TRABAJO FIN DE MÁSTER
MÁSTER DE AUTOMÁTICA E INFORMÁTICA INDUSTRIAL

SERVIDOR *OPC-UA* SOBRE *BEAGLEBONE BLACK*

Autor: Villarroya Alfonso, Benjamín

Tutor: Blanes Noguera, Juan Francisco

Valencia, Septiembre 2015

RESUMEN:

Servidor *OPC-UA* sobre *BeagleBone Black*

Autor: Villarroya Alfonso, Benjamín

Tutor: Blanes Noguera, Juan Francisco

En este trabajo fin de máster se aborda la implementación de un servidor de datos basado en el protocolo *OPC-UA* sobre un sistema empotrado, en su caso una placa *BeagleBone Black*.

Concretamente el trabajo desarrollará una capa de *software* implementada en *Java.* Para permitir que el servidor *OPC-UA* acceda a los periféricos de E/S de los que dispone el *hardware* del sistema formado por una placa *BeagleBone Black Rev-C*, sobre la que se ha conectado un módulo *CAPE* específico denominado *ROBOcape*, siendo en su conjunto, un *hardware* económico y de prestaciones limitadas. Por otra parte, este trabajo evaluará sus posibilidades para la ejecución de servicios vinculados al protocolo *OPC-UA*.

Para la implementación de de la capa de *software*, se realizará las modificaciones y ampliaciones de código sobre una versión de prueba de la implementación *Java* del servidor de la empresa *Prosys*. Planteándose también como objetivo del proyecto el validar la funcionalidad de dicho servidor mediante el acceso desde uno o varios clientes *OPC-UA* conectados desde una máquina remota.

Finalmente, además de desarrollar en el trabajo los aspectos propios del protocolo en términos de la supervisión y la actuación, se dotará al servidor de la capacidad de regular procesos mediante la implementación de un controlador básico que se vincula mediante un bucle de control de entradas y salidas del sistema *hardware.*

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

CAPÍTULO 1: INTRODUCCIÓN Y OBJETIVOS

1.1 OBJETIVOS DEL TRABAJO

El objetivo principal de este trabajo, el implementar una capa de *software* programada en *Java*, que permita la interacción con un servidor *OPC-UA* (versión en *Java SDK de Prosys*) que proporciona acceso mediante dicho protocolo a sensores y actuadores de un sistema empotrado basado en *BeagleBone Black*. Y de esta forma, hacer que esta información sea accesible desde un cliente remoto para el desarrollo de sistemas de supervisión y control de procesos. Ahora bien, los objetivos específicos son:

- Desarrollo de un sistema de configuración del árbol de *tags* (nodos) del servidor mediante fichero de configuración.
- Desarrollo de una capa de acceso al hardware *BeagleBone Black* + *ROBOcape.*
- Desarrollo de la interfaz de acceso desde el servidor a sensores y actuadores conectados a la *BeagleBone Black.*
- Validación y test del sistema mediante clientes *OPC-UA.*

Para el desarrollo de estos objetivos se hará uso de una placa *BeagleBone Black* y un módulo de expansión *ROBOcape*. La combinación de estos dos elementos es la que forma el sistema empotrado sobre el que se ejecuta el servidor, siendo la *BeagleBone Black* la encargada de la ejecución y la *ROBOcape* la que le proporciona la capa *hardware* necesaria para acceder a los elementos que se citan a continuación:

- Supervisión de entradas analógicas para sensores *Sharp*.
- Supervisión de Acelerómetro, giroscopio y magnetómetro mediante el bus *I2C*.
- Actuación sobre el ciclo de trabajo de salidas *PWM*.
- Actuación sobre el ángulo de servos.
- Implementación de un control electrónico sencillo.

En lo que a la capa de *software* respecta, ésta debe dotar al servidor de los métodos necesarios para acceder a cada uno de los elementos de los que dispone la *ROBOcape* y poder ofrecer el acceso a los mismos desde uno o varios clientes mediante la creación de nodos de información.

1.2 INTRODUCCIÓN A LOS ELEMENTOS PRINCIPALES

Dado que el presente trabajo fin de máster, trata de la ejecución de un servidor OPC-UA sobre un sistema empotrado, introduciendo las modificaciones necesarias para cumplir con los objetivos antes citados, es necesario hacer un inciso en que es un sistema empotrado y un servidor.

1.2.1 DEFINICIÓN DE SISTEMA EMBEBIDO

Un sistema embebido ("*embedded*") o empotrado es un sistema electrónico diseñado para realizar, de forma óptima, una o varias funciones concretas, en muchos casos dentro de un sistema mayor compuesto por más componentes y del cual éste forma parte. Por tanto, el enfoque de estos sistemas es muy distinto al de los ordenadores de propósito general, que están diseñados para cubrir un amplio rango de necesidades.

Hoy en día estos dispositivos se encuentran en casi todos los elementos de uso cotidiano. Desde la electrónica de un reloj digital, televisiones, una lavadora, etc. Hasta sistemas mucho más complejos, como son la industria del automóvil y la aviónica, que requieren el uso de sistemas más avanzados y que trabajen en conjunto de una forma óptima. Como se puede observar en la figura 1.

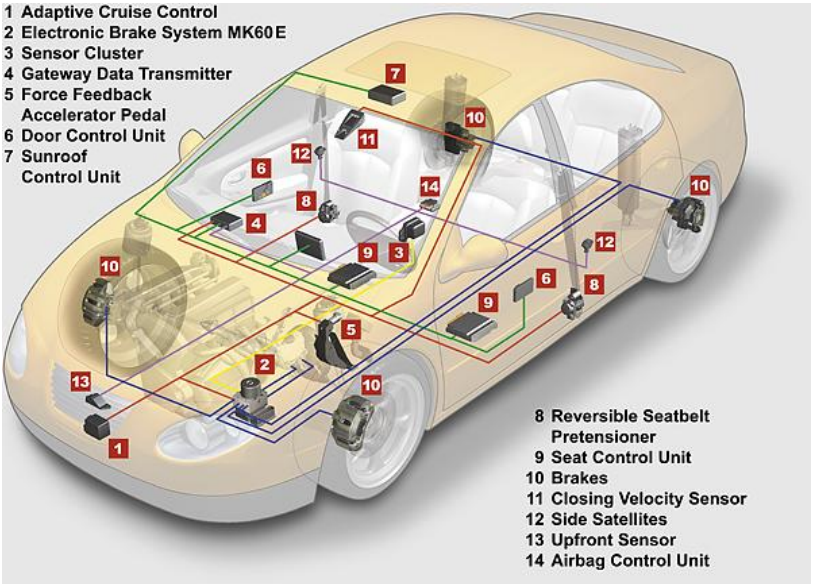

Figura 1: Sistemas empotrados dentro de un automóvil

A nivel de *hardware*, en un sistema embebido la mayoría de sus componentes se encuentran integrados en su placa base. Siendo su núcleo y elemento principal una o varias *CPU*, que es la que se encarga de dotar de "inteligencia" al sistema al que ayuda a gobernar. Lo más común es que se trate de uno de estos tipos:

- Microprocesador.
- Microcontrolador de 4, 8, 16 o 32 bits.
- *DSP* de punto fijo o flotante.
- *FPGA*.

Por lo general, los sistemas embebidos suelen ser diseñados con unas directrices, a nivel de hardware, que deben cumplir dependiendo de la aplicación a la que estén destinados. Las más importantes son:

- Tamaño reducido.
- Márgenes de temperatura de funcionamiento, en función de la aplicación:
  - Gran consumo (0ºC hasta 70ºC)
  - Industrial y automoción. Márgenes de temperatura hasta 125ºC
  - Aeroespacial
  - Militar
  - Medicina
- Consumo de energía. (Sobre todo en sistemas que necesiten de baterías)
- Robustez mecánica. (Vibraciones, golpes, humedad, ambientes corrosivos)
- Coste.

A nivel de *software*, según la aplicación se tendrá una serie de limitaciones o restricciones. Ya que, no dispone de recursos ilimitados sino que la cantidad de memoria suele ser escasa y la capacidad de cálculo limitada en comparación con sistemas de propósito general.

Los sistemas embebidos se pueden programar directamente en el lenguaje ensamblador o también, utilizando los compiladores específicos, pueden utilizarse lenguajes de alto nivel como *C* o *C++*.Y en algunos casos, cuando el tiempo de respuesta de la aplicación no es un factor crítico, pueden usarse lenguajes interpretados como *JAVA*.

En los últimos años, muchos sistemas empotrados incluyen un sistema operativo adaptado para sistemas embebidos, sobre el cual corre el software de control.

1.2.2 Definición de servidor

Un servidor es una aplicación en ejecución que es capaz de atender las peticiones realizadas por uno o varios clientes y a su vez, devolverle una respuesta acorde a la petición realizada.

En muchos casos los servidores corren sobre computadoras dedicadas, a las cuales se les considera como propios servidores. Pero en realidad, los servidores pueden ser ejecutados desde cualquier máquina con la capacidad de cómputo necesaria. Siendo posible, en la actualidad, que una misma computadora se encargue tener varios servidores en funcionamiento y de a su vez, de proveer otros servicios adicionales. Ahora bien, existe una gran ventaja de ejecutar los servidores en máquina dedicadas, la seguridad.

Los servidores se valen de una arquitectura cliente-servidor como la que se muestra en la figura 2. Es decir, los servidores son programas que se encargan de atender a las peticiones de los clientes. Por lo que, las tareas que realiza el servidor, están orientadas a solventar las peticiones que realizan estos clientes, permitiéndoles compartir datos, información y recursos de *hardware* y *software*.

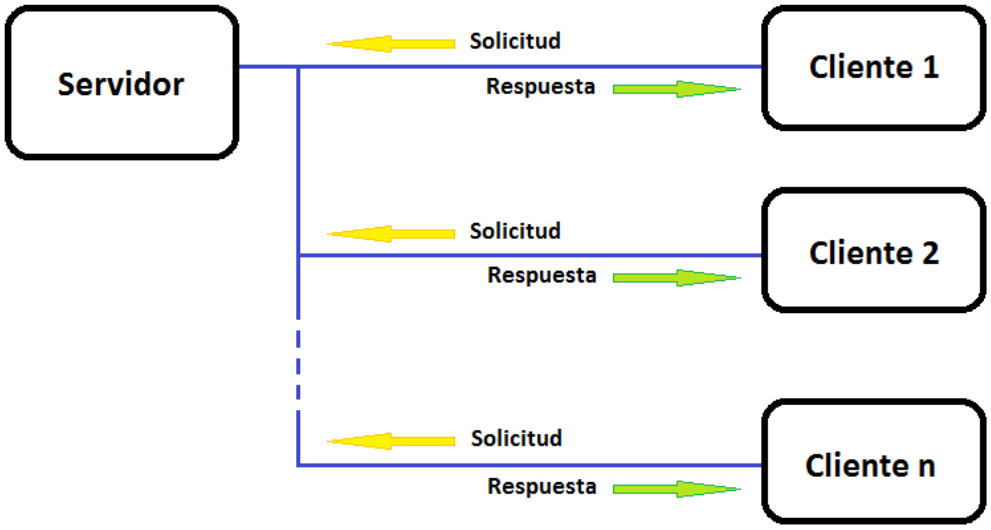

Figura 2: Modelo Cliente-Servidor.

Los tipos de servidores más comunes son: servidor de base de datos, servidor de archivos, servidor de correo, servidor de impresión, servidor web, servidor de juego y servidor de aplicaciones.

CAPÍTULO 2: HARDWARE UTILIZADO

Para la realización del trabajo fin de máster se ha utilizado diversos elementos de *hardware* que permitiesen la ejecución del servidor *OPC-UA* y que permitiesen el cumplimiento de los objetivos citados con anterioridad. Es el caso de la placa *BeagleBone Black* y del módulo de expansión de la misma la *ROBOcape*. Además, se ha utilizado un PC de propósito general para realizar la programación del código necesario y para comunicarse con la placa.

2.1 *BEAGLEBONE BLACK*

La placa *BeagleBone Black* es un dispositivo de bajo coste comercializado en 2013 como revisión de la anterior *BeagleBone* y de las primeras *BeagleBoards*. Todas ellas surgidas como plataformas de desarrollo de código abierto ofreciendo un amplio soporte en la comunidad de internet.

La primera *BeagleBoard* fue desarrollada por *Texas Instruments* en asociación con *Newmark element14* y *Digi-Key*. Con la finalidad de ser utilizada como plataforma educacional en asignaturas de programación. El primer modelo que se comercializó fue la *BeagleBoard-xM*, que fue lanzada en 2010 y tenía un tamaño de 82,55 mm x 82,55 mm. Este modelo, implementaba un procesador de 1 GHz y disponía de conexión para *Ethernet*, cuatro puertos *USB*, puerto para el conexionado de tarjetas *SD*, conectores de *audio Jack*, puerto RS-232, conexión HDMI, entre otros periféricos. Estas características le dotaban de una gran versatilidad. El aspecto de la placa se muestra en la figura 3.

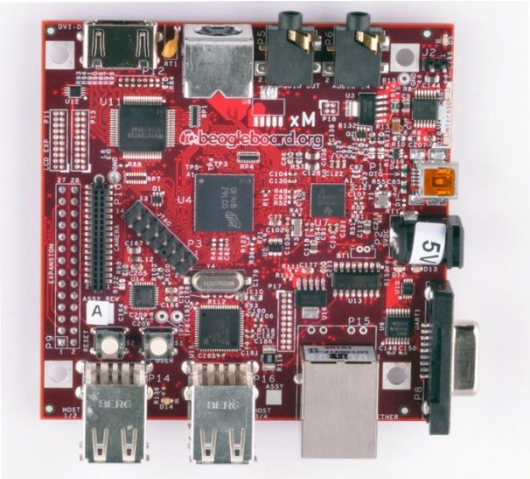

Figura 3: Aspecto de la *BeagleBoard-xM*

Fue la salida de la *Beaglebone* en 2011, la que impuso la línea de diseño que luego se seguiría con la *BeagleBone Black*. La principal característica de la *BeagleBone*, era que su núcleo era un procesador *ARM Cortex – A8 Sitara* que funcionaba a una frecuencia de reloj de 720 MHz. Además de que estaba diseñada de forma que se pudiesen conectar capas de ampliación mediante los conectores de los que dispone a ambos lados de la misma. Los cuales se pueden observar en la parte izquierda de la figura 4, mientras que en la parte derecha se puede observar un ejemplo de módulo de expansión *HDMI*.

