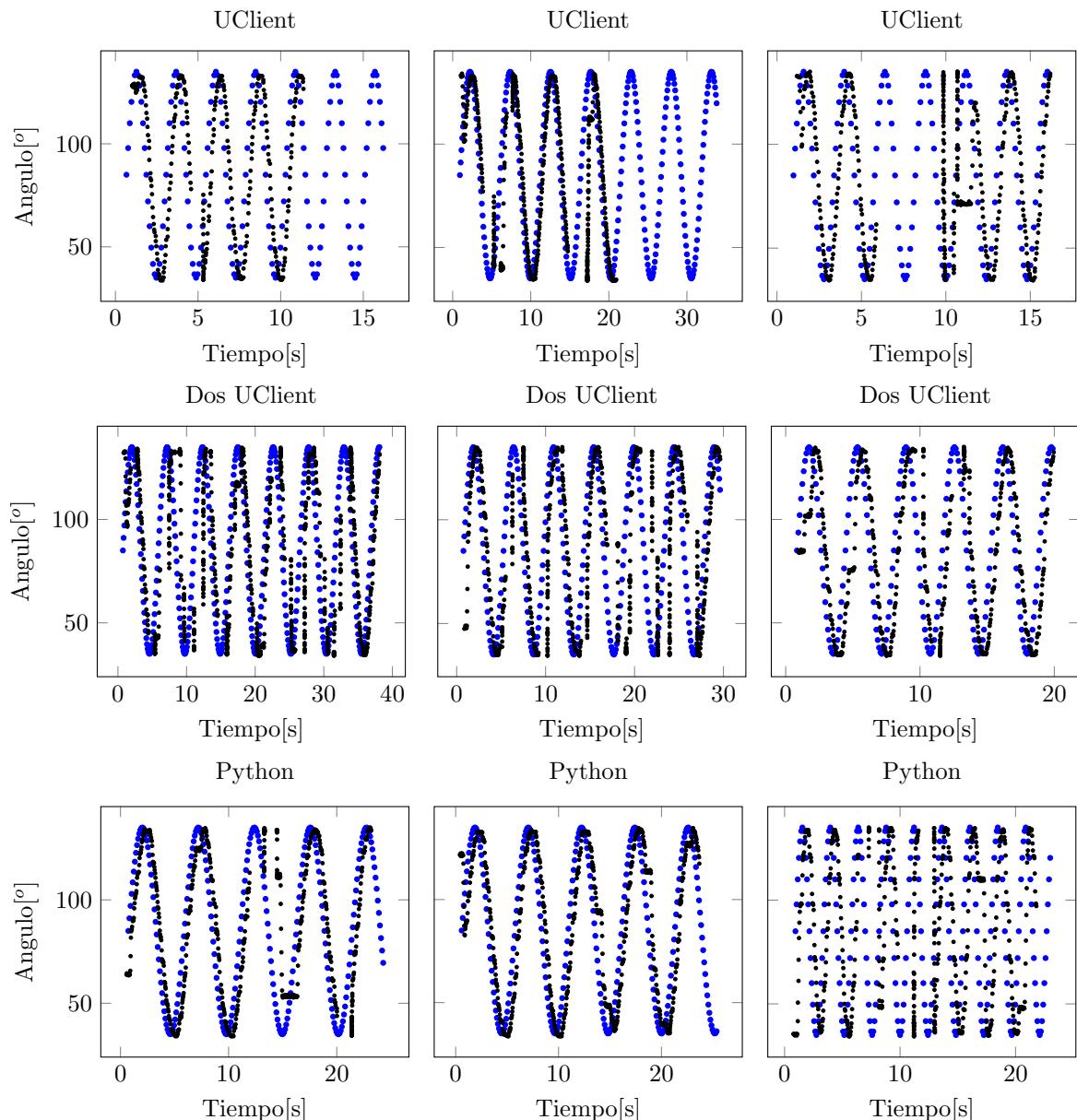


Figura 4.13: En azul, las posiciones enviadas a 10Hz y en negro las lecturas.

En los gráficos anteriores se pueden distinguir dos tipos de errores que provocan que la trayectoria no sea la correcta:

- La articulación no se mueve: Reflejado como una recta negra horizontal y el modulo responsable es el de escritura.
- No se recibe la posición actual: Reflejado como un periodo sin datos. En este caso no se puede saber si la articulación se ha movido<sup>5</sup>.

<sup>5</sup>Por inspección visual se ha comprobado que ambos errores son independientes.

Se puede concluir que el seguimiento de la trayectoria es independiente de la frecuencia utilizada en este rango. Por otro lado también se observa que con un solo cliente liburbi se producen grandes bloqueos en la recepción y muestra una trayectoria más discontinua e inconstante. En los que respecta a los otros dos métodos, dos clientes con liburbi y python, tienen un comportamiento aceptable y no se puede concluir que uno sea mejor que otro.

Para discernir cual de los dos métodos es más eficiente se han analizado las diferencias de tiempo entre el momento en que se envía la orden de los picos de la sinusoida y el momento en que se recibe que ha llegado a dicha posición. Para comprobar si existe diferencia entre las series se realiza una ANOVA sobre los datos de la Figura 4.14. Con un intervalo de confianza del 95 % el p-valor es de 2,1e-7, por lo tanto no se acepta la hipótesis nula y se puede concluir que el retraso con C++ es significativamente menor.

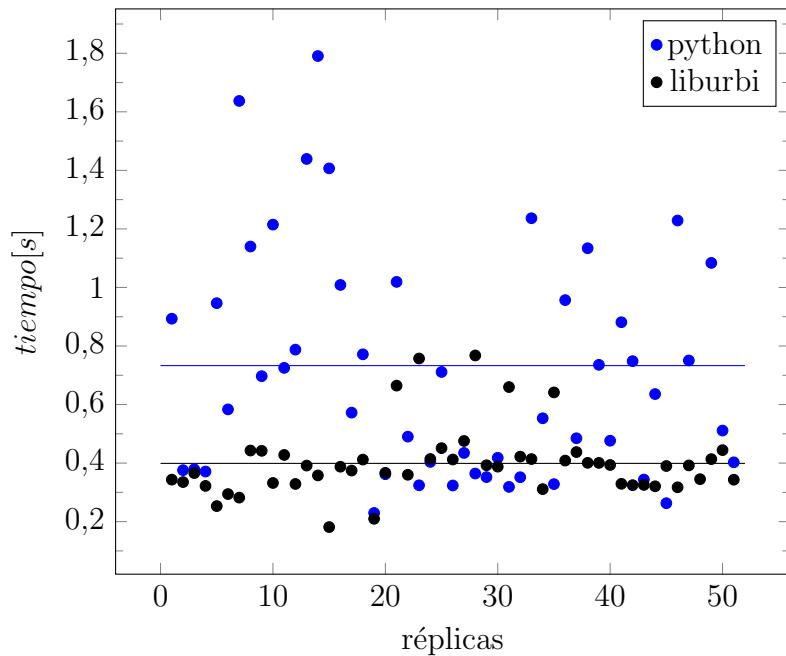


Figura 4.14: Retrasos entre la orden enviada y la acción.

### 4.3. Elección del método

La elección del método que garantizará un mejor funcionamiento del paquete a implementar se realiza conforme a los criterios de evaluación resumidos en la Tabla 4.2

	<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 [Hz]	24.966,09	102.300,09	102.300,09
Frecuencia media de lectura [Hz]	13.610,99	2.999,50	2.999,50
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	Mayor	Menor	
Bloqueos en el envío	Si	Si	Si

Tabla 4.2: Esumen de los criterios de elección.

En síntesis, el método liburbi se ha demostrado más eficaz en lo relativo a la recepción de datos, muestra un comportamiento similar a python en el seguimiento de las trayectorias con el uso de dos clientes y además se obtiene un retraso de seguimiento menor que python estadísticamente significativo. Adicionalmente, se ha de tener en consideración que el paquete será más complejo que los programas utilizados en los experimentos y dado que C++ se ejecuta a mayor velocidad que python las diferencias de ejecución podrían llegar a ser criticas. En consecuencia la librería liburbi usando dos clientes es el método elegido para implementar el paquete de ROS.

# Capítulo 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 unas librerías para ayudar a desarrollar software para aplicaciones de robótica. Proporciona entre otras facilidades abstracciones de hardware, controladores para dispositivos, herramientas de visualización, comunicación por mensajes, administración de paquetes y todo ello bajo licencia de software libre. La principal característica sobre ROS es su sistema de comunicación. A bajo nivel está basado 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 suscribir desde un terminal o bien cualquier objeto de ROS denominados *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 ejecutan sobre ROS deben ser creados dentro de un paquete y todo paquete de ROS debe contener una serie de archivos necesarios para su compilación. Los archivos necesarios son los siguientes.

- **manifest.xml:** Incluye información sobre el nombre del paquete, el autor, tipo de licencia y paquetes externos necesarios.
- **CMakeLists.txt:** Indica el uso de librerías, el uso de mensajes propios del paquete y se declaran los ejecutables.
- **mainpage.dox:** Contiene un resumen explicativo del paquete.



- Archivos ejecutables: ROS permite usar su API con C++ y 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.

### 5.1.1. Paquete aibo\_server

Se pretende implementar un paquete que se conecte al AIBO dentro de una red local. 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 del tópico al que está suscrito y enviarlos al servidor con el fin de actuar sobre la plataforma.

