Instituto de Computación
Facultad de Ingeniería
Universidad de la República

API de alto nivel genérica para desarrollo de aplicaciones de domótica.

Informe de Proyecto de Grado

26 de febrero del 2010
Montevideo - Uruguay

Tribunal:

Javier Baliosián, Gonzalo Tejera, Diego Vallespir

Autores:

Juan Manuel Picerno
Ken Tenzer

Tutores:

Andrés Aguirre
Ariel Sabiguero

Resumen

La domótica es el campo que relaciona la automatización, el hogar y la tecnología brindando beneficios a los usuarios de dichos sistemas teniendo en cuenta la seguridad, el confort de las personas y el ahorro energético.

Las soluciones existentes de domótica suelen estar altamente acopladas con una tecnología específica, lo cual simplifica la interacción con los distintos dispositivos y dificulta el uso de diversas tecnologías simultáneamente.

Este trabajo presenta en sus primeros capítulos el relevamiento del Estado del Arte de la domótica, para introducir al lector al concepto central que es la especificación de una API genérica de domótica. Esta API permite la programación de aplicaciones con un alto grado de portabilidad, implantarse en plataformas con escasas prestaciones y es lo suficientemente genérica para contemplar diferentes interfaces de entrada/salida, así como, protocolos de comunicación heterogéneos.

La solución propuesta está basada sobre una plataforma con una arquitectura orientada a servicios desarrollada en Java, llamada OSGi. Tiene en cuenta aspectos de seguridad, evitando dejar vulnerable la privacidad, la autenticación y la autorización sobre los diferentes componentes del sistema. La API permite interactuar con distintas tecnologías de forma homogénea, los cuales incluso pueden estar distribuidos entre varios sistemas.

Para mostrar su aplicabilidad se realizó un prototipo para la automatización de un baño en una plataforma con recursos limitados, usando el proyecto Usb4All que permite el control de dispositivos genéricos desde el puerto USB.

Palabras claves: Domótica, OSGi, Bajos recursos, JamVM, *Middlewares*, Usb4All, OpenWrt.

Agradecimientos

Agradecemos especialmente a nuestros tutores: Andrés Aguirre y Ariel Sabiguero por soportarnos durante todo el año.

Reconocemos también la grandeza de aquellas personas que nos facilitaron información:

- Jessica Díaz, "A Generic Gateway for Testing Heterogeneous Components in Acceptance Testing Tools" [15].
- Marco Aiello, "The role of web services at home" [3].
- Dario Bonino, "Domotic House Gateway" [74].
- Dario Russo, "An Open Standard Solution for Domotic Interoperability" [52].
- Larry Page y Sergey Brin por su aporte incondicional durante toda la carrera, vida profesional, y esperemos por muchos años más!

No es menor, nuestro reconocimiento a los Ingenieros Eléctricos por su ayuda en dicha área: Edgardo Vaz, Santiago Reyes, Alfonso González, Uri Tenzer y Daniel Goldenberg.

Índice general

Capítulo 1

Introducción

En este capítulo se presenta el interés por la domótica, la motivación de este trabajo, los objetivos planteados, el público al cual está dirigido este documento y una breve guía de lectura del mismo.

1.1. Motivación

Las nuevas tecnologías de la información están cambiando muchos aspectos de la sociedad. No solamente transforman el modo en que las personas se comunican sino que también tienen un gran impacto en el ambiente de trabajo y en el hogar.

Hasta hace muy poco, todas las redes de comunicación eran redes físicas independientes (como las redes de telefonía, TV cable, datos). Actualmente existe una tendencia a unificar todas las redes en una sola red multiservicios, en la cual se puede tener acceso a todos los servicios que se desee y necesite. En nuestro país, un ejemplo de este fenómeno, es la aparición del Plan Cardales que implantará el *Triple play*: servicio de televisión por cable, Internet y teléfono en forma conjunta.

Internet, revolucionó (y sigue revolucionando) las tecnologías y el ámbito social de las personas transformándose en una fuente de información de una importancia tal, que se crean nuevas redes de datos específicamente para Internet.

Por otra parte, gracias a la convergencia de contenidos audiovisuales, telecomunicaciones, comercio electrónico y el teletrabajo, se está gestionando un nuevo mundo de servicios con un extraordinario valor agregado para las empresas. En este escenario, la casa deja de ser simplemente un espacio para vivir, transformándose en un centro de comunicación que permite a sus habitantes estar permanentemente conectados con el resto del mundo. Es así que en la casa hay múltiples dispositivos específicos conectados entre sí, y a su vez se conectan con otros dispositivos en la oficina, en el auto, en la calle o en cualquier parte del mundo.

La domótica "proviene de la unión de las palabras *domus*, que significa casa en latín, y *tica*, de automática, palabra en griego, 'que funciona por sí sola"' [105]. Según el diccionario de la Real Academia Española, se trata de un conjunto de sistemas que automatizan las diferentes instalaciones de una vivienda. En "The

role of Web Services at Home" [3], se define domótica como el campo dónde las viviendas y la tecnología están entrelazadas en sus diversas formas: informática, robótica, mecánica, ergonomía y comunicación, para proveer mejores casas desde el punto de vista de la seguridad y confort. Sus campos de aplicación son diversos: se pueden encontrar sistemas que pretenden mejorar la calidad de vida de los adultos mayores, integrar la seguridad del hogar, permitir un ahorro energético inteligente, brindarle autonomía a las personas con discapacidades y sistemas que simplemente buscan atender el confort de las personas.

Mientras Internet ya no puede ser definido como un fenómeno emergente, sino como una realidad consolidada y en fuerte expansión, con la domótica no sucede lo mismo. Aunque desde hace años se hacen previsiones optimistas sobre el crecimiento de este mercado, el resultado no ha sido el esperado, tanto por los costos que involucra, como por el acercamiento de las personas al mundo de la domótica [29].

1.2. Objetivo

Considerando el marco presentado, los objetivos principales de este trabajo son realizar un relevamiento del estado del arte de la domótica y proponer una API (*Application Programming Interface*) genérica de domótica para dispositivos embebidos que permita independizar el desarrollo de las aplicaciones de una tecnología concreta.

En concreto, en la primera parte se analizan los avances tecnológicos y trabajos académicos de la domótica desde el punto de vista de la interoperabilidad de tecnologías heterogéneas.

Luego de haber realizado el estudio anterior, se definió una API genérica, de control domótico para sistemas embebidos, siguiendo las ultimas tendencias tecnologías existentes.

Debe considerarse que la información referente al estado del arte de la domótica fue relevada hasta finales del mes de abril del año 2009, por lo que, tomando en cuenta la velocidad de los avances tecnológicos, las conclusiones no necesariamente están actualizadas a la fecha de hoy.

1.3. Público Objetivo

El público objetivo de este documento son personas relacionadas al área de IT (*information and technology*), con conocimientos teóricos de sistemas embebidos, arquitecturas de computadores, seguridad informática y *firmware*. Se recomienda tener conocimientos prácticos del sistema operativo Linux. Ninguno de estos requisitos impiden al lector entender en forma general lo presentado, pero pueden dificultar su análisis y comprensión detallada.

1.4. Organización del Documento

Para facilitarle al lector su encuentro con este documento, se presenta a nivel general la estructura del mismo y una breve guía de lectura.

1.4.1. Estructura

Este documento está dividido en dos partes: la primera consiste en la presentación del informe, donde aparecen los puntos más importantes del estado del arte, se detallan los distintos experimentos, el diseño de la solución propuesta, y las conclusiones extraídas. En la segunda parte, se encuentran los apéndices que exponen en profundidad, complementan y justifican las diversas secciones del documento principal.

1.4.2. Guía de Lectura

- Las citas de artículos académicos se presentan con el siguiente formato: "título del *paper*" [referencia].
- A lo largo del documento se pueden encontrar las palabras en inglés en itálica, los nombres de archivos, comandos y nombre de funciones con estilo de fuente: "máquina de escribir".
- Las URL (*Uniform Resource Locator*) de menor relevancia, en particular los foros, se encuentran dentro del texto y las más importantes, en particular enlaces a artículos, dentro de la bibliografía. Las páginas web de acuerdo a su rol dentro del texto pueden estar incluidas o referenciadas.

Capítulo 2

Estado del Arte

En este capítulo se analiza al comienzo, los elementos físicos de la domótica, los sensores y actuadores, como también las tecnologías para su conexión y gestión.

El hecho de trabajar en un ambiente restringido en recursos encausa la elección de cualquier tecnología, ya que el uso de recursos es un factor determinante. Se presenta un estudio con dicho enfoque en cuánto a sistemas operativos y lenguajes de programación los que servirán de base para el sistema a diseñar.

Por último se hace un análisis de trabajos previos relacionados con la interconexión de protocolos heterogéneos en redes domóticas.

2.1. Sensores y Actuadores

Según "*Abstract virtual reality for teaching domotics*" [54], los sensores, también llamados receptores, son los elementos que reciben información del medio ambiente (como la cantidad de luz) o acciones humanas (como presionar un botón). Por otra parte los actuadores, son los dispositivos físicos con los cuales se puede actuar, o generar un cambio en el entorno.

La domótica resuelve, entre unos de sus cometidos, el problema de realizar las acciones cotidianas de un hogar en forma automática. Para eso se desarrolla *software* que interactúa con el mundo físico. En este sentido, por medio de los sensores, el *software* recibe información del mundo externo (percibe el entorno) y a través de los actuadores realiza acciones sobre el mismo (lo modifica). Es por eso que ambos son las herramientas básicas de la domótica.

En el apéndice A se relevan los distintos tipos de sensores, las medidas físicas que transforman y se presenta una clasificación de los mismos.

2.2. Estándares y *middlewares*

Antes de continuar comenzamos definiendo varios conceptos. Se define "protocolo" como el intercambio de datos entre dos o más partes para garantizar el entendimiento. Un "Estándar" define normas en cierto contexto [25]. Por último, según *"Home networking tecnologies and standards"* [128], la palabra *"middleware"* refiere a las tecnologías de software, arquitecturas y

sistemas que permiten la interconexión entre distintos dispositivos de *hardware* y características de redes. "Un *middleware* es un *software* de conectividad que ofrece un conjunto de servicios que hacen posible el funcionamiento de aplicaciones distribuidas sobre plataformas heterogéneas" [103].

Para que la lectura no resulte pesada, en el apéndice B se encuentra el relevamiento, junto con una breve descripción, de cada estándares y *middlewares* que de una manera u otra están relacionados con la domótica: *Home Audio Video interoperavility, IEEE 1394, RS232, Universal Serial Bus, Inter-Integrated Circuit, Serial Peripheral Interface, Bluetooth, ZigBee, Consumer Electronic Bus, Digital Living Network Alliance, KNX, LonWorks, OpenWebNet, X10, Universal Plug and Play, Open Services Gateway Initiative, Service Oriented Framework, P2P Universal Computing Consortium* (PUCC), *Zeroconf, Service Location Protocol y Jini,* entre otros de menor importancia.

Sin embargo, muchos de los protocolos y estándares mencionados son sustancialmente distintos, por lo que a continuación se muestran las diferentes categorizaciones y clasificaciones para facilitar su entendimiento, compararlos y relacionarlos entre sí.

2.2.1. Medios de Comunicación

Se muestran los estándares y protocolos listados ut supra de acuerdo al medio de comunicación que utilizan: Cableado e Inalámbrico. En el cuadro 2.1 se muestra distintos estándares y *middlewares* clasificadas en cableados y/o inalámbricos.

Para los protocolos que no implementan la capa física y de enlace (de ahora en adelante, "protocolos alto nivel"), se destaca el protocolo que utilizan para resolver este problema.

2.2.2. Interoperabilidad

La interoperabilidad, en este contexto, se define como la posibilidad de comunicación entre dos o más tecnologías. Teniendo en cuenta esta definición, en la domótica existe una gran cantidad de *bridges/gateways* (comunicación únicamente entre dos) que resuelve la interoperabilidad entre *middlewares* (M2M *Middleware-to-Middleware*).

- *Bridge* entre HAVi y Jini desarrollado por Philips, Sony y Sun (ver referencia [93]).
- *Bridge* entre UPnP y HAVi, CEBus, LonWorks o X10 (ver referencia [128]).
- *Bridge* PUCC y UPnP/DLNA o IEEE 1394 (ver referencia[69]).
- OSGi tiene *gateways* para UPnP y Jini (ver referencia [42]).

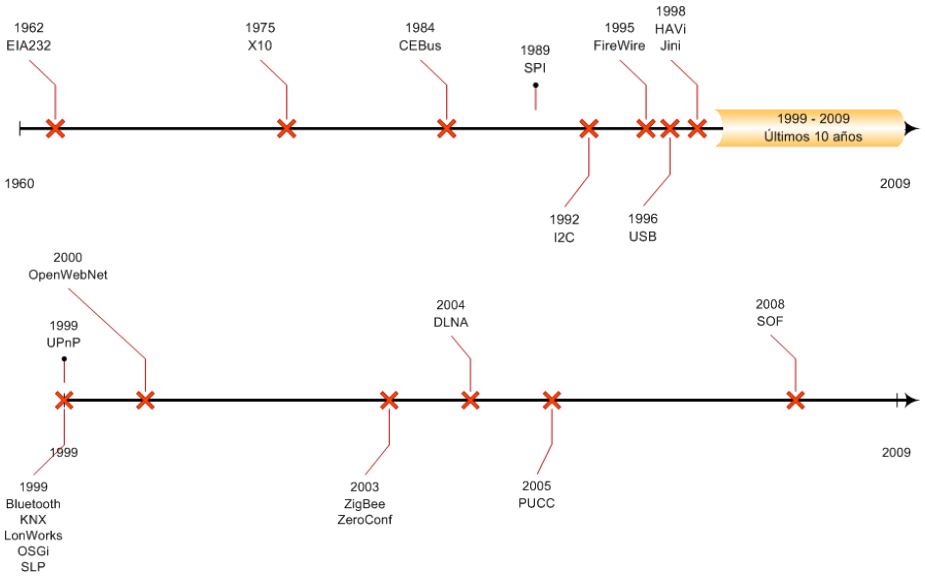
En la tabla 2.2 se sintetiza la información relevada: se marca con una cruz los *middlewares* para los cuales existen dispositivos capaces de interconectarlos. Se puede ver que existe una tendencia importante a la integración y comunicación M2M.

Estándar	**Plataforma**		
	Cableados	**Inalámbricos**	**Protocolo base**
Bluetooth		X	-
CEBus	X		-
DLNA	X	X	UPnP
HAVi	X		IEEE 1394
IEEE 1394	X		-
I^2C	X		-
KNX	X	X	-
Jini	X	X	Independiente
LonWorks	X	X	LongTalk
OpenWebNet	X	X	MyHome
OSGi	X	X	Independiente
PUCC	X	X	Independiente
RS 232	X		-
SLP	X	X	IP
SPI	X		-
UPnP	X	X	IP
USB	X		-
X10	X	X	-
Zeroconf	X	X	IP
ZigBee	X	X	-

Cuadro 2.1: Clasificación de los distintos estándares y *middlewares*.

Middleware	***Middleware***									
	HAVi	**Jini**	**UPnP**	**CEBus**	**LonWorks**	**X10**	**PUCC**	**DLNA**	**IEEE 1394**	**OSGi**
HAVi	X	X	X							
Jini	X	X								X
UPnP	X		X	X	X	X	X			X
CEBus			X	X						
LonWorks			X		X					
X10			X			X				
PUCC			X				X	X	X	
DLNA							X	X		
IEEE 1394							X		X	
OSGi		X	X							X

Cuadro 2.2: M2M relevados.



Leyenda: Se denota con una estrella las fechas exactas y con un punto las aproximadas.

Figura 2.1: Cronología de la aparición de protocolos y *middlewares*.

El problema que presenta el enfoque M2M, es que no deja de ser una solución parcial y localizada, ya que únicamente interconecta 2 tecnologías. Estas soluciones son difícilmente escalables, por ende se dificulta el desarrollo de sistemas interconectados.

Por otra parte, en el artículo "*Middlewares for Home Monitoring and Control*"[59] se destaca la existencia de *bridges* UPnP/non-IP y *gateways* como IEEE1394/IP lo cual presenta una solución un poco más genérica, ya que interconecta *middlewares* utilizados en domótica con protocolos más genéricos de red (IP).

No parece descabellado estandarizar la interoperabilidad usando como referencia el protocolo IP. Si bien eleva los requerimientos de procesamiento de la comunicación para los protocolos de menor sofisticación, se podría concluir que es es una solución simple para los de alto nivel si esta iniciativa estuviese estandarizada.

2.2.3. Cronología

Para comprender como ha sido la evolución de los protocolos y estándares, en la figura 2.1 se presentan ordenados de forma cronológica.

Se puede ver que de los 24 protocolos y *middlewares* presentados, 8 surgieron a en los primeros 40 años (se toma el comienzo del período la fecha de creación del protocolo EIA 232) y 16 en los últimos 10. Esto no es un dato estadístico ya que existe una gran cantidad de protocolos no presentados en este trabajo, y por lo tanto la muestra tomada está sesgada a los que directamente están

Sistema operativo	*Footprint*	Código fuente
FreeBSD	5 MB	*open-source*
QNX Neutrinos NTOS	-	libre para uso no comercial
OpenDomo	6,7 MB	*open-source*
Openmoko	-	*open-source*
OpenWrt	1 MB	*open-source*
SlugOS	-	*open-source*
ThreadX	2 kB	*royality free*
Windows Embedded CE	300 kB	*shared-source*
μClinux	600 kB	*open-source*

Nota: El símbolo "-" en la columna *footprint*, significa que no se han encontrado referencias al respecto.

Cuadro 2.3: Footprints y distribución del código fuente de los EOS.

vinculados a la domótica. Sin embargo, en lo que al tema concierne, se puede concluir que en los últimos años ha habido un crecimiento muy importante.

Es apreciable una concentración mayor alrededor del año 2000 (+/- 5 años) que lo que sucede actualmente. Una posible explicación es una desaceleración en la investigación, aunque es más probable que los protocolos y *middlewares* aún no hayan tomado protagonismo o no hayan sido suficientemente difundidos.

2.3. Sistemas Operativos Embebidos

Los sistemas operativos embebidos o EOS(*Embedded Operating System*) se diferencian de los no embebidos por su diseño orientado a *hardware* con bajas prestaciones, generalmente para aplicaciones de tiempo real, con alta disponibilidad, confiabilidad y recuperación ante fallas.

Para que un código máquina sea compatible con un CPU (*Central Process Unit*) es necesario que se respecten las características de esta arquitectura: *set* de instrucciones, tamaño de palabra, manejo de entrada/salida y direccionamiento de memoria. Los sistemas operativos están en contacto directo con estas particularidades, por eso deben ser generados específicamente para cada arquitectura. Los EOS suelen trabajar con *hardware* de base menos estándar que el PC común, ya que su uso responde a otros requerimientos. Por lo que una característica altamente deseable de un EOS es su portabilidad.

En el apéndice C se describe brevemente algunos EOS: Openmoko, OpenWrt, SlugOS, μClinux, OpenDomo, ThreadX, Windows Embedded CE, QNX Neutrino NTOS y BSD.

2.3.1. Comparación

En el cuadro 2.3 se muestra los *footprint* de cada EOS y la disponibilidad de sus códigos fuentes. Se puede apreciar que el hay EOS con un *footprint* de 2 kB hasta 6,7 MB (dentro de los relevados). Por otra parte es interesante ver como ningún EOS tiene todo su código propietario, probablemente para facilitar el desarrollo a bajo nivel.

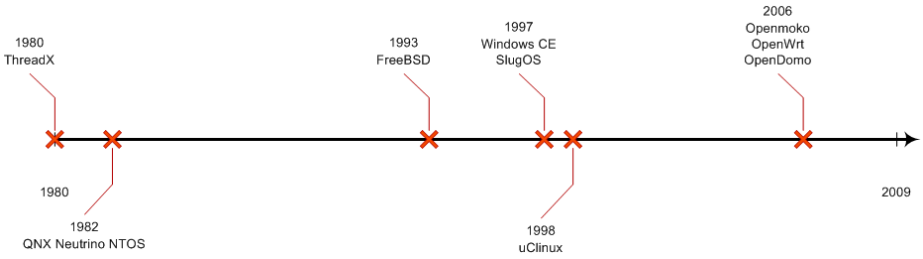


Figura 2.2: Cronología de la aparición de los EOS.

2.3.2. Cronología

En la figura 2.2 aparecen los EOS antes descritos ordenados de forma cronológica. Cada hito marca la aparición de la primera versión disponible, sin embargo, todas las distribuciones han evolucionado o incluso han sido tomadas como base para nuevas distribuciones. En particular este es el caso de distintas versiones del *kernel* de Linux que han dado vida a OpenDomo, Openmoko, OpenWrt, SlugOS, μClinux.

2.4. Lenguajes

Existen muchos lenguajes de programación, cada uno con características propias: paradigma de programación, portabilidad, *overhead*, entornos de desarrollo disponibles, capacidad de *debugging*, etc.

En el apéndice D se describen algunos de los más conocidos para dispositivos embebidos: C/C++, Python, Perl, PHP, Lua y Java.

Para poder entender cuáles son los lenguajes más aptos para el desarrollo sobre sistemas embebidos, a continuación se muestra una comparación entre los mismos. Por otra parte, se presenta la evolución de los lenguajes a lo largo del tiempo.

2.4.1. Comparación

Una posible comparación es tomar únicamente en cuenta el *footprint*. Usando esta métrica, los más apropiados son aquellos que no requieren de una VM (*Virtual Machine*): C/C++, PHP compilado y luego (en orden) Lua, Java ME (*Micro Editon*), Python (ver cuadro 2.4). En este cuadro, todos los *footprints* presentados son en base a VMs recortadas (en los casos que se utiliza VM). Una VM es recortada cuando se le quitan algunas bibliotecas y prestaciones para reducir su tamaño. Esto puede ser una limitante en tiempo de implementación ya que se podría no encontrar funciones que comúnmente vendrían incluidas en las bibliotecas de un lenguaje. En particular, Java ME y Java SE *Embedded* usan *Java Virtual Machine* (JVM) con distinto *footprint* ya que sus bibliotecas difieren.

Sin embargo, no alcanza con analizar sólo el *footprint,* esta característica es determinante, pero existen otras, como *performance* o portabilidad, que no son menores en un entorno de bajos recursos.

Lenguajes	Interpretado	Huellas de VM
C/C++	no	0 kB
PHP	*scripting* si, compilado no	-
Lua	si	64 kB
Java ME	JVM si, Java *processor* no	128 kB - 4 MB
Python	si	$<$ 200 kB
Perl	si	1 MB
Java SE *Embedded*	JVM si, Java *processor* no	$<$ 32 MB

Cuadro 2.4: Comparación de los lenguajes de programación, enfocado a dispositivos de bajos recursos.

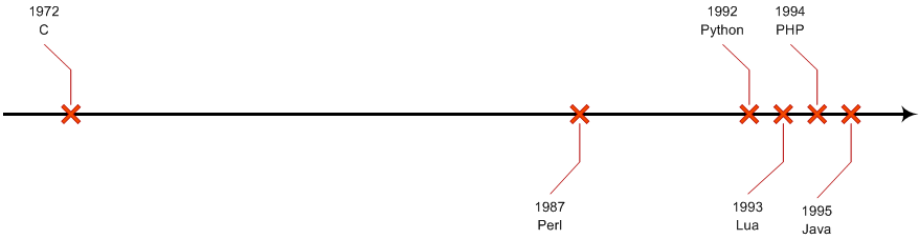


Figura 2.3: Cronología de la aparición de los lenguajes de programación.

Por lo tanto, para hacer una elección adecuada del lenguaje, es necesario conocer las posibilidades brindadas por el *hardware* así como los requerimientos de la aplicación a desarrollar.

2.4.2. Cronología

En la figura 2.3 se muestra una cronología de la aparición de los lenguajes antes mencionados. En esta imagen, todos los lenguajes aparecen en una fecha fija, sin embargo la mayoría de los lenguajes, al igual que los sistemas operativos, evolucionan tanto en la expresividad de los mismos (como el caso de C/C++), en los compiladores, sus optimizaciones (tamaño del código generado, uso de la memoria y/o cache, tiempos de compilación) y en las funcionalidades brindadas.

2.5. Trabajos Relacionados

Durante el relevamiento del estado del arte, se seleccionaron algunos artículos relevantes para el proyecto o que tienden a innovar en el área de la domótica. Estos trabajos se describen en el apéndice F. De dichos trabajos se seleccionó aquellos con *middlewares* heterogéneos (clasificación S3, S4 según ITEA: F.1).

El cuadro 2.5 muestra las ventajas y desventajas de cada una de las implementaciones de los trabajos relacionados. Esto da una buena noción de cuáles son las líneas de trabajo que se están siguiendo en este momento.