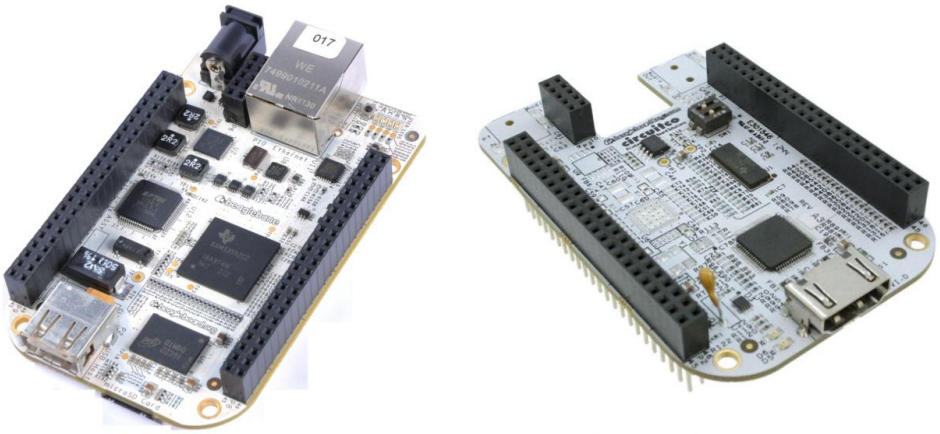

Figura 4: Aspecto de la *BeagleBone* y del módulo de expansión *HDMI*

Ahora bien, la *Beaglebone Black* nació como mejora de la *BeagleBone*. Incorporando múltiples mejoras respecto a su predecesora: Ampliando la velocidad del procesador, su memoria *RAM*, la memoria *flash eMMC* y añadiendo un puerto *HDMI* para la salida de vídeo y audio. Siendo su aspecto y distribución de los elementos principales los de la figura 5.

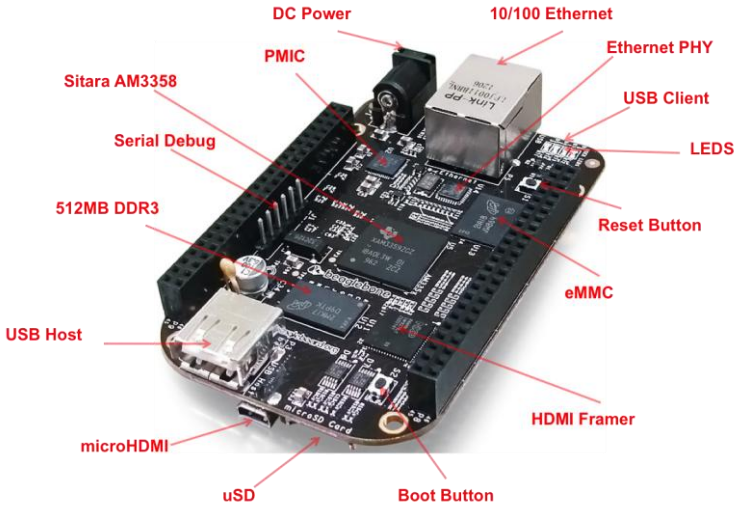

Figura 5: Aspecto de la placa *BeagleBone Black.*

2.1.1 CARACTERÍSTICAS TÉCNICAS DE LA *BEAGLEBONE BLACK*

Las características más importantes que incorpora la placa *BeagleBone Black* [2], son las enumeradas a continuación, que han sido extraídas de las especificaciones que se muestran en la tabla 1:

- Procesador: *AM335x 1GHz ARM® Cortex-A8*
  - *512MB* de memoria *RAM DDR3*.
  - *4GB 8-bit eMMC* de almacenamiento interno.
  - Acelerador gráfico 3D.
  - Acelerador de punto flotante *NEON*.
  - 2 PRU de 32-bit. (Unidades de procesado en tiempo real)

- Conectividad
  - Cliente para alimentación y comunicaciones *USB*.
  - *USB host*.
  - Conector *Ethernet*.
  - Puerto *HDMI*.
  - 2 Conectores de 46 pines.

- Compatibilidad de *software*
  - *Debian.*
  - *Android.*
  - *Ubuntu.*
  - Y muchos más.

Se ha escogido esta placa debido a la elevada capacidad de cómputo que posee, junto con las múltiples posibilidades que ofrece a la hora de expandir su funcionalidad mediante capas de *hardware*. Ya que es posible utilizarla para multitud de funciones, como se hace en este trabajo, ya sea como sistema empotrado de adquisición de datos o como sistema de control.

El sistema operativo que incorpora de serie es la revisión 3.8 de *Debian*. La cual se ha utilizado durante el trabajo, ya que este es totalmente funcional a la hora de manejar el código y las funciones necesarias para su realización.

	Feature	
Processor	Sitara AM3358BZCZ100 1GHz, 2000 MIPS	
Graphics Engine	SGX530 3D, 20M Polygons/S	
SDRAM Memory	512MB DDR3L 800MHZ	
Onboard Flash	4GB, 8bit Embedded MMC	
PMIC	TPS65217C PMIC regulator and one additional LDO.	
Debug Support	Optional Onboard 20-pin CTI JTAG, Serial Header	
Power Source	miniUSB USB or DC Jack	5VDC External Via Expansion Header
PCB	3.4" x 2.1"	6 layers
Indicators	1-Power, 2-Ethernet, 4-User Controllable LEDs	
HS USB 2.0 Client Port	Access to USB0, Client mode via miniUSB	
HS USB 2.0 Host Port	Access to USB1, Type A Socket, 500mA LS/FS/HS	
Serial Port	UART0 access via 6 pin 3.3V TTL Header. Header is populated	
Ethernet	10/100, RJ45	
SD/MMC Connector	microSD , 3.3V	
User Input	Reset Button Boot Button Power Button	
Video Out	16b HDMI, 1280x1024 (MAX) 1024x768,1280x720,1440x900 ,1920x1080@24Hz w/EDID Support	
Audio	Via HDMI Interface, Stereo	
Expansion Connectors	Power 5V, 3.3V , VDD_ADC(1.8V) 3.3V I/O on all signals McASP0, SPI1, I2C, GPIO(69 max), LCD, GPMC, MMC1, MMC2, 7 AIN(1.8V MAX), 4 Timers, 4 Serial Ports, CAN0, EHRPWM(0,2),XDMA Interrupt, Power button, Expansion Board ID (Up to 4 can be stacked)	
Weight	1.4 oz (39.68 grams)	
Power	Refer to Section 6.1.7	

Tabla 1: Características completas de la *Beaglebone Black*

2.2 *ROBOCAPE*

La *ROBOcape* es una placa electrónica especialmente diseñada para su uso con la placa *BeagleBone Black*. El uso conjunto de ambas ofrece la posibilidad de diseñar cualquier tipo de sistema robótico y la programación del mismo. Esto se debe a la potencia de cómputo de la *BeagleBone Black* y a la gran cantidad de periféricos que la *ROBOcape* ofrece [3].

2.2.1 CARACTERÍSTICAS TÉCNICAS DE LA *ROBOCAPE*

- Sistema de alimentación DC/DC independiente, capaz de ofrecer hasta 3A y alimentar directamente a la *BeagleBone Black*.
- Cargador de baterías lipo de 2 células.
- 4 conexiones *PWM* para servos mediante buffer de salida.
- 4 puentes H para control de motores.
- 2 entradas para encoder.
- 6 conectores para sensores de ultrasonido *HC-SR04*.
- 4 conectores para sensores tipo Sharp de salida analógica.
- 2 conectores para conexión red servos *Dynamixel (RS-485)*.
- 2 puertos serie con salida digital 3v3.
- *IMU* 9 ejes *(MPU-9150).*
- Altímetro y termómetro.
- *GPS* integrado (*FGPMMOPA6H*), con toma para antena externa.
- *HUB USB 2.0* de 4 puertos.

El aspecto de esta placa que hace de módulo de expansión de la *BeagleBone Black*, tiene el aspecto mostrado en la figura 6, donde además se puede observar la distribución de todos los elementos que la forman.

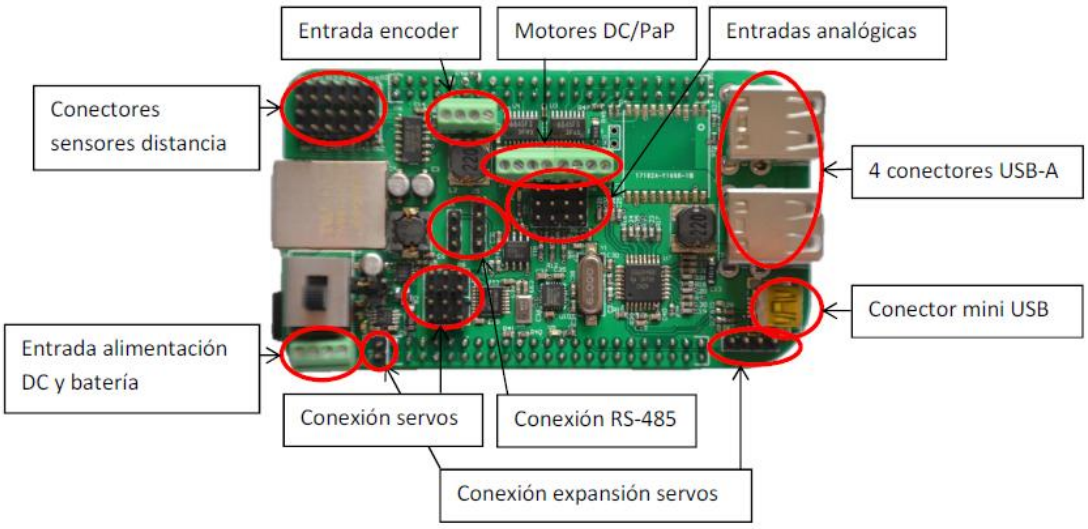

Figura 6: Aspecto y disposición de los dispositivos de la *ROBOcape*

2.2.2 INTERCONEXIÓN CON LA *BEAGLEBONE BLACK* Y ALIMENTACIÓN

La *ROBOcape*, asegura el correcto funcionamiento al conectarla con la *BeagleBone Black*, pero no con las versiones anteriores. En teoría, es posible alimentarla mediante la conexión *USB* de la *BeagleBone Black* al PC. Pero es aconsejable alimentar la *Beaglebone Black* mediante un adaptador AC/DC de 5V/1A. En la figura 7 se puede observar el aspecto de ambas placas conectadas.

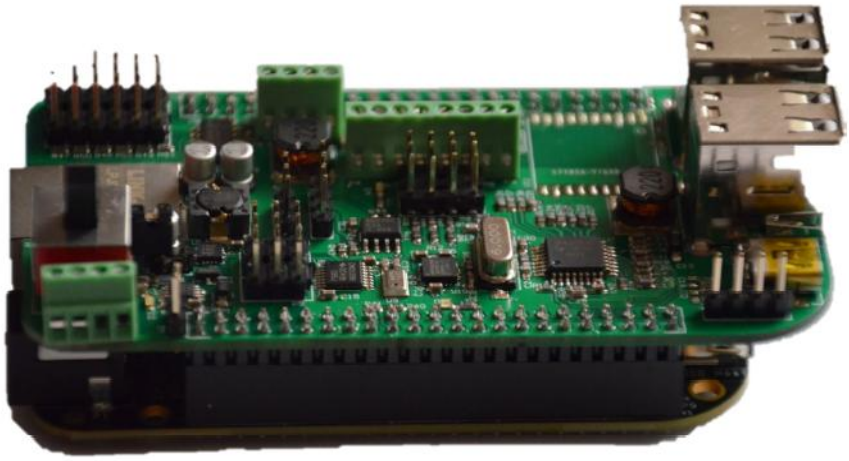

Figura 7: *BeagleBone Black* y *ROBOcape* conectadas

2.2.3 *IMU* DE 9 EJES

Para la realización de este trabajo, se han utilizado las entradas analógicas para sensores *Sharp*, además de las salidas *PWM* (Motores DC/PaP) y para el control de Servos. Pero un elemento muy importante es la *IMU*. Se trata de un elemento de gran utilidad para el mundo de la robótica, ya que es la encargada de obtener la posición espacial gracias a sistemas como acelerómetros, giroscopios y magnetómetros.

El sistema del que dispone la *ROBOcape* es un *MPU-9150*, diseñado para requisitos de baja potencia, bajo coste y alto rendimiento [4]. El *MPU-9150* es un encapsulado que combina en su interior dos chips: el *MPU-6050*, que contiene un giroscopio de 3 ejes, un acelerómetro de 3 ejes y un procesador integrado (*Digital Motion Processor*) y el *AK8975*, un magnetómetro digital de 3 ejes.

- **Acelerómetro:** Entre las funciones que incluye el acelerómetro se destacan:
  - Está formado por sensores de aceleración en los ejes X, Y y Z.
  - Rango de escala programable por el usuario: ±2g, ±4g, ±8g y ±16g.
  - Convertidor *ADC's* integrado de 16 bits.
  - Detección de orientación y señalización.
- **Giroscopio**
  - Sensores de velocidad angular en los ejes X, Y y Z.
  - Rango de escala programable por el usurario: ±250, ±500, ±1000 y ±2000 °/seg.
  - Convertidor *ADC's* integrado de 16 bits.
  - Señal de sincronismo externo conectado al pin *FSYNC* soporta imagen, vídeo y sincronización con el *GPS*.
- **Magnetómetro**
  - Sensor magnético de efecto Hall de silicio, de eje triple, con concentrador magnético.
  - La resolución de los datos de salida es de 13 bit (0,3 µT por *LSB*).
  - Amplio rango de medición dinámico y alta resolución con el menor consumo de corriente posible.
  - Rango de medida de fondo de escala: ±1200 µT.