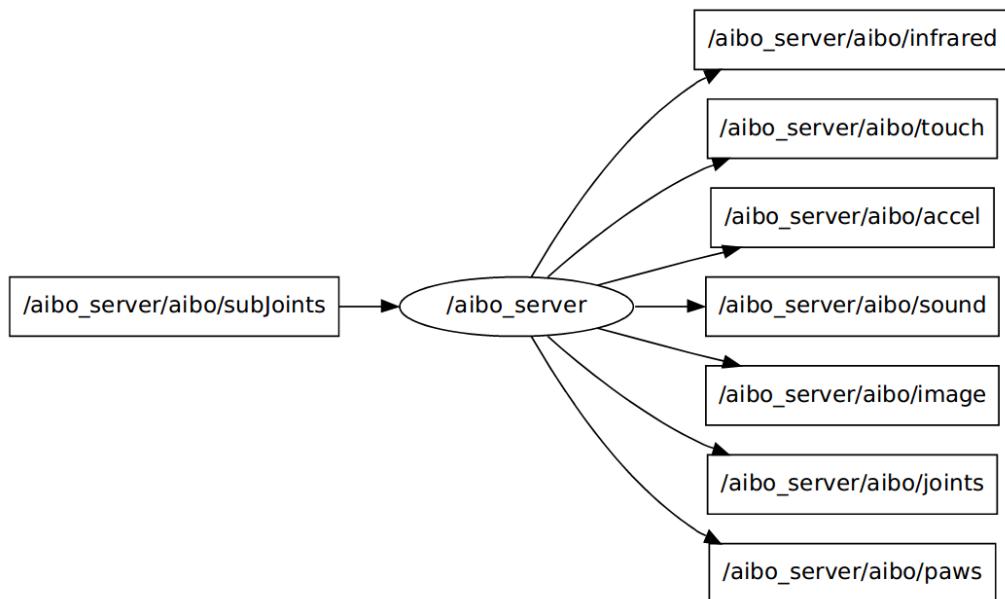


Figura 5.1: Nodo aibo\_server.

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: Para utilizar la API ROS para C++.
  - std\_msgs: Permite el uso de mensajes estándar.
  - sensor\_msgs: Permite el uso de mensajes especialmente destinados a ciertos tipos de sensores.
- Implementar los archivos necesarios para el ejecutable:
  - AiboNode.cpp: Archivo a partir del cual se crea el ejecutable del paquete. En él reside la estructura del programa.
  - AiboServer.cpp y AiboServer.h: Se define la clase aibo en la que se basa AiboNode.cpp.
  - AiboParams.h: Se definen las constantes.
- Definir 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.
- Modificar el CMakeList.txt para incluir los mensajes y los archivos C++ anteriores junto con la librería liburbi.

### 5.1.2. Estructura del programa

Conforme a lo indicado en la sección anterior, Sección 4, se procede a la implementación del paquete de ros usando liburbi y dos clientes, uno para la recepción y otro para el envío.

La estructura del programa implementado se puede ver en la Figura 5.2.



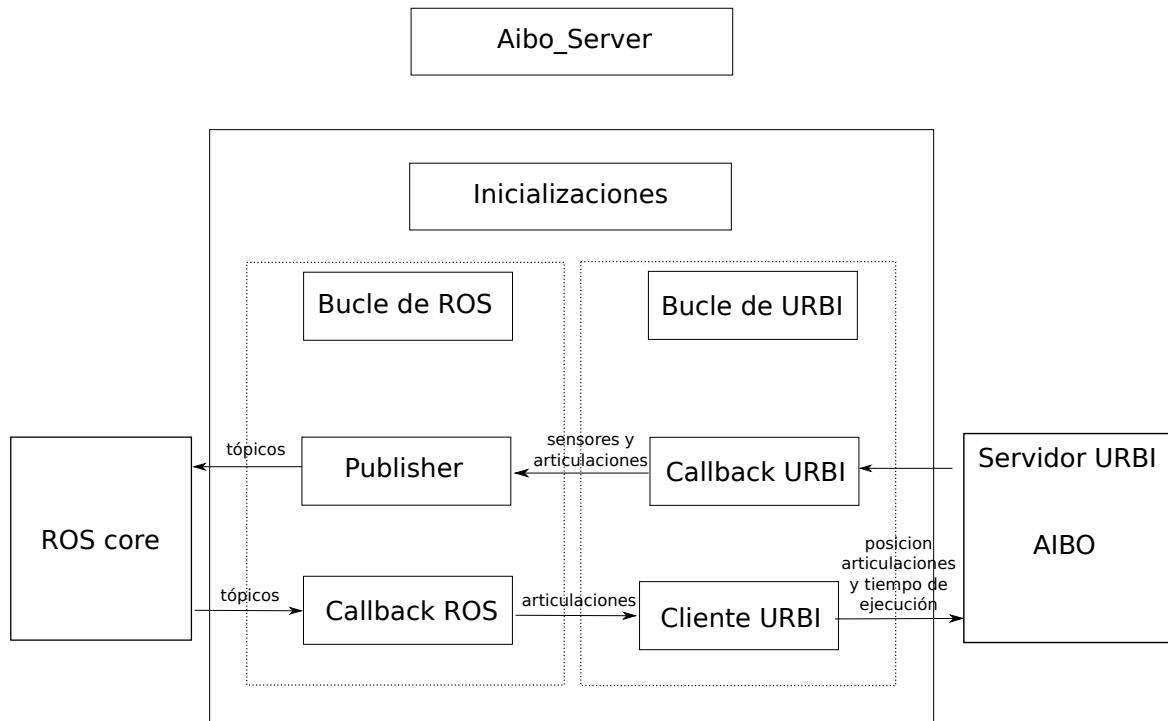


Figura 5.2: Estructura básica del ejecutable aibo\_server.

Siguiendo el esquema en la Figura 5.2 se detallan los diferentes bloques.

■ 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 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 tópicos.
  - 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 a cabo 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.

### 5.1.3. Resultados del aibo\_server

Por lo que respecta la adquisición de los valores de sensores y articulaciones se ha conseguido obtenerlos trabajando a una frecuencia de 10Hz. Con el objeto de verificar que el refresco de los tópicos es correcto se ha accedido a mostrar por el terminal todos ellos junto con su frecuencia de refresco Figura 5.3, y al mismo tiempo, en la Figura 5.4 se representa el valor de un sensor. El hecho de que el gráfico muestre un cierto ruido indica que el valor se está refreshando correctamente, por el contrario si el valor de la gráfica es constante indica que no se está refreshando correctamente.



```

diego@diego:~          diego@diego:~ 83x9          diego@diego:~ 82x20          diego@diego:~ 82x20
average rate: 9.940      min: 0.079s max: 25.104s std dev: 0.12273s window: 41508 average rate: 9.939      min: 0.079s max: 25.104s std dev: 0.12397s window: 40684
average rate: 9.940      min: 0.079s max: 25.104s std dev: 0.12272s window: 41518 average rate: 9.939      min: 0.079s max: 25.104s std dev: 0.12395s window: 40694
average rate: 9.940      min: 0.079s max: 25.104s std dev: 0.12270s window: 41528 average rate: 9.939      min: 0.079s max: 25.104s std dev: 0.12394s window: 40704
average rate: 9.940      min: 0.079s max: 25.104s std dev: 0.12269s window: 41538 average rate: 9.939      min: 0.079s max: 25.104s std dev: 0.12392s window: 40714
average rate: 9.937      min: 0.079s max: 25.104s std dev: 0.12545s window: 39729 average rate: 9.939      min: 0.079s max: 25.104s std dev: 0.12389s window: 40725
average rate: 9.937      min: 0.079s max: 25.104s std dev: 0.12543s window: 39739 average rate: 9.939      min: 0.079s max: 25.104s std dev: 0.12388s window: 40745
average rate: 9.937      min: 0.081s max: 25.104s std dev: 0.12542s window: 39749 average rate: 9.939      min: 0.079s max: 25.104s std dev: 0.12386s window: 40755
average rate: 9.938      min: 0.081s max: 25.104s std dev: 0.12540s window: 39759 average rate: 9.939      min: 0.079s max: 25.104s std dev: 0.12385s window: 40766
average rate: 9.938      min: 0.079s max: 25.104s std dev: 0.12540s window: 39759 average rate: 9.939      min: 0.079s max: 25.104s std dev: 0.12383s window: 40776
min: 0.077s max: 25.104s std dev: 0.12489s window: 40088 min: 0.074s max: 25.104s std dev: 0.12431s window: 40464
average rate: 9.938      min: 0.077s max: 25.104s std dev: 0.12487s window: 40099 average rate: 9.939      min: 0.074s max: 25.104s std dev: 0.12429s window: 40474
average rate: 9.938      min: 0.077s max: 25.104s std dev: 0.12485s window: 40109 average rate: 9.939      min: 0.074s max: 25.104s std dev: 0.12427s window: 40485
average rate: 9.938      min: 0.077s max: 25.104s std dev: 0.12484s window: 40119 average rate: 9.939      min: 0.074s max: 25.104s std dev: 0.12426s window: 40495
average rate: 9.938      min: 0.077s max: 25.104s std dev: 0.12482s window: 40129 average rate: 9.939      min: 0.074s max: 25.104s std dev: 0.12424s window: 40505
average rate: 9.938      min: 0.077s max: 25.104s std dev: 0.12481s window: 40139 average rate: 9.939      min: 0.074s max: 25.104s std dev: 0.12423s window: 40515
average rate: 9.938      min: 0.077s max: 25.104s std dev: 0.12479s window: 40150 average rate: 9.939      min: 0.074s max: 25.104s std dev: 0.12421s window: 40525
average rate: 9.938      min: 0.077s max: 25.104s std dev: 0.12478s window: 40160 average rate: 9.939      min: 0.074s max: 25.104s std dev: 0.12420s window: 40536
average rate: 9.938      min: 0.077s max: 25.104s std dev: 0.12476s window: 40170 average rate: 9.939      min: 0.074s max: 25.104s std dev: 0.12418s window: 40546
average rate: 9.938      min: 0.077s max: 25.104s std dev: 0.12474s window: 40180 average rate: 9.939      min: 0.074s max: 25.104s std dev: 0.12417s window: 40556

```

Figura 5.3: Consulta de la frecuencia de publicación en los diferentes tópicos de aibo\_server.