Paper	Ventajas	Desventajas
The Role of Web Services at Home [3]		-*Overhead* de los WS.
INTERACT-DDM [49]	-Poco *overhead* en el procesamiento de la información debido a SNMP.	-Centralizado.
DOG [8]	-Semantic context.	-*Overhead* de JVM (OSGi).
PUCC: PUCC Architecture [85] A Framework for Connecting Home Computing Middleware [40]	-Balance de carga. -Descubrimiento de servicios.	-falta información
Domotic House Gateway [73]	-Arquitectura flexible. -Comunicación versátil.	-*Overhead* de JVM.
Una implementación de un sistema de control domótico basada en servicios web [72]		-Hecho en ASP.NET. -*Overhead* de los WS.
DomoNet [52]	-Balance de carga. -Descubrimiento de servicios. -Arquitectura clara.	-*Overhead* de JVM. -*Overhead* de los WS.
Dynamic discovery and semantic reasoning for next generation intelligent environments [48]	-Balance de carga. -Descubrimiento de servicios. -*Semantic context.*	-*Overhead* de JVM (OSGi).
A Generic Gateway for Testing Heterogeneous Components in Acceptance Testing Tools [15]		-*Overhead* de JVM (OSGi).
A Framework for Connecting Home Computing Middleware [93]		-*Overhead* de JVM.
UHMN [55]	-Auto-configurable.	-*Overhead* de JVM.

Cuadro 2.5: Cuadro comparativo de los distintos trabajos relacionados.

Paper	Tecnología de base	Año
The Role of Web Services at Home [3]	WS	2006
INTERACT-DDM [49]	SNMP	2003
DOG [8]	OSGi	2008
PUCC Architecture [85]	PUCC	2007
Home Appliance Control Using Heterogeneous Sensor Networks [40]	PUCC	2009
Domotic House Gateway [73]	-	2006
Una implementación de un sistema de control domótico basada en servicios web [72]	WS	2008
DomoNet [52]	WS	2006
Dynamic discovery and semantic reasoning for next generation intelligent environments [48]	OSGi	2008
A Generic Gateway for Testing Heterogeneous Components in Acceptance Testing Tools [15]	OSGi	2008
A Framework for Connecting Home Computing Middleware [93]	-	2002
UHMN [55]	-	2003

Cuadro 2.6: Tecnologías de base usadas por en los distintos trabajos.

Además se realiza el cuadro comparativo 2.6 que muestra las tecnologías de base para cada uno de los trabajos antes mencionados. Otra característica de interés, presentada en este cuadro, es la fecha de publicación del trabajo. Las características de estos trabajos fueron tomadas como un punto de partida para el proyecto.

Tomando en cuenta los 12 trabajos que soportan redes heterogéneas, del cuadro 2.6 se puede concluir que OSGi, PUCC, WS, SNMP resuelven la interoperabilidad.

2.6. Interpretación

La domótica cada vez tiene un mayor protagonismo en la vida de las personas. La destacable proliferación de sistemas domóticos (ver apéndice E), protocolos y estándares de base (ver apéndice B) y una cantidad importante de trabajos de desarrollo intelectual sobre el tema (ver apéndice F) son en su conjunto, un claro indicador del desarrollo del área. También se puede considerar que esta proliferación se debe a una tendencia creciente de las personas a integrar la tecnología a su vida cotidiana por las influencias del marketing, confort y estatus asociado a la tecnología.

La variedad de sensores y actuadores que existen actualmente en el mercado

Tecnología de base	Ventajas	Desventajas
OSGi	- Plataforma específica. - Existen implementaciones *open-source*. - Implementación de sistema domótico disponible.	- Necesita JVM.
PUCC	- Plataforma específica.	- Información disponible insuficiente.
SNMP	- Simple. - Ampliamente difundido.	- MIBs distribuidos.
WS	- Ampliamente difundido.	- Costo computacional de SOAP.

Cuadro 2.7: Tecnologías de base, ventajas y desventajas.

(ver apéndice A) hace posible que el *software* interactúe prácticamente sin restricciones con el hogar. Uno puede comprar sensores que simplemente traduzcan una magnitud a otra (como el caso del termómetro de mercurio), o una cámara de video con soporte UPnP. Por lo tanto, las limitaciones parecen estar del lado de la interoperabilidad entre las soluciones específicas lo cual motiva este proyecto.

Tomando como punto de partida la interoperabilidad y los trabajos de la sección 2.5 se puede analizar el cuadro 2.7, donde para cada tecnología de base presentada en los trabajos relacionados, se comparan las ventajas y desventajas inherentes a la tecnología de base.

A pesar de las excelentes críticas a los trabajos que usan PUCC, que lo definen como una plataforma sumamente ventajosa, el gran problema de esta plataforma es que la información técnica sólo es accesible para miembros del consorcio.

Otras dos alternativas también utilizadas en problemas de domótica son: WS y SNMP. El mayor problema de SNMP es la distribución de los MIBs, lo cual requiere un previo análisis de factibilidad en el contexto de bajos recursos. Por otra parte la tendencia actual de la domótica no va en este sentido. Al analizar la factibilidad del uso WS, se puede observar que existen algunas consideraciones para disminuir el *overhead* (su mayor desventaja). Una forma para disminuir el costo de procesamiento de la comunicación es utilizar *Binary* SOAP (como kSOAP, gSOAP, Fast-SOAP, etc). Sin embargo según *"A Generic Gateway for Testing Heterogeneous Components in Acceptance Testing Tools"* [15] el uso de WS no es apropiado para ambientes de bajos recursos.

2.7. Tendencias

Tomando en cuenta las 4 tecnologías analizadas: OSGi, PUCC, SNMP y WS, en el cuadro 2.8 se agrupan los trabajos en función del año y la tecnología que utilizan.

Como el criterio de elección de esta muestra fue sesgado en principio, no es

Tecnología	2003	2006	2007	2008	2009
SNMP	1				
WS		2		1	
PUCC			1		1
OSGi				3	

Cuadro 2.8: Cantidad de trabajos usando cada tecnologías, relevado entre los trabajos relacionados.

representativo. Por lo tanto, se buscó expresamente más trabajos sobre SNMP, y PUCC (relacionados con la domótica) ya que no se poseían suficiente datos sobre estas soluciones. Sin embargo no se encontraron más *papers* al respecto. Tanto SNMP, como PUCC no parecen ser soluciones muy usadas o al menos no hay una cantidad de trabajos académicos al respecto.

Capítulo 3

Análisis

Este proyecto se centra en la interacción con *middlewares* heterogéneos, es decir, se pretende abstraerse de los protocolos utilizados por los sensores y actuadores. A partir de lo expuesto en el capítulo 2, se intentó seguir con algunas de las líneas de investigaciones en el área de la domótica.

Se decidió definir las tecnologías con las cuales se harán las pruebas de concepto, y posterior desarrollo de la API. Para esto pareció conveniente determinar: primero el *hardware* de base para el sistema, luego el sistema operativo, el lenguaje y por último, si se considera apropiado, una tecnología existente. Para esta decisión era importante tener presente que la API debía funcionar en una plataforma de *hardware* con escasas prestaciones.

3.1. Hardware

Considerando que un las hipótesis del trabajo es desarrollar en una plataforma de bajos recursos, fue necesario determinar sobre qué plataforma de *hardware* trabajar. Debido a la disponibilidad y precios en el mercado local, una buena opción para utilizar este tipo de plataformas, es utilizar un *router* cuyo *firmware* pueda ser re-programado con un EOS *open-source* (utilizar *software* *open-source* también fue una hipótesis de trabajo). Surge así, la recomendación de usar un dispositivo común como ser un *router* como plataforma de *hardware*.

Ya que existe una gran variedad de *routers* disponibles, se tomó como referencia para la capacidad de procesamiento, la capacidad del *hardware* de un *router* para uso hogareño (sin *overclocking*) y como capacidad de RAM: 32 MB.

Las opciones que se evaluaron fueron: Linksys WRT54G y Asus wl500w, sus especificaciones se pueden ver en la tabla 3.1.

Para poder tomar la decisión de qué *router* utilizar, hubo que tener en cuenta que *Usb4All* era una hipótesis de trabajo, y este *middleware* requiere de un puerto USB para su comunicación con los *baseboards*. Por lo tanto, se descartó el Linksys WRT54 por no poseer esta interfaz y se decidió el uso del *router* Asus wl500w como placa objetivo del sistema.

Característica	Router	
	Linksys WRT54G	**Asus wl500w**
Arquitectura	MIPS	MIPS
Frecuencia de reloj	125 a 200 MHz	264 MHz
Memoria RAM	16 MB	32 MB
Memoria flash	4 MB	8 MB
Puertos USB	ninguno	2 x v2.0

Cuadro 3.1: Comparación de las especificaciones técnicas de los *routers* WRT54G y Asus wl500w.

3.2. EOS

En las secciones 2.4 y 2.3 se describen posibles entornos para el *deploy* del sistema.

Un problema no menor, es encontrar un EOS que brinde todas las prestaciones necesarias sobre el cual se pueda correr el sistema, dado que terminará siendo uno de los pilares del proyecto. Una restricción, es que dicho *software* tiene que ser *open-source*. Una posibilidad es usar OpenWrt, una distribución de Linux orientada a dispositivos embebidos, completamente *open-source*. La misma permite elegir los paquetes que se incluirán en la imagen del sistema operativo pudiendo así, generar una configuración "a medida".

3.3. Lenguajes

En el caso de elegir un lenguaje interpretado, uno de los factores más importantes a tener en cuenta es el *footprint*. Si bien una VM presenta un costo fijo en dicho aspecto, en el caso de Java, es posible "eliminarlo" usando un procesador capaz de procesar directamente *bytecode*. Teniendo en cuenta que la solución debe ser genérica, esta alternativa se plantea simplemente a modo de completar el análisis hecho sobre la elección del lenguaje.

Otra particularidad de usar un lenguaje interpretado es que la VM deberá ser compatible con la arquitectura de la plataforma de *hardware* y del EOS. Por lo tanto, la existencia de VMs portables a distintas plataformas hace que su lenguaje sea independiente de la arquitectura.

De los estudios preliminares, considerando el *footprint* y la portabilidad, se pudo afirmar que cualquiera de los lenguajes descritos en la sección 2.4 podrían usarse para la solución. Ya que el *footprint* no es la única característica determinante es necesario hacer pruebas de factibilidad o de concepto para eliminar riesgos técnicos.

3.4. Deploy

En la sección 2.2 se presenta una cantidad significativa de estándares y *middlewares* utilizados en domótica. Sin embargo, dadas las características del proyecto, se dio más énfasis a los protocolos de más alto nivel ya que el enfoque

de este proyecto pretende abstraerse de las diferentes tecnologías para lograr una API capaz de interactuar con las diferentes *middlewares*.

Considerando los trabajos relacionados (ver sección 2.5) el uso de OSGi, PUCC, SNMP y WS resuelve el problema que se desea atacar. Según lo expuesto en la sección 2.6, el uso de WS no es apropiado para plataformas de bajos recursos. Si se toma en cuenta lo presentado en el estado del arte (ver sección 2.7), SNMP parece haber sido una solución circunstancial y no es trascendente en la tendencia actual de la domótica. En cuanto a PUCC, no se poseen los detalles técnicos de esta tecnología y solamente están disponibles para miembros del consorcio.

Examinando el cuadro 2.7, que se encuentra en el capítulo anterior, que compara las ventajas y desventajas de trabajos sobre distintas plataformas en cuanto a la generalidad de la solución, portabilidad e interacción, todo parece apuntar a OSGi. Sin embargo, con la información disponible, no era claro que esta opción sea compatible en un entorno con recursos limitados, ya que requiere hacer el *deploy* de la JVM en conjunto con los paquetes usados por OSGi en el *router*.

A continuación se hace una breve reseña de OSGi, ya que resultó ser una de las tecnologías más destacadas para el proyecto.

3.4.1. OSGi

OSGi es una plataforma de servicios especificada por OSGi Alliance [66]. Define una arquitectura orientada a servicios (SOA - *Service-Oriented Architecture*) para proveer servicios de forma estandarizada. Al ser de especificación abierta, potencia la creación de nuevos servicios y aumenta la colaboración entre los mismos. El *framework* se centra en gestionar el ciclo de vida de las aplicaciones: publicación, registro y acceso a los servicios. De esta forma, OSGi es una tecnología que permite agregar funcionalidades dentro de una aplicación, con bajo acoplamiento a su entorno.

Un ejemplo de OSGi es Eclipse (IDE de programación), el cual está desarrollado sobre está tecnología y permiten extender y personalizar de manera muy sencilla sus funcionalidades en tiempo de ejecución (ver apéndice G).

Tal como se expone en el apéndice G.6, el *framework* ha tenido una evolución sostenida durante casi 10 años y sigue creciendo. El hecho de que no haya desaparecido, ni haya sido remplazado y que se siga adaptando a los nuevos requerimientos, hace pensar que OSGi se ha ganado un lugar en el mercado de *software* y no es simplemente una curiosidad tecnológica. A su vez, la existencia de trabajos académicos en el área de domótica basados en OSGi (ver apéndice H) sugiere que se seguirán dando avances sobre esta plataforma.

Considerando cada actuador y sensor (o sus operaciones) como servicios, una plataforma SOA como OSGi, permite publicar estos servicios en un entorno controlado. En este sentido, OSGi parece ser una plataforma apropiada para domótica, más aún tomando en cuenta los trabajos relacionados que utilizan dicha plataforma: DOG [8],"*Dynamic discovery and semantic reasoning for next generation intelligent environments*" [48] y "*A Generic Gateway for Testing Heterogeneous Components in Acceptance Testing Tools*" [15].

3.5. JVM

Utilizando como punto de partida la tendencia por OSGi, el próximo paso fue buscar antecedentes de casos exitosos de implantación de Java (y si es posible de OSGi) en ambientes de bajos recursos. Algunos se pueden ver en:

- Concierge + Mika + Linksys WRT54GS
  `http://concierge.sourceforge.net/platforms/mika.html`
- JamVM + GNU-Classpath + Linksys WRT54G
  `https://forum.openwrt.org/viewtopic.php?id=16445`
- Knopflerfish + JamVm + sablevm-classpath-full + Asus WL-500d Deluxe
  `https://forum.openwrt.org/viewtopic.php?id=6991`

Ya que es posible utilizar OSGi y JVM en un *router*, se decidió usar OSGi quedando pendiente qué implementación de OSGi y JVM utilizar. De acuerdo al relevamiento hecho, las JVM más utilizadas para estos ambientes son JamVM y Mika. Estas dos alternativas difieren radicalmente en las bibliotecas de Java que utilizan (*Classpath*).

JamVM se puede utilizar con cualquier *Classpath* y en particular con GNU-Classpath. Su versatilidad permite proveerle a la JVM un sub-conjunto de estas bibliotecas. De esta forma, se puede compilar una distribución de Java "a medida" pudiendo eliminar las bibliotecas que no se usen (en particular, para una implantación en un *router*, a priori, las bibliotecas gráficas no son necesarias). Por otra parte, en el artículo *"Java VMs Compared"* [82] se destaca la *performance* de OSGi con JamVM.

Mika, por su parte, trae sus propias bibliotecas, lo que podría llegar a ser un riesgo en lo que a su confiabilidad respecta (ver 4.2.2). Su gran ventaja es que fue portado a MIPS y en particular se utilizó OpenWrt Whiterussian.

Dadas estas características, se decidió comenzar portando JamVM, dado que es más adecuada para ambientes de bajos recursos, y en caso de no ser posible, usar Mika.

3.6. Interconexión

Luego de haber elegido el *framework* OSGi, se buscaron distintas alternativas de distribución o publicación de servicios entre distintas instancias del *framework*.

Se encontraron las siguientes soluciones para *remoting* (o interconexión con objetos de otra JVM) dentro de OSGi:

SOAP: Knopflerfish tiene varias implementaciones de WS (cliente y servidor) disponibles en forma de *bundles*. En particular, kSOAP (cliente de *web service* para *binary* SOAP) está enfocada para aplicaciones con recursos limitados y JVMs reducidas.

Remote Framework: es un conjunto de *bundles* provistos por Knopflerfish que permite controlar otro *framework* de forma remota, con una arquitectura cliente-servidor. Se basa en conexiones kSOAP por lo que su arquitectura es costosa en recursos. Por otra parte precisa de AWT

(*Abstract Window Toolkit*) de Java para cargar el *bundle* "*desktop*". Esta biblioteca no sería normalmente necesaria en el *router* ya que no tiene interfaz gráfica.
Haciendo pruebas se encontró un *bug* el cual fue reportado y arreglado `http://sourceforge.net/forum/message.php?msg_id=7431604`.

RMI *bundle*: en "*Remote services in osgi framework using rmi connection*"[41] se encuentra publicada un solución con "*remote framework*" pero usando RMI (*Remote Method Invocation*) cuya *performance* es mejor que la de SOAP.

R-OSGi: es un conjunto de *bundles* publicados por ETH (Eidgenössische Technische Hochschule) Zürich [26] que permite trabajar con *bundles* distribuidos en distintos *frameworks* de forma transparente generando un *Proxy* local que simula el servicio remoto.

jSLP: es una implementación de SLP en Java puro. Permite el descubrimiento de servicios mediante *multicast* y la conexión entre los *User Agent* y SLP (*Service Location Protocol*) *Service Agent* por medio de *unicast*. Por más información, ver "jSLP"[27].

Jini: es una API desarrollada por Sun Microsystems que facilita el desarrollo de aplicaciones distribuidas. Internamente trabaja sobre RMI, por más información ver: la sección B.7.

DOSGi: Distributed OSGi es un componente que aparecerá en la especificación OSGi 4.2 bajo el ECF RFC 119 D-OSGi. En la conferencia anual de Eclipse de marzo 2009 se dieron a conocer los avances de este proyecto [81].

HTTP *bundles*: en los OBR (OSGi Bundle Repository) de cada distribución de OSGi, se encuentran distintos *bundles* que proveen una implementación de *web servers* como un servicio de OSGi.

En el *paper* "*Distributed Applications through Software Modularization*" [79], se explica que R-OSGi es más eficiente que RMI (en la mayoría de los casos) y dos ordenes de magnitud más eficiente que UPnP. Según está publicado, R-OSGi utiliza un protocolo de red eficiente, un *footprint* pequeño y es ideal para dispositivos embebidos. Por lo cual se entendió que era la más adecuada para un proyecto de estas características. A continuación se explica más en detalle este protocolo.

3.6.1. R-OSGi

R-OSGi se encarga de distribuir los servicios entre distintos nodos, basadándose, en el envío de mensajes binarios utilizando la biblioteca NIO(*Non blocking Input Output*) sobre TCP. Además provee la interfaz de descubrimiento de *hosts* pero la implementación es delegada a otros proyectos como jSLP. En particular, ETH tiene 2 implementaciones de descubrimiento de servicios: *Bluetooth Service Discovery Protocol* (SDP) y jSLP (que cumple con el RFC 2608).

R-OSGi puede usarse con y sin descubrimiento de servicios. La forma más simple, consta de publicar un servicio con una interfaz conocida en una URI

(Uniform Resource Identifier, por ejemplo: "`r-osgi://192.168.1.1:9278`"), el consumidor puede usar dicho servicio simplemente conociendo la URI y la interfaz. En todos los casos, el publicador debe publicar el servicio con la propiedad `R_OSGi_REGISTRATION`, de lo contrario el servicio no podrá ser consumido.

El *bundle* de jSLP implementa el descubrimiento de servicios definido en la interfaz de R-OSGi por lo que simplemente hay que definir en qué dirección de *multicast* se publicarán los *hosts*.

El proyecto jSLP que originalmente también fue desarrollado por ETH, fue recientemente donado al proyecto Eclipse ECF(*Eclipse Communication Framework Project*) por lo cual, en este momento, se encuentra en una etapa de transición.

3.7. Middleware

Uno de los requerimientos definidos para la API es interactuar con diferentes *middlewares*, por lo tanto su definición debe ser independiente de las capas que controlen la información relativa a la comunicación con el *hardware*. Estos aspectos son resueltos por cada manejador de la tecnología específica.

Para la definición de la API se podría elegir el *middleware* de forma arbitraria, usando por ejemplo, elementos simples como un puerto paralelo para interactuar con sensores y actuadores directamente conectados en los pines de entrada y salida. Mapeando el puerto a una dirección de entrada o salida a un espacio de memoria las operaciones de *read* y *write* serían los comandos de comunicación con los elementos físicos. Sin embargo, enfocar el trabajo en un *middleware* tan simplificado podría sesgar la definición de la API, no considerando casos como lecturas bloqueantes, *streaming* u otros.

Una hipótesis de trabajo fue usar Usb4All [1] como *middleware* de referencia, lo que se justifica por: su portabilidad, genericidad y también para la reutilización de un proyecto de grado del In.Co. En este escenario el Usb4All tendría interacción directa con los sensores y actuadores, mientras que el sistema se comunicaría utilizando un manejador del mismo.

3.8. Resumen

Sintetizando las ideas del análisis previo, se destacan las tecnologías (*softwares* y *middlewares*) con los que se comenzaron las pruebas de concepto para el desarrollo de la API, arquitectura de la solución y prototipos.

Como dispositivo físico (o *hardware*): se resolvió usar como plataforma de *hardware* un *router* Asus wl500w, al cual se determinó re-programar su *firmware* con el EOS OpenWrt.

Se eligió Java como Lenguaje de programación, quedando pendiente la experimentación con dos JVM: JamVM en conjunto de GNU-Classpath y como segunda opción Mika. Esta elección posibilita el uso de OSGi como *Framework* de base.

Se decidió utilizar R-OSGi en conjunto con jSLP para lograr la distribución e interconexión del sistema.

Finalmente, como *middleware* se utilizó Usb4All, el cual fue una hipótesis de trabajo.

Si bien, por lo presentado en 3.5, se sabía que la configuración Java + OSGi podía implantarse en el ambiente deseado, era preciso verificarlo y comprobar la factibilidad de la elección todas las tecnologías en su conjunto. Para esto, no se tenían datos suficientes en cuanto al *footprint*, tiempo computacional, y otros factores determinantes, por lo que fue inminente hacer una prueba de factibilidad tanto de cada componente elegido como de la solución en su conjunto.

Capítulo 4

Experimentos y Resultados

Dado las conclusiones y recomendaciones obtenidas sobre las líneas de trabajo alcanzadas como resultado de la investigación y posterior análisis del estado del arte, en este capítulo se detallan las pruebas realizadas con los sistemas operativos, JVMs, *remoting* y *middlewares* más relevantes para este proyecto.

Unos de los problemas enfrentados fue obtener un ambiente de desarrollo estable y lo más actualizado posible. Para ello se armó una máquina virtual de compilación donde se generaron: las diferentes imágenes del sistema operativo, los paquetes utilizados y los programas que se cross-compilaron.

4.1. Sistema Operativo

Como se mencionó en el capítulo 3 se eligió OpenWrt como EOS, en esta sección se explican las distintas pruebas realizadas, y alternativas consideradas con respecto a dicho EOS.

En las primeras etapas del proyecto no se contaba con la placa objetivo para poder realizar las pruebas de concepto, por lo tanto, se intentó utilizar la *Virtual Machine* qEmu para poder emular la arquitectura MIPS dentro del PC, pudiendo así validar las pruebas de concepto. Sin embargo, el emulador de MIPS de qEmu presentaba fallas, algunos de los errores encontrados ya fueron arreglados, ver `https://forum.openwrt.org/viewtopic.php?pid=92868`. Dado que no se pudo utilizar dicho emulador, se hicieron las pruebas directamente sobre una arquitectura x86 hasta tener acceso a la placa.

Una de las primeras pruebas que se hizo fue compilar la versión *OpenWrt 8.09* con kernel 2.4 Broadcom la cual fue la más utilizada en los inicios del proyecto. Luego, se experimentó con la configuración ideal considerando tener los últimos avances y la mayor estabilidad posible: la última versión estable del sistema operativo: *Kamikaze* 8.09.1 (r:16278) y el último *kernel* disponible: 2.6.

Sin embargo, al re-programar esta imagen en la plataforma de *hardware* la misma no booteaba correctamente. Al no disponer de una consola serial para diagnosticar la causa del error, se dejó de lado y se probó la última revisión del repositorio (en su momento) r:17438 con kernel 2.6 la cual funcionó correctamente, por lo tanto se continuó con esta última configuración.

Para tener mayor flexibilidad en cuanto al tamaño del código de las pruebas,

se le agregó un *pendrive* USB a la placa objetivo, dejando invariante la memoria flash de la placa (por más información ver: `http://oldwiki.openwrt.org/HowToInstallOntoUsbDrive.html`).

4.2. JVM

Por lo concluido en el capítulo 3, los esfuerzos iniciales se dedicaron en probar la factibilidad de cross-compilar una JVM e instalarla en la placa. Esto permitiría poder ejecutar aplicaciones Java en la placa. Las pruebas se realizaron con las dos implementaciones mencionadas: JamVM y Mika.

4.2.1. JamVM

Entre las primeras pruebas realizadas, estuvo la compilación de JamVM, junto con GNU-Classpath, para la arquitectura x86, lo cual se logró siguiendo los pasos descritos en `http://sourceforge.net/forum/forum.php?thread_id=3208847&forum_id=256481`. Un detalle no menor es que para compilar JamVM es necesario tener otra JVM y un *pre-build classpath*.