Las hojas de características de estos elementos y el mapeo de registros de dichos dispositivos, se pueden encontrar en las referencias [5] y [6].

CAPÍTULO 3: TECNOLOGÍA UTILIZADA

En este apartado se van a describir las diferentes tecnologías de *software* que se han utilizado para la realización del TFM tanto para el servidor que corre sobre el sistema empotrado formado por la *BeagleBone Black* y la *ROBOcape*, como para el cliente que se ejecutará desde un *PC* de propósito general.

3.1 TECNOLOGÍA *OPC*

El estándar *OPC* (*OLE for Process Control*) nació debido a la cada vez mayor integración del *software* basado en *PC* dentro de los sistemas de automatización. Concretamente para unificar el protocolo de comunicación entre los distintos dispositivos dentro de una red.

Su aparición se debe a que en un principio, los dispositivos se conectaban con el *PC* mediante buses y protocolos diferentes, además de que los distintos interfaces de red tenían un interfaz de programación propio de cada distribuidor. Este hecho, unido al gran número de distribuidores, hacía que los costes de desarrollo y mantenimiento fuesen muy elevados. La figura 8 es la encargada de mostrar este problema.

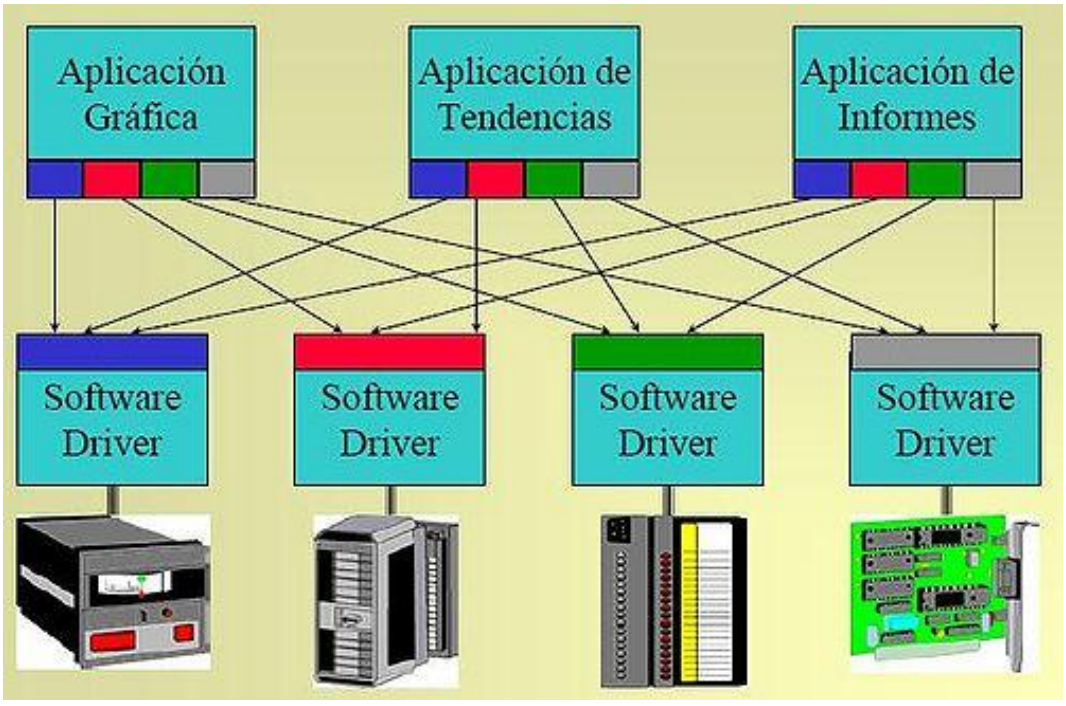

Figura 8: Conexión sin *OPC*

Como solución al problema anterior surgió el estándar *OPC* clásico, que engloba varias especificaciones del mismo dependiendo de la funcionalidad que se necesite. Siendo la primera de todas ellas *OPC-DA*.

3.1.1 *OPC-DA*

La especificación *OPC-DA* (*Data Access*). Es una agrupación de estándares que se centra en la comunicación de datos en tiempo real entre dispositivos de adquisición de datos con dispositivos de visualización e interfaces *HMI*. Siendo la máxima prioridad de esta especificación el flujo continuo de información. Esta especificación ofrece las siguientes características:

- API's que permitan acceder a diferentes datos de proceso.
- Interfaces para los datos de proceso, eventos y alarmas.
- Usa como base la tecnología *COM / DCOM* de *Microsoft Windows*.
- Los vendedores de hardware pueden proveer servidores *OPC* como driver estándar, haciendo posible que los vendedores de software puedan acceder a datos de proceso implementando un solo driver como cliente *OPC*.

Dado que esta especificación solo trabaja con datos en tiempo real, es necesario utilizar las otras especificaciones del estándar *OPC* clásico cuando se necesita acceder a:

- Historial de datos de proceso es necesario utilizar *OPC-HDA* (*Historical Data Access*).
- Alarmas y eventos requiere el uso de *OPC-AE* (*Alarms and Events*)

En la figura 9 se puede observar la simplificación de la comunicación entre servidores y clientes que supuso la implementación del protocolo *OPC-DA* y el uso de drivers.

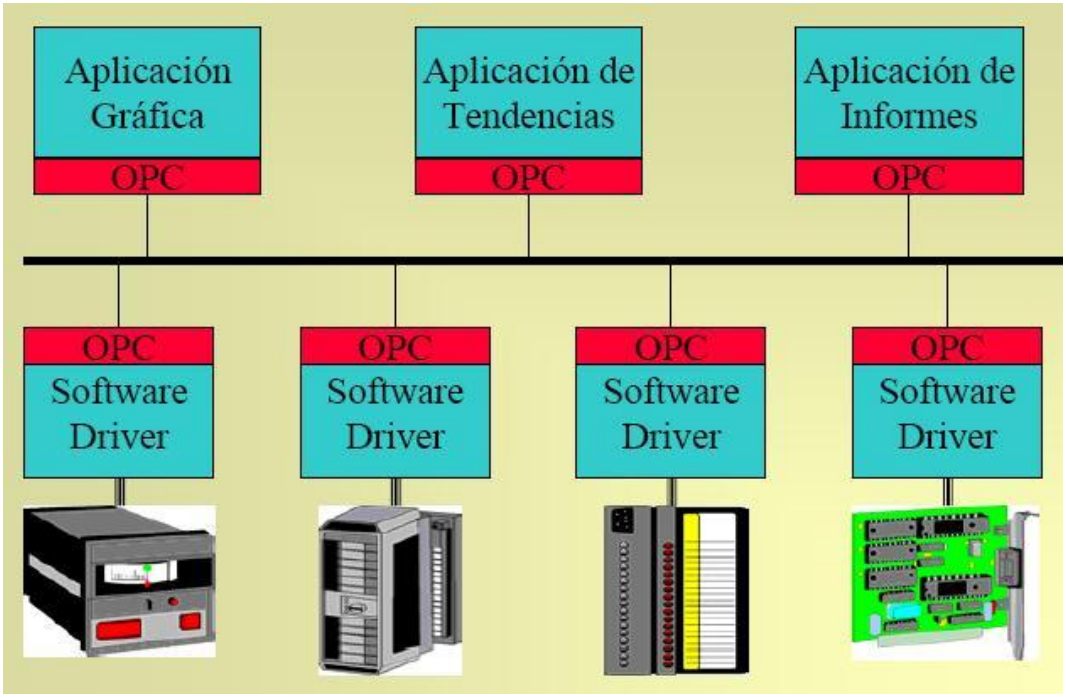

Figura 9: Conexión mediante el protocolo *OPC-DA*

3.1.2 *OPC-UA*

Sin embargo, dadas las necesidades de hoy en día, ha surgido una evolución de este estándar *OPC* llamada *OPC-UA* (*Unified Architecture*). Este nuevo protocolo integra las funcionalidades de las anteriores especificaciones de *OPC* clásico y corrige varias las lagunas del mismo, como son:

- Frecuentes problemas de configuración con *DCOM*.
- *Time-outs* no configurables.
- Dependencia de la plataforma *Microsoft Windows*.
- Falta de control sobre *DCOM* (Los desarrolladores no tenían acceso a los recursos, dificultándose la depuración y la corrección de *bugs*).
- Soporte para redundancia.
- *Heartbeat* para las conexiones, que permite tanto a cliente como a servidor sabes si el otro sigue en funcionamiento y detectar posibles interrupciones.
- *Buffering* de la información para que las pérdidas de conexión no impliquen la pérdida de datos.

OPC-UA es una plataforma que extiende las capacidades del modelo clásico ofreciendo: adquisición de datos, modelado de la información y comunicación entre planta y aplicaciones de una forma fiable y segura. Además, dispone de una arquitectura orientada a servicios, con el objetivo de integrar los diferentes elementos de *software* de una forma flexible. Por esta razón, se descarta la tecnología *COM/DCOM* de *Microsoft*, a favor de una tecnología independiente. Gracias a esto se amplía la cantidad de sistemas y dispositivos que podrán compartir información, no solo sistemas del propietario *Microsoft*.

La flexibilidad de este protocolo queda reflejada en la figura 10, en la que se puede observar la comunicación entre distintos sistemas con un lenguaje de programación distinto, eliminando dependencias del protocolo anterior.

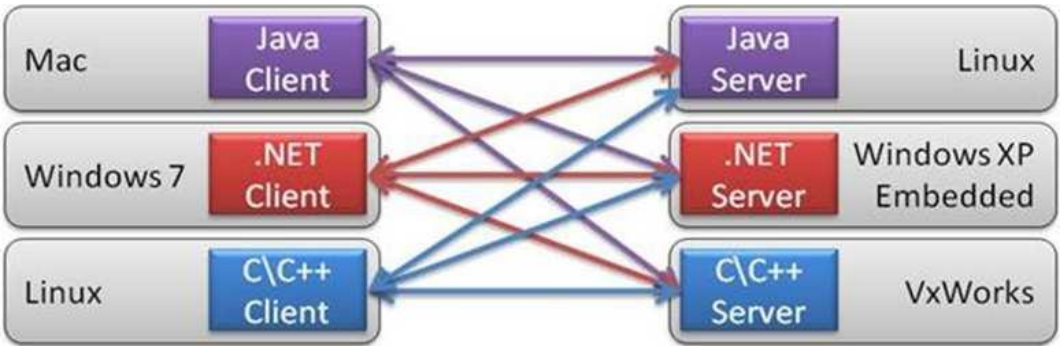

Figura 10: Conexión mediante el protocolo *OPC-UA*

Además, *OPC-UA* facilita en gran medida la comunicación a través de internet eliminando la configuración *DCOM*. Siendo suficiente abrir un único puerto en el *firewall*, ya que la protección la garantizan los mecanismos de seguridad que integra el protocolo. Para ello, sustituye dicha configuración por uno de los dos protocolos distintos que se nombran a continuación. El primero es un protocolo binario basado *TCP* orientado a un buen rendimiento (eficiente y optimizado) y el segundo está orientado a servicios *Web*.

El protocolo binario *TCP* ofrece el mejor rendimiento debido a que necesita menos recursos para su funcionamiento, muy importante para sistemas empotrados, debido a que no requiere de *XML parser, SOAP o HTTP*. Además, ofrece la mejor interoperabilidad un solo puerto *TCP*, escogido arbitrariamente, que facilita la comunicación a través de un *firewall*.

El protocolo de servicios *Web* (*SOAP*) tiene un buen soporte siendo amigable con los *firewall* y hace uso de los puertos estándar http/https.

En la figura 11, se puede observar que dicha seguridad está basada en estándares e incluye opciones para autenticación de usuario e instancias de *software* (certificados), firma y cifrado de los mensajes que se transmiten, lo que garantiza la confidencialidad e integridad de la información. Este esquema de seguridad es válido para los protocolos de servicios Web y el binario *TCP*.

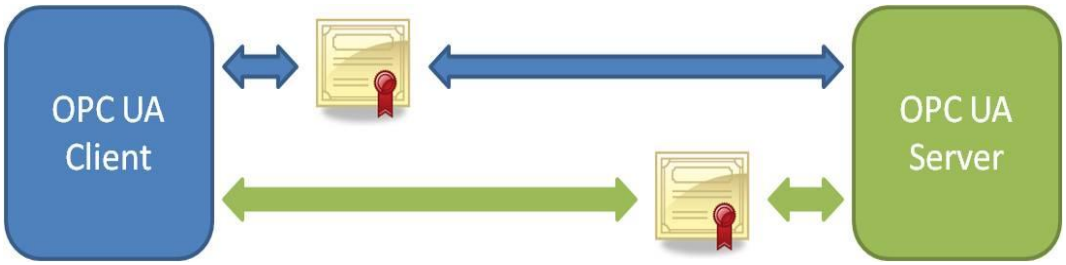

Figura 11: Uso de certificados *OPC-UA*

El modelo de datos de *OPC-UA* se basa en nodos, que pueden contener cualquier tipo de información. Estos nodos son considerados como objetos, al igual que en un lenguaje de programación orientado a objetos. Este concepto ofrece la posibilidad de representar datos de proceso, datos históricos, alarmas y llamadas a funciones con traspaso de parámetros. Pudiendo relacionar todos estos datos entre sí mediante los distintos métodos de los nodos.

Dadas las múltiples mejoras que aporta el nuevo estándar *OPC-UA*, está pensado para que termine sustituyendo al modelo OPC clásico y sus distintas variantes, aunque éste es capaz de coexistir con ellos.

3.2 *JAVA*

Partiendo de que toda la programación se ha implementado en el lenguaje de programación Java, es necesario nombrar los elementos de software de los que se ha hecho uso.