Ángulo[°]

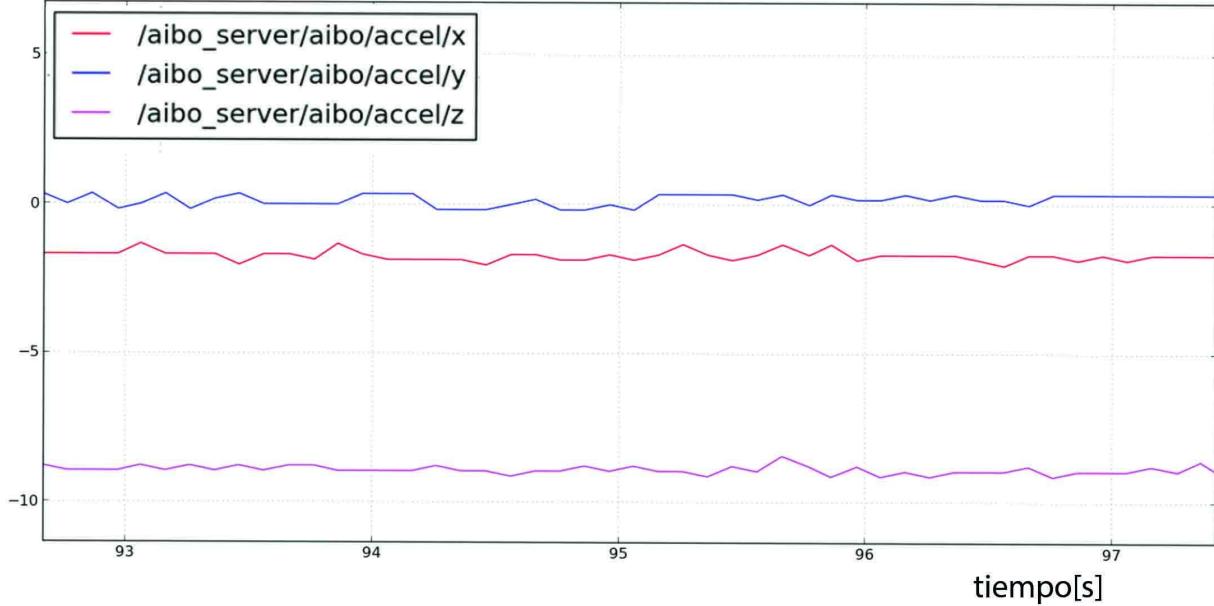


Figura 5.4: Valores de los acelerómetros usando la herramienta rqt\_plot.

Para realizar una valoración del envío de las posiciones necesita de un paquete de ROS que permita cuantificar los resultados.

El test aplicado es similar al realizado en las experimentaciones de la Sección 4. Se trata

de enviar punto a punto una trayectoria sinusoidal a las posiciones de una articulación. Para la realización del test se ha implementado un sencillo programa<sup>1</sup> que crea un nodo que publica sobre el tópico `/aibo_server/aibo/subJoints/jointRF1`. La estructura entre los nodos se muestra en la Figura 5.5.

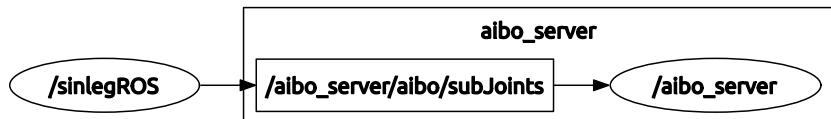


Figura 5.5: Estructura de ROS con los nodos aibo\_server y SinLeg.

De los resultados obtenidos en la realización de varias réplicas a diversas frecuencias se deduce que el seguimiento de la trayectoria lleva un retraso mínimo entorno a los 0.5 segundos, dato que coincide con el retraso medio obtenido en las experimentaciones de la sección 4.2. Si bien, como se puede observar en las Figuras 5.6, 5.7 y 5.8 dicho retraso no es la norma general estando el retraso medio es mayor al obtenido en las experimentaciones.

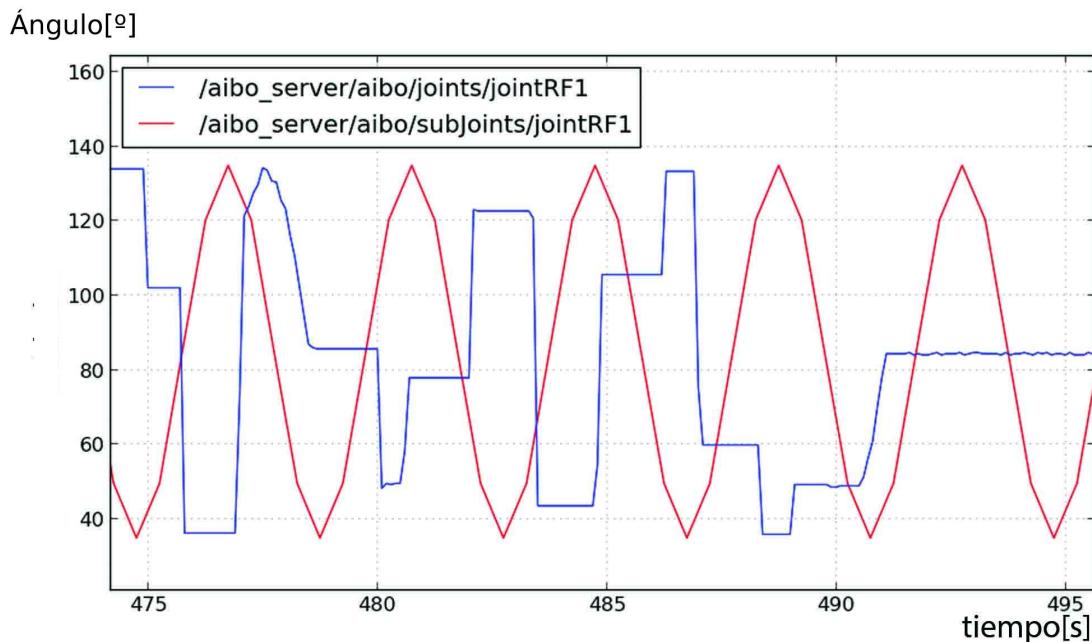


Figura 5.6: Entrada y respuesta del sistema ante una señal sinusoidal de enviando puntos a 2Hz.

<sup>1</sup>El código se puede encontrar en el Anexo B.5

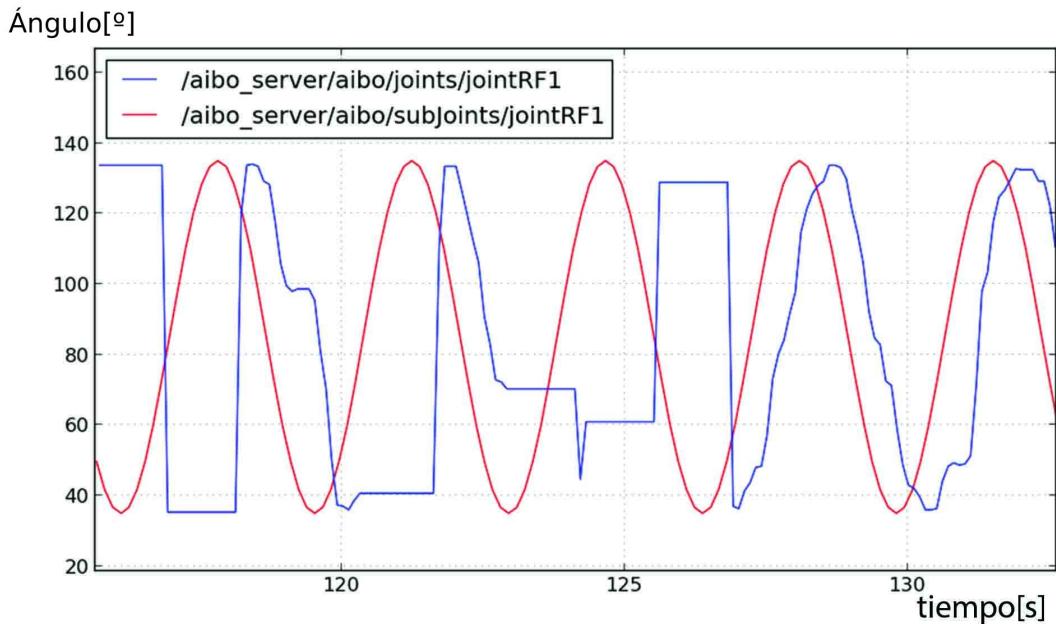


Figura 5.7: Entrada y respuesta del sistema ante una señal sinusoidal de enviando puntos a 7Hz.

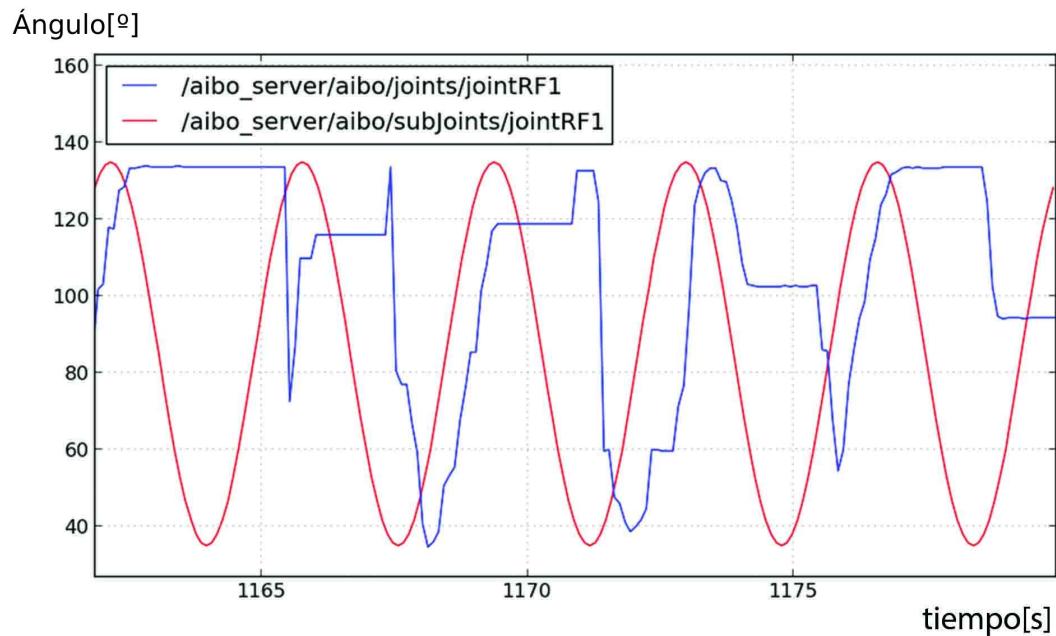


Figura 5.8: Entrada y respuesta del sistema ante una señal sinusoidal de enviando puntos a 10Hz.