Al comienzo de las pruebas no se tuvo ningún problema en obtener JamVM compilado para MIPS con las librerías del sistema operativo de la placa, ya que en los repositorios de todas las versiones de OpenWrt se encuentra el paquete con los binarios de JamVM, con lo cual, no fue necesario cross-compilar JamVM. Sin embargo, como JamVM necesita un *classpath,* la compilación del mismo para MIPS tuvo sus complicaciones.

Para la utilización de JamVM en conjunto con GNU-Classpath se encontraron dificultades. La elección de las versiones de los mismos (dado que no todas las versiones de GNU-Classpath son compatibles con las versiones de JamVM), la elección de los compiladores y la cross-compilación del código para MIPS. A continuación se describen las diferentes pruebas realizadas y los resultados obtenidos.

4.2.1.1. Compilación de JamVM y GNU-Classpath para x86

Para las primeras pruebas se utilizó JamVM 1.5.2 y GNU-Classpath 0.9.7. En este caso el mayor problema fue la elección del compilador de Java para la compilación de las clases del Classpath. Se probó la compilación con GNU JDK 1.5, GNU JDK 1.6, el compilador de Eclipse ECJ, Jikes y Sun JDK 1.6 los cuales no dieron buenos resultados.

Luego se probó utilizar JamVM 1.5.3 y GNU-Classpath 0.97 con los mismos compiladores que en las pruebas anteriores. Utilizando Sun JDK 1.6 se logró el cometido y se documentó esta solución, pudiendo replicarla siguiendo el *mini-HowTo* que se encuentra en el apéndice K.

Por último, también se probó con los últimos fuentes de JamVM, pero considerando la estabilidad requerida, no pareció adecuado seguir con esta versión.

Dado que satisfactoriamente se logró lo propuesto para x86, se decidió realizar pruebas similares utilizando el entorno de la placa.

4.2.1.2. Cross-compilación de JamVM y GNU-Classpath para OpenWrt sobre MIPS

El camino para resolver la cross-compilación fue más sinuoso de lo esperado, principalmente porque no se tenía una guía de los pasos a seguir, hubo poco apoyo en los foros y no se tenía experiencia en cross-compilación y más precisamente para OpenWrt. Inicialmente se hicieron pruebas sobre OpenWrt 8.09 con kernel 2.4 y al no obtener buenos resultados, se hicieron pruebas sobre OpenWrt *trunk* r:17438 con *kernel* 2.6.

OpenWrt 8.09 con *kernel* 2.4: En primer lugar, se instaló la versión de JamVM que se encontraba en el repositorio oficial (jamvm_1.5.0-2_mipsel.ipk), lo cual se pensó que iba a aliviar el trabajo. Sin embargo, GNU-Classpath no se encontraba en el repositorio, por lo cual se obtuvo desde `http://ipkg.nslu2-linux.org/feeds/optware/openwrt-brcm24/cross/unstable/` una versión del GNU-Classpath (classpath_0.98-1_mipsel.ipk), pero dichas versiones eran incompatibles.

Luego se instaló desde otro repositorio JamVM 1.5.3 y GNU-Classpath 0.9.8-1: se obtuvo desde `http://ipkg.nslu2-linux.org/feeds/optware/openwrt-brcm24/cross/unstable/` una versión del GNU-Classpath (classpath_0.98-1_mipsel.ipk), en la cual no hubo problemas al instalarlo, ya que se utilizo una memoria externa USB. Sin embargo no se consiguió ejecutar un simple "*Hello World*", lo cual se puede ver en el siguiente *POST* `http://www.nabble.com/UnsatisfiedLinkError-JamVM1.5.3-GNU-Classpath-0.98-1-on-mips-td25311909.html`.

Finalmente se decidió probar la compilación de GNU-Classpath 0.9.8 con el *toolchain* (programas, *scripts* y bibliotecas para compilar, cross-compilar y linkeditar) de OpenWrt 8.09 y utilizar la JamVM que se encontraba en el repositorio oficial (jamvm_1.5.0-2_mipsel.ipk). Se encontró, que por defecto, el "*configure*" de GNU-Classpath utiliza el *path* de los compiladores, *linkers* y bibliotecas de la máquina donde se ejecuta el *script*. Esto hace que genere los binarios para una arquitectura incorrecta. Luego de modificar los *Makefiles*, al correr los binarios sobre la placa, se encontraron incompatibilidades entre las versiones del Classpath y JamVM.

OpenWrt *trunk* r:17438 con *kernel* 2.6: La principal motivación para cambiar la versión de OpenWrt fue que en esta revisión, se encuentra un *classpath* para poder compilarlo junto con la compilación del sistema operativo o usando el SDK. Sin embargo, al copiar el *classpath* generado a la placa y usarlo junto a la versión de JamVM disponible en el repositorio oficial, también se obtuvo malos resultados.

Finalmente, se comprobó que las versiones de JamVM y GNU-Classpath presentaban serias incompatibilidades y que las pruebas fallidas no sólo se debían a errores en los procedimientos efectuados. Se descubrió que para hacer la correcta instalación, el principal problema es encontrar (o fabricar) versiones compatibles entre JamVM y GNU-Classpath las cuales se pueden ver en la tabla 4.1. Luego de encontrar la causa de las pruebas fallidas: la incompatibilidad entre las versiones de JamVM con GNU-Classpath, se pudo ejecutarlos en la placa. El procedimiento de cómo cross-compilar satisfactoriamente JamVM junto con GNU-Classpath se encuentra documentado en el apéndice L.

JamVM	Classpath		
	0.97.1	**0.97.2**	**0.98**
1.5.1	compatibles	incompatibles	incompatibles
1.5.2	incompatibles	incompatibles	incompatibles
1.5.3	incompatibles	incompatibles	compatibles

Cuadro 4.1: Compatibilidad de versiones.

4.2.1.3. Diferencias con Java-Sun

Al estar desarrollando sobre diferentes JVM y *classpath* se encontraron dos diferencias entre el funcionamiento de la JVM de Sun y JamVM GNU-Classpath.

- Manejo de propiedades:
  El método `Java.util.Properties stringPropertyNames()` que se encuentra en la versión distribuida por Sun no se encuentra en GNU-Classpath.
- Comparación entre certificados de archivos JARs:
  Aunque en la especificación de Sun, la comparación entre los objetos se tiene que realizar mediante la función `equals()`, en la versión de Sun los certificados se pueden comparar mediante el operador "==", sin embargo esto no funciona con JamVM y GNU-Classpath.

Se puede ver que estas diferencias son menores y que el impacto en el pasaje de un entorno con una JVM a otro, no fue significativo. Sin embargo, estas diferencias son la prueba y la justificación de el modo de trabajo que se utilizó, así como parte de las dificultades encontradas (ver sección 7.2).

4.2.2. Mika

Mika no se eligió como primera opción, principalmente porque se encontraron varios indicios negativos en forma temprana:

- Mika no funciona en forma nativa sobre arquitectura de 64 bit. `http://forums.k-embedded-Java.com/forum/discussion.php?webtag=GENERAL&msg=3.1`
- Los *links* de la página oficial para descargar los binarios estaban desactualizados. `http://forums.k-embedded-Java.com/forum/discussion.php?webtag=GENERAL&msg=4.1`

Otro indicador negativo fue la baja actividad en el foro, reflejo de poca popularidad.

Sin embargo, dados los inconvenientes encontrados en un comienzo con JamVM, se decidió proseguir experimentando con Mika.

Se encontraron *bugs* en la cross-compilación de MIPS `http://forums.k-embedded-Java.com/forum/index.php?webtag=GENERAL&msg=8.1`, los cuales se resolvieron en conjunto con desarrolladores de Mika posibilitando su *deploy* en la placa MIPS con OpenWrt.

Luego de hacer el *deploy* en la placa, se comprobó que la performance era extremadamente baja con el *kernel* 2.6 (debido a cambios recientes en Mika

relacionados con la seguridad de java, ver: `http://forums.k-embedded-java.com/forum/index.php?webtag=GENERAL&msg=8.14`), por lo que se decidió usar el *kernel* 2.4 mientras se seguía avanzando con Mika.

Finalmente se encontraron varios *bugs* importantes que impidieron el uso definitivo de Mika para el proyecto:

- La distribución de `jni.h` de Mika para MIPS no compila con g++: ver `http://forums.k-embedded-Java.com/forum/discussion.php?webtag=GENERAL&msg=10.1`, lo que impedía el uso directo de la API de Usb4All nativo en Java, ya que está escrita en C++.
- La implementación de *sockets* de Mika tiene un comportamiento no estándar y por lo tanto también distinto al de Java-Sun (independientemente de la arquitectura): ver: `http://forums.k-embedded-java.com/forum/discussion.php?webtag=GENERAL&msg=12.1`. Esto se pudo comprobar utilizando nuestros prototipos junto a Concierge RC2.

4.3. OSGi

En el artículo "*Concierge: a service platform for resource-constrained devices*" [80] se evalúan implementaciones de OSGi y se concluye que las más adecuadas para dispositivos embebidos de bajos recursos son primero Concierge y luego Knopflerfish. En este artículo, se documentan los resultados de pruebas de performance comparativas. Concierge resultó ser mejor que el resto en casi todas las métricas,en particular se destacan: menor consumo de *heap*, menor *footprint* y menor tiempo de ejecución para determinados *benchmarks.*

Las implementaciones existentes de Concierge cumplen con el *estándar* OSGi *release* 3, mientras que Knopflerfish cumple con el estándar OSGi *release* 4. En el apéndice G se detalla la especificación 4.1 y en el apéndice G.6 se pueden ver las diferencias entre la especificación 3 y la 4.1.

Concierge está orientado específicamente para dispositivos embebidos, eficientes y muy pequeños. Dado que para este proyecto hay restricciones en cuanto a los recursos de *hardware* disponibles, se optó por sacrificar funcionalidades resueltas en pro de ganar eficiencia.

Durante el uso de Concierge se encontraron serios problemas (presentados a continuación), por lo que, dado que Knopflerfish funcionaba correctamente en estos casos, se cambió a Knopflerfish. En comparación con Concierge tiene un tamaño un poco mayor (358 kB contra 86 kB, sin contar con los *bundles* adicionales), cuenta con una mayor comunidad de usuarios, mejor soporte, más documentación, y por sobretodo, para los casos probados tiene mayor estabilidad.

4.3.1. Concierge

Para esta implementación de OSGi se utilizaron dos versiones: RC2 y RC3, ambas cumplen el estándar OSGi *release* 3 por lo que algunas funcionalidades especificadas en el estándar 4 son implementadas mediante *bundles* auxiliares.

En el comienzo del proyecto, mientras no se contaba con la placa objetivo, se testeó en PC utilizando Sun Java VM, JamVM y Mika. Esta última con malos resultados, los cuales se detallan en la sección 4.2.2, por otra parte se

encontró otro problema con la versión Concierge RC3 y Mika: al intentar obtener las propiedades del sistema falla salvo que se definan en forma manual (por detalles, ver `http://sourceforge.net/projects/concierge/forums/forum/605731/topic/3389434`).

Las pruebas en PC (x86 y amd64) fueron satisfactorias, no así los prototipos en la placa, dónde se observaba que algunos servicios publicados en el *framework*, desaparecían después de unos segundos y no era posible consumirlos desde otros servicios. Luego de varias pruebas tanto en la placa como en PC, se descubrió que el problema era causado por la implementación de Concierge al limitar a menos de 64 MB la memoria máxima asignada a la JVM. Este problema imposibilita trabajar con OSGi de forma adecuada, ya que es central para el descubrimiento y consumo de servicios.

4.3.2. Knopflerfish

No fue directo diagnosticar que el problema del descubrimiento de servicios era propio de la implementación de Concierge, sin embargo una vez descubierta la causa, dado que se estaba en el un momento crítico del proyecto y que Knopflerfish funcionaba correctamente (por no tener dicho *bug* y por tener una *performance* aceptable) se decidió cambiar a Knopflerfish.

Con esta implementación se habían realizado pruebas en los comienzos del proyecto, ya que es más difundida que Concierge y por lo tanto se realizaron pruebas utilizando diversas JVM, Mika inclusive, y en ninguna se encontraron problemas propio de esta implementación de OSGi.

4.4. Remoting

Luego de haber elegido el *framework* OSGi y dada la decisión de utilizar R-OSGi (ver capítulo 3) como interconexión entre los diferentes *frameworks*, se continuó experimentando con esta tecnología.

Originalmente R-OSGi se desarrolló para la especificación 3 de OSGi y específicamente para Concierge. Actualmente se está adaptando para soportar la especificación 4 de OSGi, pero esta versión aún no fue liberada.

En primera instancia R-OSGi dio muy buenos resultados. Debido a los tiempos restringidos no se probó completamente antes de su elección. Durante el proyecto, los *bug* reportados no fueron corregidos a tiempo por parte de los desarrolladores del proyecto R-OSGi, por lo tanto para los que impedían su funcionamiento, se corrigieron sin la validación de la comunidad R-OSGi.

El primer problema que se encontró fue que R-OSGi no publicaba correctamente la interfaz de recepción o envió de cada *host*. Los servicios eran publicados en *localhost* o `127.0.0.1`, impidiendo la comunicación entre los distintos *hosts*. Como se tuvo insuficiente apoyo en el foro (ver `http://sourceforge.net/projects/r-osgi/forums/forum/533563/topic/3416196` y `http://sourceforge.net/projects/r-osgi/forums/forum/533563/topic/3299482`), se decidió modificar el código para que funcionara publicando directamente las direcciones de IP y no sus nombres. Aún después de estas modificaciones se encontraron otros *bugs*:

- Relacionado al no uso de Concierge (dado que R-OSGi estaba acoplado a dicha implementación) donde los servicios fallaban al inicializar no bien son descubiertos. Este *bug* se arregló y luego se reportó en `https://sourceforge.net/projects/r-osgi/forums/forum/533563/topic/3433383`.
- Error al intentar descubrir nuevamente un servicio, luego que un nodo fue reiniciado, reportado en `https://sourceforge.net/projects/r-osgi/forums/forum/533563/topic/3438673`.
- Error en la resolución de la interfaz de comunicación por la cual mandar los paquetes, esto se traduce en que los servicios del *router* no son publicados hacia los demás nodos (ver `https://sourceforge.net/projects/r-osgi/forums/forum/533563/topic/3438704`). Este error también causa que si dos máquinas están conectadas por dos redes distintas, los servicios no son publicados entre ambas máquinas.
- En la ultima revisión (r1285 al momento que se escribe este documento 17/10/09) falla el *autocompile* de Maven (herramienta de administración de compilaciones, ver post `http://sourceforge.net/projects/r-osgi/forums/forum/533563/topic/3422530`).

Si bien se puede ver que persiste una cantidad importantes de errores remanentes y que tienen un impacto considerable en el funcionamiento, se logró implementar un prototipo que funcione en el siguiente caso de uso:si se tiene una red con 2 *hosts*, ambos con distintos servicios, al descubrirse se publican mutuamente sus servicios y son consumibles desde su contraparte. Si a este escenario se le agregan más *hosts* seguirá funcionando correctamente, sin embargo, en caso de que alguno se reinicie, esté temporalmente inalcanzable en la red, o algún *host* esté en otra red, el prototipo no funcionará como es esperado. Por otra parte, queda pendiente el funcionamiento de R-OSGi en topologías más complejas.

4.5. Middleware

En este capítulo se detallan los *middlewares* usados para probar la API. Como pilar se usó el Usb4All, y para probar el soporte de *middleware* heterogéneos se usaron dispositivos Bechamp.

4.5.1. Usb4All

El proyecto Usb4All [1] brinda un entorno de ejecución concurrente a un conjunto de bloques lógicos (llamados *user modules*) encargados de mantener el estado y el comportamiento especifico de cada dispositivo electrónico conectado al *baseboard*.

Anteriormente al proyecto, existían al menos dos interfaces para la comunicación con la placa:

Usb4All-API: desarrollada en C++ que además posee un *wrapper* llamado Usb4AllAPIWrapper el cual publica una interfaz Java a las operaciones de la API. A su vez, el módulo Usb4All-API tiene 2 modos de acceso al bus USB: mediante un *driver* basado en libusb o mediante un *kernel module.*

Bibliotecas y Servidor TCP: desarrollados en Lua publicando una interfaz de *sockets* para la comunicación con el *baseboard*. Extiende la API original con descubrimiento de servicios de bajo nivel, hoy en día esta interfaz es utilizada para otros proyectos de bajos recursos.

4.5.1.1. Usb4All-API

Una evaluación temprana de las alternativas hizo tender la decisión a usar la Usb4All-API para mantener una solución nativa en Java, sin embargo, mientras se portaba la solución original a la placa se encontraron varios inconvenientes.

En primer lugar, la comunicación entre C++ y Java fue hecha con JNative (ver `http://sourceforge.net/projects/jnative/`) lo que facilita el desarrollo en tempranos momentos del desarrollo de un proyecto (razón por la que se usó en el proyecto Usb4All), pero dificulta la compatibilidad, portabilidad y mantenibilidad. Si se desea portar a otra arquitectura, no solamente hay que re-compilar el código C++ de la API, sino que también hay que re-compilar JNative para las diferentes arquitecturas. Por lo tanto, se modificó el código original para que usara directamente JNI (Java *Native Interface*), eliminando la dependencia con JNative.

Por otra parte, el proyecto Usb4All fue desarrollado con *Code-Blocks* (ver: `http://www.codeblocks.org/`) por lo que no poseía un Makefile. Para cross-compilar dicho proyecto, se hizo uno genérico para poder usarlo dentro del SDK de OpenWrt y otro específico para la cross-compilación fuera del SDK.

Inicialmente se quería fabricar una imagen de OpenWrt sin bibliotecas redundantes (lo más pequeña posible) por lo que no se agregaron las bibliotecas `glibc` (ver `http://www.gnu.org/software/libc/libc.html`), `libgcpp` y `libstdcpp` a la imagen de fabricada. Con el fin de utilizar únicamente `uClibc` (ver `http://www.uclibc.org/`), se modificó el modo en que se escribía la información de *log*. Sin embargo existían más dependencias, por lo que fue necesario utilizar las bibliotecas: `libgcpp` y `libstdcpp`, no así `glibc.`

Finalmente, se encontró que el código original para plataforma Linux no funciona con el *driver* basado en libusb, con lo cual era necesario cross-compilar el *kernel-module* o reimplementar la comunicación libusb para poder utilizarlo en la placa.

Colateralmente se descubrieron dos problemas que no impidieron seguir con el trabajo, pero son interesantes de resaltar.

Se descubrió un una falla en ambientes virtualizados. La comunicación con el *baseboard* al usar VMware-workstation-6.5.2, con sistema operativo Ubuntu 9.04 (en arquitectura x86), sobre un sistema de base Windows XP, presentaba fallas en la comunicación con el *bus* USB. Se verificó replicando el problema en una instalación "limpia" dentro de un ambiente emulado, y siguiendo los mismos pasos en una en una instalación "limpia" dentro de un ambiente nativo, no presentaba fallas. En particular: se recibía *timeout* al ejecutar `libusb.bulk_read` tanto con la interfaz Lua, como con la API en Java / C++ y se obtenía un error similar con el *kernel module*. Este error se puede ver reportado en: `http://www.vmwareforum.org/cgi-bin/yabb2/YaBB.pl?num=1264779580/0#0`. Una posible explicación para este error, es que estas versiones de VMware y Ubuntu son experimentalmente soportadas (ver `ttp://www.vmware.com/resources/compatibility/detail.php?device_cat=software&device_id=10723~16&release_id=70`).

Otra limitación que presenta la solución original de Usb4All es que el código no está portado para la arquitectura AMD64.

4.5.1.2. Prototipos

Dado que el funcionamiento de la solución en Lua era garantido en arquitectura MIPS con OpenWrt, se decidió su uso para la primera prototipación y mitigación de riesgos de la API. La forma en que se interconectó el sistema hecho en OSGi (Java), fue utilizando sockets TCP/IP con el servidor Lua.

4.5.2. *Middleware* Secundario

Para probar la interoperabilidad entre *middlewares,* fue necesario implementar el sistema con al menos dos distintos: el primero fue Usb4All, como segundo se eligió un dispositivo controlado por radio frecuencia (RF) para el cual se poseía un *driver* para plataforma Linux con arquitectura x86.

El sistema funciona utilizando un control conectado al puerto USB, el cual mediante radio frecuencia permite controlar los diferentes dispositivos del sistema. Para el caso concreto de prueba, se utilizo un dispositivo para prender y apagar luces.

Un inconveniente que se presentó para poder utilizar esta tecnología como protocolo en la placa, es que el fabricante no cuenta con un *driver* para MIPS y el código fuente no es entregado junto con los dispositivos. Una alternativa para resolver su portabilidad, era hacer ingeniería inversa de la comunicación USB con la placa de RF. Una vez obtenidos los comandos de cada control, se podría reproducir la comunicación creando un *driver* propio, usando por ejemplo, libusb. Para probar el uso de *middlewares* heterogéneos, no es necesario probarlo en arquitectura MIPS, por lo tanto no se portó el *driver*.

4.6. Resumen

En este capítulo se presentaron los diferentes experimentos y resultados obtenidos con las tecnologías elegidas en el capítulo anterior. A continuación se extraen los principales temas recorridos en este capítulo.

Inicialmente se intentó crear un ambiente de trabajo que emulara el ambiente objetivo, sin embargo esto no fue posible por lo que hubo que trabajar en distinta arquitectura haciendo *deploy* periódicamente para poder probar lo desarrollado.

Se consideraron y probaron ambas JVMs, inicialmente se usó Mika ya que cross-compilar JamVM llevo más tiempo que Mika. Sin embargo Mika presentaba limitaciones que impidieron su uso.

Con respecto a las implementaciones de OSGi, se trabajó hasta fases avanzadas del desarrollo con Concierge, habiendo hecho pruebas con Knopflerfish únicamente como primero contacto con OSGi. En determinado momento Concierge presentó fallas importantes y se descartó, eligiendo así, la segunda mejor opción: Knopflerfish.

En lo que a R-OSGi respecta, presentó incompatibilidades con Knoplerfish y en particular estando atado a la especificación R3 de OSGi. Además de no estar lo suficientemente maduro para un sistema de producción. Sin embargo, sí se pudo observar que tiene buenas perspectivas, resuelve el problema que se quería

atacar originalmente. Por otra parte, la donación del proyecto JSlp a eclipse hace pensar que aumentará su estabilidad por tener una comunidad de usuarios mayor.

Por último, se pudo probar dos *middlewares*, requisito mínimo para la definición de la API, generando una API para USB4All en Java independiente de la arquitectura, y usando la tecnología RF en x86.

Capítulo 5

Solución Propuesta

En este capítulo se presenta la solución de *software* para resolver los problemas planteados.

En primer lugar se explicitan y detallan los requerimientos de la solución, luego se expone la solución y las justificaciones sobre las decisiones tomadas.

Se introduce el término nodo, para nombrar a cada instancia del *framework*.

5.1. Requerimientos

En esta sección se especifican los requerimientos funcionales y no funcionales que deben estar contemplados por la API.

5.1.1. Funcionales

Los requerimientos funcionales se listarán con el formato RFX dónde X es el número de requerimiento.

Se espera que la arquitectura provea las siguientes características funcionales:

RF1 *Framework* distribuido: Comunicación transparente entre nodos.

RF2 Servicios distribuidos: Forma homogénea de trabajar con servicios tanto del propio nodo como de los nodos remotos.

RF3 Servicios colaborativos: Combinación de varios servicios en un nuevo servicio más complejo.

RF4 Nodos colaborativos: Colaboración entre distintos nodos permitiendo combinar sus servicios publicados.

RF5 Servicios inteligentes: Entorno que favorece la creación de consumidores de servicios que tomen decisiones de acuerdo a la información de los servicios presentes.

RF6 Acceso uniforme a *middlewares*: Forma única de invocación a los métodos tanto para obtener mediciones de sensores o producir cambios con los actuadores. En principio, el consumidor del servicio no tiene porqué saber cuál es la tecnología de base de los dispositivo.

5.1.2. No Funcionales

Los requerimientos no funcionales se listarán con el formato RNFX dónde X es el número de requerimiento.

Se espera que la arquitectura provea las siguientes características no funcionales:

RNF1 Seguridad:

1. El creador de un servicios puede elegir si dicho servicio puede ser consumido por cualquier servicio o si los consumidores requieren autenticación previa.
2. La API debe prevenir que se instalen servicios no autorizados y que consuman los servicios seguros de la API.

RNF2 Bajos recursos: Se usa un *router* Asus wl500w como *test-bed* de una plataforma objetivo pudiendo ampliarle la memoria flash (espacio dónde se encuentran almacenados los programas). El *framework* junto con los servicios deben poder implantarse en dicho dispositivo.

RNF3 Portabilidad: el *framework* debe ser portable entre cualquier sistema operativo y cualquier arquitectura de CPU (que cuenten con un compilador para las mismas).

RNF4 *Open-source*: el *framework,* plataformas y bibliotecas utilizadas deben ser *open-source.*

5.2. Arquitectura

Al haber realizado las pruebas y prototipos, como se explica en el capítulo 4, es factible ejecutar Java sobre una plataforma de bajos recursos y montar sobre él OSGi. Por lo tanto, el uso de esta configuración, es viable en el escenario deseado (**RNF2**).