3.2.1 INTERFAZ DE PROGRAMACIÓN *ECLIPSE*

Eclipse es el entorno de programación que se ha utilizado para la programación de este trabajo. Se trata de un *IDE* (*Integrated Development Environment*), es decir, una aplicación de *software* que se encarga de aportar a los programadores facilidades a los programadores para la edición y desarrollo de *software*. Un *IDE* por lo general lo forman: Un editor de código,

herramientas para la construcción del código y un depurador. Y en algunos casos, como es el caso del propio *Eclipse* o *NetBeans*, incluyen también un compilador.

Eclipse se caracteriza por trabajar en un *workspace* base, donde se configura el proyecto que contendrá los programas desarrollados. Además, se caracteriza por su sistema extensible mediante *plugins* para personalizar el entorno. Este entorno de programación está escrito casi por complete en *Java*, siendo su principal uso el de desarrollar aplicaciones en ese mismo lenguaje, pero su sistema de *plugins* permite desarrollar aplicaciones en otros muchos lenguajes, como son: *Ada, ABAP, C, C++, COBOL, Fortran, Java Script, Perl*, etc.

En la figura 12 se puede observar el interfaz gráfico de éste entorno de programación.

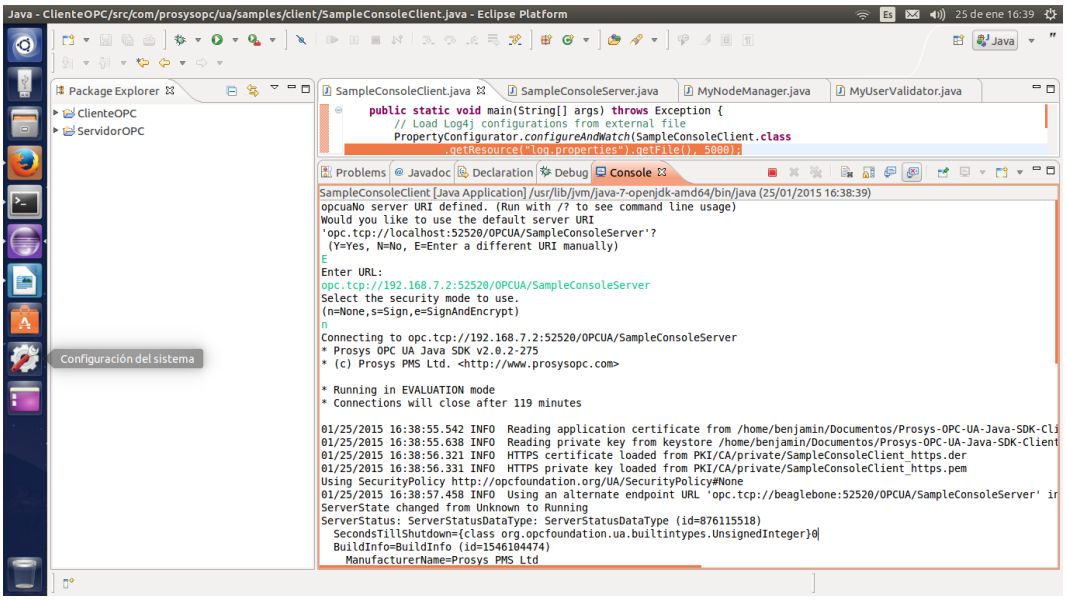

Figura 12: Interfaz gráfico del entorno de desarrollo *Eclipse*

3.2.2 SERVIDOR *PROSYS OPC-UA JAVA SDK*

Como punto de partida para este trabajo se ha utilizado la versión de prueba del *Prosys OPC-UA Java SDK* (versión 2.0.2-275).

Se ha optado por esta versión programada en *Java* debido a que *Java SDK* se encarga de todos los de talles de la comunicación *OPC-UA*, por lo que supone una gran ayuda de cara al programador, ofreciendo un interfaz de programación de alto nivel que permite la creación de aplicaciones de una forma sencilla y rápida. Esta versión es una versión de prueba que permite tener el servidor en ejecución limitado (2 horas).

3.2.3 LIBRERÍA *LIBBULLDOG*

La librería *Libbulldog* es una librería de *Java* de distribución libre y fácil uso. Su finalidad es el garantizar el acceso a periféricos de bajo nivel como son pines *GPIO* o buses *I2C* mediante *Java*, que es lenguaje de alto nivel y orientado a objetos. Además, ofrece soporte para dispositivos comerciales como son:

- *Beaglebone Black*.
- *Raspberry Pi* (Revisiones A y B)
- *Cubietruck* (*Cubieboard* 3)

3.2.4 CLIENTE GRÁFICO *PROSYS*

El cliente gráfico *Prosys OPC-UA Client* es un cliente programado en *Java*. Este cliente permite conectarse con el servidor *OPC-UA* de la placa *BeagleBone Black* mediante la dirección *IP* de la placa y estableciendo el puerto de comunicaciones que abre el servidor al ejecutarse. De esta forma, permite el acceso a los nodos del servidor. La figura 13 muestra una captura del interfaz gráfico del cliente.

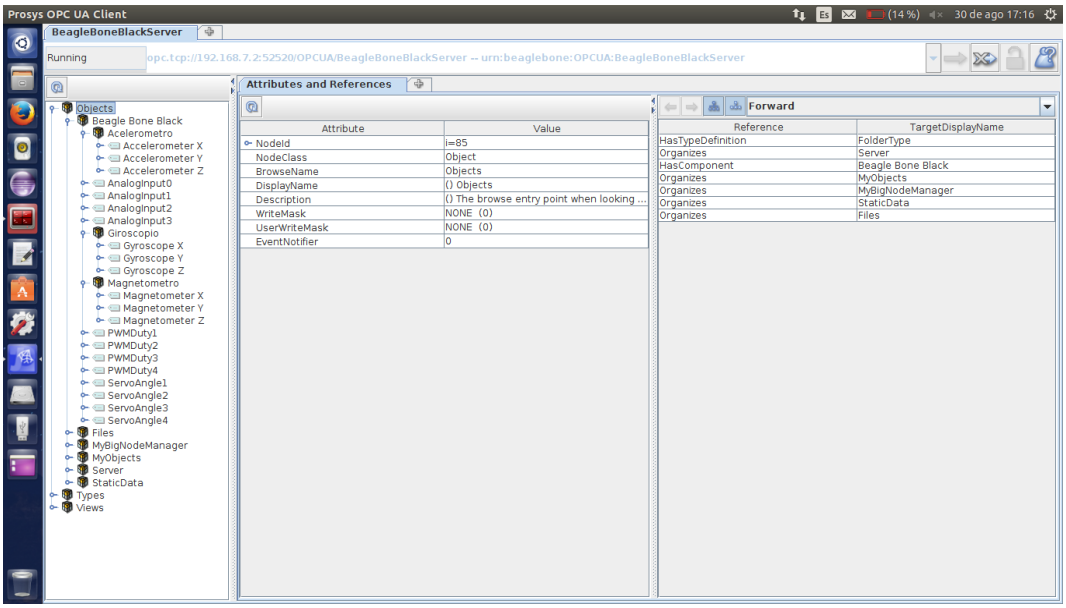

Figura 13: Interfaz gráfico del cliente gráfico *Prosys*

Este cliente *OPC* permite ver con claridad y cómodamente los nodos de información del servidor, ya que tiene la característica la *Data View*. Esta característica se abre como una pestaña más, donde se pueden arrastrar los nodos que se deseen del árbol de carpetas y se encarga de mostrar su valor refrescándolo a con la frecuencia que se le indique. Además permite la edición del valor de dichos nodos, siendo posible una variación manual por parte del usuario de las salidas de la *ROBOcape*. En la figura 14 se puede ver como se muestran los nodos mediante el *Data View*.

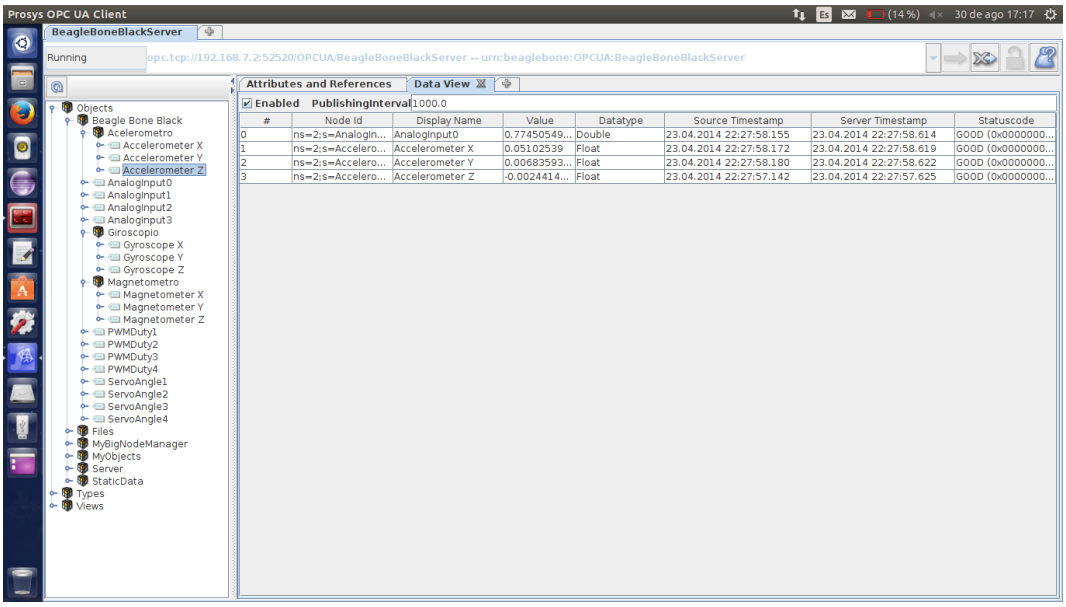

Figura 14: Función *Data View* del cliente gráfico *Prosys*

CAPÍTULO 4: ESTRATEGIA DE DISEÑO

La estrategia que se ha seguido para implementar una capa de software, que sea capaz de intermediar entre los dispositivos de la *ROBOcape* y el servidor *OPC-UA* que ejecutará la *BeagleBone Black*, se ha dividido en varias partes. Siendo el objetivo final de esta capa el poder compartir la información con un cliente *OPC-UA* que se ejecuta desde una máquina remota, en este caso desde un *PC* de propósito general. Siendo el esquema general de la comunicación el que se puede observar en la figura 15.

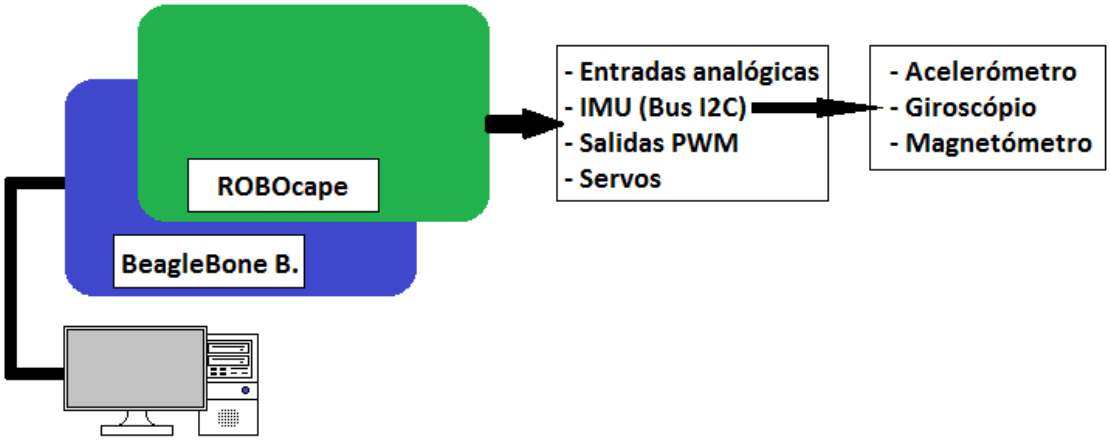

Figura 15: Esquema de la comunicación del *hardware*

4.1 ELECCIÓN DEL LENGUAJE DE PROGRAMACIÓN Y LA LIBRERÍA DE ACCESO

Al haberse elegido como único lenguaje de programación el lenguaje Java, debido a su gran flexibilidad, a su portabilidad una vez compilado y a que el trabajo parte de un servidor ya programado en dicho lenguaje. Además, Java permite que todo el código de ejecución puede ser comprimido en un fichero (*.jar) que incluya ya todas las librerías necesarias en su interior.

Ha sido necesario el uso de una librería externa para el acceso a los dispositivos de bajo nivel como son las E/S y la *IMU* de la *ROBOcape* de una forma eficiente. Concretamente, se ha hecho uso de la librería *Libbulldog*. Esta librería, implementa los métodos de acceso y configuración necesarios para construir la capa de *software* que garantiza el acceso del servidor a las E/S.

4.2 CAPA DE *SOFTWARE* IMPLEMENTADA

La capa de software, lleva el nombre de *BeagleBoneBlack* y ha sido implementada como una clase de Java. En esta clase, se ha implementado y configurado los métodos de la librería *Libbulldog* necesarios para poder acceder a cada dispositivo de la *IMU* y de E/S según las peticiones del servidor. Es decir, permite el acceso individual a cualquier dispositivo de la *ROBOcape* independientemente del tipo al que pertenezca. Siendo en su totalidad:

- 4 entradas analógicas para sensores Sharp.
- 4 salidas de modulación *PWM*.
- 4 Salidas para establecer el ángulo de los *Servos.*
- La IMU, que dispone de acelerómetro, giroscopio y magnetómetro.

Además de esta clase, es necesario variar el código del propio servidor para que cree los nodos de información necesarios y los actualice con los valores obtenidos de la lectura de los dispositivos de la *ROBOcape*. Por otra parte, debe ser capaz de escribir valores en los dispositivos de salida según las peticiones del cliente que se conecte al servidor. La figura 16 representa gráficamente la tarea de la capa implementada.