Por otro lado el modulo implantado no ha mejorado los resultados obtenidos en la sección 4.2 en lo relativo a los bloqueos. Por lo tanto, cabe concluir que el módulo no

proporciona las condiciones necesarias para aquellas aplicaciones en las que se requiere una lectura y una respuesta rápida del sistema.

#### 5.1.4. Mejoras del aibo\_server

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

- Cortas congelaciones en la recepción de los valores de los sensores y articulaciones:  
Ya se ha probado que en las experimentaciones anteriores que los resultados no podían mejorar usando liburbi.
- Envío de la orden correcto pero el movimiento se realiza con una demora: Esto es debido a un problema interno en el tratamiento de las órdenes del servidor URBI y por lo tanto des del punto de vista del cliente no se puede aportar ninguna solución.
- Bloqueos en el envío de la orden de movimiento: Dichos bloqueos se originan en la función de liburbi que envía órdenes provocando un bloqueo completo del nodo.

Aunque para los dos primeros problemas no existe una solución si se ha encontrado una solución para el bloqueo en el envío de las órdenes de movimiento. En primer lugar, cuando se produce un bloqueo en el envío, 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. Para ello se inicializa un tercer cliente URBI que reemplace al cliente de envío en caso de que éste se bloquee. Igualmente en caso de bloqueo del nuevo cliente se espera que el primero, que ya se habrá reinicializado, tome el relevo. Bajo estas ideas se ha reescrito el callback de ROS como muestra el diagrama de la figura 5.9.

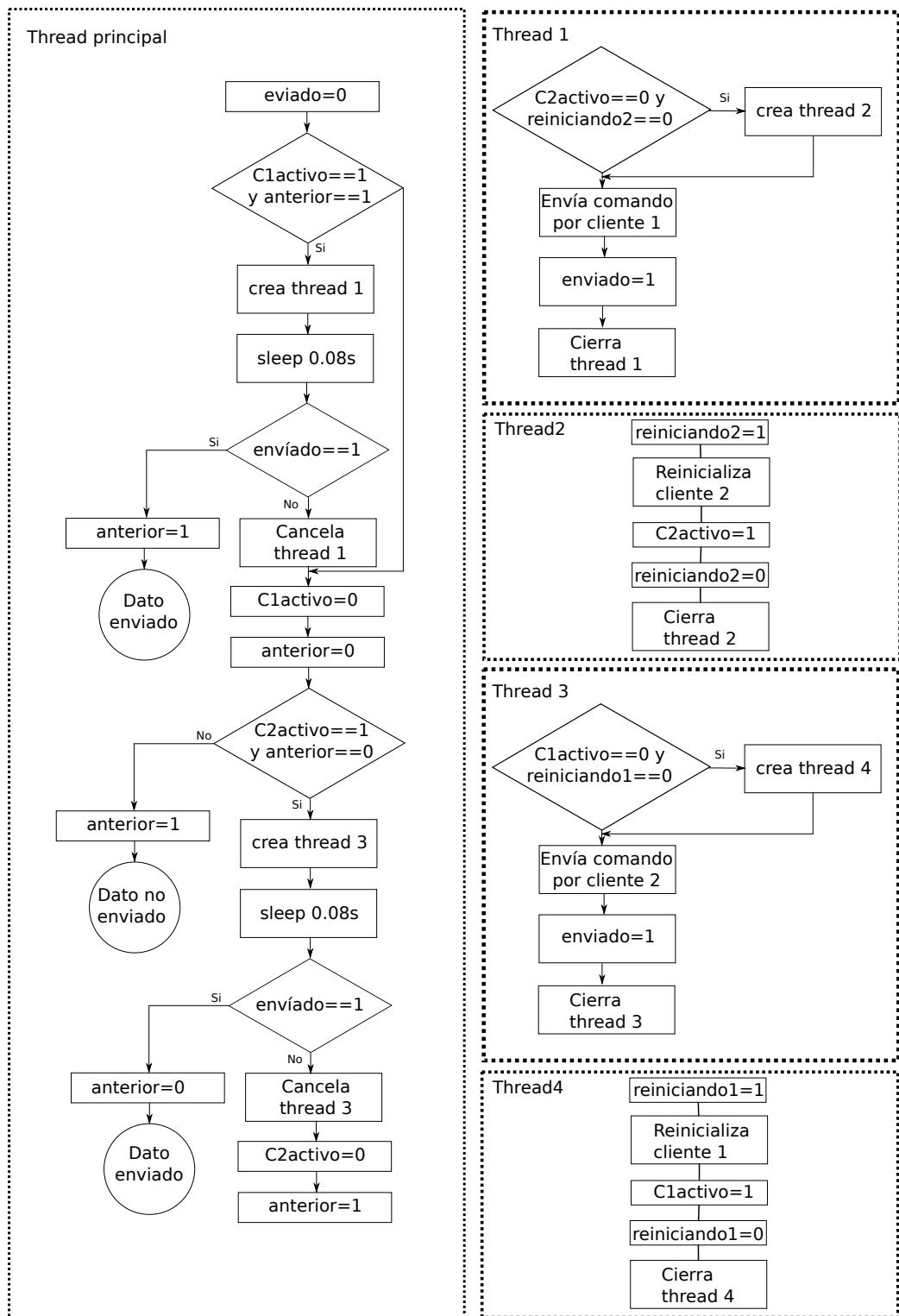


Figura 5.9: Diagrama del callback de ROS. Los flags *anterior*, *C1activo* y *C2activo* se inicializan con valor 1 y los flags *reiniciando1* y *reiniciando2* se inicializan a 0.

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

Con la modificación introducida se vuelve a aplicar el test. En la Figura 5.10 se ve como tras un primer seguimiento aceptable de la trayectoria empieza una zona donde el seguimiento empeora. Durante este periodo de tiempo que continua en la Figura 5.11 la comunicación con el AIBO es realmente mala y donde se originaran los bloqueos. Tras implementar el nuevo callback la comunicación no se corta, pero la trayectoria no se sigue correctamente.

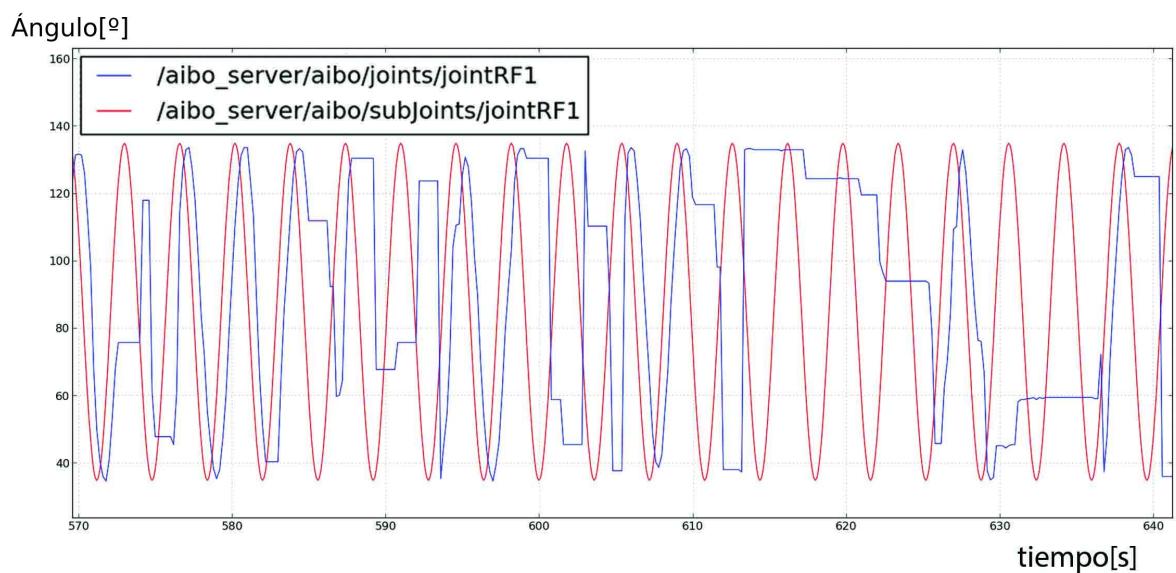


Figura 5.10: Empiezan a producirse bloqueos.

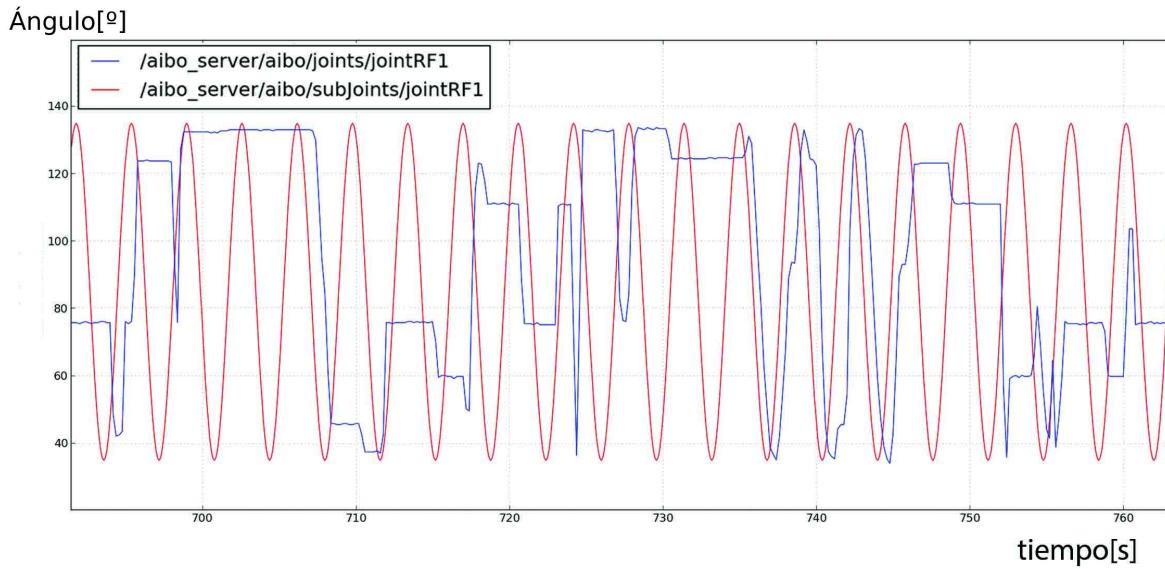


Figura 5.11: Se producen bloqueos que se intentan evitar.