El uso de Java permite obtener sistemas con la portabilidad requerida (**RNF3**). Por otra parte, dado que hay varias implementaciones de Java y de OSGi, open-source, cumplen con **RNF4**.

OSGi permite la flexibilidad de manejar el ciclo de vida de los servicios, potenciar la colaboración entre estos, el descubrimiento y uso de servicios de forma dinámica con bajo acoplamiento entre ellos (**RF3** y **RF5**).

La distribución y colaboración de nodos (**RF2** y **RF4**), se resuelve con R-OSGi y jSLP. La API provee un servicio de publicación remota (de ahora en más, *RemoteManager*) especializado en la colaboración entre nodos, permitiendo una forma transparente para el descubrimiento y consumo de servicios en todo el despliegue de las solución.

El acceso uniforme de los recursos de los *middleware* (**RF6**) y la seguridad de acceso a la utilización de ciertos servicios específicos de los *middlewares* (**RNF1**) están contemplados por la API (dentro del componente *SecurityManager*).

Los conceptos que se utilizan para poder entender el verdadero potencial y funcionamiento de la arquitectura y de la API propuesta son:

- Cada *middleware* requiere un *TechManager* para controlar dicha tecnología. El *TechManager* es el encargado de encapsular las características propias de la tecnología específica (Usb4All, X10, Jini, UPnP, etc) y brindar una interfaz uniforme a los servicios dentro de la API (concepto basado en DomoNet [52]).

- Los *TechManager* son los productores básicos de servicios dentro del *framework*. Los servicios que ofrecen los *TechManager* tienen que poder ser accedidos de forma controlada para poder reducir los problemas de seguridad, *security* & *safety* (ver sección 5.3). Además proveen una interfaz específica que brinda funciones administrativas para el conjunto de servicios que cada uno posee.

- Los servicios que provee el *TechManager* representan a los diferentes actuadores y sensores que el *middleware* controla. Las principales características de estos servicios es que tienen que poder ser accedidos de forma simple, uniforme y homogénea pudiéndose requerir la autenticación del invocador (mediante un servicio especializado para dicho propósito, ver sección 5.3).

- Dentro de esta arquitectura se definió el *Proxy*. Es el publicador de servicios públicos (cualquier servicio, esté autenticado o no puede consumirlo). Las responsabilidades del *Proxy* son: encapsular, componer y filtrar los servicios publicados por los *TechManager* hacia el resto de los consumidores. Aunque en el caso hipotético que no se quiera usar seguridad se podría obviar, se entiende que es importante para una solución más robusta la inclusión de este componente en la arquitectura. Esto ayuda a delimitar cuáles son los servicios privados a la solución del problema y cuales son los servicios accesibles al resto del mundo.

- Los *TechManager* tienen la posibilidad de exigir a los consumidores que estén autenticados. En el caso que lo requieran, sólo consumidores que estén autenticados podrán consumir sus servicios (ver sección 5.3).

Una de las características más importantes, aunque ya se ha comentado, a la hora de definir la arquitectura es el problema de la seguridad, por lo que en la figura 5.1 se puede observar como los diferentes componentes están interrelaciones y cuales son los componentes dentro y fuera del sistema.

A continuación se explica las funcionalidades del *framework* OSGi y las de la tecnología de R-OSGi para desarrollar luego la implementación de los conceptos presentados.

5.2.1. OSGi

Cómo primer paso se explica como es el funcionamiento de los *bundles* dentro de OSGi y luego la gestión de los servicios.

5.2.1.1. Bundles

Como se mencionó antes, y como se explica detalladamente en el apéndice G, los *bundles* son la forma en que el desarrollador puede empaquetar y distribuir

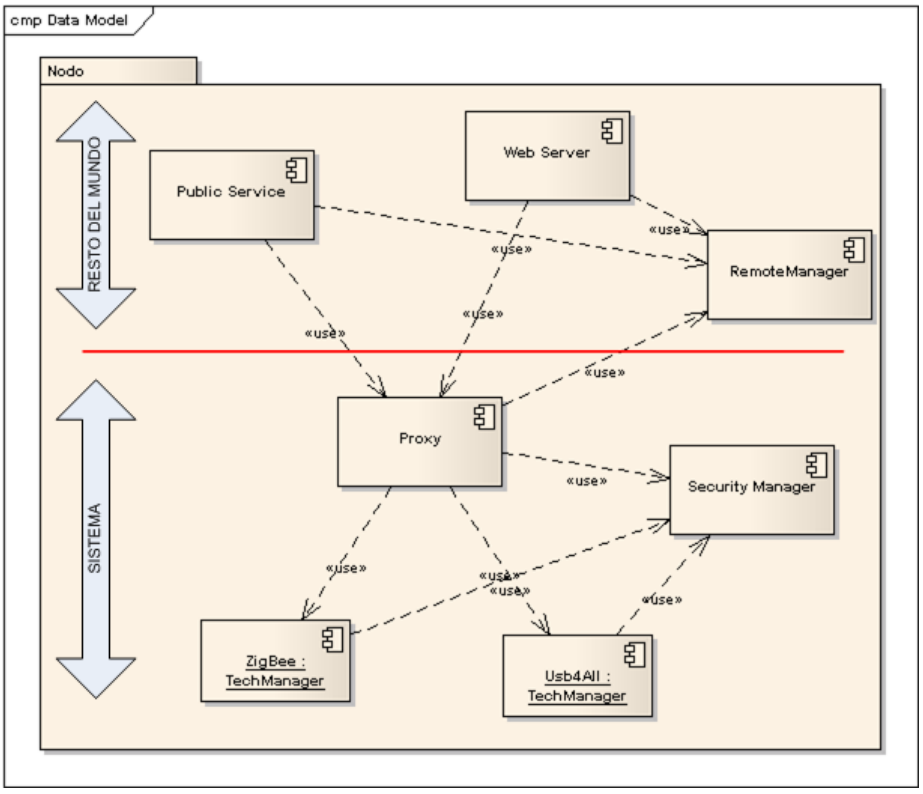


Figura 5.1: Arquitectura y separación en capas de los diferentes conceptos.

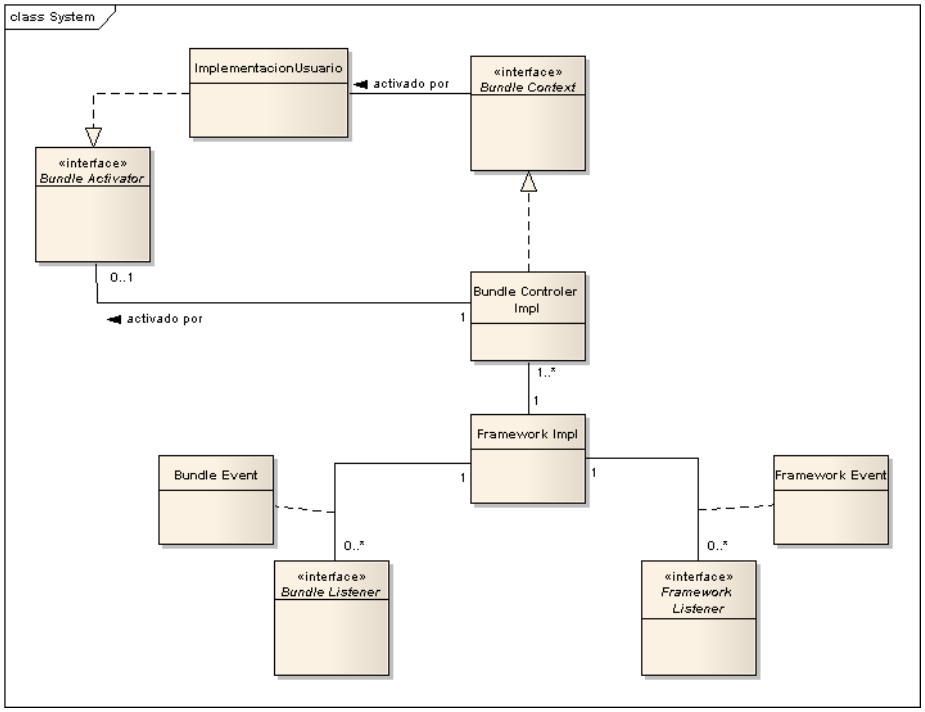


Figura 5.2: Diagrama de clases de gestión de los *bundles* en el *framework* OSGi.

las funcionalidades implementadas. Es responsabilidad del *framework* OSGi gestionar el ciclo de vida de dichos servicios.

Para poder entender el uso de los *bundles* se hace referencia a la figura 5.2 como medio de visualización de la explicación.

Los *bundles* dentro del *framework* están representados por clases, clases abstractas e interfaces. Estas son: *"Bundle Controller Impl", "Bundle Context", "Bundle Activator"* e *"ImplementacionUsuario"*. Para generar un nuevo *bundle* simplemente hay que desarrollar la clase *"ImplementacionUsuario". ImplementacionUsuario* implementa *"Bundle Activator",* los métodos que define esta interfaz, determina el comportamiento durante el ciclo de vida de este nuevo *bundle*. El ciclo de vida (que se puede ver detalladamente en G.3.1.2) está completamente gestionado por la implementación del *framework*. Por lo tanto el desarrollador simplemente debe especificar qué acciones realizará su *bundle* en las transiciones de estados.

Además del manejo de estados, el *framework* provee manejo de eventos que permite al desarrollador poder identificar, filtrar y reaccionar ante las acciones de otros *bundles* o eventos del *framework* los cuales son accesibles implementando las interfaces *"Bundle Listener" y "Framework Listener"* respectivamente.

5.2.1.2. Servicios

Cada *bundle* puede publicar varios servicios, para ello el desarrollador debe crear una o varias interfaces de acceso a dicho servicio, y la implementación de la misma. En la figura 5.3 se muestra un diagrama de clases simplificado con

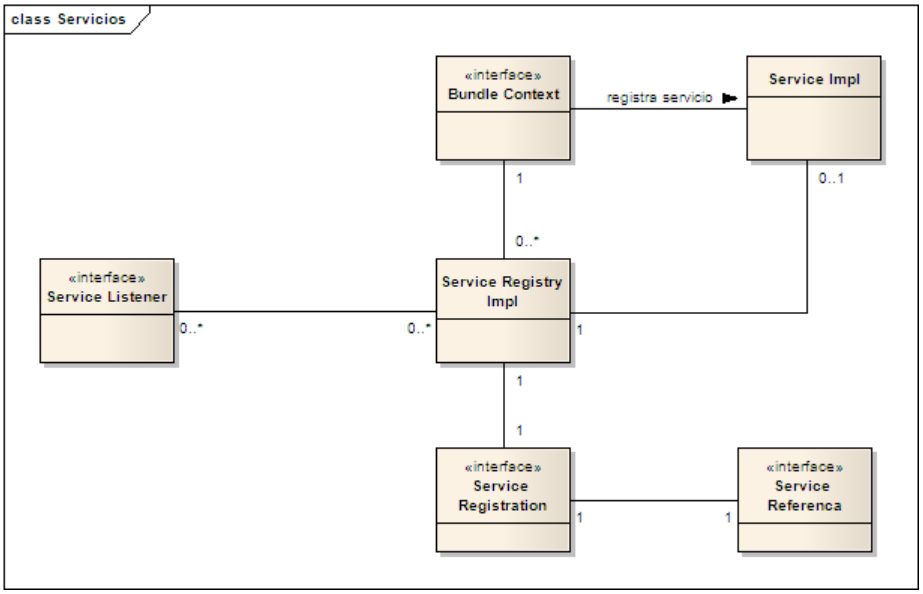


Figura 5.3: Diagrama de clases para de gestión de servicios en el *framework* OSGi.

los elementos más relevantes para el uso de servicio en OSGi.

El "*Bundle Context*" brinda la funcionalidad necesaria para publicar los servicios dentro del *framework* y que sea accesible para los consumidores. Cada vez que un servicio es publicado, el *framework* retorna un "*Service Registration*" qué es el registro de dicho servicio. El consumidor de un servicio, implementando la interfaz "*Service Listener*" recibe notificación de los cambios en el ciclo de vida de los servicios. Con cada notificación, se puede obtener un "*Service Reference*" mediante el cual es posible acceder al servicio y utilizarlo.

5.2.2. R-OSGi

Las funcionalidades de R-OSGi son provistas por medio de tres *bundles*, dos para el descubrimiento de servicios y publicación remota, y otro para el descubrimiento de *frameworks*.

El *bundle* que desea publicar los servicios simplemente debe especificar la propiedad `RemoteOSGiService.R_OSGi_REGISTRATION`, R-OSGi detecta los servicios con esta propiedad y los publica de forma remota. El cliente, debe registrar un servicio de `ServiceDiscoveryListener` con dos métodos: `announceService(...)` y `discardService(...)`, mediante los cuales el *framework* le notifica la publicación o anulación de servicios remotos. R-OSGi se encarga de generar servicios locales, por lo que el consumidor final no distingue entre un servicio local y uno remoto.

5.2.3. API

Luego de haber desarrollado los conceptos de arquitectura anteriores y la explicación de OSGi y R-OSGi, se materializaron en la solución desarrollada en

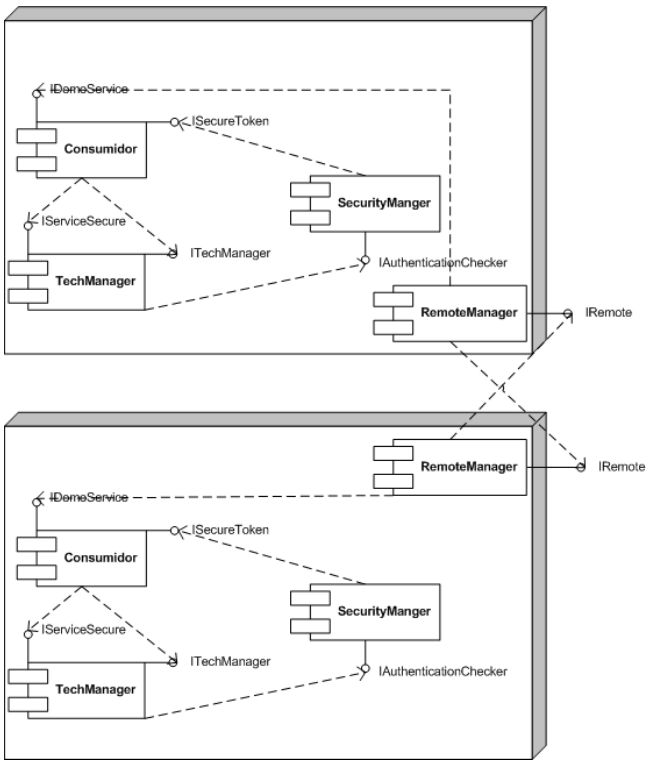


Figura 5.4: Componentes del sistema.

cinco componentes: *TechManager, SecurityManager, RemoteManager, Proxy,* Consumidores / Productores *custom.*

En la figura 5.4 se muestran dos nodos, cada uno con cuatro componentes: Consumidor (que podría ser el *Proxy, HttpBinder* u otro *Custom*), el *SecurityManager,* un *TechManager* y el *RemoteManager* junto con las dependencias de interfaces.

Por claridad, se omite el prefijo *uy.edu.fing.domo.api* en todos los paquetes, clases e interfaces de la API. Se recomienda al lector, leer el apéndice **??** para ver los detalles de cada clase.

La API desarrollada permite dos tipos de enfoques: desarrollar todas las interfaces definidas, o utilizar la API como un *framework.*

La primera opción es la más compleja para el desarrollador. Para cada subsistema de la solución, debe implementar todos las interfaces y se debe encargar no solo de la solución que desea implementar sino también de la publicación y gestión de los servicios dentro del *framework.*

La segunda opción es la recomendada, ya que al utilizar la API como *framework* el desarrollador únicamente deberá ocuparse de implementar su solución, el resto: la gestión, el descubrimiento, la obtención y seguridad de los servicios, los brinda el *framework.* Esto permite que los tiempos de desarrollo sean más cortos y que los *bundles* generados tengan un tamaño más reducido. Un ejemplo de como utilizar la API como *framework* se puede ver en "*Getting Started*" apéndice J.

5.2.3.1. TechManager

Son los productores básicos de servicios dentro del *framework*. Cada uno tiene tres responsabilidades:

1. Gestionar el *middleware* para el cual fue diseñado.
2. Publicar los servicios dentro de la API.
3. Reaccionar frente a los eventos de la API.

Para cumplir con su cometido, publica 2 tipos de servicios:

1. Servicios que representan a los dispositivos físicos, accesibles mediante la interfaz `IServiceSecure`. La estructura de la interfaz fue diseñada utilizando el concepto que define "Java Reflection", de esta forma se asegura el acceso homogéneo y simple a los servicios.
2. Servicios propios del *TechManager* implementa la interfaz `ITechManager` la cual permite realizar acciones administrativas como: la publicación, la actualización y remover la publicación de los servicios en el sistema.

Como todo *bundle* dentro de OSGi, posee un *Activator*: `TechManagerActivator` que es la interfaz de acceso al *bundle*. En la figura 6.3 se muestra el diagrama de clases de los *TechManager* implementados por la API. Cada *TechManager* debería implementar dos clases abstractas: `TechManagerActivator` y `TechManagerDomo` para poder publicar los servicios dentro de la API. `TechManagerActivator` define 4 métodos y `TechManagerDomo` define 6 más, muchos de los cuales pueden dejarse vacíos, e igual así, tener una implementación operativa.

La clase `TechManagerDomo` que implementa la interfaz `ITechManager` es la encargada de gestionar la publicación de los servicios del *middleware* en el *framework*. Esta gestión se hace de forma transparente para el desarrollador, lo que facilita su desarrollo. Un ejemplo de estas características es el uso de la autenticación. Definiendo únicamente el resultado de la función `isUsingSecurity()` definida en la interfaz `ITechManager`, el desarrollador se asegura que la API verifique o no la autenticación de los invocadores de dicho *TechManager*. Por lo tanto, el desarrollador que implementa `ITechManager`, puede elegir definiendo que la gestión a través de la requiera o no autenticación.

Además de implementar estas clases el desarrollador deberá implementar con una subclase la clase abstracta `ServiceDomoSecure` que realiza `IServiceDomoSecure`.

La interfaz `IServiceDomoSecure` es la interfaz de servicios que los *TechManager* publican para cada dispositivo de entrada o de salida presentes en el *middleware*.

Al implementar la subclase de `ServiceDomoSecure`, el implementador solamente tiene que ocuparse de definir qué funciones se utilizarán con su dispositivo en el *middleware*, implementarlas y si requiere o no autenticación al igual que en `TechManagerDomo` la validación de la autenticación esta definida en la clase padre. Al publicar estos servicios utilizando la propiedad `DOMO_API_PROXY_INSECURE`, el *Proxy* encapsula este servicio y lo publique como `IServiceDomo,` lo que permite que se puede acceder de forma controlada.

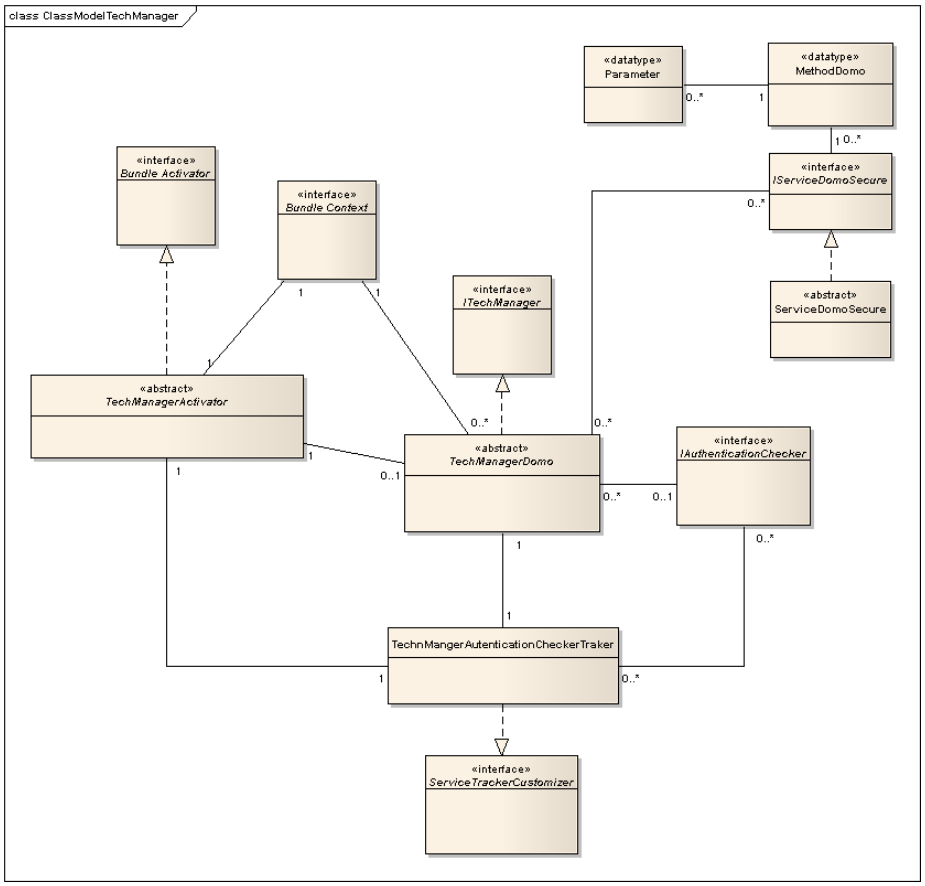


Figura 5.5: Diagrama de clases del *TechManager*.

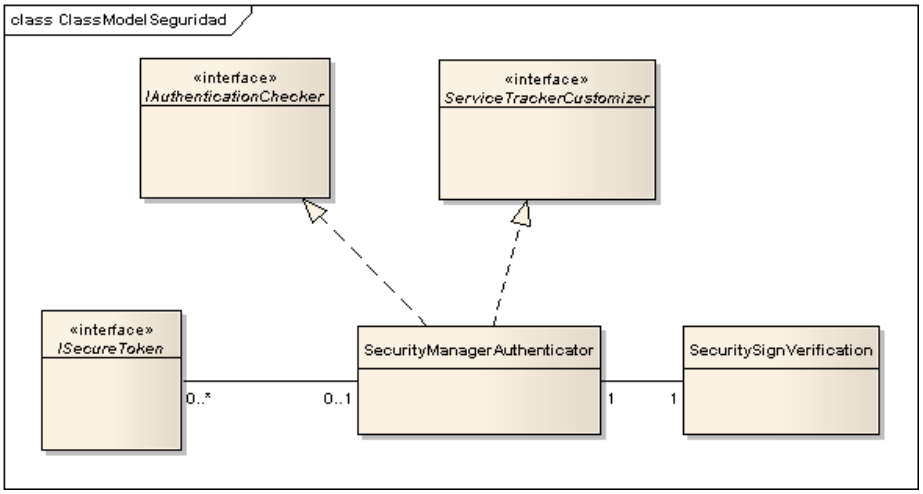


Figura 5.6: Diagrama de clases del *SecurityManager*.

La clase `TechManagerAuthenticationCheckerTracker`, la cual implementa la interfaz `org. osgi. util. tracker. ServiceTrackerCustomizer`, es utilizada para obtener el `IAuthenticationChecker` desde el *framework* y entregársela a `TechManagerDomo`. El servicio `IAuthenticationChecker` es utilizado para validar la autenticación de los invocadores.

5.2.3.2. Security Manager

Es el componente de la API responsable de la lógica de seguridad del sistema. Aunque una opción posible sería implementarlo como un *bundle* separado, para reducir vulnerabilidades, este componente está desarrollado dentro de la API. Esto permite que cada vez que se inicia la API en el *framework*, los servicios de este componente estarán disponibles para su uso.

Para tener un mejor entendimiento de las explicaciones a continuación, en la figura 5.6 se presenta un diagrama de clases reducido con las clases e interfaces involucradas.

Este componente publica un servicio con la interfaz `security. IAuthenticationChecker` para que los *TechManager* puedan validar a sus invocadores.

También dentro de este componente se encuentra un servicio de *tracker* el cual utiliza la interfaz `org. osgi. util. tracker. ServiceTrackerCustomizer` para detectar a los consumidores que implementen la interfaz `security.ISecureToken`, a los cuales le entrega el *token* para poder consumir los servicios que brindan los *TechManager*. Este tema se desarrolla más en profundidad en la sección: 5.3.

5.2.3.3. Remote Manager

Es el componente encargado de publicar los servicios entre las distintas instancias del *framework* o nodos. El corazón del componente está compuesto por R-OSGi y jSLP y `uy. edu. fing. domo. remoteManager`, los que en

conjunto brindan una forma de descubrimiento de servicios entre nodos sin necesidad de saber la ubicación ni la dirección de los otros nodos, ni conocer a priori sus servicios.

El usuario del *framework* puede abstraerse del problema de descubrimiento de nodos y servicios y centrar sus desarrollos en los servicios y no en la topología.

La API tiene la particularidad que todos los nodos son potencialmente productores y consumidores de servicios al mismo tiempo. Por esta característica de la API, el *RemoteManager* filtra los servicios publicados mediante R-OSGi que hayan sido generados en el *framework* local y se encarga de la publicación remota.