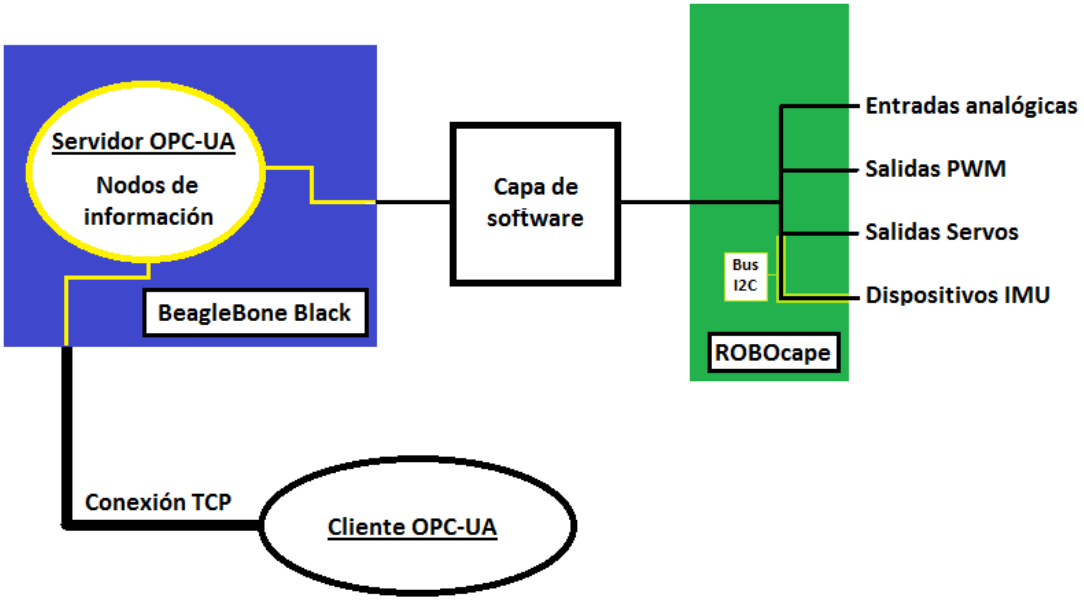

Figura 16: Esquema de diseño de la capa de *software*

4.3 CONFIGURACIÓN DE ARRANQUE DEL SERVIDOR

Por otra parte, entre las modificaciones que se han realizado en el servidor, se ha implementado una función que permita la selección de los dispositivos a los que se desea acceder, a partir de un fichero de texto plano. Esto se realiza en el arranque del servidor, donde se establece a que E/S accederá el servidor y los nodos de información que creará para proporcionar los datos al cliente. En la figura 17 se puede observar dicha selección.

Figura 17: Funcionamiento del fichero de configuración

Una vez seleccionados los elementos y a los que se desea acceder, el servidor solo creará los nodos de información correspondientes y actualizará los valores de los dispositivos que se le indiquen.

El fichero de configuración tiene la estructura que se muestra en el siguiente cuadro de texto, partiendo de que se desease acceder a todas las características de la placa:

```
AnalogInput0 double AnalogInput 0
AnalogInput1 double AnalogInput 1
AnalogInput2 double AnalogInput 2
AnalogInput3 double AnalogInput 3
PWMDuty1 float PWM 1
PWMDuty2 float PWM 2
PWMDuty3 float PWM 3
PWMDuty4 float PWM 4
ServoAngle1 float Servo 1
ServoAngle2 float Servo 2
ServoAngle3 float Servo 3
ServoAngle4 float Servo 4
Accelerometer float IMUDevice A
Gyroscope float IMUDevice G
Magnetometer float IMUDevice M
PID float PID ON
```

4.4 ACTUALIZACIÓN DE LOS NODOS DEL SERVIDOR

Para la actualización del valor de los nodos del servidor existen dos métodos. Para las entradas analógicas y los valores de los ejes del acelerómetro, giroscopio y magnetómetro se ha decidido implementar un hilo de ejecución que se encargue de leer periódicamente el valor de los dispositivos y de ir volcándolo en los nodos correspondientes. Mientras que, para las salidas *PWM* y el ángulo de los *Servos* se ha implementado un *listener* que está pendiente de los cambios que pueda haber en el nodo del servidor, provenientes de un cliente, para así actualizar su valor físico. En la figura 18 se puede apreciar una representación gráfica de la creación de los nodos a partir de los dispositivos a los que se desea acceder.

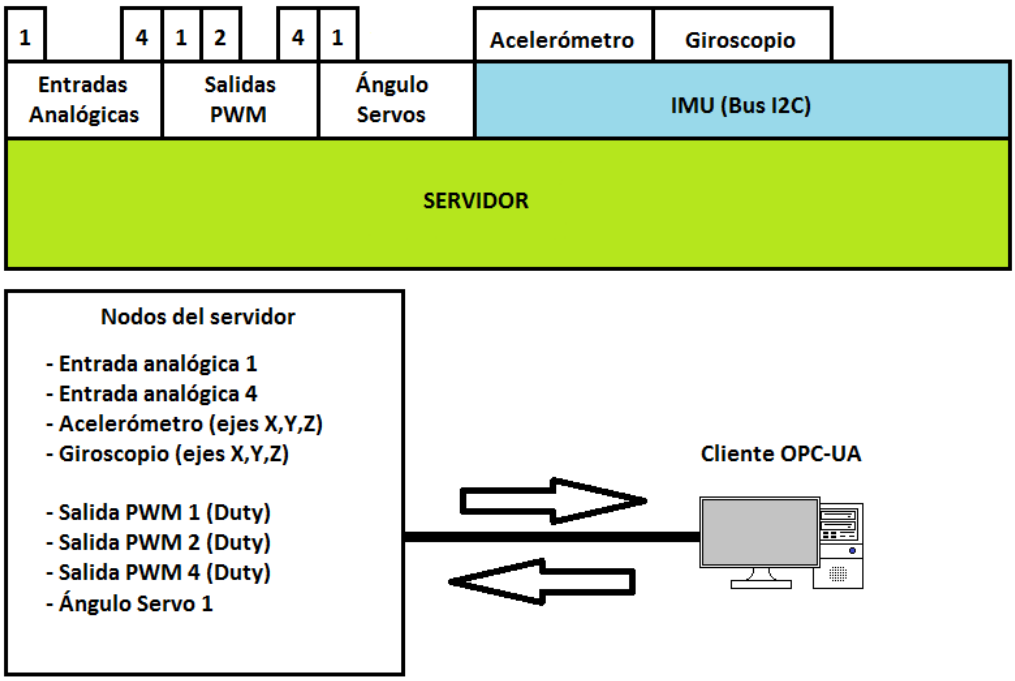

Figura 18: Nodos del servidor

4.5 EJEMPLO DE IMPLEMENTACIÓN DE CONTROL

A parte de la supervisión y la actuación que permite realizar la comunicación cliente servidor. Dada la potencia de la *BeagleBone Black* y que Java es un lenguaje concurrente, se ha implementado una clase capaz de ejercer un control *PID*. Dada las múltiples ventajas que supone la implementación de un control en el sistema empotrado que permita controlar el proceso de forma local, sin el uso de buses. Aunque éste control no sea adecuado para plazos de tiempo real.

El ejemplo de control se basa en la actuación sencilla sobre la salida *PWM* 1 tomando como referencia el valor leído por el sensor *Sharp* conectado a la entrada analógica 0. Siendo posible incluir este control entre las funcionalidades del servidor mediante el fichero de configuración. Además, dicho control se ejecuta concurrentemente mediante un hilo de ejecución dentro del servidor *OPC-UA*.

4.6 FUNCIONAMIENTO GENERAL DEL SERVIDOR

Las modificaciones implementadas en el servidor hacen que éste, una vez arrancado se quede trabajando en atender a los clientes que se conecten, escuchando mediante los *listeners* los cambios que introduzcan dichos clientes en las salidas, y proporcionándoles periódicamente la lectura de las entradas y los dispositivos de la *IMU*. Por otra parte, también trabajará en el control, como se muestra en la figura 19.

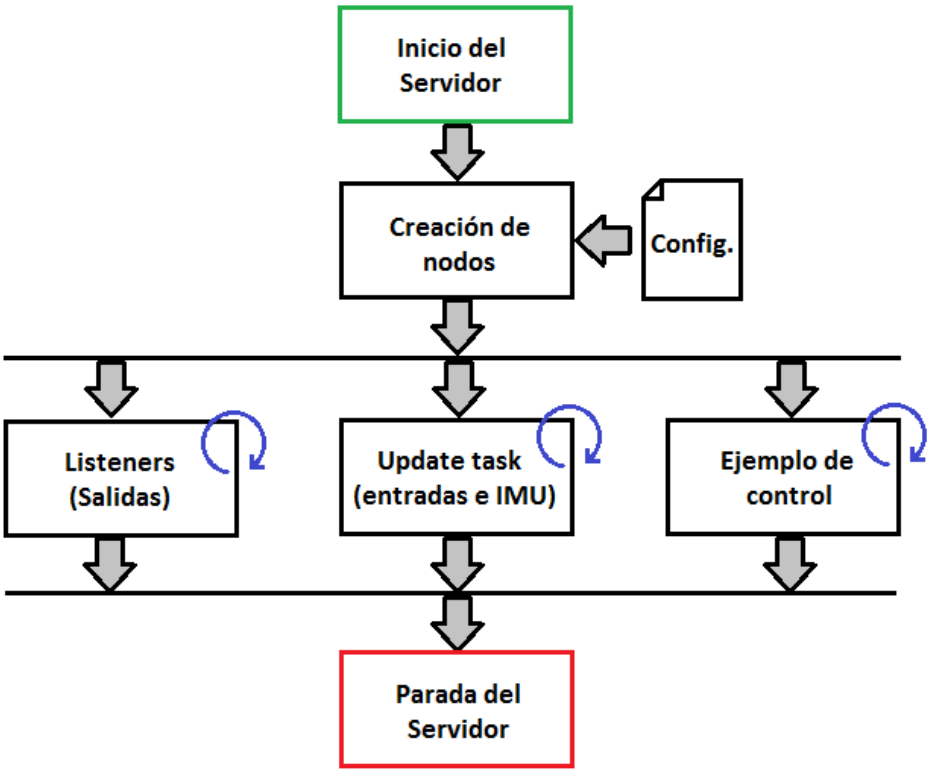

Figura 19: Esquema de funcionamiento del servidor

Estas funciones que se han citado se realizarán en bucle hasta que se dé al servidor la orden de que se cierre. Cabe decir que los *listeners* no se ejecutan en un bucle o hilo independiente del servidor sino que saltan cuando detectan un cambio en la variable a la que escuchan siendo ésta el nodo correspondiente a una salida *PWM* o de *Servo*.

CAPÍTULO 5: DESARROLLO DEL PROYECTO

En este apartado se va a ahondar en los pasos seguidos y el código que desarrollado para que el servidor realice las tareas deseadas.

5.1 PREPARACIÓN DEL ENTORNO DE DESARROLLO

Para realizar la programación de la capa de *software* y las distintas variaciones en el servidor *OPC-UA*, se procedió a instalar la versión 3.8 de *eclipse* para *Ubuntu*, en la que se creó un proyecto de Java nuevo, donde se importaron los distintos ficheros Java que componen la versión de evaluación del servidor *Prosys OPC-UA Java SDK Server*. En dicho proyecto se importaron además las distintas librerías para que funcionase.

A partir de ese momento, se empezó el desarrollo. Siendo el primer cambio realizado sobre el servidor el cambio de su nombre, tanto del archivo principal como dentro del código siempre que se le referenciaba. Pasando a llamarse *BeagleBoneBlackServer* en vez de *SampleConsoleServer*.

5.2 CREACIÓN DE LA CAPA DE *SOFTWARE*

La capa de software, dejando al margen las modificaciones que se realizan sobre el servidor, se compone por tres ficheros programados en *Java*:

- *BeagleBoneBlack.java*
- *imuDevice.java*
- *datos3axis.java*

Siendo los ficheros *imuDevice* y *datos3axis* ficheros de apoyo para la lectura de los dispositivos de la *IMU*. El *imuDevice* se encarga de configurar la comunicación por el bus *I2C*, estableciendo las direcciones de cada dispositivo, mientras que *datos3axis* es una variable compartida para almacenar los valores de las lecturas y utilizarlas en el servidor.

5.2.1 FICHERO *BEAGLEBONEBLACK.JAVA*

En primer lugar se deben importar las librerías de *Libbulldog* necesarias para la configuración y el acceso a cada uno los dispositivos de bajo nivel de la *ROBOcape* a través de los pines *GPIO* de la *BeagleBone Black*. Las líneas de código implementadas aparecen en el apartado A.2.1 del anexo I.

A partir de ahí, se procede a crear la clase partiendo de la declaración de las variables necesarias asignando los pines de la placa para el acceso a entradas y salidas. Además de la inicialización del el objeto *imuDevice* (basado en la clase que contiene *imuDevice.java*) para acceder a los elementos de la *IMU* a través del bus *I2C* y las variables necesarias para acceder a las distintas entradas y salidas. Como se muestra en el apartado A.2.2.

Una vez declarados todos los elementos, se procede a configurar aquellos que lo requieran en el constructor de la clase *BeagleBoneBlack*. Como se puede observar en el apartado A.2.3, es necesario inicializar el objeto *imu* para que los métodos estén disponibles para su posterior uso. Además, las salidas *PWM* no asociadas a los Servo necesitan de activación y que se les indique a la frecuencia a la que trabajarán.

Por último, en el apartado A.2.4 aparecen los métodos de acceso a los elementos de la *ROBOcape* que se han implementado. Los cuales permiten el acceso a los dispositivos de la *IMU* (acelerómetro, giroscopio, magnetómetro), la lectura de las entradas analógicas y la variación del *Duty* de las salidas *PWM* y del ángulo en las salidas *PWM* destinadas a los servos.