En la Figura 5.12 se ve como , tras una serie de sucesivos bloqueos de ambos clientes, se recupera un buen seguimiento de la trayectoria.

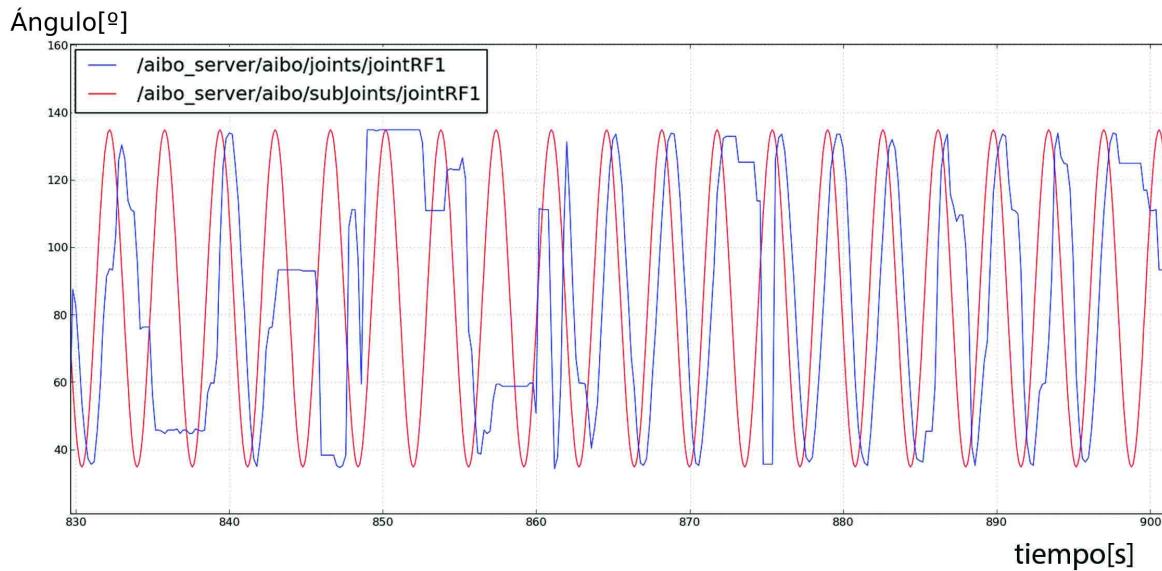


Figura 5.12: Recuperación del seguimiento de la trayectoria.

Finalmente en la Figura 5.13 se puede observar como en algunos momentos el seguimiento de la trayectoria es satisfactorio.

Cabe señalar que en los gráficos obtenidos mediante la herramienta de ROS rqt\_plot no permiten diferenciar entre un error recepción de datos o un seguimiento de la trayectoria incorrecto.

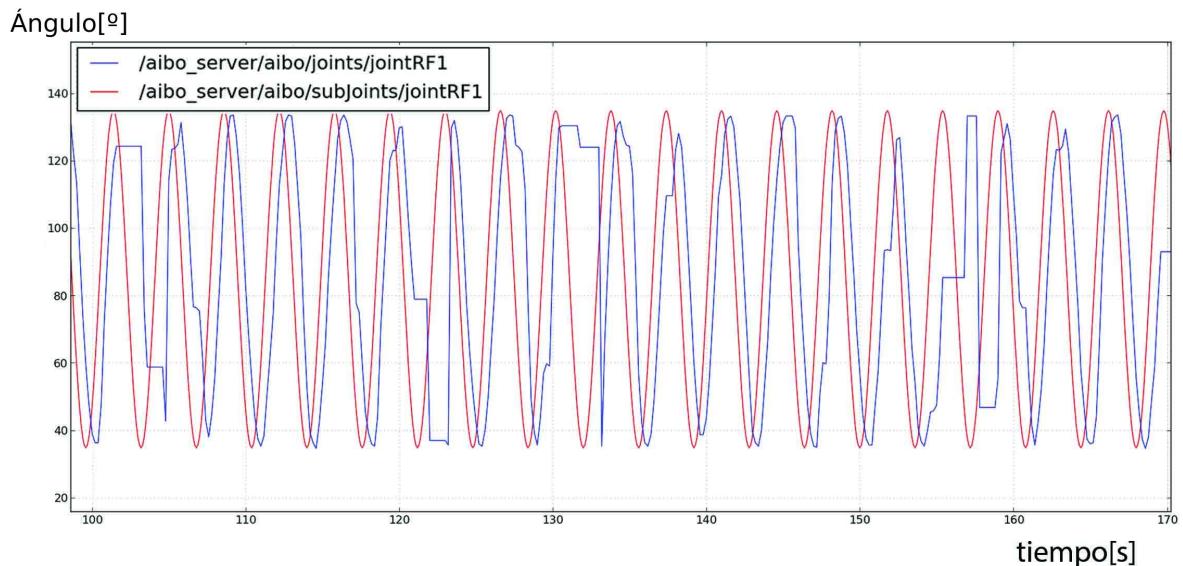


Figura 5.13: Buen seguimiento de la trayectoria.

## 5.2. 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>2</sup>, 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 del modelo de AIBO para su representación en imagen 3D y la implementación de un nodo que sincronice el modelo con el robot.

### 5.2.1. El modelo

Para generar un modelo, se debe editar un xml con formato URDF<sup>3</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

<sup>2</sup><http://wiki.ros.org/rviz>

<sup>3</sup><http://wiki.ros.org/urdf>

han descartado en el modelo la boca y las orejas. En la Figura 5.14 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.

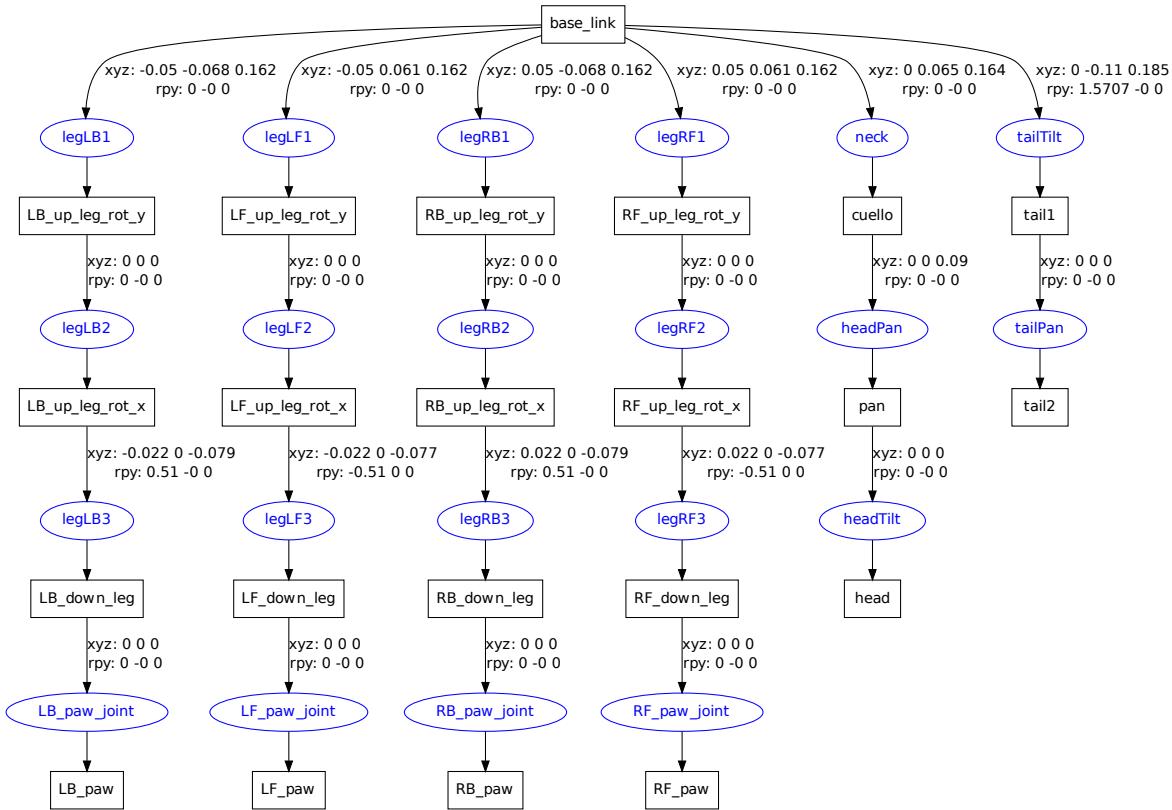


Figura 5.14: Estructura del modelo URDF.

En cada una de las partes se debe definir la geometría y sus dimensiones. Con el fin de darle un aspecto visual más atractivo y mas semejante más al AIBO real la geometría ha sido definida mediante archivos .stl que contienen las figuras de cada una de las partes. A partir de un modelo sólido se han dividido y modificado algunas piezas usando los softwares de apoyo FreeCAD<sup>4</sup> y Netfabb<sup>5</sup>. El resultado desde la ventana de rviz se muestra en la Figura 5.15.