Gracias a la encapsulación de la API, definiendo la propiedad `DomoApiConstant.DOMO_API_PROXY_INSECURE`, los servicios que implementen `IServiceDomo` son automáticamente publicados de forma remota (asumiendo que las propiedades de interfaces están correctamente definidas).

El *bundle* `uy. edu. fing. domo. remoteManager` esta compuesto por una clase que implementa `BundleActivator` que utiliza otra que implementa la interfaz `ch. ethz. iks. r_osgi. service_discovery. ServiceDiscoveryListener` la cual es utilizada para filtrar y publicar en el *framework* local los servicios publicados por otros nodos.

5.2.3.4. Proxy

Es el intermediario entre servicios básicos `IServiceDomoSecure` y otros servicios; encapsulando, filtrando y componiendo servicios generando nuevo `IServiceDomo`.

Al implementar una subclase de ProxyDomo permite opcionalmente publicar de forma automática todos los servicios que estén marcados como "públicos". Estos servicios son encapsulados mediante la clase `DomoSecureWrapperPublic` y publicados como `IServiceDomo.` En este caso, la autenticación se realiza en las invocaciones del *Proxy* (intermediario) usando `ISecureToken` y no en las del consumidor final del servicio.

Además, al igual que `IServiceDomoSecure` la interfaz `IServiceDomo` utiliza el mismo concepto de "Java Reflection".

El diagrama de clases que involucra los componentes del *Proxy* se puede ver en 5.7.

5.2.3.5. Observer

Uno de los problemas encontrados a la hora de desarrollar usando la API, fue cómo hacer que el *Proxy* o *bundle* de la aplicación, obtenga la información desde los *TechManager* de forma asincrónica.

En el caso más simple, este componente invoca los servicios de los *TechManager* y/o *Proxys*. En otros escenarios es necesario obtener notificaciones de cambios generado en un *middleware* en particular, esto se resuelve usando el patrón *observer* junto con un `ServiceTracker` lo que se muestra en la figura 5.8.

5.2.3.6. Consumidores / Productores *custom*

Consumidores / Productores *custom*: son los puntos de extensión de la lógica del *framework* (a modo de referencia se provee una implementación de

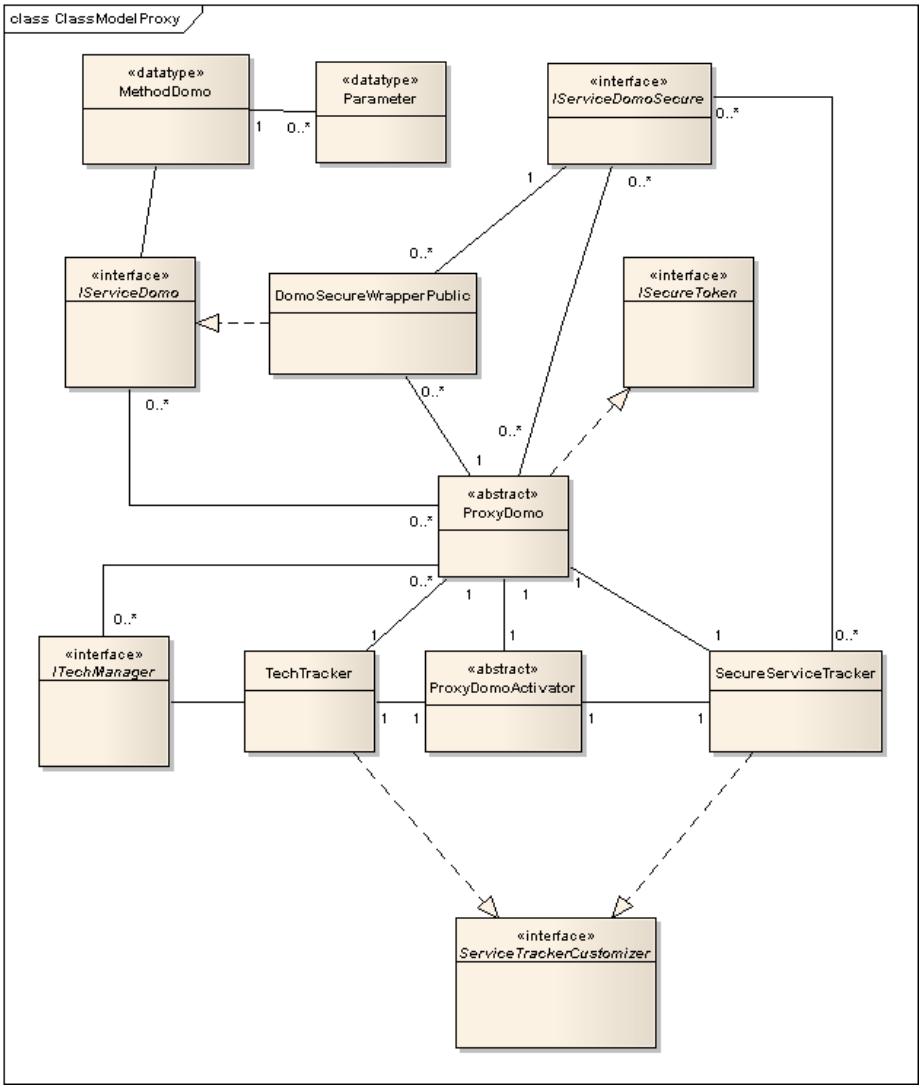


Figura 5.7: Diagrama de clases del *Proxy*.

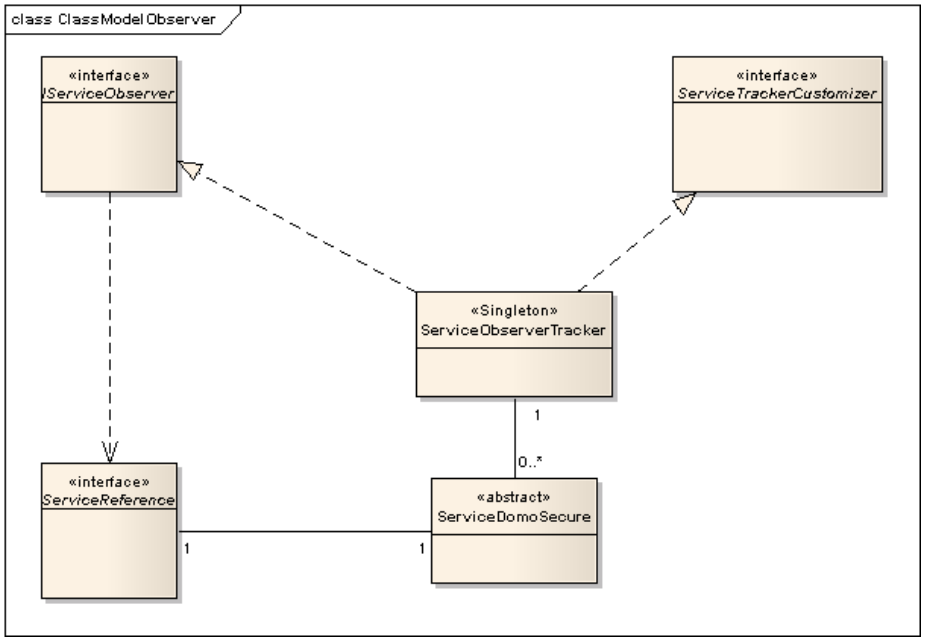


Figura 5.8: Diagrama de clases de los Consumidores / Productores *custom*.

HttpBinder). Dadas las características de OSGi, es simple cambiar la lógica de estas implementaciones en *runtime*. El *HttpBinder* provee una interfaz de comunicación HTTP mediante la cual se puede consumir los servicios del *framework* (local o remoto).

5.3. Seguridad

Como ya se ha dicho en secciones anteriores todo servicio puede definir si quiere ser publicado a otros nodos o no. El creador del servicio (en la implementación que controla el *middleware*: el *TechManager*) es el encargado de decidir cuáles servicios son filtrados y cuáles son publicados.

La API brinda un sistema de seguridad opcional para cada productor de servicios. Si este lo requiere, todos los consumidores de servicios de los *TechManagers*, deben estar previamente autenticados. Esto se logra a través de un mecanismo de firmas mediante clave pública y privada.

Todo *bundle* que quiera utilizar servicios autenticados debe estar firmado. Cuando el *SecurityManager* lo autentica, le da un *token* con el cual puede acceder a todos los métodos que requieren autenticación. Este *token* será usado por el productor de servicios para validar la invocación. Si un consumidor no está autenticado o entrega un *token* falso, el *TechManager* podrá filtrar las invocaciones.

En la figura 5.9 se muestra la secuencia (mediante un DSS) de cómo el *Proxy* obtiene el *token* para luego acceder a los *TechManager*. Para simplificar el escenario, se asume que el *bundle* API (que incluye *SecurityManager*) fue previamente cargado en el sistema.

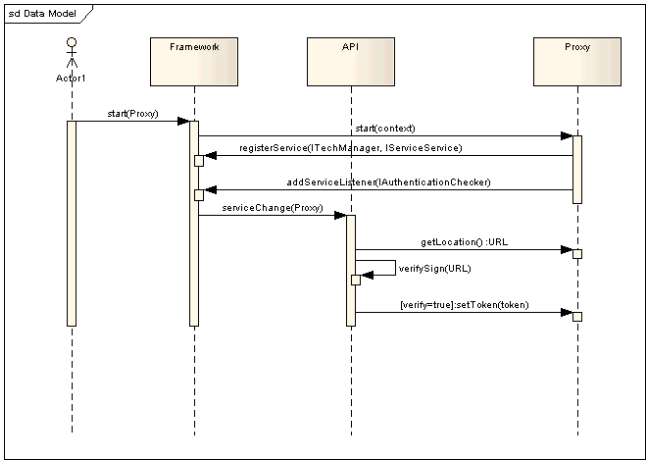


Figura 5.9: DSS de obtención de *token*.

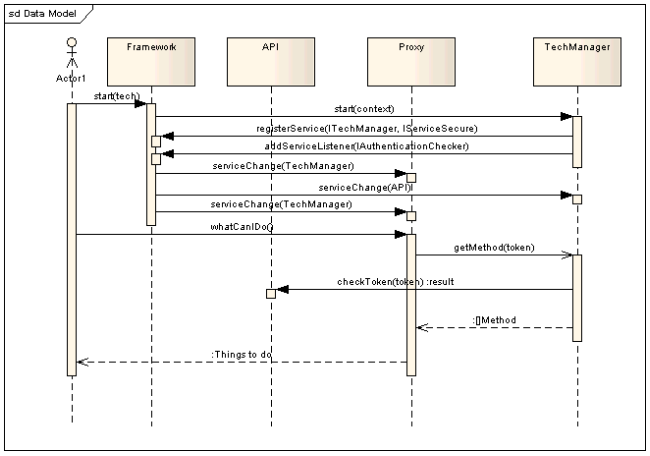


Figura 5.10: DSS invocación a método.

En la figura 5.10 se muestra un DSS de cómo el *Proxy* se comunica con un *TechManager* y el mismo valida el *token* del *Proxy* (nuevamente se asume que los *bundles* de API y *Proxy* ya se encuentran cargados).

5.3.1. Token: Obtención y Manipulación

Cuando un consumidor que implemente la interfaz `ISecureToken` es publicado en el *framework*, el *SecurityManager* obtiene el `org. osgi. framework. ServiceReference` del servicio consumidor recién levantado.

Con la referencia al servicio, el *SecurityManager* puede obtener el *org. osgi. framework. Bundle.* Al haber obtenido esta interfaz, es posible obtener la ubicación del archivo JAR (dónde esta la implementación del *bundle*) desde el cual se publica la interfaz `org. domo. api. security. ISecureToken`.

Cuando se obtiene la ubicación se realizan los siguientes pasos:

1. Analizar si el archivo está firmado y si no está corrupto. Si el archivo no cumple alguna de estas dos condiciones, se descarta.

2. En caso de no ser descartado (paso 1), se compara que alguna de las firmas con las que esta firmado el *bundle* en cuestión esté incluida en el *bundle* donde se encuentra el *SecurityManager*. Si no se tiene registro de esta firma, se descarta.

3. En caso de no ser descartado, (paso 1 o 2), se genera el *token*. El *token* se obtiene generando el MAC (Message Authentication Code) con los datos del archivo JAR más la fecha y hora del momento de generación y la URL del mismo. La función de *hash* utilizada es MD5 (*Message-Digest Algorithm 5*).

4. Luego de haber generado el *token* se guarda en una base interna, para poder validar el *token* desde los *TechManagers* (en la innovación a sus servicios) y se le entrega al *bundle* que fue interceptado con la interfaz `ISecureToken` mediante el método `setToken(String token)`.

Cada vez que un *bundle* anula la publicación del servicio `ISecureToken`, el *SecurityManager* elimina de su base el *token* para el *bundle* que esté dando de baja el servicio. Una vez rescindido el *token*, este no podrá ser utilizado por ningún otro servicio.

5.3.2. Usurpación de Identidad

En el MANIFEST se puede setear la propiedad *singleton* asociada al *Bundle-SymbolicName* la cual no permite instalar varios *bundles* con el mismo nombre simbólico. Esta propiedad recién se introdujo en R4 (por lo que no se puede utilizar con Concierge) dado que Knopflerfish soporta dicha especificación de OSGi se decidió usar esta alternativa para evitar un ataque de usurpación de identidad a los *bundles*.

5.4. Decisiones sobre la API

En esta sección se especifican las decisiones tomadas y su justificación en todos los casos.

5.4.1. Invocación a Métodos

Para definir las interfaces de los servicios de los distintos dispositivos se evaluaron 2 posibilidades.

Una primera alternativa, es generar una interfaz de marca por cada tipo de servicio de los diferentes sensores, actuadores y dispositivos en general, que podrían ser publicados por los *TechManager*. Su principal virtud es poder distinguir el tipo de servicio de forma sencilla utilizando un `ServiceListener` o un `ServiceTracker`. Definiendo una taxonomía de interfaces, presenta una forma sencilla e intuitiva de trabajo. Sin embargo, esta solución es rígida y la especificación de todas las posibles interfaces para cualquier dispositivo existente, no parece ser una tarea simple. Esta limitante se podría paliar de dos maneras:

1. Redistribuir una versión nueva de la API con cada interfaz nueva.
   Tiene como desventaja que es necesario recompilar y distribuir la API cada vez que se desee agregar una interfaz de servicios no prevista. Aunque OSGi brinda una gran flexibilidad para el manejo dinámico de *bundles*, todos los nodos deberían tener la misma versión de la API para el correcto funcionamiento del *RemoteManager*.

2. Generar un nuevo *bundle* con la taxonomía de interfaces. El cual habría que distribuirlo aparte de la API para poder utilizar las nuevas interfaces.

En ambos casos no sólo es necesario mantener el sincronismo de versiones entre los nodos sino que habría que modificar los consumidores para que puedan aceptar estas las interfaces.

El segundo enfoque (que resultó ser la forma elegida), es que los servicios implementen una interfaz que semejante a la interfaz definida en *reflection*. Esto permite tener una interfaz única donde todos los servicios se pueden filtrar con una sola interfaz. La forma de poder identificar cada uno de los dispositivos se realiza mediante las propiedades definidas en la publicación de la interfaz. Cada dispositivo genera las funciones necesarias para su utilización. Una desventaja que se puede encontrar en esta alternativa, sin herramientas que agreguen procesamiento semántico, es que los nombres y los parámetros de las funciones tiene que ser de conocimiento mutuo entre los programadores que desarrollen los programas o seguir alguna descripción de servicios estándar.

Sería un interesante punto de extensión de este proyecto, enriquecer el descubrimiento de servicios con una ontología que describa el modelo y mediante un motor de inferencia definir la similitud entre los servicios. Una opción sería integrar soluciones como las propuestas en [57], con la cual se pueda obtener las relaciones entra propiedades de los servicios, métodos y parámetros. De ser así, el acoplamiento entre productor y consumidor de servicio sería puramente semántico. Notar que esto es independiente de la opción elegida.

5.4.2. Descubrimiento de Servicios

Dada la decisión de usar un mecanismo similar a *reflection*, se define una interfaz homogénea para todos los servicios. Usando la potencialidad de OSGi, el desarrollador puede ser notificado de la aparición de un nuevo servicio (ya que conoce la interfaz que implementa). Sin embargo es necesario otro mecanismo para identificar el servicio encontrado, dicha semántica se logra añadiendo propiedades (objetos de OSGi) a los servicios. Estas propiedades son definidas por el publicador del servicio, el cual puede definir tanto la propiedad como su valor, de esta forma se define una propiedad por cada característica que sea relevante.

Un ejemplo para esclarecer esto es una luz conectada por Usb4All. En primer lugar interesa el tipo de servicio (independientemente del nombre de la función a invocar), para ello se puede definir una propiedad `DomoApiConstant. DOMO_API_SERVICE_TYPE`, también es relevante de qué luz es el servicio, para ello se define la propiedad `DomoApiConstant. DOMO_API_SERVICE_LOCATION` con la ubicación física del servicio:

- `DomoApiConstant.DOMO_API_SERVICE_TYPE=''LUZ''`

- `DomoApiConstant.DOMO_API_SERVICE_LOCATION=''Baño fondo''`

Con estas propiedades se pueden conseguir agrupaciones semánticas sobre los servicios, lo que permitiría poder filtrar todas las luces de la casa, todos los dispositivos del Baño del fondo, todos los servicios que hayan sido publicados por el *TechManager* de un *middleware* específico o combinación de los mismos.

Es necesario que el desarrollador del servicio productor y el desarrollador del servicio consumidor usen las mismas propiedades para que este mecanismo funcione. Si bien la API brinda esta posibilidad, su uso queda a criterio de cada implementación.

5.4.3. Composición de Servicios

Se entiende por composición de servicios, la agrupación de servicios en uno más complejo. El *Proxy* es el encargado de la generación de estos servicios. Un ejemplo sería brindar la posibilidad de prender todas las luces de una casa, con un sólo comando. Para realizar esto, el *Proxy* debería tener dos `Tracker`:

- El primero, para encontrar los servicios `IServiceSecure` publicados por los *TechManager* (esto ya está resuelto con la implementación de las clases abstractas que ofrece la API para implementar un *Proxy*).

- El segundo, para encontrar los servicios `IServiceDomo` publicados por los otros *Proxys,* el cual debe filtrar para que no tome los que el mismo *bundle* publica.

Por otra parte, deberá filtrar todos estos servicios que tengan la propiedad que define las luces (por ejemplo) `DomoApiConstant.DOMO_API_SERVICE_TYPE` = `''Light''`. Luego, debe generar un nuevo servicio que prenda todas las luces, lo cual hará que cada vez que se llame a la función **prender,** llame a todos los servicios simples, invocando la función análoga para cada dispositivo. Notar que se usa el mismo mecanismo de propiedades. Se puede ver que la composición de servicios se basa en el uso de propiedades al igual que el descubrimiento de los mismos.

5.4.4. Seguridad

La forma utilizada para la validación de los invocadores de los *TechManagers* se base en el modelo *"The white-board Model"* [76]. En este modelo se recomienda que los *bundles* registren un servicio en el *framework* OSGi, para que el servicio de generación de *tokens* les brinde uno. Esta solución es opuesta a lo que comúnmente se haría: el bundle obtendría el servicio de generación de tokens y le invocaría el método para generar token. Esta solución es vulnerable a ataques de robo de identidad de *bundles*, esto permitiría que se consumiera servicios sin permisos.

La decisión más importante en cuanto a la seguridad es su integración a la API, a diferencia del *RemoteManager*, el *SecurityManager* se distribuye junto con la API. Esto minimiza vulnerabilidades en cuanto a la usurpación de identidad. Dado el sistema de firmas usado, para saber qué código es confiable y cuál no, es crucial que quién valide esto, esté dentro de la API.

La API debe ser firmada por todas las entidades confiables, de esta forma, un código es confiable cuando la API fue previamente firmado con la misma firma que dicho código. Las firmas de la API ofician de lista de fabricantes confiables.

Los mecanismos de firmas y validación están dentro de la API y no pueden deshabilitarse (para prevenir su vulneración). Sin embargo cada *TechManager* y servicio publicado puede especificar si desea que la API le garantice la identidad del invocador. Esto presenta varias ventajas: más eficiencia (se realiza un chequeo de firmas *dummy* cuando `isUsingSecurity = false` con menor *overhead* que el chequeo real), por otra parte durante el desarrollo no es necesario firmar los *bundles* para poder probarlos.

A modo de evaluación se presentan algunos posibles ataques y la forma en que la API los previene.

- Una forma de vulnerar dicha seguridad es firmando nuevamente la API, con la firma del atacante. Sin embargo, la única forma de hacer esto, es que el atacante tenga total acceso al nodo donde esta corriendo la API. Si bien es un escenario posible, con dichos permisos podría invocar directamente el *driver* del *middleware,* de requerir permisos especiales del sistema para esto, también los precisaría para levantar la API.

- Otra forma de ataque podría ser introducir un nuevo *SecurityManager*, dado que se usa OSGi R4, existe una propiedad para los *bundles* (dentro del MANIFEST: `singleton=true`) la cual no permite levantar varias instancias del mismo *bundle* dentro de un *framework*, con esta propiedad se evita este ataque. Si bien se podría modificar esta propiedad, al estar dentro del JAR, la modificación alteraría la firma dejándola invalida.

Para evitar vulnerabilidades con el uso de *Remoting*, se decidió que los servicios publicados por nodos remotos no son verificados. Es decir si el desarrollador decide publicar de forma remota un servicio que requiere seguridad su invocación lanzará una excepción. Esto no presenta restricciones en cuanto a la publicación de servicios, ya que siempre se pueden hacer *Wrappers* para dichos servicios. En este caso la seguridad queda a cargo de quién genera el *Wrapper*. Por lo tanto *Remoting* sólo puede consumir servicios inseguros ya que la identificación y validación del invocador presenta riesgos adicionales.

Una aplicación práctica de dicha decisión sería el control de un un portón electrónico en la entrada de un garaje, teniendo conectado el motor del portón y un sensor de proximidad (por razones de *safety*). Una posible solución a la automatización del portón sería publicar el control del mismo de forma segura (y por lo tanto local) pero publicar el sensor de proximidad de forma remota. De esta forma se puede saber el estado del portón, pero no controlarlo. Sin embargo resulta aún más interesante poder controlarlo y que el control sólo tenga efecto si no hay peligro (físico de las personas y bienes). Se puede realizar una composición de servicios y publicar el control como una composición del sensor y el motor (en el cual no se cierre si hay un objeto interfiriendo el haz de luz del sensor), este servicio puede ser publicado de forma remota sin presentar peligros de *safety*.

Capítulo 6

Caso de Estudio

Para prototipar la API se definió un problema de domótica donde se quiere automatizar algunos aspectos de un baño. El objetivo de este prototipo es modelar un "problema real" con la API propuesta. La implementación del caso de estudio permite hacer los ajustes necesarios a la API en general, y a los componentes específicos de la solución planteada para el caso de estudio.

6.1. Descripción del Problema

Se desea prototipar la automatización del baño del InCo con la API de domótica propuesta, usando como *middleware* Usb4All.

Para el problema en cuestión, es de interés resolver problemas prácticos que ocurren en dichos baños. En primer lugar se desea poder ahorrar energía con la puesta en producción del sistema, evitando que se prenda la luz eléctrica cuando la luz natural es suficiente, y que la luz eléctrica esté apagada cuando no hay gente adentro del baño.

Por último se desea brindar un servicio de consulta del estado de los baños (ocupado o libre) para optimizar el tiempo de quienes utilizan dichos baños.

6.2. Requerimientos

Se dividen los requerimientos del sistema en funcionales y no funcionales.

6.2.1. Funcionales

Los requerimientos funcionales se listarán con el formato RFX dónde X es el número de requerimiento.

RF1 El sistema debe detectar la presencia de gente en cada baño.

RF2 El sistema debe detectar la intensidad de luz de cada baño y determinar si la luz artificial es suficiente o no.

RF3 El sistema debe poseer un mecanismo, mediante el cual el usuario puede prender la luz y notificarle al sistema que el baño está ocupado.

RF4 El sistema debe ser accesible de forma remota para poder consultar la presencia de gente en cada baños.

RF5 El sistema debe controlar la luz eléctrica de ambos baños y prendiéndola sólo cuando hay gente dentro del baño y luz natural es insuficiente o cuando se activa el mecanismo del requerimiento **RF3**.

6.2.2. No Funcionales

Los requerimientos no funcionales se listarán con el formato RNFX dónde X es el número de requerimiento.

RNF1 Se debe usar la API planteada para modelar el problema.

RNF2 Se debe usar Usb4all nativo en Java, como *middleware* para resolver la interacción con sensores y actuadores.

RNF3 Se debe usar un dispositivo de bajos recursos como plataforma de base para el sistema.

RNF4 Se debe configurar un mecanismo de auto-arranque en caso de reinicios (o cortes de luz).

RNF5 Se debe prever un mecanismo de recuperación ante fallas de *hardware*.

RNF6 Se debe tener conexión a la red *Ethernet* para permitir las consultas remotas.