Ahora bien, como se ha dicho con anterioridad, el acceso a los dispositivos de la *IMU* se sustenta de la clase *imuDevice* y de la variable compartida datos3axis. Que se encargan de establecer las direcciones necesarias para manejar el bus *I2C* y la posibilidad de devolver los datos de lectura al servidor, cuando éste llame al método de acceso a cualquier dispositivo de la *IMU*.

5.2.2 CONTENIDO DEL FICHERO *IMUDEVICE.JAVA*

Como esta clase también está destinada al acceso de los dispositivos de bajo nivel de la *ROBOcape*, es necesario importar las librerías necesarias pertenecientes a *Libbulldog* tal y como aparece en el apartado A.3.1.

Ahora bien, esta clase se encarga de establecer las direcciones, mostradas en el apartado A.3.2, donde se encuentran los dispositivos de la *IMU* a los que se desea acceder a través del bus *I2C*, ya que son necesarias para poder hacer uso de los métodos de acceso.

Para acceder la comunicación con el bus *I2C* y al magnetómetro. es necesaria la inicialización de los mismos, de lo que se encarga el método *initMode() mostrado en el apartado* A.3.3, que luego se ejecutará en el constructor de la clase *BeagleBoneBlack*. Una vez hecho eso, se procede a implementar los métodos para que accedan a los dispositivos que se desea leer de la *IMU* haciendo uso de las direcciones declaradas con anterioridad.

Para la lectura del acelerómetro, giroscopio y magnetómetro, lo que se hace es leer la conversión de un convertidor *ADC* de 16 bits que implementan, por lo que se debe leer la parte alta (8 bits) y la baja (8 bits) de cada uno de los dispositivos, para cada uno de sus ejes, ya sea X, Y o Z. Una vez hecho eso se concatenan la parte alta con la baja y se convierten a un valor *float* para que se pueda introducir su valor en el nodo de información correspondiente.

De esta forma se incorpora el método de lectura para cada uno. Se ha utilizado la variable compartida dato3axis, mostrada en el apartado A.4, para devolver los valores de los ejes X, Y y Z de cada uno de los dispositivos. Y de esta forma, que en el servidor se pueda actualizar el valor de de lectura de cada uno de los 3 ejes de dichos dispositivos en sus correspondientes nodos de información.

Siendo la implementación de la lectura del acelerómetro la que se muestra en el apartado A.3.4. Como su nombre indica, éste mide la aceleración según el movimiento del dispositivo en m/s^2.

La implementación de lectura del giroscopio la mostrada en el apartado A.3.5, que se encarga de medir los cambios en la inclinación del dispositivo la cual se mide en °/s.

Y por último, en el apartado A.3.6 se puede observar el código necesario para acceder al magnetómetro. Que se encarga de realizar la medida de la dirección y fuerza del campo magnético que incida sobre el dispositivo, expresando la medida en µT.

5.3 VARIACIONES HECHAS SOBRE EL SERVIDOR *BEAGLEBONEBLACKSERVER.JAVA*

Este fichero contiene la clase principal del servidor y además las modificaciones que se han realizado para que el servidor acceda a los distintos dispositivos de la robocape y los sincronice con los nodos creados.

5.3.1 DECLARACIÓN DE VARIABLES NECESARIAS Y CONSTRUCTOR

En las líneas de código a continuación, se pueden observar las variables globales que se han declarado para la creación de los nodos y poder acceder a los distintos dispositivos de la *ROBOcape*, mediante la capa de *software* que se ha creado. Además, se observa como en el constructor se crea el objeto *bbb*, que es indispensable para tener acceso a los dispositivos y también las variables compartidas para leer los datos de la *IMU.*

```
private UaObjectNode BeagleBoneVariable;
private UaObjectNode AccelerometerVariable;
private UaObjectNode GyroscopeVariable;
private UaObjectNode MagnetometerVariable;

private final BeagleBoneBlack bbb;
private final datos3axis da;
private final datos3axis dg;
private final datos3axis dm;

public BeagleBoneBlackServer() throws IOException {
   bbb = new BeagleBoneBlack();
   da = new datos3axis();
   dg = new datos3axis();
   dm = new datos3axis();
}

private CacheVariable[] NodeArray = new CacheVariable[50];
private String[] periphArray = new String[50];
private String[] IDArray = new String[50];
private int numNodes=0;

private CacheVariable acelerometroX;
private CacheVariable acelerometroY;
private CacheVariable acelerometroZ;

private CacheVariable giroscopioX;
private CacheVariable giroscopioY;
private CacheVariable giroscopioZ;

private CacheVariable magnetometroX;
private CacheVariable magnetometroY;
private CacheVariable magnetometroZ;
```

5.3.2 CREACIÓN DE NODOS CON EL FICHERO DE CONFIGURACIÓN (*LOADNODESTASK*)

La esta función es la encargada de acceder al fichero de configuración, que se le pasa como parámetro. Donde el *StringTokenizer* se encarga de separar las palabras que hay en el fichero de texto y almacenarlas en variables para realizar la gestión adecuada de los nodos de información.

Mientras se lea una nueva línea, querrá decir que hay un nuevo nodo a considerar, y según los parámetros que se lean en dicha línea del fichero se ejecutará uno de los cases del *switch*. Donde se configurará un nuevo nodo de información en el servidor y se inicializará el tipo de información que contendrá. Además, las variables que almacenan los parámetros para crear estos nodos, luego son utilizadas por las funciones que actualizan los valores de los nodos, para saber cuales tienen que actualizar o no.

Tras la creación de todos los nodos deseados se cierra el fichero y la función acaba.

```
public void loadNodesTask(int ns, String archivo) throws
FileNotFoundException, IOException, StatusException{

   String cadena;
   int n=0;
   FileReader f = new FileReader(archivo);
   BufferedReader b = new BufferedReader(f);
   while((cadena = (b.readLine()))!=null) {

      StringTokenizer st = new StringTokenizer(cadena," ");

      String name = st.nextToken();
      String type = st.nextToken();
      periphArray[n] = st.nextToken();
      IDArray[n] = st.nextToken();

      UaType doubleType =
      server.getNodeManagerRoot().getType(Identifiers.Double);
      UaType intType =
      server.getNodeManagerRoot().getType(Identifiers.Int32);
      UaType boolType=
      server.getNodeManagerRoot().getType(Identifiers.Boolean);
      UaType floatType =
      server.getNodeManagerRoot().getType(Identifiers.Float);

      switch(periphArray[n]){
         …
      }
   }
   b.close();
   numNodes=n;
}
```

Para la realización de esta memoria, se han sustituido los distintos *cases* por unos puntos suspensivos. Esto se debe a que la función tiene una gran cantidad de líneas de código y de esta forma se puede explicar el contenido de la misma de una forma más visual en los siguientes cuadros de texto.

El primer case, es el encargado de lanzar el hilo de ejecución del control *PID*, si este aparece en el fichero de configuración.

```
case "PID":

   C.start();
   n++;
break;
```

Los tres cuadros de texto que aparecen a continuación son la implementación del case del acceso a la *IMU*. Que a su vez contiene otro *switch*, que se encarga de implementar los nodos correspondientes al acelerómetro, giroscopio o el magnetómetro, dependiendo de a cuál de ellos se necesite acceder.

```
case "IMUDevice":
   switch(name){
      case "Accelerometer":

         // Accelerometer folder (inside BeaglBoneBlack folder)
         final NodeId AccelerometerId = new NodeId(ns, "Accelerometer");
         AccelerometerVariable = new UaObjectNode(myNodeManager,
         AccelerometerId, "Acelerometro", myNodeManager.getDefaultLocale());
         BeagleBoneVariable.addReference(AccelerometerVariable,
         Identifiers.HasComponent,false);

         // Acelerometro X variable
         final NodeId AccelerometerXId = new NodeId(ns, "Accelerometer X");
         acelerometroX = new CacheVariable(myNodeManager, AccelerometerXId,
         "Accelerometer X", myNodeManager.getDefaultLocale());
         acelerometroX.setDataType(floatType);
         AccelerometerVariable.addComponent(acelerometroX);

         // Acelerometro Y variable
         final NodeId AccelerometerYId = new NodeId(ns, "Accelerometer Y");
         acelerometroY = new CacheVariable(myNodeManager,AccelerometerYId,
         "Accelerometer Y", myNodeManager.getDefaultLocale());
         acelerometroY.setDataType(floatType);
         AccelerometerVariable.addComponent(acelerometroY);

         // Acelerometro Z variable
         final NodeId AccelerometerZId = new NodeId(ns, "Accelerometer Z");
         acelerometroZ = new CacheVariable(myNodeManager, AccelerometerZId,
         "Accelerometer Z", myNodeManager.getDefaultLocale());
         acelerometroZ.setDataType(floatType);
         AccelerometerVariable.addComponent(acelerometroZ);

      break;
```

```
case "Gyroscope":

   // Gyroscope folder (inside BeaglBoneBlack folder)
   final NodeId GyroscopeId = new NodeId(ns, "Gyroscope");
   GyroscopeVariable = new UaObjectNode(myNodeManager, GyroscopeId,
   "Giroscopio", myNodeManager.getDefaultLocale());
   BeagleBoneVariable.addReference(GyroscopeVariable,
   Identifiers.HasComponent, false);

   // Giroscopio X variable
   final NodeId GyroscopeXId = new NodeId(ns, "Gyroscope X");
   giroscopioX = new CacheVariable(myNodeManager, GyroscopeXId,
   "Gyroscope X", myNodeManager.getDefaultLocale());
   giroscopioX.setDataType(floatType);
   GyroscopeVariable.addComponent(giroscopioX);

   // Giroscopio Y variable
   final NodeId GyroscopeYId = new NodeId(ns, "Gyroscope Y");
   giroscopioY = new CacheVariable(myNodeManager, GyroscopeYId,
   "Gyroscope Y", myNodeManager.getDefaultLocale());
   giroscopioY.setDataType(floatType);
   GyroscopeVariable.addComponent(giroscopioY);

   // Giroscopio Z variable
   final NodeId GyroscopeZId = new NodeId(ns, "Gyroscope Z");
   giroscopioZ = new CacheVariable(myNodeManager, GyroscopeZId,
   "Gyroscope Z", myNodeManager.getDefaultLocale());
   giroscopioZ.setDataType(floatType);
   GyroscopeVariable.addComponent(giroscopioZ);

Break;
```

```
      case "Magnetometer":

         // Magnetometer folder (inside BeaglBoneBlack folder)
         final NodeId MagnetometerId = new NodeId(ns, "Magnetometer");
         MagnetometerVariable = new UaObjectNode(myNodeManager,
         MagnetometerId, "Magnetometro", myNodeManager.getDefaultLocale());

         BeagleBoneVariable.addReference(MagnetometerVariable,
         Identifiers.HasComponent, false);

         // Magnetometro X variable
         final NodeId MagnetometerXId = new NodeId(ns, "Magnetometer X");
         magnetometroX = new CacheVariable(myNodeManager, MagnetometerXId,
         "Magnetometer X", myNodeManager.getDefaultLocale());
         magnetometroX.setDataType(floatType);
         MagnetometerVariable.addComponent(magnetometroX);

         // Magnetometro Y variable
         final NodeId MagnetometerYId = new NodeId(ns, "Magnetometer Y");
         magnetometroY = new CacheVariable(myNodeManager, MagnetometerYId,
         "Magnetometer Y", myNodeManager.getDefaultLocale());
         magnetometroY.setDataType(floatType);
         MagnetometerVariable.addComponent(magnetometroY);

         // Magnetometro Z variable
         final NodeId MagnetometerZId = new NodeId(ns, "Magnetometer Z");
         magnetometroZ = new CacheVariable(myNodeManager, MagnetometerZId,
         "Magnetometer Z", myNodeManager.getDefaultLocale());
         magnetometroZ.setDataType(floatType);
         MagnetometerVariable.addComponent(magnetometroZ);

      break;

   }
   n++;
break;
```

Por último, en el cuadro de texto a continuación se muestran las opciones para las entradas analógicas y por último el case default para las salidas *PWM* y las de los servo. Se usa el mismo case para ambas debido a que el nodo que almacena a ambas gasta la misma estructura y a ambas se les sincroniza un *listener*.

```
case "AnalogInput":

   final NodeId analogId = new NodeId(ns, name);
   NodeArray[n] = new CacheVariable(myNodeManager, analogId, name,
   myNodeManager.getDefaultLocale());
   switch(type){
      case "double": NodeArray[n].setDataType(doubleType);
      break;
      case "int": NodeArray[n].setDataType(intType);
      break;
      case "bool": NodeArray[n].setDataType(boolType);
      break;
      case "float": NodeArray[n].setDataType(floatType);
      break;
   }
   NodeArray[n].setValue(new DataValue(new Variant(0)));

   BeagleBoneVariable.addComponent(NodeArray[n]);
   n++;
break;

default:
   final NodeId pwmId = new NodeId(ns, name);
   NodeArray[n] = new CacheVariable(myNodeManager, pwmId, name,
   myNodeManager.getDefaultLocale());
   switch(type){
      case "double": NodeArray[n].setDataType(doubleType);
      break;
      case "int": NodeArray[n].setDataType(intType);
      break;
      case "bool": NodeArray[n].setDataType(boolType);
      break;
      case "float": NodeArray[n].setDataType(floatType);
      break;
   }
   NodeArray[n].setValue(new DataValue(new Variant(0)));
   NodeArray[n].addDataChangeListener(new dataChange());

   BeagleBoneVariable.addComponent(NodeArray[n]);
   n++;
break;
```

Esta función es llamada desde la función *createCompilanceNodes()* que ya traía implementada el servidor, para que cree los nodos necesarios para la aplicación una vez creados todos los nodos que necesita el servidor para su funcionamiento interno. Además se le pasa como parámetro la ruta del fichero de configuración.