<sup>4</sup><http://freecadweb.org/>

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

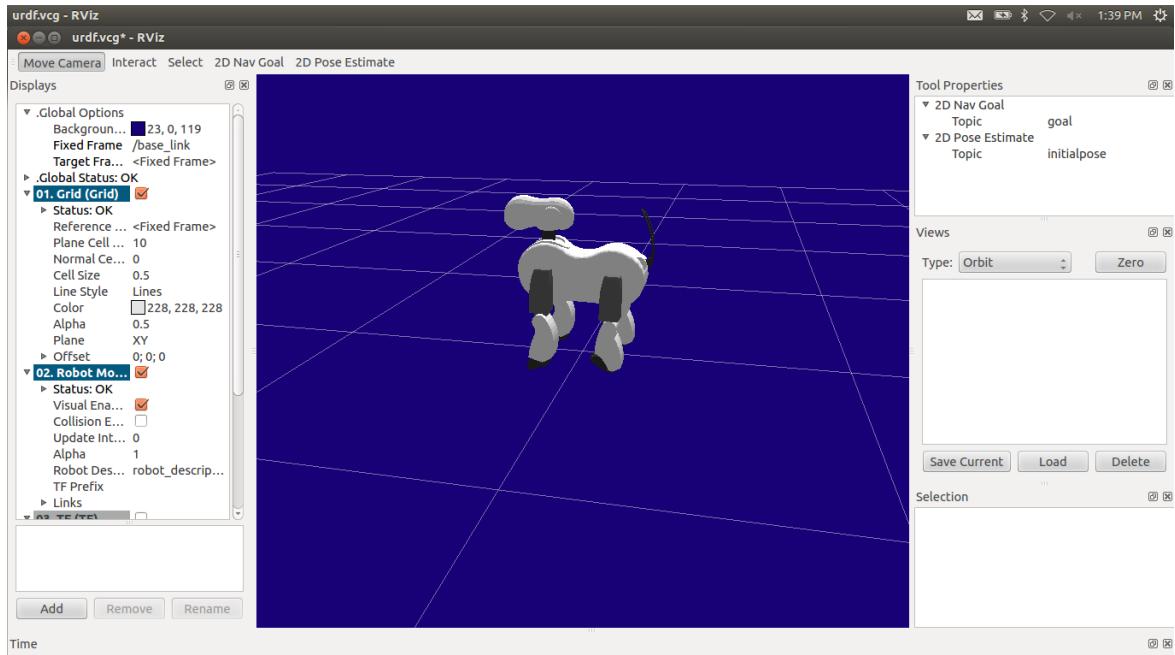


Figura 5.15: Modelo visto des de el framework de rviz.

### 5.2.2. El nodo state\_publisher

El nodo state\_publisher está creado mediante un programa escrito en C++ (`state_publisher.cpp`) y está suscrito al tópico `/aibo_server/ aibo/joints`, donde se encuentran las posiciones de las articulaciones del robot real, y las publica en un tópico llamado `joint_states` al que accede 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: Donde se publica el estado de las articulaciones.
  - odom\_trans: Donde se envía 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.

### 5.2.3. Estructura de paquete

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 5.2.2.
- 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 parámetro robot\_description el archivo Aibo.urdf.
  - Se lanza un nodo del tipo robot\_state\_publisher del paquete con el mismo nombre. Dicho nodo está suscrito al tópico joint\_states sobre el que publica el nodo implementado, state\_publisher, y se encarga de leer el URDF del 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 5.16 se puede observar cómo queda la estructura de los nodos con el visualizador rviz y el nodo aibo\_server en funcionamiento.



Figura 5.16: Estructuras de los nodos de ROS con aibo\_server y el modelo visualizado con rviz.



# Capítulo 6

## Presupuesto

El coste total del proyecto asciende a 18.200€ de los cuales el 82,42 %corresponden a gastos de personal. Los costes asociados al material solo vienen dados por el coste de la plataforma de estudio. Ya que en ser un proyecto fundamentalmente de software y todo el software usado es con licencia libre, des de el sistema operativo hasta el procesador de textos,

### 6.1. Coste material

Concepto	Coste unitario [€/unid.]	Cantidad	Coste total [€]
AIBO ERS-7	1.600	2	3.200

Tabla 6.1: Coste material detallado.

### 6.2. Coste personal

Concepto	Coste horario [€/hora]	Horas	Coste total [€]
Estudio preliminar	30	300	9.000
Diseño y depuración	30	150	4.500
Programación	15	100	1.500
<b>Total</b>		550	15.000

Tabla 6.2: Coste personal detallado.



# Capítulo 7

## Impacto ambiental

El objetivo de este proyecto conlleva como causa inmediata el incremento de la vida útil de una plataforma ya en desuso. Y en este sentido cabe entenderlo como el reciclaje de un objeto que de otro modo acabaría por desecharse en poco tiempo.

El ahorro en los costes ambientales asociados al uso de la plataforma se puede dividir en chasis, electrónica y batería.

El chasis esta hecho de ABS (Acrilonitrilo Butadieno Estireno, en inglés). Se trata de un termoplástico amorfo difícil de reciclar y en su proceso de fabricación usa acrilonitrilo, que es altamente tóxico.

Tanto la electrónica como las baterías contienen metales pesados altamente contaminantes si no son tratados correctamente.

Las baterías son de iones de litio y aunque su impacto ambiental no está del todo claro, tienen potencial de contaminar el suministro de agua subterránea debido a que contiene metales como cobalto.



# Conclusiones

Se ha alcanzado el objetivo implementando los paquetes **aibo\_server** y **aibo\_description**. Estos paquetes están disponibles para su uso o modificación con tal de complementarlos o mejorar los resultados (<https://github.com/Diamg/AiboRosPackages>). El módulo implementado permite adquirir los datos de los sensores y articulaciones del AIBO de forma satisfactoria a velocidades superiores a 10 Hz en forma de tópicos de ROS. Esta parte del módulo por si solo ya facilita enormemente la programación y sobretodo el testeo y la depuración con el uso de las herramientas que facilita ROS. Además en esta linea se ha incluido un modelo para el visualizador rviz con un aspecto visual y funcional complementando el módulo anterior. Por otro lado la parte implementada que se encarga de enviar las ordenes al AIBO permitiendo actuar sobre el no se ha dado muy buenos resultados y no permite el envío de trayectorias punto a punto con este método. De todos modos URBI tiene su propio generador de trayectorias por lo que permite enviar funciones de movimiento más complejas o posiciones concretas enviadas a un ritmo mas bajo del implementado.

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 cabe la duda de si el comportamiento de envío sería mejor usando OPEN-R, al eliminar 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, permitiendo entre otros proyectos de odometría.

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 a ayudado a mejorar las habilidades de desarrollo de software con los lenguajes C++ y python.



# Agradecimientos

Este proyecto no podría haber sido posible sin el apoyo de la familia y amigos que han sido fuente de motivación y ánimo.

Un especial agradecimiento a los tutores de éste proyecto, Manel Velasco y Cecilio Angulo tanto por su implicación y atención como por facilitar el material y el espacio de trabajo.

Por último agradecer al doctor Ricardo Tellez de quien salió la idea de este proyecto y quien le dio su primer impulso, además de su apoyo y consejo durante su realización.



# Bibliografía

- [1] Arevalillo-Herráez, Miguel and Moreno-Picot, Salvador and Millán, Vicente Caveiro, *Arquitectura para Utilizar Robots AIBO en la Enseñanza de la Inteligencia Artificial*. 2012. <http://dblp.uni-trier.de/db/journals/ieee-rita/ieee-rita6.html#Arevalillo-HerraezMM11>,
- [2] Quigley, Morgan and Conley, Ken and Gerkey, Brian P. and Faust, Josh and Foote, Tully and Leibs, Jeremy and Wheeler, Rob and Ng, Andrew Y., *ROS: an open-source Robot Operating System*. 2009.
- [3] CEA, Comité Español de Automàtica, *El libro blanco de la robótica en España*. 2011.
- [4] Sony Corporation, *OPEN-R Programmer's Guide*. 2004.
- [5] Baillie, *URBI: Towards a Universal Robotic Low-Level Programming Language*. 2005.
- [6] Xavier Perez, *Vision-based Navigation and Reinforcement Learning Path Finding for Social Robots*. 2010.
- [7] Ángel Montero Mora, Gustavo Méndez Muñoz, José Ramon Dominguez Rodriguez, *Metodologías de diseño de comportamientos para AIBO ERS7*. 2009.
- [8] RoboCup, <http://www.robocup2014.org/>
- [9] Jesus Morales, *Localización de objetos y posicionamiento en el escenario de RoboCup Four-Legged con un robot AIBO*. 2007.
- [10] Jesús Martinez Gómez, *Diseño de un teleoperador con detección de colisiones para robots AIBO*. 2006.
- [11] Ricardo A. Tellez, *Aibo Programming*. <http://www.ouroboros.org/aibo.html>
- [12] Ethan Tira-Thompson, *Tekkotsu Quick Reference*, ERS-7, Tekkotsu 3.0.

- [13] 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>
- [14] *URBI*, <http://www.urbiforge.org/index.php/Main/Robots>
- [15] Jean-Christophe Baillie, *URBI Doc for Aibo ERS2xx ERS7 Devices documentation*. 2005.
- [16] ROS new by kwc, *EusLisp now open source*. 2010. <http://www.ros.org/news/2010/07/euslisp-now-open-source.html>
- [17] Kecsap, *Porting URBI 2.x for AIBO*. 2010. <http://kecsapblog.blogspot.com.es/2010/12/porting-urbi-2x-for-aibo.html>

# Apéndice A

## Manual de usuario

### A.1. Requisitos previos

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

- Instalar Ubuntu 12.04 (Precise) de 32 bits.

<http://releases.ubuntu.com/12.04/>

- Descargar librería liburbi 1.5 para C++:

<http://www.gostai.com/downloads/urbi/1.5/>

- Descomprimir la librería, que ya está compilada, en el directorio /usr del sistema.
- Instalar de ROS.