6.3. Casos de uso

Los casos de uso se listarán con el formato CUX dónde X es el número del caso de uso, luego se presenta un nombre y una descripción.

CU1 Normal: La persona abre la puerta, el sistema detecta la presencia y verifica la intensidad de la luz natural, en caso de no ser suficiente enciende la luz eléctrica. El sistema mantiene el estado ocupado por un tiempo determinado para permitirle al usuario poner el pasador de la puerta. Luego que el usuario puso el pasador, el estado del baño se mantiene ocupado hasta un tiempo determinada después que saca el pasador y se vuelve a abrir la puerta.

CU2 Consultas: Se dispondrá de un servicio web, que brindará información de la disponibilidad de los baños. Si la conexión de red está activa, se podrá saber si hay gente o no en cada uno de ellos en forma remota.

CU3 Falla Sensor: La persona abre la puerta y el sistema no detecta la apertura de la puerta, al no haber luz, el usuario aprieta el botón existente, el sistema prende la luz artificial (independientemente de la luz natural) y de ser consultado vía web aparecerá en estado ocupado. El usuario pone el pasador, el sistema permanece en estado ocupado hasta un tiempo después que se saca el pasador dejándole tiempo suficiente a la persona para salir.

Requerimiento	Caso de Uso				
	CU1	CU2	CU3	CU4	CU5
RF1	X	X			
RF2	X				
RF3	X		X		
RF4		X			
RF5	X				
RNF1					
RNF2				X	
RNF3					
RNF4					X
RNF5			X	X	
RNF6		X			

Cuadro 6.1: Matriz de trazabilidad.

CU4 Falla USB: En caso de interferencia electromagnética (EMI: *Electromagnetic Interference*) que reinicien los dispositivos USB, se debe intentar una reconexión a las mismas.

CU5 Falla Corriente: En caso de un corte de luz, desconexión del cable de corriente o reinicio manual, una vez restablecida la corriente el sistema debe ejecutarse automáticamente sin intervención humana.

6.4. Matriz de Trazabilidad

En el cuadro 6.1 se presenta la matriz de trazabilidad dónde para cada caso de uso se muestran los requerimientos que involucra o viceversa.

Se puede ver que el escenario normal, que es el primer caso de uso es el que involucra más cantidad de requerimientos funcionales. Por otra parte, el segundo caso de uso (consultas remotas), involucra dos requerimientos funcionales, el de detección de presencia y el consultas remotas y un requerimiento no funcional relacionado con el acceso a la red. Los últimos tres casos de uso, especifican el funcionamiento del sistema ante distinto tipo fallas (en el sensor de movimiento, en la comunicación con el *baseboard* Usb4All y en la alimentación eléctrica).

6.5. Solución

En esta sección se presentan la resolución de los tres aspectos centrales del problema planteado: el uso de Usb4All, el modelado del uso del baño en base a los sensores disponibles, y la resolución del acceso remoto.

Para cada uno de estos aspectos se presentará la vinculación con la API.

6.5.1. *Usb4All*

A continuación se detalla cómo se resolvió la interacción con el *middleware* mediante Usb4All, el diseño del *firmware*, y la implementación de la solución utilizando la API.

6.5.1.1. *Driver*

En la sección 4.5.1 se detallan los problemas iniciales encontrados al usar la API en Java Usb4All, sin embargo por el requerimiento **RNF2** se debía implementar una solución nativa en Java. Por lo tanto, se decidió implementar una solución inspirada en el interfaz Lua (que utiliza libusb) para resolver la comunicación con los *baseboards* sin usar la API (Java - JNI - C++) ya existente.

En la figura 6.1 se muestra los distintos componentes de las soluciones: API Usb4All original, la solución implementada y la solución en LUA.

La API original de Usb4All utiliza básicamente cuatro componentes: la API en Java, el proyecto JNative para la interacción con código C++, una API en C++ que encapsula la comunicación con el bus USB, pudiendo conectarse mediante un *kernel-module* (cuya interfaz es un KO *Kernel Object*) o mediante libUsb.

La solución LUA, por su parte, es una arquitectura cliente servidor basada en comunicación TCP. En la imagen se presenta únicamente el servidor que es quién resuelve la comunicación con el bus USB. El servidor tiene tres componentes: el LubotServer que tiene las entidades de la placa (es análogo a la API original en Java) y se comunica con lualibusb (ver lualibusb [2]). El proyecto lualibusb genera un SO *Shared Object* que es un *wrapper* de libusb lo cual permite el control del bus USB desde código LUA y las clases correspondiente a la API en LUA.

La solución planteada implementa de forma nativa en Java la comunicación con el *firmware* Usb4All usando libusbJava que es un *wrapper* en Java de libusb (ver el proyecto Java libusb [12]) para resolver la comunicación con el bus USB. Esta implementación es una traducción del Lubot-Server a Java, a diferencia que publica una interfaz Java y no espera pedidos TCP la lógica correspondiente a la API en Java se detalla en el apéndice **??**. Por otra parte, el proyecto libusbJava es básicamente un *wrapper* de libusb usando JNI, por lo que tiene un subcomponente en C++ que fue necesario cross-compilar para generar el SO que permite la comunicación con libusb.

6.5.1.2. Firmware

La implementación en *firmware* está fuertemente ligada a los sensores y actuadores que se deseen usar. En este caso, se usaron los componentes disponibles: dos *baseboard* Usb4All, foto resistencias (LDR *Light-Dependent Resistor*) para medir la intensidad de luz, LEDs (*Light Emitting Diodes*) para visualizar el estado del baño, un sensor de movimiento y distintos *switches*. Por último fue necesario implementar un circuito para poder controlar una carga de 220 V con las señales del *baseboard* Usb4All.

La lógica de los *baseboards* Usb4All está en su PIC (*Programmable Integrated Circuit*) modelo 18F4550 de Microchip, en el cuadro 6.2se puede ver algunas de sus principales características.

Por detalles, ver el *data sheet* del PIC.

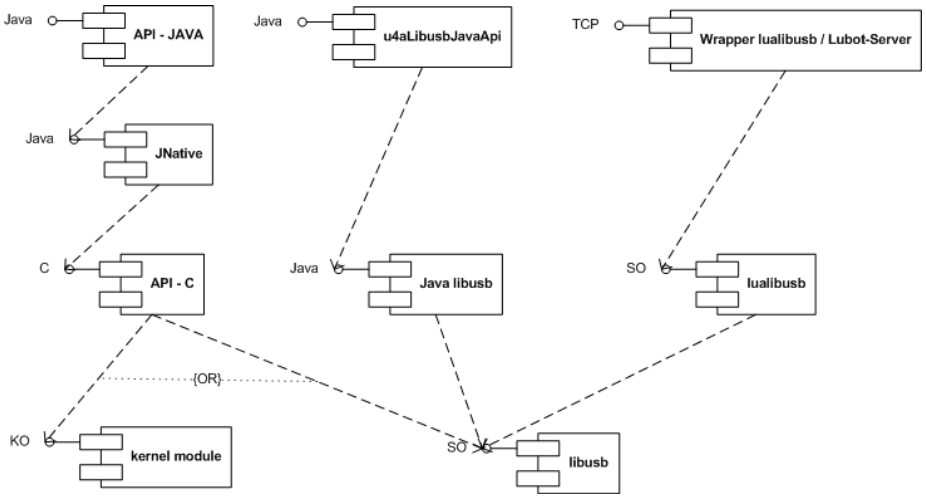


Figura 6.1: Comparación de componentes de la API Usb4All y la solución implementada.

Característica	PIC 18F4550
Memoria de Programa	32 kB
Memoria de Datos	2 kB
USB	2.0 full speed
E/S	hasta 35
Precisión A/D	13 bits

Cuadro 6.2: Principales características del PIC 18F4550.

Criterio	Diseño	
	Monolítica	**Modular**
Complejidad	Código más complejo. Simplifica la comunicación USB ya que sólo hay cuatro *EndPoints* (dos de entrada y dos de salida) controlados por el usuario.	Cada *user-module* es simple. Es necesario evaluar alternativas para controlar el BUS (un *read* bloqueante por módulo o un módulo de administración de las notificaciones).
Mantenibilidad	Simplifica *debug* de errores de comunicación y sincronismo.	Repite código, en caso de haber *bugs* hay que corregirlo en todos los módulos.
Legibilidad	Funcionalmente es más complejo.	Muy simple de repetir y hacer nuevos módulos similares con distintos dispositivos.
Reutilización de código	Más compleja.	Favorece la reutilización de código.

Cuadro 6.3: Evaluación del diseño de firmware.

Diseño Para interactuar con la placa, a priori, se consideraron dos diseños de *firmware*:

1. Crear un *user module* por cada elemento físico a controlar.
2. Crear un *user module* que administre y controle varios dispositivos.

Se evaluaron las ventajas y desventajas de cada una de estas opciones:

La opción modular tiene como ventaja que cada módulo es más simple, por lo que el código del *firmware* es más entendible, y en ese sentido más fácilmente mantenible. Sin embargo, repite código, principalmente en el caso de sensores bloqueantes y de haber errores es necesario corregirlo en todos los módulos. Por otra parte, favorece la reutilización de código, tomando un módulo que implemente funcionalidades similares a las requeridas y haciendo las adaptaciones necesarias.

En cuanto a la solución monolítica, de controlar varios dispositivos mediante un único *user module* hace que el *firmware* sea más complejo aunque no necesariamente con menos cohesión (en el caso de manejar todos los componentes de un mismo tipo, los módulos quedarían agrupados por funcionalidad).

En el cuadro 6.3, se pueden ver sintetizadas las ventajas y desventajas de cada una de las soluciones.

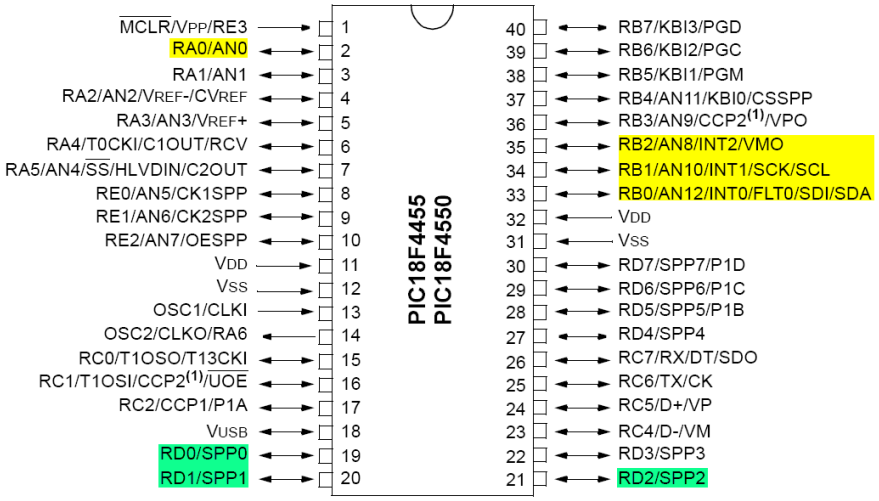


Figura 6.2: Pines usados del PIC para entrada y salida.

Para mostrar la factibilidad de ambos, inicialmente se decidió implementar parte de la solución de acuerdo a cada una de estas filosofías. Por un lado se decidió usar un *user module* para todos los sensores y un *user module* por cada actuador. Para ello fue necesario habilitar un *EndPoint* para cada módulo en el archivo `usbdsc.c` (archivo de *firmware* de Usb4All).

Durante la implementación de la solución siguiendo este enfoque, se encontraron problemas de concurrencia: al hacer un `read(...)` y `open(...)` simultáneos en la misma placa, se observaban comportamientos inesperados, por más que los `write(...)` para un mismo *EndPoint* estuviesen mutuo-excluidos. Inicialmente no se sabía que este era el problema y se migró la solución para usar un módulo para todos los sensores y otro para todos los actuadores. Si bien, esto no arregló el problema, luego se encontró la causa del error y se le brindó el uso exclusivo del bus antes de llamar a la función `open(...)`.

Para el caso de estudio planteado y con el *hardware* disponible se implementó la lógica necesaria para la recepción digital (el sensor infrarrojo y el de contacto), la recepción analógica (la fotocélula) y otro para la emisión digital (que con el circuito adecuado se potencia a 220V para el tubo-luz).

En la figura 6.2 se muestra en amarillo los puertos del PIC usados como entrada y en verde los puertos usados como salida.

Para simplificar el el uso de las entradas y salidas del PIC, se decidió usar todo el puerto B como entrada digital (aunque sólo hay tres pines configurados) y dónde el RB2 está programado para conectarse al botón. El puerto A como entrada analógica (con el RA0 programada para el LDR) y finalmente el puerto D está configurado como salida. En la tabla 6.4 se muestra qué debe ir conectado a cada puerto.

Puerto	Característica	
	E / S	**Componente**
RB2	E	Botón
RB1	E	Indistinto.
RB0	E	Indistinto.
RA0	E	Sensor de luz.
RD2	S	Indistinto.
RD1	S	Indistinto.
RD0	S	Indistinto.

Cuadro 6.4: Puertos y componentes.

6.5.1.3. Modelado con la API

En la sección 5.2.3.1 se explican las responsabilidades de los *TechManagers* (TM) y se muestra un diagrama de clases de diseño simplificado, dónde se pueden ver las clases de la API involucradas en la gestión de los TM y las firmas de los métodos más relevantes.

TechManagerUsb4All es una instancia concreta de TM, hace uso de los mecanismos de gestión de TM de la API, implementa la lógica correspondiente a los dispositivos que controla y encapsula las características de la tecnología Usb4All.

Este servicio se implementa en forma de *bundle,* su responsabilidad es controlar la comunicación con el *baseboard* (hacer uso de la API basada en Java libusb), crear y mantener actualizados los servicios de dicha tecnología. En este caso, se creará un servicio (accesible desde la API) por cada sensor y actuador, a los servicios de tipo sensor se les podrá pedir el valor actual `getValue`(...) y a los servicios de tipo actuador se los podrá modificar el valor actual `setValue`(...).

Este *bundle* está compuesto por seis clases (se omiten las clases interfaces de *test*): *ActivatorTMUsb4All* y *TechManagerUsb4All* que definen el TM, *Outputs*, *Inputs* y *DeviceChecker* que controlan los actuadores y sensores, y *TechManagerUsb4AllException* que define las excepciones relacionadas con este TM.

ActivatorTMUsb4All extiende de `TechManagerActivator`, es la encargada de crear el `TechManagerUsb4All`, así como de detener el servicio en el método `stopOtherThingsTechManagerActivator` en el caso de Usb4All esto es particularmente relevante ya que hay que cerrar los *user modules* del *baseboard* (en general, es el método dónde se liberan los recursos).
Las demás funcionalidades de la API para los `TechManagerActivator`, no son usadas en este TM.

TechManagerUsb4All es la clase central de este *bundle*, extiende `TechManagerDomo`, heredando así los métodos para cumplir con las funcionalidades de un *TechManager*.
En `isUsingSecurity(...)` se especifica si dicho manejador requiere que sus consumidores estén autenticados en el sistema. En este caso de estudio, se seteará en `false` ya que no se tendrá acceso al *router* sin una contraseña. Por lo que sólo los administradores podrán modificar qué *bundles* están levantados en el *framework.*

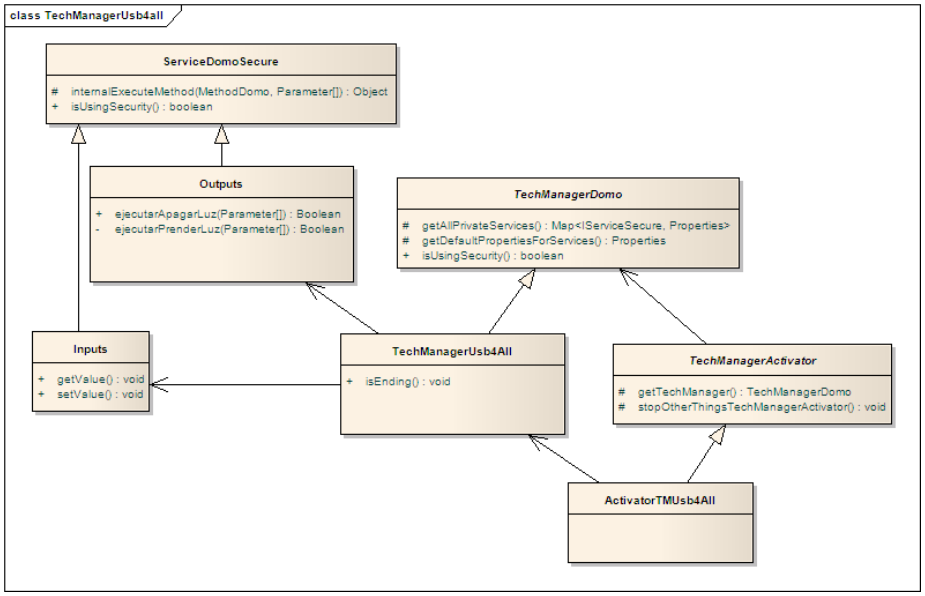


Figura 6.3: Diagrama de clases de diseño simplificado del *TechManagerUsb4All* simplificado.

En el método `getAllPrivateServices(...)` se definen los servicios que el `TechManagerUsb4All` publicará. A nivel de la API, cada sensor y cada actuador, es representado como un servicio, estos están definidos por las clases *Outputs* e *Inputs* y son distinguibles por las propiedades con las que son publicadas.
Las propiedades variables están definidas en dos archivos: `techManagerUSB4All.properties` y `sensores.properties`. El primero define a qué placa se corresponde su número de serie, el número de serie es una característica propia del *middleware* y no tiene porqué corresponderse con el número de placa que el consumidor del servicio conoce, y también especifica la ubicación de la placa. El segundo archivo define el nombre de cada sensor, a qué placa pertenece y si este es normal abierto o normal cerrado.

Outputs extendiendo *ServiceDomoSecure* articula entre la definición de los servicios en la API y el control del *middleware*. Funcionalmente, es la clase que define el control de los actuadores, que para este caso de estudio, tienen dos operaciones: una para el prendido y otra para el apagado. Para que esto se vea reflejado a nivel de la API, `getInternalMethods(...)` debe devolver un `array` de `MethodDomo` con un método para cada uno de estas funcionalidades e `internalExecuteMethod(...)` debe hacer la traducción de la invocación del `MethodDomo` a la invocación de la funcionalidad correspondiente en el *middleware*. En el caso de Usb4All, *Ouputs* pose el acceso a la entidad `DeviceUsb4All` (clase perteneciente al proyecto u4aLibusbJavaApi) y los códigos de las operaciones que puede invocar, encapsulando así la comunicación especifica con el *user module*

del *baseboard*.

Inputs es similar al servicio *Outputs*: hacia la API define el método `getValue(...)` que devuelve el valor del sensor, en caso de ser normal cerrado invierte el valor.
La API permite notificar sobre cambios en los sensores a los consumidores de dichos servicios, sin necesidad de hacer *polling* (ver subsección 5.2.3.5), para ello esta clase posee un método público `setValue(...)` que invocado por la clase `DeviceChecker` cada vez que recibe una actualización del *middleware* internamente este método invoca `updateAllObservers(...)`, propagando así la información en la API.

DeviceChecker es la clase encargada de actualizar los valores de los diferentes sensores (*Inputs*) cuando la placa detecta un cambio en alguno de sus valores. Para ello está en comunicación directa con la clase `DeviceUsb4All` del proyecto u4aLibusbJavaApi.
Para correr concurrentemente con la ejecución del programa es una clase `Runnable`. Efectúa continuamente un `read` bloqueante sobre el *user module* que controla los sensores quién únicamente lo desbloquea cuando los valores de sus sensores cambian. Cada vez que este *thread* es desbloqueado, ejecuta el método `setValue(...)` de la clase *Inputs* correspondiente al sensor que cambió.

6.5.2. Estados

A continuación se presenta el modelado del uso del baño en base a los sensores disponibles, así como el control de la luz eléctrica, y la implementación de esta lógica usando la API.

6.5.2.1. Lógica

Cada baño cuenta con cuatro sensores distintos: uno para la detección de la apertura de la puerta, uno para la detección del estado del pasador (abierto / cerrado), un botón de encendido de la luz y el LDR. Además cada baño cuenta con el sistema de control de luz eléctrica, y dos LEDs (uno verde y otro rojo) para poder visualizar el estado de cada baño.

La detección de la apertura de la puerta se hará en un baño con un sensor magnético (normal cerrado) y en el otro baño con un sensor de movimiento (normal abierto). Esta diferencia simplemente afecta en la definición de los sensores en el archivo `sensores.properties` del *TechManagerUsb4All*. Por lo que, no hay diferencia en el modelado de los baños.

La detección de presencia y control de luz se modelaron mediante un diagrama de estados.

Ya que el LDR potencialmente puede estar dentro del baño, y sus mediciones disparan el encendido de la luz, que a su vez afectan las mediciones del LDR, esto puede hacer que el sistema entre en un `loop` infinito. Para evitar dicho problema y no restringir la ubicación del LDR respecto a la luz, se decidió que una vez encendida la luz eléctrica, no se apagará hasta que el usuario salga del baño. Esto hace que los estados relacionados a la detección presencia se vean

Estado	Descripción
LIB CLA	No hay nadie en el baño y la luz natural es suficiente.
IND CLA	No se sabe si hay alguien en el baño, de ser consultado, el sistema responderá que el baño está ocupado. La luz natural es suficiente.
OCU CLA	Hay alguien en el baño y la luz natural es suficiente. De ser consultado, el sistema responderá que el baño está ocupado.
LIB OSC	No hay nadie en el baño y la luz natural es insuficiente.
IND OSC	No se sabe si hay alguien en el baño, de ser consultado, el sistema responderá que el baño está ocupado. La luz natural fue insuficiente en algún momento desde que el baño dejó de estar libre, por lo que la luz es encendida.
OCU OSC	Hay alguien en el baño. La luz natural fue insuficiente en algún momento desde que el baño dejó de estar libre, por lo que la luz es encendida. De ser consultado, el sistema responderá que el baño está ocupado.

Cuadro 6.5: Estados y sus descripciones.

duplicados en: los que hay luz natural (CLA por "claro") y en los que la luz natural es insuficiente (OSC por "oscuro"). De esta forma se cubre **RF2** y **RF5**.

Se desea evitar que en caso de fallar el sistema, el usuario entre al baño y no pueda prender la luz, por eso se dispondrá del botón, que mediante una rutina programada a nivel de *firmware* (sin intervención de la API) prenderá la luz eléctrica. Además funcionará de *preset* de estados para la detección de presencia, poniéndolo en estado ocupado y OSC. El sistema sólo saldrá de dicho estado si se remueve el pasador, independientemente de que esté puesto o no. De esta forma se cubre **RF3**.

Por otra parte, pensando en la usabilidad del baño, se detectó la necesidad de incorporar un *timer* en la detección de estados. En caso de que el usuario abra e inmediatamente cierre la puerta, y que no ponga el pasador, no hay forma (con estos sensores y sin más datos) de saber si el usuario se encuentra dentro o fuera del baño. Por lo tanto se añade un *timer* (T fin, $!T$ corriendo, $T = T_0$ reset) para determinar un período de incertidumbre (IND por "indeterminado"), durante el cual el baño está ocupado.

En la tabla 6.5 se resume la información referente a los estados.

En la figura 6.4 se muestra el diagrama de estados: los estados, las transiciones y las salida. Las entradas $< B, C, M, T, L >$ se corresponden: al botón B, al sensor de contacto C (conectado al pasador de la puerta), al sensor de movimiento M, al *timer* T, y la luz natural L (para el diagrama se simplifica la lectura del LDR en: luz suficiente o insuficiente). Las salidas $< T, L >$se

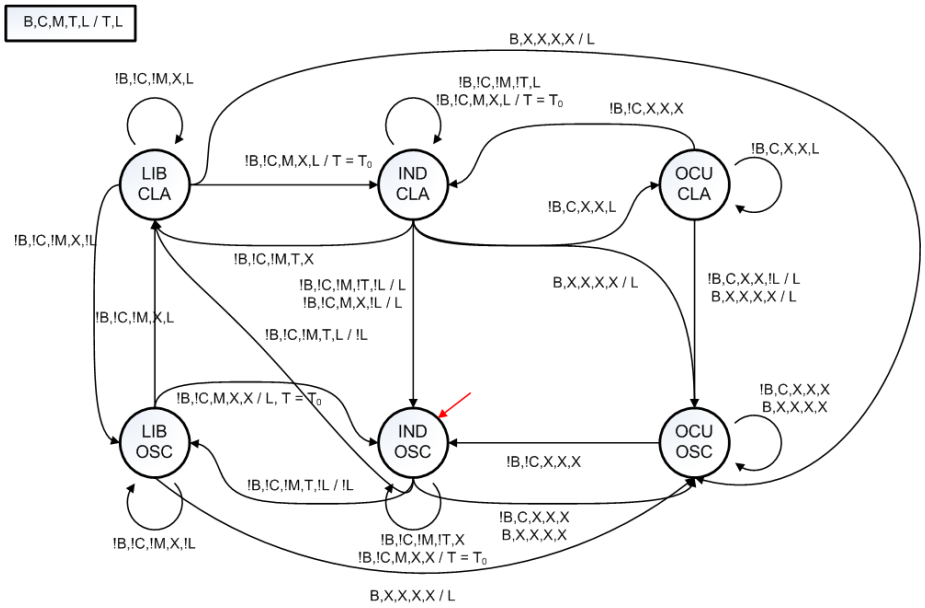


Figura 6.4: Diagrama de estados para detección de presencia.

corresponden: al *timer* T y la luz eléctrica L.