5.3.3 LISTENER (*DATACHANGE*)

Esta clase implementa los *listener*, mediante cases para diferenciar si luego se los implementa a las salidas *PWM* o las de *Servo*.

```
class dataChange implements DataChangeListener{
   @Override
   public void onDataChange(UaNode node, DataValue oldValue, DataValue
   newValue) {
      String nodeStr = node.getDisplayName().getText();
      double newValueD = newValue.getValue().doubleValue();
      switch (nodeStr) {
         case "PWMDuty1":
         case "PWMDuty2":
         case "PWMDuty3":
         case "PWMDuty4":
            try {
               bbb.writePWMDuty(newValueD,nodeStr);
            } catch (IOException e) {
               // TODO Auto-generated catch block
               e.printStackTrace();
            }
         break;

         case "ServoAngle1":
         case "ServoAngle2":
         case "ServoAngle3":
         case "ServoAngle4":
            try {
               bbb.writeServoAngle(newValueD,nodeStr);
            } catch (IOException e) {
               // TODO Auto-generated catch block
               e.printStackTrace();
            }
         break;
      }
   }
}
```

Los *listeners*, una vez creados se los asigna a los nodos del servidor cuando estos son creados en la función *loadNodesTask*().

5.3.4 LECTURA PERIÓDICA DE LAS ENTRADAS E *IMU* (*UPDATETASK THREAD*)

```
Thread uT = new updateTask();

public class updateTask extends Thread {

public void run(){
   while (threadsEnd){
      try {
         for (int n=0 ; n<numNodes ; n++){
            switch (periphArray[n]){

               case "AnalogInput":
                  NodeArray[n].setValue(bbb.getAnalogInput(IDArray[n])*1.8);
               break;

               case "IMUDevice":
                  switch(IDArray[n]){
                     case "A":
                        bbb.read_accel(da);
                        acelerometroX.setValue(da.X_f);
                        acelerometroY.setValue(da.Y_f);
                        acelerometroZ.setValue(da.Z_f);
                     break;

                     case "G":
                        bbb.read_giro(dg);
                        giroscopioX.setValue(dg.X_f);
                        giroscopioY.setValue(dg.Y_f);
                        giroscopioZ.setValue(dg.Z_f);
                     break;

                     case "M":
                        bbb.read_magnet(dm);
                        magnetometroX.setValue(dm.X_f);
                        magnetometroY.setValue(dm.Y_f);
                        magnetometroZ.setValue(dm.Z_f);
                     break;
                  }
               break;
            }
         }
         Thread.sleep(1000);

      } catch (StatusException | IOException | InterruptedException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
      }
   }
}
}
```

Este *thread* es lanzado en el *run()* del servidor, para que empiece a actualizar los nodos así poner a disposición del cliente que se conecte la información deseada.

5.3.5 CONTROL *PID* (*PIDCONTROLLER THREAD*)

Para el control se ha implementado el siguiente *thread*, que se encarga de crear el objeto de configurar los parámetros necesarios y ponerlo en marcha. Todo esto se realiza a partir de una clase de código abierto contenida en el fichero *PIDController.java* [9].

```
Thread C = new controlTask();

public class controlTask extends Thread {
public void run(){

   final PIDController pidController = new PIDController(2, 0.5, 0);
   pidController.setInputRange(0, 1.8); // The input limits
   pidController.setOutputRange(0, 1); // The output limits
   pidController.setSetpoint(0.9); // My target value (PID should minimize
   the error between the input and this value)
   pidController.enable();
   double input = 0;
   double output = 0;
   while (threadsEnd) {
      try {

         input = bbb.getAnalogInput("0")*1.8;

         pidController.getInput(input);
         output = pidController.performPID();

         bbb.writePWMDuty(output, "PWMDuty1");

      } catch (IOException e1) {
         // TODO Auto-generated catch block
         e1.printStackTrace();
      }

      try {
         Thread.sleep(1000);
      } catch (InterruptedException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
      }
   }
}
}
```

Este *thread* se lanza dentro de la función *loadNodes()* solo si se detecta en el fichero de configuración la petición de que se incluya el control PID en el servidor.

CONCLUSIONES

Una vez presentado brevemente el desarrollo de cada una de las partes que componen el Trabajo Fin de Máster, siguiendo las pautas proporcionadas por el tutor y los puntos marcados como esenciales desde la organización del propio Máster, se pasa a continuación a reflejar una serie de conclusiones que se han obtenido a raíz de la solución implementada respecto a los objetivos que se fijaron inicialmente.

De tal manera que, partiendo de la distribución de evaluación del servidor *OPC-UA Prosys* (*Java SDK*), de su ejecución en un sistema empotrado basado en la *BeagleBone Black* y con un sistema operativo *Debian*, se pretendía acceder a dispositivos de bajo nivel de la placa y a su vez, gestionar la lectura y escritura en dichos dispositivos para poder realizar tareas de supervisión y actuación desde una máquina remota, en este caso un *PC* de propósito general. Para ello, ha sido necesario el diseño e implementación de una serie de elementos de *software*.

Seguidamente se incluyen las conclusiones en relación con los objetivos marcados y los resultados obtenidos.

1. En primer lugar, se ha desarrollado con éxito una capa software que permite el acceso al *hardware* de la dupla *BeagleBone Black + ROBOcape*. Junto a ello, se ha desarrollado una interfaz que permita el acceso del servidor a los distintos sensores y actuadores conectados a la *ROBOcape* mediante la capa de *software* creada.
2. En segundo lugar, se ha cumplido con el desarrollo de un sistema de configuración del árbol de *tags* (nodos) del servidor mediante un fichero de configuración. Dicho sistema se ha implementado de forma que se pueda seleccionar los dispositivos de la placa a los que se desea acceder con el servidor, con la finalidad de incluir los datos de lectura en los nodos. Estos dispositivos a los que se pretende acceder son:
    - Supervisión de un máximo de 4 entradas analógicas para sensores *Sharp*.
    - Supervisión de Acelerómetro, giroscopio y magnetómetro mediante el bus *I2C*.
    - Actuación sobre el ciclo de trabajo de un máximo de 4 salidas *PWM*.
    - Actuación sobre el ángulo de un límite de 4 servos.
    - Implementación de un sistema de control sencillo.

3. En tercer lugar, se ha realizado la validación y test del sistema mediante clientes *OPC-UA*. En este caso se ha podido observar la correcta disposición de los nodos según se hayan configurado, y la supervisión y actuación sobre los distintos nodos según se requiera.

En consecuencia, después de implementar todas estas funcionalidades exitosamente, se puede afirmar que se han logrado superar todos los objetivos que se habían marcado al principio de este trabajo.

Finalmente, tras la realización de este Trabajo Fin de Máster y teniendo en cuenta el contexto en el que se engloba dentro del Máster de Automática e Informática Industrial, se pueden enunciar algunas conclusiones relacionadas con la aplicabilidad del mismo:

a) Por una parte, ofrece un amplio abanico de posibilidades dada la flexibilidad de la tecnología *OPC-UA*, a la programación en *Java* del servidor y a la de las soluciones implementadas; lo que hace de este software un elemento portable y con muchas aplicaciones dentro del mundo de la robótica y la automatización. Sin embargo, es conveniente tener en cuenta que si la arquitectura del sistema empotrado de destino es distinta, se deberían realizar las modificaciones pertinentes.
b) Por otra parte, se trata de es un sistema muy versátil por todas las funcionalidades que es capaz de ofrecer: supervisión, actuación y control en un mismo sistema empotrado; por lo que, puede ser aplicado en sistemas de mayor envergadura, ya sean industriales como de electrónica de consumo.

Por último, se comprende que el Trabajo Fin de Máster supone un inicio o las bases para un trabajo de investigación posterior. De manera que, se ofrece una posible vía de ampliación que responde a las características de la *BeagleBone Black* y de sus *PRU*. Así, dada la potencia de la *BeagleBone Black* y la disponibilidad que tiene de sus *PRU,* se podría seguir trabajando sobre la implementación de controladores más avanzados que funcionen en tiempo real y aplicar un sistema de mayor complejidad de supervisión, actuación y control real.

BIBLIOGRAFÍA

[1] Colaboradores de *Wikiedia*. *Wikipedia*, La enciclopedia libre, 2015.
http://es.wikipedia.org/wiki

[2] *BeagleBone Black What is BeagleBone Black?*
http://beagleboard.org/black

[3] Miguel Albero Gil. *ROBOcape for BeagleBone Black (rev 1f beta)* - Guía de usuario.

[4] InvenSense: Sensor Function Calibration Software - MPU-9150.
http://www.invensense.com/products/motion-tracking/9-axis/

[5] *MPU-9150: Product Specification. Revision 4.0.*
https://www.sparkfun.com/products/11486

[6] *MPU-9150: Register Map and Descriptions. Revision 4.0.*
https://cdn.sparkfun.com/datasheets/Sensors/IMU/MPU-9150-Register-Map.pdf

[7] Tecnología *OPC-UA*
https://**opc**foundation.org/
http://matrikonopc.es/opc-ua/index.aspx

[8] *Prosys OPC UA Java SDK Server SDK Tutorial.*
www.**prosysopc**.com/

[9] *PID Controller Java, Googlecode*
https://code.google.com/p/frcteam443/source/browse/trunk/2010_Post_Season/Geisebot/src/freelancelibj/PIDController.java?r=17

ANEXO I

A.1 COMANDOS DE CONTROL MEDIANTE LA TERMINAL

A.1.1 CONEXIÓN CON LA *BEAGLEBONE BLACK*

La conexión con la placa y el *PC* de desarrollo se realiza mediante *USB* y una conexión *ssh* mediante la terminal del sistema operativo Ubuntu instalado en el *PC*. Para conectarse y manejar la placa es necesario conectarse en modo administrador (*root*) siendo la dirección *IP* de la placa la (192.168.7.2). Éste método proporciona control total sobre la placa.

```
ssh root@192.168.7.2
```

A.1.2 PREPARACIÓN DE LA *BEAGLEBONE BLACK*

Para la realización del proyecto se procedió en primer lugar a instalar *Java* en la placa *BeagleBone Black*. Para que esta tenga la capacidad de ejecutar el código compilado en el *PC* donde se desarrolla el programa. Siendo necesario bajar de la página oficial de *Oracle* la versión de *Java 8 (JDK)* para procesadores *ARM*, ya que el procesador de la placa es de dicha familia.

```
gunzip jdk-8-linux-arm-vfp-hflt.tar.gz
tar xfv jdk-8-linux-arm-vfp-hflt.tar
mv jdk1.8.0 /usr/
```

Con estos comandos lo que hacemos es descomprimir la carpeta que contiene el *Java 8 (JDK)* y por último copiarlo al directorio deseado. Ahora bien, es necesario el indicar al sistema operativo *Debian* de la placa el *path* de la carpeta que contiene la versión de java y el JAVA_HOME. Para ello es necesario implementar las siguientes líneas en el fichero *.bashrc* de la placa.

```
export JAVA_HOME=/usr/jdk1.8.0
export PATH=$PATH:$JAVA_HOME/bin
```

Para confirmar que todo se ha realizado de la forma correcta y funciona la versión de java instalada. Es posible introducir el siguiente comando en la terminal:

```
java -version
```

Lo que nos indicará la versión de java que actualmente está instalada en la placa, tal y como se puede observar en la figura 20. Esto nos permitirá compilar y ejecutar código escrito en dicho lenguaje de programación.

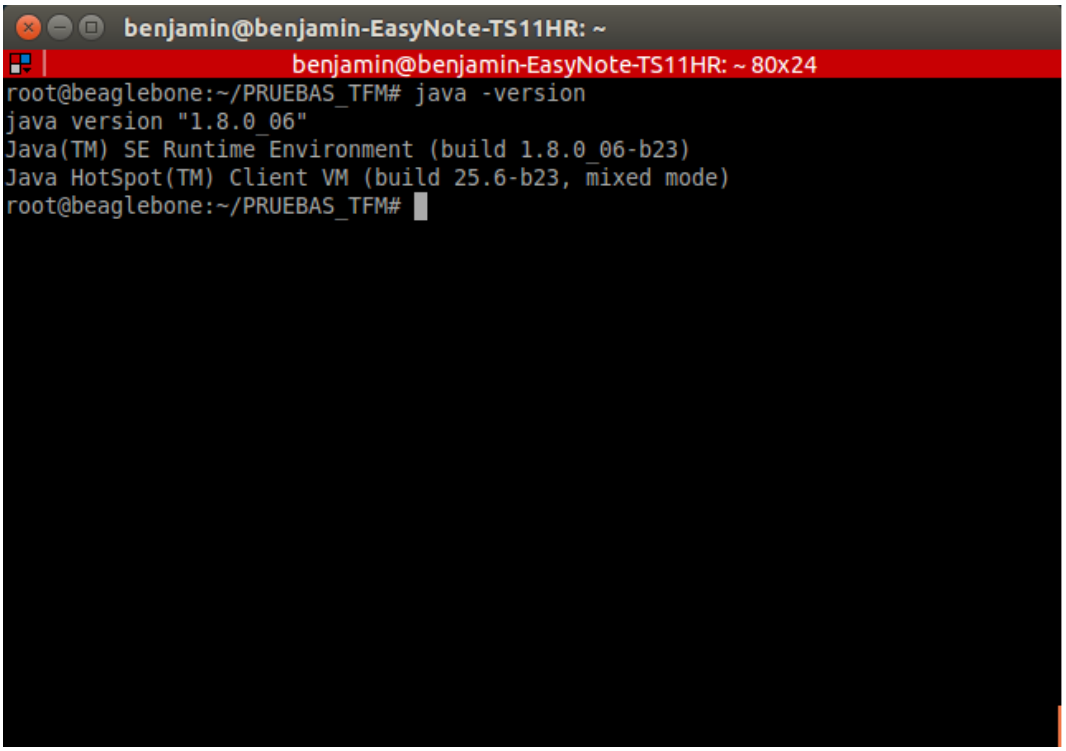

Figura 20: Comprobación de la versión de *Java* instalada en la *BeagleBone Black*