- Preparar el ordenador para aceptar el software de ROS:

```
1 sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu precise main" > /etc/apt/sources.list.d/ros-latest.list'
```

- Preparación las claves:

```
1 wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

- Instalación:



```
1 sudo apt-get install ros-fuerte-desktop-full
```

- Preparación del entorno:

```
1 echo "source /opt/ros/fuerte/setup.bash" >> ~/.bashrc
. ~/.bashrc
```

- Crear un workspace:

```
rosdep init ~/fuerte_workspace /opt/ros/fuerte
```

```
1 sudo apt-get install python-rosinstall
```

- Crear un sandbox donde irán los nuevos paquetes y añadir al ROS\_PACKAGE\_PATH:

```
1 mkdir ~/fuerte_workspace/sandbox
2 rosws set ~/fuerte_workspace/sandbox
3 echo "source ~/fuerte_workspace/setup.bash" >> ~/.bashrc
. ~/.bashrc
```

- Confirmación de la correcta instalación del entorno:

Al ejecutar:

```
1 echo $ROS_PACKAGE_PATH
```

Se debe obtener algo similar a:

```
1 /home/your_user_name/fuerte_workspace/sandbox:/opt/ros/fuerte/
share:/opt/ros/fuerte/stacks
```

- Asegurarse que los siguientes paquetes de ROS han sido instalados:

- rviz:

```
1 rosdep install --from-paths src --ignore-src -r -y
```

- urdf

```
1 rosdep install --from-paths src --ignore-src -r -y
```

- robot\_state\_publisher

```
1 rosdep install --from-paths src --ignore-src -r -y
```

- tf

```
1 rosdep install --from-paths src --ignore-src -r -y
```

Deben devolver la ubicación del paquete.

- Descargar los paquetes:

- aibo\_server:

[https://github.com/Diamg/AiboRosPackages/tree/master/aibo\\_server](https://github.com/Diamg/AiboRosPackages/tree/master/aibo_server)

- aibo\_description

[https://github.com/Diamg/AiboRosPackages/tree/master/aibo\\_description](https://github.com/Diamg/AiboRosPackages/tree/master/aibo_description)

- Copiar la tarjeta de memoria, Figura A.1 para el AIBO con URBI:

[https://github.com/Diamg/AiboRosPackages/tree/master/MS\\_URBI](https://github.com/Diamg/AiboRosPackages/tree/master/MS_URBI)



Figura A.1: Tarjeta de memoria para AIBO.

- Configuración de la conexión wireless con el AIBO.

Editar el archivo WLANDFLT.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 traves 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.
- APMode: 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

WEPPENABLE = 1

WEPKEY = AIBO2

APMODE = 2

CHANNEL = 3

## A.2. Instalación

- Guardar los paquetes en el workspace de ROS.
- Ejecutar en un terminal:

```
1 rosmake --pre-clean aibo_server  
rosmake --pre-clean aibo_description
```

## A.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  
(La frecuencia por defecto y recomendada son 5 Hz).

### A.3.1. Tópicos

El nodo aibo\_server permite suscribirse a los siguientes tópicos:

- **/aibo\_server/aibo/infrared:**

Da la información de los tres infrarrojos del AIBO guardada en forma de mensaje de tipo aibo\_server/IRArrays.

- **/aibo\_server/aibo/touch:**

Da la información de los sensores de presión de la barbilla, la cabeza y los tres de la espalda en un mensaje de tipo aibo\_server/TouchArray.

- **/aibo\_server/aibo/accel:**

Da la información de los acelerometros en un mensaje de tipo aibo\_server/Accel.

- **/aibo\_server/aibo/joints:**

Da la información de las posiciones de las articulaciones en un mensaje de tipo aibo\_server/Joints.

- **/aibo\_server/aibo/paws:**

Da la información de los sensores de contacto de las tres patas en forma de mensaje del tipo aibo\_server/BumperArray.

Y publicar sobre el tópico:

- **/aibo\_server/aibo/subJoints:**

Permite controlar la posición de las articulaciones en un mensaje de tipo aibo\_server/Joints.



### A.3.2. Mensajes

- aibo\_server/IRArrays

```

1 Header header
sensor_msgs/Range[] IRS

```

El array de mensajes de tipo std\_msgs/Range esta ordenado:

Sensor	Máximo	Mínimo	Unidades
Pecho	90	19	[cm]
Morro de corto alcance	50	5.7	[cm]
Morro de largo alcance	20	150	[cm]

- sensor\_msgs/Range

```

Header header
2 uint8 ULTRASOUND=0
uint8 INFRARED=1
4 uint8 radiation_type
float32 field_of_view
6 float32 min_range
float32 max_range
8 min_range==max_range
float32 range

```

Se trata de un mensaje estándar de ROS<sup>1</sup>

- aibo\_server/TouchArray

```

Header header
2 float64[] touch

```

Los sensores están ordenados en la array de la siguiente forma:

---

<sup>1</sup>[http://docs.ros.org/api/sensor\\_msgs/html/msg/Range.html](http://docs.ros.org/api/sensor_msgs/html/msg/Range.html)

Sensor	Máximo	Mínimo	Unidades
Barbilla	0	1	[booleano]
Espalda delantero	60	0	[ $\mu\text{Pa}$ ]
Espalda del medio	60	0	[ $\mu\text{Pa}$ ]
Espalda trasero	60	0	[ $\mu\text{Pa}$ ]
Cabeza	35	0	[ $\mu\text{Pa}$ ]

■ aibo\_server/Accel:

```

1 Header header
2   float64 x
3   float64 y
4   float64 z

```

- x: Valor del acelerometro en el eje x.
- y: Valor del acelerometro en el eje y.
- z: Valor del acelerometro en el eje z.

Sensor	Máximo	Mínimo	Unidades
x: Valor del acelerometro en el eje x.	-19,613	19,613	[ $m/s^2$ ]
y: Valor del acelerometro en el eje y.	-19,613	19,613	[ $m/s^2$ ]
z: Valor del acelerometro en el eje z.	-19,613	19,613	[ $m/s^2$ ]

■ aibo\_server/Joints

```

1 Header header
2   float64 jointLF1
3   float64 jointLF2
4   float64 jointLF3
5   float64 jointLH1
6   float64 jointLH2
7   float64 jointLH3
8   float64 jointRF1
9   float64 jointRF2
10  float64 jointRF3

```

```
12   float64 jointRH1
    float64 jointRH2
    float64 jointRH3
14   float64 tailPan
    float64 tailTilt
16   float64 headTilt
    float64 headPan
18   float64 headNeck
    float64 mouth
```

Donde:

- jointLF1: Articulación de rotación del hombro del de la pata izquierda delantera.
- jointLF2: Articulación de apertura del hombro de la pata izquierda delantera.
- jointLF3: Articulación del codo de la pata izquierda delantera.
- jointLH1: Articulación de rotación del hombro del de la pata izquierda trasera.
- jointLH2: Articulación de apertura del hombro de la pata izquierda trasera.
- jointLH3: Articulación del codo de la pata izquierda delantera.
- jointRF1: Articulación de rotación del hombro del de la pata derecha delantera.
- jointRF2: Articulación de apertura del hombro de la pata derecha delantera.
- jointRF3: Articulación del codo de la pata derecha delantera.
- jointRH1: Articulación de rotación del hombro del de la pata derecha trasera.
- jointRH2: Articulación de apertura del hombro de la pata derecha trasera.
- jointRH3: Articulación del codo de la pata derecha trasera.
- tailPan: Articulación de la cola para el movimiento lateral.
- tailTilt: Articulación de la cola para el movimiento vertical.
- headTilt: Articulación de la cabeza para el movimiento vertical.
- headPan: Articulación de la cabeza para el movimiento lateral.
- headNeck: Articulación del cuello.
- mouth: articulación de la boca.

<b>Sensor</b>	<b>Máximo</b>	<b>Mínimo</b>	<b>Unidades</b>
jointLF1	134	-120	[°]
jointLF2	91	-9	[°]
jointLF3	119	-29	[°]
jointLH1	134	-120	[°]
jointLH2	91	-9	[°]
jointLH3	119	-29	[°]
jointRF1	120	-134	[°]
jointRF2	91	-9	[°]
jointRF3	119	-29	[°]
jointRH1	120	-134	[°]
jointRH2	91	-9	[°]
jointRH3	119	-29	[°]
tailPan	59	-59	[°]
tailTilt	63	2	[°]
headTilt	44	-16	[°]
headPan	91	-91	[°]
headNeck	2	-79	[°]
mouth	-3	-58	[°]

- aibo\_server/BumperArray

```
Header header
2 float64 [] paws
```

La array de paws esta ordenada con el siguiente orden:

<b>Sensor</b>	<b>Máximo</b>	<b>Mínimo</b>	<b>Unidades</b>
Delantero izquierdo	1	0	[booleano]
Trasero izquierdo	1	0	[booleano]
Delantero derecho	1	0	[booleano]
Trasero derecho	1	0	[booleano]

## A.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.



# Apéndice B

## Scripts y programas

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

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

9 std :: ofstream myfile;

11 //adquisicion del tiempo
void getTime(double& tim)
13 {
    struct timespec tsp;
15    clock_gettime(CLOCK_REALTIME, &tsp);
    tim = (double)(tsp . tv_sec+ tsp . tv_nsec*0.000000001);
17    return ;
}
19 //escritura en archivo
void saveData( double tim ,double value){
21    myfile << tim ;
    myfile << "\t";
23    myfile << value ;
    myfile << "\n";
25    return ;
```

```

}
27
//callback de URBI
29 urbi :: UCallbackAction onJointSensor( const urbi :: UMessage &msg)
{
31     double value = (double)msg.value->val;
32     if (msg.tag==”legRF1”)
33         double tim;
34         getTime(tim);
35         saveData(tim , value);
36         return urbi :: URBI_CONTINUE;
37     }

39
int main( int argc , char** argv )
40 {
    //inicializacion del cliente URBI
41     urbi::UClient* client = new urbi::UClient(argv[1] , 54000);
    //inicializacion del callback
43     client->setCallback ( urbi::callback(onJointSensor) ,”legRF1” );
    //demanda del bucle de envio
45     client->send(”loop legRF1 << legRF1.val , ” );
    //abre el archivo de escritura
47     myfile.open (”Data1.txt”);
    myfile.precision(15);
    //bucle de URBI
49     urbi::execute();
    //cierra el archivo de escritura
50     myfile.close();
51     return (0);
52 }

```

src/getDataOneLeg/getDataC++/getData.cpp

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

```

import telnetlib
2 import time

4 def tractarData(data):