En dicho diagrama se puede ver que al apretar el botón B se pasa del estado actual al estado OCU OSC, es decir, se considera ocupado y que la luz natural es insuficiente. Dejando de lado estas transiciones, sólo se pasa del los estados CLA a OSC si se cumple $!L$, es decir, si la luz natural es insuficiente. Por otra parte a excepción del botón, nunca se pasa de un estado LIB a OCU directamente salvo al poner el pasador, que es el único caso en el que se está seguro que hay gente adentro. Finalmente, se puede ver que únicamente se setea el *timer* en las transiciones que inciden en los estados IND y que se vuelve a LIB, cuando se cumple $!B, !C, !M, T$, es decir, que no se está apretando el botón, no se está el pasador puesto, no hay movimiento y paso un tiempo T_0desde que se seteó el *timer*.

6.5.2.2. Modelado con la API

Se definió un *Proxy* para la resolución de estados ya que dentro de la API, es la entidad que puede componer servicios más complejos en base a otros servicios existentes. Una primera aproximación a esta entidad, revela que tiene 2 responsabilidades bien distintas: el control de los estados que se resolvió implementando lo diseñado en el punto anterior, y la gestión de esta implementación en base a los servicios y notificaciones publicados por la API.

El *Proxy* fue diseñado asumiendo que existe otra entidad que generará un servicio por cada sensor o actuador: *TechManagerUsb4All*. El *Proxy* reconoce los servicios, por medio de las propiedades que estos tienen, dichas propiedades se definen en el archivo: `ProxyBanio.properties`. Por otra parte, cada sensor y actuador se publica con una propiedad que define dónde está ubicado: `DomoApiConstant.DOMO_API_SERVICE_LOCATION`, por cada valor distinto, si

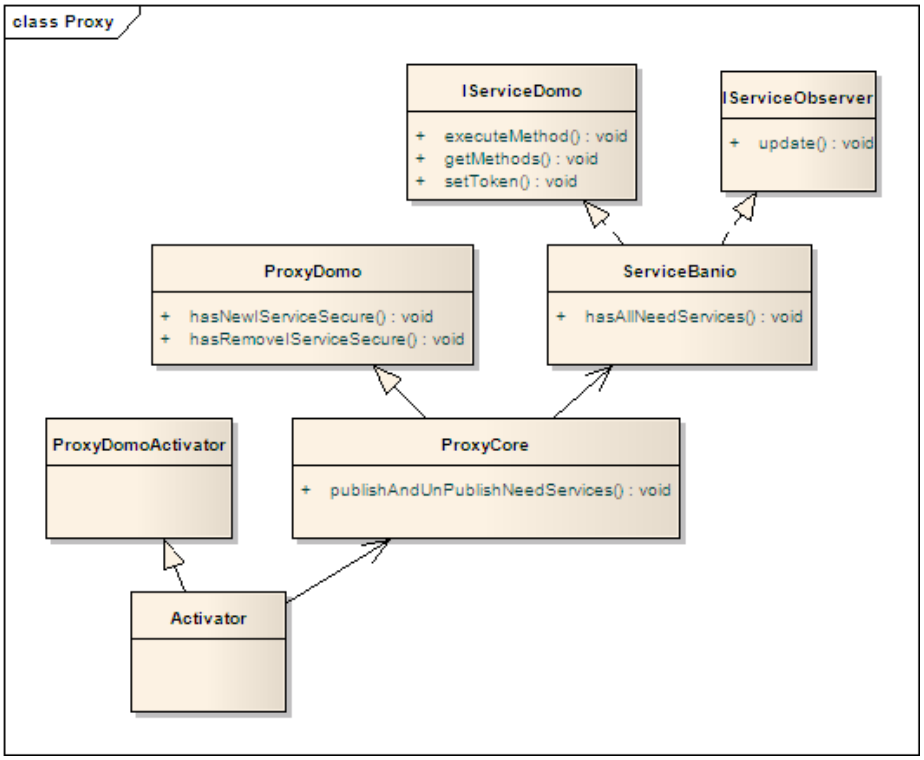


Figura 6.5: Diagrama de clases simplificado de diseño del *Proxy*.

existen todos los sensores necesario, se crea un servicio "estado de baño".

En la figura 6.5 se muestra un diagrama de clases de diseño simplificado del *Proxy* en cuánto a la gestión de servicios. Cómo todo *bundle*, este tiene un *Activator*, para cumplir con la definición de *Proxy* de la API, extiende *ProxyDomoActivator*. Las dos clases centrales de este *bundle* son *ProxyCore* y *ServiceBanio*. *ProxyCore* extiende de *ProxyDomo*, recibiendo así notificaciones de cambios en los servicios, de esta forma púbica y anula la publicación de *ServiceBanio* que es el servicio compuesto que este *bundle* agrega a la API. *ServiceBanio* implemente *IServiceDomo* definiendo e implementando el método "¿Hay alguien?". Por otra parte, por implementar *IServiceOberver* recibe notificación de cambios en los valores de los sensores. Por último, esta clase es la encargada de la máquina de estados que se implementó en otro paquete con varias clases auxiliares.

6.5.3. Acceso Remoto

Para cumplir con el único requerimiento funcional aún no resuelto: **RF4**, en primer lugar se decidió buscar soluciones existentes ya que el acceso web es un problema frecuente en diversos sistemas. En el OBR de Oscar [65] se encontró una implementación de *web server* muy liviana (600 kB de *footprint* incluyendo bibliotecas auxiliares) y *open-source*, basado en Jetty (ver `http://jetty.codehaus.org/jetty/`).

Desde el punto de vista de la API, este *bundle* es simplemente un consumidor de servicios. La resolución de la publicación del estado de los baños es la implementación de páginas web, utilizando *servlets* que se comunica directamente con el *Proxy*.

Mediante un *tracker* el *http server* detecta la publicación de un servicio cuya propiedad `DomoApiConstant. DOMO_API_SERVICE_TYPE` vale "BANIO" dentro de la API y puede consumirlo ya que este cumple con la interfaz `IServiceDomo.`

Para el sitio web, se realizaron dos paginas: una para todos los usuarios montada sobre la raíz de la ruta al servidor la cual muestra el estado de los Baños. La página para administradores, montada sobre `/Services` la cual requiere autenticación, permite ver todos los servicios, sus propiedades y ejecutar los métodos. Los nombres de usuarios y contraseñas se encuentran en el archivo de propiedades: `login.properties`.

El puerto en el cuál Jetty publica las páginas, está definido por la propiedad `org.osgi.service.http.port` que puede estar tanto en el archivo de inicio de OSGi o como pasarse como parámetro a la JVM.

6.6. Deploy

Por el requerimiento no funcional **RNF3** se requiere el uso de una plataforma de *hardware* de bajos recursos, y por lo expuesto en 3.8 se usó un *router* Asus wl500w. Esto también asegura que el EOS esté disponible en caso de reinicios, con lo que faltaría levantar el sistema automáticamente con el inicio de sesión para cumplir con **RNF4**. Para ello, se implementó un *script* de auto-arranque de Knopflerfish, qué pasándole como parámetro su archivo de inicio, define cuáles son los *bundles* (y propiedades) que iniciará.

6.7. Adaptaciones

Luego de haber hecho el prototipo, se encontraron nuevos desafíos, los cuales se presentan a continuación.

6.7.1. USB

Inicialmente el *hardware* presentaba varias fallas. Paulatinamente se fueron eliminando o mitigando cada uno de los problemas que se pudieron identificar.

Se encontró que en el *router* no se podía hacer funcionar ambos relés al mismo tiempo, lo qué no pasaba experimentando en las máquinas de escritorio. Simplificando hipótesis, se descubrió que se debía al uso de un *hub* USB (necesario en el *router* ya que sólo tiene 2 puertos USB y uno estaba en uso por un *pendrive*). Probando el mismo escenario con otro *hub* que contaba con alimentación externa y viendo que funcionaba, se descubrió que el problema se debía a la corriente del USB que no era suficiente para alimentar ambas placas a la vez. Por lo que, para hacer el *deploy* del sistema en el *router* es necesario un *hub* con alimentación externa.

Otra particularidad relacionada al USB es la calidad de los cables. Si bien, muchos cables funcionan para diversas aplicaciones, en este caso en particular, se encontraron problemas relacionados a los cables que se estaban usando, teniendo incluso que cambiar uno de ellos.

6.7.2. Circuito de interacción

Unos de los grandes desafíos que se tuvo en el proyecto, fue el diseño del circuito electrónico para el control de los diferentes sensores y actuadores, principalmente por el poco conocimiento que se tiene en esta área, sin embargo se realizaron las pruebas y consultas necesarias a expertos en el área.

Para poder realizar la configuración del circuito se recurrió a la ayuda de Ingenieros Eléctricos. La mayoría de las configuraciones fueron probadas antes de realizarlas en las placas definitivas en una *protoboard* para mitigar riesgos, sin embargo esto no fue suficiente, teniendo que realizar adaptaciones como se explica a continuación.

6.7.2.1. Configuración inicial

En la figura J.1, se muestra la primera configuración del circuito. No hubo necesidad de realizar cambios sustantivos sobre esta configuración, sino pequeñas modificaciones que se irán viendo en esta sección.

Para poder interactuar con los sensores y actuadores hubo que resolver 4 problemas: controlar el encendido de una luz (que trabaja con 220v), obtener la lectura de un LDR (*Light Dependent Resistor*) que mide la intensidad de la luz, prender LEDs y obtener las lecturas de los sensores de movimiento, contacto y apertura.

Para resolver el problema de obtener la lectura del LDR desde el PIC, se tuvo que utilizar un Amplificar Operacional (AO) en modo seguidor para mantener el valor de la resistencia del LDR constante mientras se realiza la operación de lectura desde el PIC.

El problema de los sensores digitales, se resolvió uniendo un conector de cada sensor a Vcc y el otro conector unido a una entrada del PIC que lee los datos para ese sensor. A su vez la entrada del PIC se conectó mediante una resistencia de 10K a Gnd para que si el sensor está abierto, el PIC no quede en alta impedancia. La resistencia se colocó, para que si el sensor está cerrado no se produzca un corto circuito con Gnd y Vcc. En el diseño del circuito para los sensores, se previó la colocación de capacitores (entre los dos conectores de los sensores) para poder eliminar el rebote de los mismos.

Los LEDs fueron conectados con un conector a Gnd y el otro al puerto salida del PIC.

La explicación del control de luz (que trabaja en 220v) se ha dejado para el final dado que sufrió varias modificaciones, las cuales se mostrarán. Para controlar la luz se utilizó un transistor NPN y un relé de un solo polo. Más específicamente se muestra en la figura 6.7 la configuración inicial del relé. La interconexión es la siguiente: uno de los conectores de la bobina del relé se conecta directamente a Vcc y el otro conector de la bobina se conecta al colector del transistor NPN. La base del transistor se conecta a la salida del PIC para que controle la apertura o cierre del circuito. El emisor del transistor se conecta directamente al Gnd. Se colocó el transistor ya que la alimentación del PIC no era suficiente para controlar el relé, además con esta configuración fue necesario colocar una fuente externa. El transistor oficia de punto de control, si se le emite una corriente I_{PIC} a la base del transistor, se generará una corriente $I_2 < I_{PIC}$ entre el colector y Gnd.

Hay que notar que con esta configuración se usa Vcc de la fuente externa (la

del PIC no está conectada), y ambos GND (PIC y fuente externa) están unidos para tener un valor de referencia común.

6.7.2.2. Modificaciones

Como se vio que la configuración realizada hacía que el relé funcionara mal en ciertas circunstancias (haciendo que el PIC dejara de responder), se investigaron otras interconexiones propuestas por Ingenieros Eléctricos.

La siguiente prueba realizada, como se muestra en la figura 6.8, fue agregar un diodo entre las conexiones de la bobina del relé. Esta prueba se realizó dado que se conoce que la bobina genera una corriente opuesta (al sentido de la corriente del circuito), la cual se supuso que era la responsable de las perturbaciones del circuito, sin embargo no se obtuvo ninguna mejora.

La tercera prueba realizada, que se muestra en la figura 6.9, fue colocar un diodo en el emisor y el colector del transistor para proteger el transistor de la corriente electro-magnética generada por la bobina del relé. Además se colocó una resistencia de 10 KΩ entre el PIC y la base del transistor y una resistencia 470 Ω entre la colector y la bobina del relé. Sin embargo esta configuración no alimentaba al relé con la corriente necesaria para que funcionara de forma adecuada. Por lo tanto hubo que bajar la resistencia de 470 Ω a una menor de 45 Ω (este dato se obtuvo empíricamente, teóricamente 470 Ω sería lo correcto, pero 45 Ω es lo máximos que se puede poner conservando el funcionamiento del relé). Con esta configuración el circuito se comporta mejor, dura más tiempo antes de obtener una falla, pero de igual manera sigue fallando con lo cual se siguió trabajando para mejorarlo.

Otra prueba para intentar mitigar los errores producidos por el relé que genera la interrupción del funcionamiento del PIC fue agregar una resistencia entre la base y el emisor del transistor (se muestra en la figura 6.10) la cual no generó grandes cambios.

6.7.2.3. Configuración alternativa

En paralelo con las mejoras al circuito original, se prototipó una configuración que no requiere de fuente externa para la alimentación de la bobina del relé, la cual se muestra en la figura 6.11. Sin embargo, no hubo cambios significativos en la estabilidad del circuito.

6.7.3. Recuperación ante fallas

En las primeras configuraciones del circuito, las fallas se manifestaban en el la comunicación con el PIC. Luego de la misma, el PIC no aparecía como un dispositivo USB conectado (no respondiendo al comando `lsusb`). Luego de la última configuración utilizando el (comando `dmesg`) se podía ver que el *router* reiniciaba la conexión con la placa (con mensajes como este: "`hub 3-0:1.0: port 1 disabled by hub (EMI?), re-enabling...`"). Dónde expresamente, indicaba una posible interferencia electro-magnética: EMI (*ElectroMagnetic interference*).

Dado que se podía volver a iniciar la comunicación con las placas, se decidió implementar un sistema de recuperación ante fallas.

El sistema de recuperación ante fallas se basa en la resolución de dos problemas. En el primer problema es la búsqueda y publicación de los servicios de las placas. Por lo que, el TM tiene un proceso que busca cada determinado tiempo, mediante *pooling* (ya que la notificación de nuevos dispositivos recién se liberará en la versión 1.1) si existen nuevas placas. Si existe genera los servicios nuevos para la placa encontrada. El segundo problema ocurre cuando la placa presenta algún error, en este caso el TM elimina todos los servicios publicados del sistema.

Por más información ver apéndice **??**.

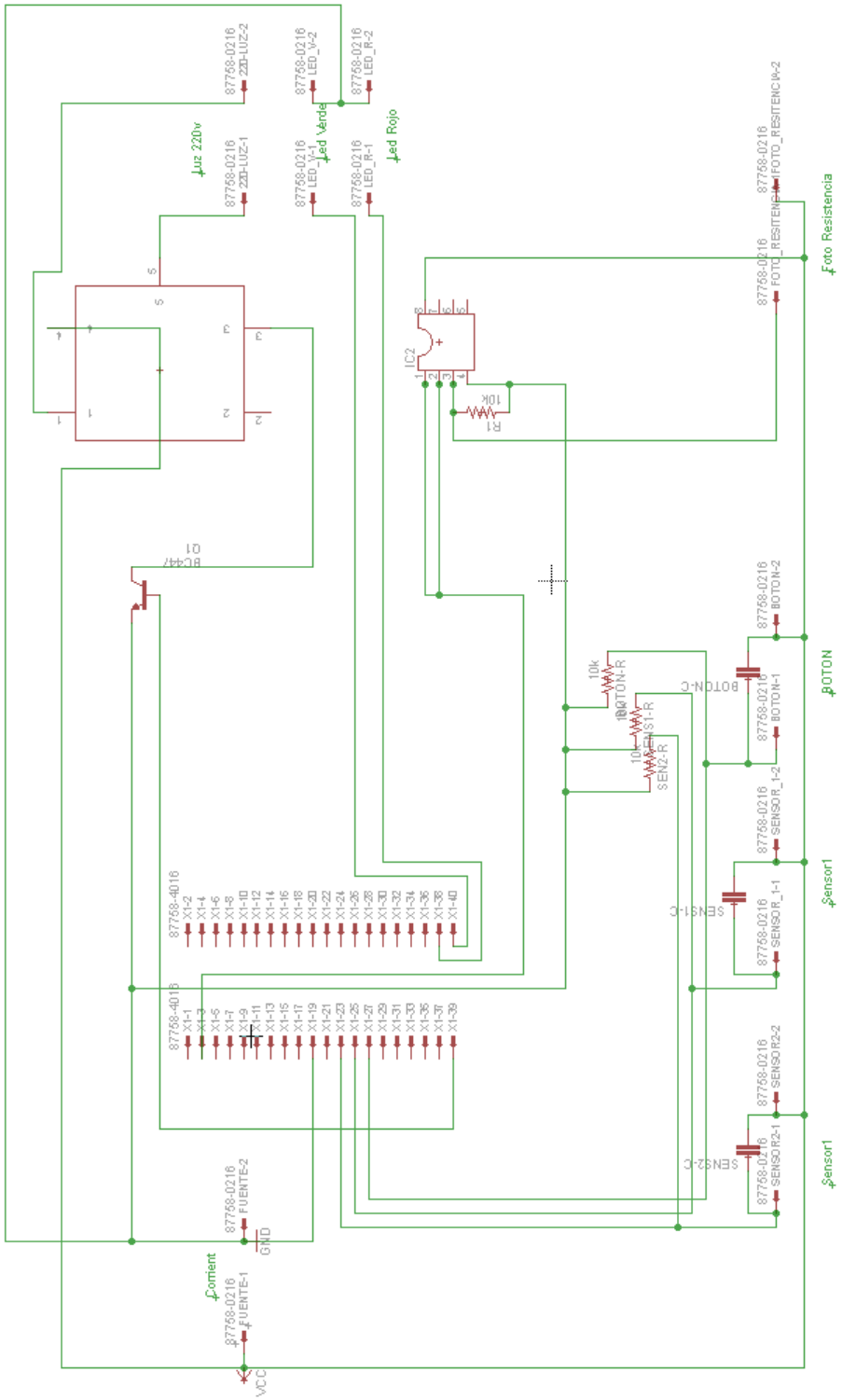


Figura 6.6: Diseño esquemático.

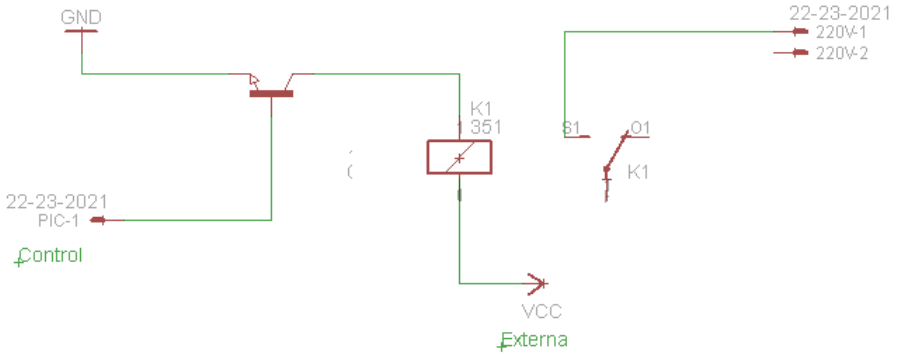


Figura 6.7: Primera Configuración con el relé.

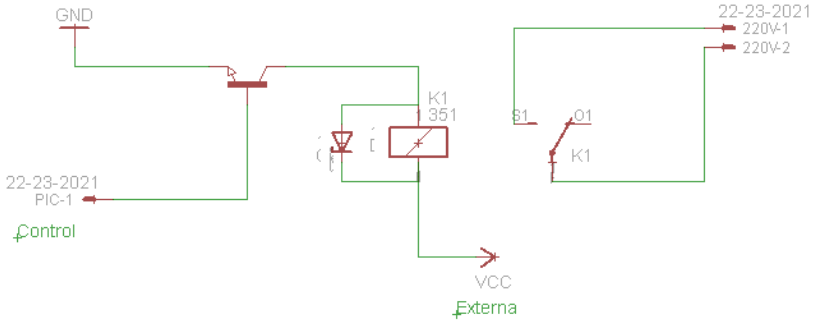


Figura 6.8: Segunda Configuración con el relé.

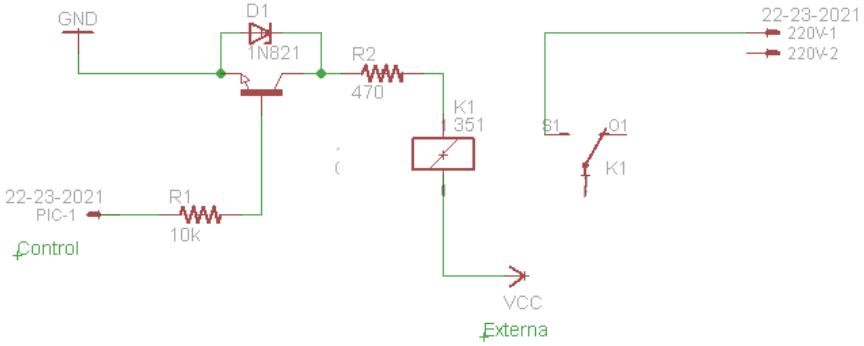


Figura 6.9: Tercera Configuración con el relé.

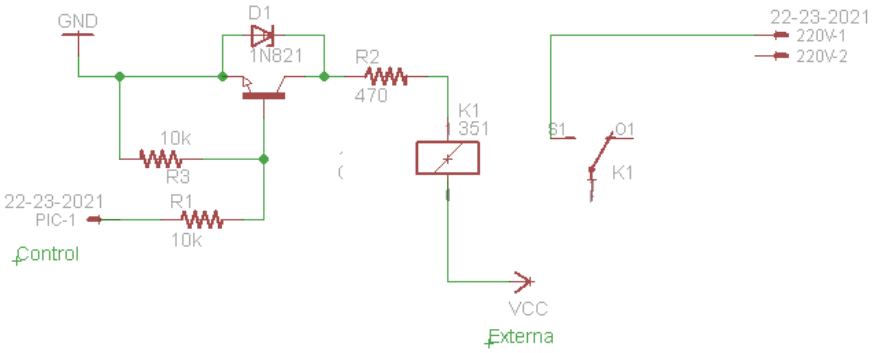


Figura 6.10: Cuarta Configuración con el relé.

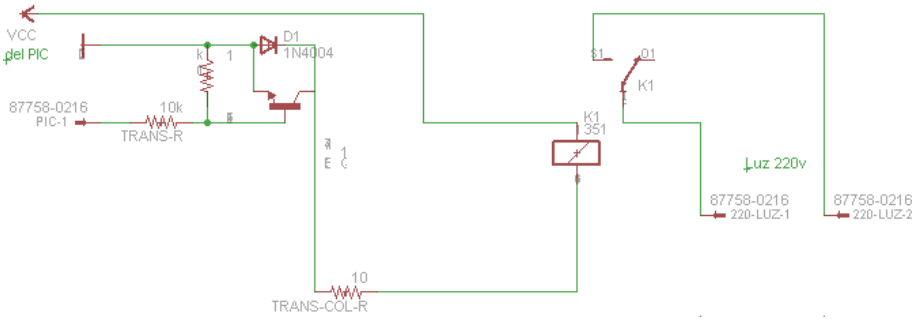


Figura 6.11: Configuración del relé sin fuente externa.

Capítulo 7

Conclusiones

En este capítulo se presentan las conclusiones generales del proyecto. Se resumen las dificultades encontradas durante el desarrollo del trabajo, los resultados obtenidos con respecto a los objetivos planteados inicialmente, los aportes generados de forma colateral, y por último, se presenta posible trabajos a futuro que se podrán realizar a partir de los resultados obtenidos.

7.1. Resultados

Se lograron los objetivos planteados: se hizo un relevamiento del estado del arte, se propuso una API de alto nivel genérica para desarrollo de aplicaciones de domótica, que posibilita implantaciones en dispositivos de bajos recursos.

Del relevamiento del estado del arte, se pudo observar cuáles son las tecnologías de punta, cuales son los desarrollos que se están generando en el área y hacia dónde se esta dirigiendo la integración de la tecnología en los hogares.

Del desarrollo de la API, al haber sido desarrollada sobre OSGi permite que las aplicaciones sean fácilmente escalables. Permitiendo al implementador enfocar el esfuerzo del desarrollo en resolver el problema de la comunicación con las tecnologías con las cuales se quiera comunicar, además de proveer una forma fácil y sencilla para generar composición de servicios a partir de los servicios básicos que las diferentes tecnologías de base.