A.1.3 EJECUCIÓN DEL SERVIDOR EN LA *BEAGLEBONE BLACK*

Para probar la ejecución del servidor en la *BeagleBone Black*, debe copiarse a la misma el código que se ha implementado en *Eclipse*. Para ello, se ha exportando el código en un fichero *BeagleBoneBlackServer.jar* haciendo uso de la conexión *ssh* con la placa. Éste fichero contiene todos los elementos necesarios en su interior, ya sean las librerías como el código ya compilado. Para exportar los ficheros necesarios se ha hecho uso del comando de consola *rcp*:

```
rcp BeagleBoneBlackServer.jar root@192.168.7.2:/root
```

Al ejecutar el fichero mediante la línea de comando indicada en el siguiente cuadro de texto, se crean una serie de subcarpetas en el directorio y empieza la ejecución normal del servidor en sí. Es indispensable que el fichero de configuración se encuentre en la ruta indicada en la llamada a la función *loadNodesTask ()* ya que si no se provocará un error de ejecución.

```
java -Djava.library.path=/root/PRUEBAS_TFM/lib/ -jar
BeagleBoneBlackServer.jar
```

A.1.4 EJECUCIÓN EL CLIENTE GRÁFICO *PROSYS* Y COMUNICACIÓN CON EL SERVIDOR

Como se ha comentado en el apartado de las tecnologías empleadas. El cliente gráfico *Prosys* es lanzado en el *PC* con el que está conectada la placa *BeagleBone Black.* Para ello se utiliza la terminal de *Ubuntu*, pero sin hacer uso de la conexión *ssh* que permite controlar la placa, sino en una terminal distinta. Para ello se debe ejecutar el siguiente comando en el directorio que se encuentra el fichero *Prosys-OPC-UA-Client.jar.*

```
java -jar Prosys-OPC-UA-Client.jar
```

A.2 CÓDIGO DEL FICHERO *BEAGLEBONEBLACK.JAVA*

A.2.1 PAQUETE Y LIBRERÍAS UTILIZADAS

```
package com.prosysopc.ua.app.beagleboneblack;

import java.io.IOException;

import org.bulldog.beagleboneblack.BBBNames;
import org.bulldog.core.gpio.AnalogInput;
import org.bulldog.core.gpio.Pwm;
import org.bulldog.core.io.bus.i2c.I2cBus;
import org.bulldog.core.platform.Board;
import org.bulldog.core.platform.Platform;
import org.bulldog.devices.servo.Servo;
```

A.2.2 DECLARACIÓN DE VARIABLES

```
public class BeagleBoneBlack {

   Board board = Platform.createBoard();
   I2cBus bus = board.getI2cBus(BBBNames.I2C_1);
   imuDevice imu = new imuDevice(bus, 0x69);

   AnalogInput analogInput0 =
   board.getPin(BBBNames.AIN0).as(AnalogInput.class);
   AnalogInput analogInput1 =
   board.getPin(BBBNames.AIN1).as(AnalogInput.class);
   AnalogInput analogInput2 =
   board.getPin(BBBNames.AIN2).as(AnalogInput.class);
   AnalogInput analogInput3 =
   board.getPin(BBBNames.AIN3).as(AnalogInput.class);

   Pwm pwm1 = board.getPin(BBBNames.EHRPWM1B_P8_34).as(Pwm.class);
   Pwm pwm2 = board.getPin(BBBNames.EHRPWM1A_P8_36).as(Pwm.class);
   Pwm pwm3 = board.getPin(BBBNames.EHRPWM2A_P8_19).as(Pwm.class);
   Pwm pwm4 = board.getPin(BBBNames.EHRPWM2B_P8_13).as(Pwm.class);

   Pwm pwm5 = board.getPin(BBBNames.EHRPWM0B_P9_29).as(Pwm.class);
   Servo servo1 = new Servo(pwm5);
   Pwm pwm6 = board.getPin(BBBNames.ECAPPWM2_P9_28).as(Pwm.class);
   Servo servo2 = new Servo(pwm6);
   Pwm pwm7 = board.getPin(BBBNames.EHRPWM0A_P9_31).as(Pwm.class);
   Servo servo3 = new Servo(pwm7);
   Pwm pwm8 = board.getPin(BBBNames.ECAPPWM0_P9_42).as(Pwm.class);
   Servo servo4 = new Servo(pwm8);
```

A.2.3 CONSTRUCTOR DE LA CLASE

```
   public BeagleBoneBlack() throws IOException {

      imu.initMode();

      pwm1.setFrequency(50.0f);        // 50 Hz
      pwm1.enable();

      pwm2.setFrequency(50.0f);        // 50 Hz
      pwm2.enable();

      pwm3.setFrequency(50.0f);        // 50 Hz
      pwm3.enable();

      pwm4.setFrequency(50.0f);        // 50 Hz
      pwm4.enable();

   }
```

A.2.4 MÉTODOS DE ACCESO A LOS DISPOSITIVOS DE LA ROBOCAPE

```
   public void read_accel(datos3axis da) throws IOException{
      imu.read_acc(da);
   }
   public void read_giro(datos3axis dg) throws IOException{
      imu.read_giro(dg);
   }
   public void read_magnet(datos3axis dm) throws IOException{
      imu.read_magnet(dm);
   }
   public double getAnalogInput(String number) throws IOException{
      double analogRead=0;
      switch (number){
         case "0": analogRead = analogInput0.read();
         break;
         case "1": analogRead = analogInput1.read();
         break;
         case "2": analogRead = analogInput2.read();
         break;
         case "3": analogRead = analogInput3.read();
         break;
      }
      return analogRead;
   }
   public void writePWMDuty(double duty, String name) throws IOException{

      switch (name){
         case "PWMDuty1": pwm1.setDuty(duty);
         break;
         case "PWMDuty2": pwm2.setDuty(duty);
         break;
         case "PWMDuty3": pwm3.setDuty(duty);
         break;
         case "PWMDuty4": pwm4.setDuty(duty);
         break;
      }
   }
   public void writeServoAngle(double angle, String name) throws
   IOException{

      switch (name){
         case "ServoAngle1": servo1.setAngle(angle);
         break;
         case "ServoAngle2": servo2.setAngle(angle);
         break;
         case "ServoAngle3": servo3.setAngle(angle);
         break;
         case "ServoAngle4": servo4.setAngle(angle);
         break;
      }
   }
}
```

A.3 CONTENIDO DEL FICHERO *IMUDEVICE.JAVA*

A.3.1 LIBRERÍAS IMPORTADAS

```
package com.prosysopc.ua.app.beagleboneblack;

import java.io.IOException;

import org.bulldog.core.io.bus.i2c.I2cBus;
import org.bulldog.core.io.bus.i2c.I2cConnection;
import org.bulldog.core.io.bus.i2c.I2cDevice;
```

A.3.2 DECLARACIÓN DE LAS VARIABLES DE DIRECCIONAMIENTO

```
public class imuDevice extends I2cDevice{
   byte MPU9150_RA_YG_OFFS_TC = 0x01;
   byte IMU = 0x69;
   byte PWR_MGMT_1 = 0x6B;
   //byte measureMode= 0x08;

   //Acelerometro
   byte ACCEL_XOUT_H = 0x3B;
   byte ACCEL_XOUT_L = 0x3C;

   byte ACCEL_YOUT_H = 0x3D;
   byte ACCEL_YOUT_L = 0x3E;

   byte ACCEL_ZOUT_H = 0x3F;
   byte ACCEL_ZOUT_L = 0x40;

   //Giroscopo
   byte GYRO_XOUT_H = 0x43;
   byte GYRO_XOUT_L = 0x44;

   byte GYRO_YOUT_H = 0x45;
   byte GYRO_YOUT_L = 0x46;

   byte GYRO_ZOUT_H = 0x47;
   byte GYRO_ZOUT_L = 0x48;

   //magneto
   byte MPU6050_RA_YG_OFFS_TC = 0x01;
   byte MPU6050_RA_ZG_OFFS_TC = 0x02;

   byte MPU9150_RA_MAG_XOUT_L=0x03;
   byte MPU9150_RA_MAG_XOUT_H=0x04;

   byte MPU9150_RA_MAG_YOUT_L=0x05;
   byte MPU9150_RA_MAG_YOUT_H=0x06;

   byte MPU9150_RA_MAG_ZOUT_L=0x07;
   byte MPU9150_RA_MAG_ZOUT_H=0x08;

   byte MPU6050_RA_ZA_OFFS_H = 0x0A;
   byte MPU9150_RA_MAG_ADDRESS=0x0C;
   byte MPU6050_RA_INT_PIN_CFG = 0x37;
```

A.3.3 MÉTODO DE INICIALIZACIÓN

```
public void initMode(){
   try {
      getBusConnection().writeByteToRegister(PWR_MGMT_1,
      MPU9150_RA_YG_OFFS_TC); //Set clock

      getBusConnection().writeByteToRegister(MPU6050_RA_INT_PIN_CFG,MPU605
      0_RA_ZG_OFFS_TC); //set i2c bypass enable pin to true to access
      magnetometer

      getBusConnection().writeByteToRegister(MPU6050_RA_ZA_OFFS_H,MPU6050_
      RA_YG_OFFS_TC); //enable the magnetometer

   } catch (IOException e) {
      System.out.println("No se ha podido inicializar el reloj de la
      IMU.");
      e.printStackTrace();
   }
}
```

A.3.4 ACCESO AL ACELERÓMETRO

```
public void read_acc(datos3axis da){
   try {
      byte hi_val_X =
      getBusConnection().readByteFromRegister(ACCEL_XOUT_H);
      byte lo_val_X =
      getBusConnection().readByteFromRegister(ACCEL_XOUT_L);
      short vlX = (short)(((hi_val_X) << 8) | (lo_val_X));

      da.X_f = vlX * 2.0f/32768.0f;//16bit ADC, full scale range of ±2g

      byte hi_val_Y =
      getBusConnection().readByteFromRegister(ACCEL_YOUT_H);
      byte lo_val_Y =
      getBusConnection().readByteFromRegister(ACCEL_YOUT_L);
      short vlY = (short)(((hi_val_Y) << 8) | (lo_val_Y));

      da.Y_f = vlY * 2.0f/32768.0f;//16bit ADC, full scale range of ±2g

      byte hi_val_Z =
      getBusConnection().readByteFromRegister(ACCEL_ZOUT_H);
      byte lo_val_Z =
      getBusConnection().readByteFromRegister(ACCEL_ZOUT_L);
      short vlZ=(short)(((hi_val_Z) << 8) | (lo_val_Z));

      da.Z_f = vlZ * 2.0f/32768.0f;//16bit ADC, full scale range of ±2g

   } catch (IOException e) {
      System.out.println("No se ha podido leer el acelerometro.");
      e.printStackTrace();
   }
}
```

A.3.5 ACCESO AL GIROSCÓPIO

```
public void read_giro(datos3axis dg){
   try {
      byte hi_giro_X =
      getBusConnection().readByteFromRegister(GYRO_XOUT_H);
      byte lo_giro_X =
      getBusConnection().readByteFromRegister(GYRO_XOUT_L);
      short giroX = (short)(((hi_giro_X) << 8) | (lo_giro_X));

      dg.X_f = giroX * 250.0f/32768.0f;//16bit ADC, full scale range of
      ±250°/sec

      byte hi_giro_Y =
      getBusConnection().readByteFromRegister(GYRO_YOUT_H);
      byte lo_giro_Y =
      getBusConnection().readByteFromRegister(GYRO_YOUT_L);
      short giroY = (short)(((hi_giro_Y) << 8) | (lo_giro_Y));

      dg.Y_f = giroY*250.0f/32768.0f;//16bit ADC, full scale range of
      ±250°/sec

      byte hi_giro_Z =
      getBusConnection().readByteFromRegister(GYRO_ZOUT_H);
      byte lo_giro_Z =
      getBusConnection().readByteFromRegister(GYRO_ZOUT_L);
      short giroZ = (short)(((hi_giro_Z) << 8) | (lo_giro_Z));

      dg.Z_f = giroZ*250.0f/32768.0f;//16bit ADC, full scale range of
      ±250°/sec

   } catch (IOException e) {
      System.out.println("No se ha podido leer el giroscopio.");
      e.printStackTrace();
   }
}
```

A.3.6 ACCESO AL MAGNETÓMETRO

```
   public void read_magnet(datos3axis dm){

      try {
         byte magnetX_H =
         getBusConnection().readByteFromRegister(MPU9150_RA_MAG_XOUT_H);
         byte magnetX_L =
         getBusConnection().readByteFromRegister(MPU9150_RA_MAG_XOUT_L);
         short magnX = (short)(((magnetX_H) << 8) | (magnetX_L));

         dm.X_f = magnX*0.3f*1200.0f/4096.0f+18.0f; //±1200 µT por 13bit
         (0.3µT por LSB)

         byte magnetY_H =
         getBusConnection().readByteFromRegister(MPU9150_RA_MAG_YOUT_H);
         byte magnetY_L =
         getBusConnection().readByteFromRegister(MPU9150_RA_MAG_YOUT_L);
         short magnY = (short)(((magnetY_H) << 8) | (magnetY_L));

         dm.Y_f = magnY*0.3f*1200.0f/4096.0f+70.0f; //±1200 µT por 13bit
         (0.3µT por LSB)

         byte magnetZ_H =
         getBusConnection().readByteFromRegister(MPU9150_RA_MAG_ZOUT_H);
         byte magnetZ_L =
         getBusConnection().readByteFromRegister(MPU9150_RA_MAG_ZOUT_L);
         short magnZ = (short)(((magnetZ_H) << 8) | (magnetZ_L));

         dm.Z_f = magnZ*0.3f*1200.0f/4096.0f+270.0f; //±1200 µT por 13bit
         (0.3µT por LSB)

      } catch (IOException e) {
         System.out.println("No se ha podido leer el magnetometro.");
         e.printStackTrace();
      }
   }
}
```

A.4 CONTENIDO DEL FICHERO *DATOS3AXIS.JAVA*

```
package com.prosysopc.ua.app.beagleboneblack;

public class datos3axis {
   public float X_f;
   public float Y_f;
   public float Z_f;
}
```