```



```

    datos=[]
6   # lectura hasta la marca
7   while (data.find("555555")<0):
8       data=data+te.read_some()
# descarta la informacion de la bateria y divide la cadena
10  if (data.find("Bat")<0):
11      dataNow=data.split("555555")[0]
12      data=data.split("555555")[1]
13      dataArray=dataNow.split("\n")
14      del dataArray[0]
15      leg=0
16      for i in range(len(dataArray)):
17          if (dataArray[i].find("tag0")>0):
18              leg=float(dataArray[i].split("]")[1])
19              return leg, data
20      else:
21          data=""
22      return 0, data

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

28 # inicializacion cliente telnet
29 te=telnetlib.Telnet("192.168.0.125",54000)
30 time.sleep(1)
31 data=1
32 # eliminacion de la cabecera URBI
33 while data:
34     data=te.read_eager()
35     print data
36 # demanda del bucle de envio
37 te.write("motors on;\r\n")
38 te.write("motors.load=0;\r\n")
39 te.write("loop tag0 << legLF1.val, loop tag19 << 5*111111;")
40 # primera lectura y tratamiento de la cadena
41 dataArray, data=tractarData(data)
42 # abre el archivo de escritura
43 f = open("Data.txt", "w")
44 #bucle de tratamiento y escritura
45 while True:
46     leg, data =tractarData(data)
        t= str(leg) + "\t"

```

```

48     writeData(f, t)
49     sec = "{0:.15f}".format(time.time()) + "\n"
50     writeData(f, sec)
51     print sec
52 # cierra el archibo de escritura
53 f.close()

```

src/getDataOneLeg/getDataPython/getData.py

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

```

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

13
14 std::ofstream f;
15 std::ofstream f2;
16 float angle=0;

17
18 urbi::UClient* client;
19 urbi::UClient* client2;

21 //guarda el tiempo
22 void getTime(double& tim)
23 {
24     struct timespec tsp;
25     clock_gettime(CLOCK_REALTIME, &tsp);
26     tim = (double)(tsp.tv_sec+ tsp.tv_nsec*0.000000001);
27     return;
28 }
29 //escribe los datos de salida
30 void saveData( double tim, double value , std::string tag){

```



```
31     char command[100];
32     sprintf (command,"%f \t %f",tim , value);
33     f << command << std :: endl;
34
35     return ;
36 }
37 //escribe los datos de entrada
38 void saveData2( double tim ,double value , std :: string tag){
39
40     char command[100];
41     sprintf (command,"%f \t %f",tim , value);
42     f2 << command << std :: endl;
43
44     return ;
45 }
46 //Bucle de envio
47 void *move( void *)
48 {
49     while(true){
50
51         float l=50*sin (angle*2*M_PI/360) + 85;
52         char command[100];
53         sprintf (command,"legLF1 . val=%f time: 100 ,",l );
54         client2->send(command);
55         angle+=10;
56         double tim;
57         getTime(tim);
58
59         saveData2(tim,l , "LF1");
60         usleep(100000);
61
62     }
63     return NULL;
64 }
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);
```

```

    saveData( tim , value , msg.tag ) ;

75
}

77     return urbi::URBLCONTINUE;
79 }

int main( int argc , char** argv )
{
    //inicia clientes URBI
83     client = new urbi::UClient(argv[1] , 54000);
84     client2 = new urbi::UClient(argv[1] , 54000);
    //Inicia callback de URBI para cada tag
85     client->setCallback( urbi::callback(onJointSensor) , "legRF1" );
86     client->setCallback( urbi::callback(onJointSensor) , "legRF2" );
87     client->setCallback( urbi::callback(onJointSensor) , "legRF3" );
88     client->setCallback( urbi::callback(onJointSensor) , "legRH1" );
89     client->setCallback( urbi::callback(onJointSensor) , "legRH2" );
90     client->setCallback( urbi::callback(onJointSensor) , "legRH3" );
91     client->setCallback( urbi::callback(onJointSensor) , "legLF1" );
92     client->setCallback( urbi::callback(onJointSensor) , "legLF2" );
93     client->setCallback( urbi::callback(onJointSensor) , "legLF3" );
94     client->setCallback( urbi::callback(onJointSensor) , "legLH1" );
95     client->setCallback( urbi::callback(onJointSensor) , "legLH2" );
96     client->setCallback( urbi::callback(onJointSensor) , "legLH3" );
97     client->setCallback( urbi::callback(onJointSensor) , "mouth" );
98     client->setCallback( urbi::callback(onJointSensor) , "headNeck" );
99     client->setCallback( urbi::callback(onJointSensor) , "headPan" );
100    client->setCallback( urbi::callback(onJointSensor) , "headTilt" );
101    client->setCallback( urbi::callback(onJointSensor) , "tailTilt" );
102    client->setCallback( urbi::callback(onJointSensor) , "tailPan" );
103    client2->send("motors on;" );
104
105    //define el modo de resolver conflictos
106    client2->send("legLF1.val->blend = cancel;" );
107
108    //demanda del loop
109    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;" );
110
111    //Abre el archivo de datos de salida

```

```

111 f . open ( "DataOut . txt" );
112 f . precision ( 15 );
113 //Abre el archivo de datos de entrada
114 f2 . open ( "DataIn . txt" );
115 f2 . precision ( 15 );
116 // crea thread para el envio
117 pthread_t t1;
118 pthread_create (&t1 , NULL, &move ,NULL);
119 pthread_join (t1 ,NULL);
120 //Bucle de URBI
121 urbi :: execute ();
122 //cierra archivos
123 f . close ();
124 f2 . close ();
125
126 return (0);
127 }
```

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

## B.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
8 angle=0

9
10 #funcion de envio
11 def move():
12     global te1
13     global angle
14     global f2
15     A=50
16     offset1=85
17     paso=15
18     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)
threading.Timer(0.2,move).start()
25 #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):

```

```

61     Joint . val . append( float( dataArray [ i ] . split( " ] " ) [ 1 ] ) )
62     if ( dataArray [ i ] . find( " tag10 " ) > 0 ):
63         Joint . val . append( float( dataArray [ i ] . split( " ] " ) [ 1 ] ) )
64     if ( dataArray [ i ] . find( " tag11 " ) > 0 ):
65         Joint . val . append( float( dataArray [ i ] . split( " ] " ) [ 1 ] ) )
66     if ( dataArray [ i ] . find( " tag12 " ) > 0 ):
67         Joint . val . append( float( dataArray [ i ] . split( " ] " ) [ 1 ] ) )
68     if ( dataArray [ i ] . find( " tag13 " ) > 0 ):
69         Joint . val . append( float( dataArray [ i ] . split( " ] " ) [ 1 ] ) )
70     if ( dataArray [ i ] . find( " tag14 " ) > 0 ):
71         Joint . val . append( float( dataArray [ i ] . split( " ] " ) [ 1 ] ) )
72     if ( dataArray [ i ] . find( " tag15 " ) > 0 ):
73         Joint . val . append( float( dataArray [ i ] . split( " ] " ) [ 1 ] ) )
74     if ( dataArray [ i ] . find( " tag16 " ) > 0 ):
75         Joint . val . append( float( dataArray [ i ] . split( " ] " ) [ 1 ] ) )
76     if ( dataArray [ i ] . find( " tag17 " ) > 0 ):
77         Joint . val . append( float( dataArray [ i ] . split( " ] " ) [ 1 ] ) )
78     if ( dataArray [ i ] . find( " tag18 " ) > 0 ):
79         Joint . val . append( float( dataArray [ i ] . split( " ] " ) [ 1 ] ) )

80
81     return Joint , data
82 else :
83     data= ""
84     return 0 , data
85
86 #escribe en el archivo
87 def writeData(f , text):
88
89     f . write( text )
90
91 #inicia cliente telnet de envio
92 te=telnetlib.Telnet("192.168.0.124",54000)
93 te . write("motors on;\r\n")
94 #selecciona modo de resolver conflictos
95 te . write("legLF1.val->blend = cancel;\r\n")
96 #inicia cliente telnet de recepcion
97 te=telnetlib.Telnet("192.168.0.124",54000)
98 te . write("motors on;\r\n")
99 time . sleep (3)
100 data=1
101 #elimina la cabecera de URBI de la cadena
102 while data:
103     data=te . read_eager ()

```

```

    print data
105 #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"
121     writeData(f ,t)
    t= str(joint.val[0]) + "\n"
123     writeData(f ,t)
#cierra archivos
125 f . close ()
f2 . close ()

```

src//sinLegRead/sinLegPython/sinlegLF1.py

## B.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
11 //publica 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     //loop de ROS
        while (ros::ok())
```

```

50    {
51        float l=50*sin( angle*2*M_PI/360) + 85;
52        angle+=10;
53        joi.jointRF1=l;
54        publishJoint();
55        r.sleep();
56    }
57    return(0);
58}

```

src/ROStests/sinlegROS.cpp

## B.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 {
14     joi.jointRF1=msg->jointRF1;
15     joi.jointRF2=msg->jointRF2;
16     joi.jointRF3=msg->jointRF3;
17     joi.jointRH1=msg->jointRH1;
18     joi.jointRH2=msg->jointRH2;
19     joi.jointRH3=msg->jointRH3;
20     joi.jointLF1=msg->jointLF1;
21     joi.jointLF2=msg->jointLF2;
22     joi.jointLF3=msg->jointLF3;
23     joi.jointLH1=msg->jointLH1;
24     joi.jointLH2=msg->jointLH2;
25     joi.jointLH3=msg->jointLH3;
26     joi.headPan=msg->headPan;
27     joi.headNeck=msg->headNeck;

```



```
29     joi.headTilt=msg->headTilt;
30     joi.mouth=msg->mouth;
31     joi.tailTilt=msg->tailTilt;
32     joi.tailPan=msg->tailPan;

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

38 int main(int argc, char** argv)
39 {
40     //inicia nodo de ROS
41     ros::init(argc, argv, "aibo_tutorials");
42     ros::NodeHandle n;
43     //define frecuencia del loop de ROS
44     ros::Rate r(10);

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

src/ROStests/mimic.cpp