A modo de exponer la factibilidad de mismo, también se realizó un prototipo el cual podría ser viable de implantarlo en el InCo.

7.2. Dificultades

En primer lugar, se resalta que no se tenía experiencia en trabajo con sistemas embebidos con las características de este proyecto. Si bien se había desarrollado *firmwares* para PIC*s* y microprocesadores (SPARC, 8086 y Z80), las diferencias fueron sustanciales. En particular, se encontraron problemas en la cross-compilación de componentes para OpenWrt, las cuales se agravaron con la existencia de poca y mala (desactualizada o incorrecta) documentación al respecto. Una de las causas de este tipo de problemas está dado por la homogeneidad del mercado informático actual y la poca educación que la

facultad brinda con respecto al desarrollo de sistemas en arquitecturas diferentes a x86 y EOS.

Este riesgo no pudo ser mitigado con antelación por falta de un ambiente que emulara el ambiente de trabajo final, ni se poseyó en forma temprana con los componentes necesarios para la experimentación. Al ser el pilar fundamental del proyecto, esto impactó en la totalidad del trabajo.

Sobre el EOS se monta la JVM, donde también se encontraron diversos desafíos. No se tuvo problemas con la cross-compilación de Mika, pero resultó ser una implementación muy poco madura. Si bien generalmente se obtuvo buen soporte, se encontraron problemas en distintos niveles, para los cuales, no había perspectivas de arreglarlos.

Por otra parte, el proceso de cross-compilación de JamVM estaba mal documentado, dónde las versiones que venían en forma nativa con Kamikaze 8.09.1 no eran compatibles. El diagnostico de este problema, llevó más tiempo de lo previsto porque se creyó que el error estaba en el proceso de cross-compilación y no en la distribución de los paquetes. Una vez montada la JVM, se encontraron diferencias menores entre las distintas implementaciones de Java.

Con respecto a OSGi, se encontró que la implementación óptima para dispositivos de bajos recursos, Concierge, tenía problemas importantes, lo cual, acompañado por escasa documentación y lento soporte, provocó un cambio en la implementación de OSGi que se decidió usar.

En lo que R-OSGi respecta, se descubrió que no es una tecnología lo suficientemente estable. Se corrigieron algunos errores de dicha implementación, pero dado que no se pudo validar estas correcciones, la solución de *remoting* es difícilmente mantenible si se quiere seguir el ritmo de futuros *releases*, además de los errores remanentes.

El uso de Usb4All resultó ser más costoso de lo estimado. En principio, ya que el proyecto de la API genérica no debía depender de ninguna tecnología específica, se esperaba contar con un *middleware* funcional y simplemente hacer la conexión (o *driver*) entre este último y el sistema a construir. Aunque la API de Usb4All es funcional en la mayoría de sus configuraciones, se trabajó bajo la premisa que Linux - libusb debían funcionar, lo cual resultó no ser así.

Por último, pero no menor, cabe destacar los obstáculos que se atravesaron durante el desarrollo de la API. En primer lugar, los tiempo de inicialización del sistema sobre la placa objetivo son extremadamente lentos (considerando los tiempo de codificación). Cada vez que se quería probar la API sobre la placa, había que esperar alrededor de tres minutos, lo que hizo que el proceso fuese tedioso. Por más que se poseyera un ambiente de desarrollo comparable al de la placa, para simularlo completamente debería tener la misma *performance* que la de la placa por lo que de todos modos, se hubiese tenido este problema.

Otra característica propia de este proyecto, fue la diversidad de los candidatos a fallar: por lo general se desarrolla sobre ambientes estables que se supone que no van a fallar y se atribuye los errores a *bugs* en lo que uno mismo está desarrollando, o a lo sumo situaciones específicas del sistema operativo, red, etc. En este proyecto, no se tenían certezas del correcto funcionamiento de ningún elemento. Los diferentes fallos se podían encontrar en: el circuito (que depende de interferencias físicas), el *firmware* (que depende del circuito y de la comunicación con su *driver*), el sistema operativo (que no necesariamente funciona igual que en x86, así como sus variantes en los paquetes), la JVM (que no necesariamente funciona igual compilada para MIPS que para x86), OSGi,

jSLP y R-OSGi los cuales eran candidatos a fallar o a tener *bugs*. El mayor problema fue que difícilmente se podían eliminar hipótesis (comprobar que el problema no estaba en uno de ellos) ya que probar en una arquitectura x86 lo mismo que en el la placa MIPS, no sólo implicaba diferencias de recursos (y de *performance*), sino que los fuentes (y más aún los binarios) eran distintos.

7.3. Aportes

Más allá de los resultados obtenidos, y pese a las dificultades encontradas, se generaron diversos aportes reflejados tanto en el informe final, como en las diferentes comunidades (*embedded*, OSGi y R-OSGi).

En primer lugar, se generó experiencia en cuanto al trabajo con EOS y particularmente con OpenWrt.

En cuanto al trabajo con JVMs para EOS, es un área donde la información no está muy difundida e incluso la existente es de mala calidad. Esto es que la mayoría de los manuales, *HOW-TO* y explicaciones en los foros no andan, tienen problemas o están desactualizadas. Un ejemplo de esto, es la incompatibilidad entre JamVM y GNU-Classpath, donde si bien los pasos a seguir pueden estar bien descritos, con la mayoría de las combinaciones de sus versiones no funcionan.

Teniendo en cuenta las limitaciones de Java y sus aspectos negativos (principalmente el *footprint* y el overhead de ser un lenguaje interpretado), igual se puede concluir que, en ambientes de bajos recursos, brinda muchas ventajas dado la portabilidad, la posibilidad de utilizar un *framework* como OSGi, y la facilidad de implantación dinámica. Por otra parte permite desacoplar completamente el desarrollo de un sistema con el *hardware* de base que se utilice (luego de portar el *driver*). Esto permite difundir el desarrollo de aplicaciones en sistemas embebidos. De la experiencia generada por este proyecto, se concluye que en caso de querer hacer una solución para un problema pequeño y puntual, de forma rápida y no necesariamente escalable, es preferible no usar Java en embebidos por el costo de instalación y adaptaciones que esto requiere. Sin embargo, si se desea hacer una solución más genérica, extendida en el tiempo y fácilmente mantenible se recomienda el uso de Java en sistemas embebidos.

Asimismo, se dinamizaron distintos foros haciendo aportes a sus comunidades, tanto por la generación de "preguntas frecuentes" para usuarios nuevos en estas tecnologías, como reportando *bugs,* e incluso corrigiendo algunos de ellos. En este sentido, se puede decir qué se participó de forma activa en la maduración de las tecnologías utilizadas.

Se experimentó con dos implementaciones de OSGi, tecnología poco difundida en la FING[1]. No sólo se contribuyó en la generación de experiencia de uso, sino que también, se encontró un punto de enlace entre dos mundos generalmente muy divididos: el área tecnológica de punta en sistemas grandes y el área de embebidos (por las características intrínsecas de lo que buscan resolver). En este proyecto se tuvo un primer acercamiento a OSGi dónde se pudo ver muchas de las ventajas de usar este *framework*: fácil manejo de servicios, portabilidad, dinamismo y reutilización de código. Se sugiere considerar el uso

[1] Búsqueda en google "site:fing.edu.uy OSGi", se encuentran dos documentos en formato pdf de la asignatura TSI3 dónde aparecen en un punteo de tecnologías SOA.

de esta tecnología, en proyectos de domótica pero también en cualquier sistema que se pueda resolver con la arquitectura SOA.

Por último, se reutilizó parcialmente el proyecto de grado Usb4All del año 2007, actualizando la información del estado en el cual se encuentra dicho proyecto, diversificando su aplicabilidad, y multiplicando el conocimiento adquirido.

7.4. Trabajos a Futuro

Durante la realización de este trabajo, quedaró pendientes pulir algunos aspectos y se encontraron nuevas vetas de expansión del mismo.

En la implementación de *remoting* es dónde se han detectado más *bugs* no resueltos. De la experiencia en corregir los *bugs* de R-OSGi, se recomienda revaluar soluciones de *remoting*. En caso de hacer otro proyecto de grado para esto, probablemente sea más rápido generar una solución propia funcional que usar R-OSGi. Incluso, se podría verificar si realmente presenta las ventajas en eficiencia que se jacta tener. Hay que tener presente que R-OSGi y jSLP son dos proyectos separados, con este último no se tuvieron grandes inconvenientes y parece tener buenas perspectivas de crecimiento por estar ahora en manos de una comunidad más grande. Por lo tanto, no sería necesario implementar el descubrimiento de *hosts*.

Por otra parte, ya que se podría implantar la solución en el baño del InCo, parece interesante realizar distintos tipos de *testing* sobre el mismo, no sólo *testing* funcional, sino también *testing* de *performance* y de carga de la solución en su conjunto. Estos testeos se podrían combinar con distintas optimizaciones que impacten en la *performance* general de la solución, como ser recortar las bibliotecas de Java utilizadas (ajustándolas al mínimo conjunto necesario para todos los *bundles* de la solución implementada), usar optimizaciones de código Java y de *bytecode*, analizando en todos los casos los tiempos de respuesta del sistema, memoria consumida, utilización de CPU, uso de *swap*. Como resultado de esta investigación, no sólo se podría optimizar los tiempos de procesamiento, sino que también se podría utilizar una placa embebida tomando en cuenta los requerimientos específicos del sistema.

Otro camino a seguir, antes de proliferar dicha solución en los baños de la FING, podría ser la inyección de fallas (a nivel de *hardware*, físicos, de red y del sistema operativo) midiendo la confiabilidad y el impacto potencial en la *safety* de las personas que utilizan el sistema.

Otra línea de trabajo pendiente, es sofisticar las medidas (y algoritmos) de control combinándolos con valores estadísticos. Midiendo, por ejemplo, el ahorro en energía que presenta implantar esta solución. Hay que tomar en cuenta el valor estimado de la cantidad de veces que usuarios del baño dejan la luz prendida y el tiempo que queda prendida y contrarrestar este valor contra el consumo energético de la placa y la alimentación externa del circuito. Siguiendo este enfoque, sería recomendable usar el diseño de la placa de control sin alimentación externa del circuito.

En las tendencias relevadas en el estado del arte, existen trabajos con el objetivo de conceptualizar el problema de domótica, donde se describen todas las relaciones posibles entre los elementos y se carga dicha información en una ontología. Aplicando motores de inferencias, es posible concluir si la acción que

el usuario quiere realizar, es posible en la configuración actual del ambiente, o si puede vulnerar una situación. Por ejemplo, el sistema podría evitar el encendido automático de la aspiradora en un cuarto en el qué se está mirando una película, o evitar que un aire acondicionado se prenda a 20° y otro a 30° dentro de la misma habitación. Esto abre un campo en el desarrollo semántico de la solución.

Finalmente, otro campo de trabajo interesante es la interacción de la solución con el usuario: HCI (*Human Computer Interaction*). Este es el típico caso en que la tecnología interactúa con la persona por diversos medios (teclado, pantalla, *LEDs*, luz, botones, etc). En esta dirección parece atractivo estudiar cuál es la interfaz más adecuada para cada lugar, así como las características que potencian la interacción persona-sistema.

Bibliografía

[1] Andrés Aguirre, Rafael Fernández, and Carlos Grossy. USB4all. http://www.fing.edu.uy/inco/grupos/mina/pGrado/pgusb/. Last accessed, 09/15/09.

[2] Andrés Aguirre and Jorge Visca. lualibusb. http://sources.luadist.org/lualibusb-0.4/. Last accessed, 12/29/09.

[3] Marco Aiello. The role of web services at home. In *AICT-ICIW '06: Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services*, page 164, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Sergio Alcántara Segura. Proyecto de sistema de control domótico. http://www.gestiopolis.com/recursos4/archivo/deger/sistem.zip. Last accessed, 04/10/09.

[5] Sergio Alcántara Segura and Christal Berengena Moreno. Sistema de riego inteligente. http://www.gestiopolis.com/recursos4/docs/ger/sirin.pdf. Last accessed, 04/10/09.

[6] Askels. Home pac xp8741. http://askels.weblodge.net/product. Last accessed, 04/10/09.

[7] The Bluetooth Special Interest Group. Core specification v2.1 + edr. http://www.bluetooth.com/Bluetooth/Technology/Works/Core_Specification_v21__EDR.htm. Last accessed, 04/10/09.

[8] Dario Bonino, Emiliano Castellina, and Fulvio Corno. Dog: An ontology-powered osgi domotic gateway. In *Tools with Artificial Intelligence, 2008. ICTAI '08. 20th IEEE International Conference on*, volume 1, pages 157–160, 2008.

[9] Andre Bottaro, Anne Gerodolle, and Philippe Lalanda. Pervasive service composition in the home network. In *AINA 07: Proceedings of the 21st International Conference on Advanced Networking and Applications*, pages 596–603, Washington, DC, USA, 2007. IEEE Computer Society.

[10] Sistema Casa. Sistema casa. http://www.sistemacasa.it. Last accessed, 10/24/09.

[11] Computer Science and Engineering University of Texas. Intelligent environments. www.eecs.wsu.edu/ holder/courses/cse6362/spr03/lectures/Networks.ppt. Last accessed, 04/10/09.

[12] Switzerland. Computer Science Laboratory at the University of Applied Sciences of Technology NTB, Buchs. Java libusb. http://libusbjava.sourceforge.net/wp/. Last accessed, 10/20/09.

[13] Consumer Electronics Association. Ansi/cea standard - introduction to the cebus standard - ansi/cea-600.10. http://www.ce.org/Standards/ANSI-CEA-600.10_Preview.pdf. Last accessed, 04/10/09.

[14] Deyanira del Socorro, Roque Leslie, Eduardo Úbeda Sequeira, and Carlos Alberto Ortega Huembes. Zigbee - trabajo de técnicas de alta frecuencia. http://www.scribd.com/doc/4559979/Zigbee. Last accessed, 04/10/09.

[15] Jessica Díaz, Agustín Yagüe, Pedro Pablo Alarcón, and Juan Garbajosa. A generic gateway for testing heterogeneous components in acceptance testing tools. In *ICCBSS*, pages 110–119. IEEE Computer Society, 2008.

[16] Rebeca P. Díaz Redondo, Ana Fernández Vilas, Manuel Ramos Cabrer, José J. Pazos Arias, Jorge García Duque, and Alberto Gil Solla. Enhancing residential gateways: a semantic osgi platform, Jan/Feb 2008.

[17] Digital Living Network Alliance. Dlna. http://www.dlna.org/. Last accessed, 04/10/09.

[18] domoticacasera.com.ar. Domótica casera. http://www.domoticacasera.com.ar/index.php. Last accessed, 04/10/09.

[19] Domotica.Net. Lonworks. http://www.domotica.net/4234.html. Last accessed, 04/10/09.

[20] Domotica.Net. Lonworks conceptos básicos. http://www.domotica.es/lonworks. Last accessed, 04/10/09.

[21] Echonet Consortium. Echonet specification - part 1 echonet overview. http://www.echonet.gr.jp/english/spec/pdf/spec_v1e_1.pdf. Last accessed, 04/22/09.

[22] Electronicstalk. Express logic enhances threadx system. http://www.electronicstalk.com/news/exp/exp149.html. Last accessed, 04/10/09.

[23] eLua Devolper Team. elua. http://www.eluaproject.net/. Last accessed, 04/21/09.

[24] Emlix. μclinux description. http://emlix.com/soluclinux.html?language=en. Last accessed, 04/10/09.

[25] Real Academia Española. Diccionario real academia española. http://www.rae.es. Last accessed, 04/12/09.

[26] ETH Zürich, Department of Computer Science. R-OSGi. http://r-osgi.sourceforge.net/. Last accessed, 09/13/09.

[27] ETH Zürich, Institute for Pervasive Computing. jSLP. http://jslp.sourceforge.net/. Last accessed, 09/13/09.

[28] ETH Zürich, Institute for Pervasive Computing. Overview of a osgi implementation. http://concierge.sourceforge.net/. Last accessed, 04/18/09.

[29] Firenze Tecnologia. *Domotica, lo stato dell'arte*. Italy, 2008.

[30] The FreeBSD Project. The freebsd proyect. http://www.freebsd.org/. Last accessed, 04/18/09.

[31] Panagiotis Gouvas, Thanassis Bouras, and Gregoris Mentzas. An osgi-based semantic service-oriented device architecture. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *OTM Workshops (2)*, volume 4806 of *Lecture Notes in Computer Science*, pages 773–782. Springer, 2007.

[32] Granpyme.com. Dog domotica. http://www.granpyme.com/empresas/hogaral/blog/dog-domotica. Last accessed, 04/10/09.

[33] Green Hills Software. Threadx real-time operating system. http://www.ghs.com/products/rtos/threadx.html. Last accessed, 04/10/09.

[34] Hall, Dean - eLua Foro. Lua footprint. https://lists.berlios.de/pipermail/elua-dev/2009-January/000304.html. Last accessed, 04/10/09.

[35] HAVi. Havi sitio oficial. http://es.wikipedia.org/wiki/Middleware. Last accessed, 04/10/09.

[36] The home gateway initiative. Hgi. http://www.homegatewayinitiative.org/. Last accessed, 04/10/09.

[37] HomePNA. Homepna home page. http://www.homepna.org/en/index.asp. Last accessed, 04/10/09.

[38] IMarketing.es. Proyecto de sistema de control domótico. http://www.gestiopolis.com/recursos4/docs/ger/sistem.htm. Last accessed, 04/10/09.

[39] Java Community. Jsr 8: Open services gateway specification. http://jcp.org/en/jsr/detail?id=8. Last accessed, 29/08/09.

[40] Y. Kato, T. Ito, H. Kamiya, M. Ogura, H. Mineno, N. Ishikawa, and T. Mizuno. Home appliance control using heterogeneous sensor networks. pages 1–5, 2009.

[41] Alexei Krasnopolski. Remote services in osgi framework using rmi connection. http://crasnopolski.com/alpha/home.htm. Last accessed, 06/09/09.

[42] Christer Larsson. Osgi release 4, version 4.2. http://www.jfokus.se/jfokus09/slides/salong4/osgi_release4_v4.2.pdf. Last accessed, 29/08/09.

[43] Choonhwa Lee, Sunghoon Ko, Seungjae Lee, Wonjun Lee, and Sumi Helal. Context-aware service composition for mobile network environments. pages 941–952. 2007.

[44] Young-Hee Lee, Chae-Kyu Kim, and Kyeong-Deok Moon. Self-configurable service middleware for pervasive digital home environment. In *CIC 2003*.

[45] Johan Lilius and Ivon Porres Paltor. Deeply embedded python, a virtual machine for embedded systems. http://www.tucs.fi/magazin/output.php?ID=2000.N2.LilDeEmPy. Last accessed, 04/18/09.

[46] Linknx Developer Team. Knxweb. http://linknx.wiki.sourceforge.net/KnxWeb. Last accessed, 04/10/09.

[47] Emiliano López. Linux embebido. http://linuxemb.wikidot.com/toc. Last accessed, 04/10/09.

[48] D. Lopez-de Ipina, A. Almeida, U. Aguilera, I. Larizgoitia, X. Laiseca, P. Orduna, A. Barbier, and J.I. Vazquez. Dynamic discovery and semantic reasoning for next generation intelligent environments. In *Intelligent Environments, 2008 IET 4th International Conference on*, pages 1–10, 2008.

[49] Antonio E. Martínez, Ruben Cabello, Francisco J. Gómez, and Javier Martínez. Interact-ddm: A solution for the integration of domestic devices on network management platforms. In Germán S. Goldszmidt and Jürgen Schönwälder, editors, *Integrated Network Management*, volume 246 of *IFIP Conference Proceedings*, pages 485–488. Kluwer, 2003.

[50] Jordi Mayné. Sensores acondicionadores y procesadores de señal. http://www.scribd.com/doc/2948891/Sensores-2003. Last accessed, 04/22/09.

[51] Microsoft. Windows embebido. http://www.microsoft.com/windowsembedded/en-us/products/windowsce/default.mspx. Last accessed, 04/10/09.

[52] Vittorio Miori, Luca Tarrini, M. Manca, and Tolomei. Domonet: a framework and a prototype for interoperability of domotic middlewares based on xml and web services. In *Consumer Electronics, 2006. ICCE '06. 2006 Digest of Technical Papers. International Conference on*, pages 117–118, 2006.

[53] MisterHouse Developer Team. Misterhouse. http://misterhouse.wikispaces.com/x10+dimming. Last accessed, 04/10/09.

[54] Ana Isabel Molina Díaz, Miguel Ángel Redondo Duque, and Manuel Ortega Cantero. Abstract virtual reality for teaching domotics.

[55] Kyeong-Deok Moon, Young-Hee Lee, and Young-Sung Son. Universal home network middleware (uhnm) guaranteeing seamless interoperability among the heterogeneous home network middleware for future home networks. In *Consumer Electronics, 2003. ICCE. 2003 IEEE International Conference on 17-19 June 2003*, pages 306–307, 2003.

[56] Caio Augustus Morais Bolzani and Marcio Lobo Netto. The engineering of micro agents in smart environments. *Int. J. Know.-Based Intell. Eng. Syst.*, 13(1):31–38, 2009.

[57] Regina Motz, Alfredo Viola, Fernando Carpani, Jorge Abin, Oscar Sena, Gustavo Núñez, Alvaro Rodriguez, Alvaro Rettich, Marco Scalone, Guzmán Llambías, and Fernando Puig. Proyecto Camaleon. https://www.fing.edu.uy/inco/grupos/csi/wiki/Camaleon. Last accessed, 09/18/09.

[58] μclinux community. μclinux. http://www.uclinux.org/. Last accessed, 04/10/09.

[59] Long Nguyen Hoang. Middlewares for Home Monitoring and Control. http://www.tml.tkk.fi/Publications/C/23/papers/NguyenHoang_final.pdf. Last accessed, 04/10/09.

[60] Onlamp.com. Misterhouse. http://www.onlamp.com/pub/a/onlamp/2004/11/11/smrthome. Last accessed, 04/10/09.

[61] OpenDomo developer Team. Opendomo seguridad y domótica libre. http://www.opendomo.org/. Last accessed, 04/10/09.

[62] Opendomotica. Misterhouse. http://opendomotica.wordpress.com/. Last accessed, 04/10/09.

[63] Openmoko community. Openmoko. http://wiki.openmoko.org/wiki/Main_Page. Last accessed, 04/10/09.

[64] OpenWrt community. Openwrt. http://openwrt.org/. Last accessed, 04/10/09.

[65] Oscar. Oscar Bundle Repository. http://oscar-osgi.sourceforge.net/. Last accessed, 09/13/09.

[66] OSGi Alliance. Osgi alliance. http://www.osgi.org/. Last accessed, 04/10/09.

[67] OSGi Alliance. Osgi service platform core specification. http://www.osgi.org/Download/File?url=/download/r4v41/r4.core.pdf. Last accessed, 29/08/09.

[68] Ouaye.net. linkknx demo. http://ouaye.net/linknx/linknxwebsim/. Last accessed, 04/10/09.

[69] P2P Universal Computing Consortium. Invitation to pucc technology. http://www.pucc.jp/PUCC/Material/English/PUCC_brochureEnglish.pdf. Last accessed, 04/22/09.

[70] P2P(Peer to Peer) Universal Computing Consortium. Pucc. http://www.pucc.jp/PUCC/English/. Last accessed, 04/10/09.

[71] Mike Pall. The luajit project. http://luajit.org/. Last accessed, 04/10/09.

[72] Mauricio Esteban Pardo, Guillermo Enrique Strack, and Diego C. Martinez. A domotic system with remote access based on web services. In *Journal of Computer Science and Technology - JCS&T. Vol 8, Num 2*, pages 91–96, 2008.

[73] Paolo Pellegrino, Dario Bonino, and Fulvio Corno. Domotic house gateway. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1915–1920, New York, NY, USA, 2006. ACM.

[74] Politenico di Torino. Dog. http://elite.polito.it/content/category/8/20/72/. Last accessed, 04/10/09.

[75] Programming Tutorials. Bluetooth technology. http://progtutorials.tripod.com/Bluetooth_Technology.html. Last accessed, 04/10/09.

[76] The Knopflerfish Project. The white-board model. http://www.knopflerfish.org/releases/current/docs/osgi_service_tutorial.html#white. Last accessed, 10/24/09.

[77] Python Software Foundation. Python programming language – official website. http://www.python.org/. Last accessed, 04/18/09.

[78] QNX Software systems. Qnx neutrino rtos. http://www.qnx.com/products/neutrino_rtos/. Last accessed, 04/18/09.

[79] Jan S. Rellermeyer, Gustavo Alonso, , and Timothy Roscoe. Distributed Applications through Software Modularization. http://www.inf.ethz.ch/personal/troscoe/pubs/middleware07-rosgi.pdf. Last accessed, 10/20/09.

[80] Jan S. Rellermeyer and Gustavo Alonso. Concierge: a service platform for resource-constrained devices. *SIGOPS Oper. Syst. Rev.*, 41(3):245–258, 2007.

[81] Scott Rosenbaum, Scott Lewis, Markus Kuppe, David Bosschaert, and Tim Diekmann. Distributed OSGi Services. http://www.eclipsecon.org/2009/sessions?id=757. Last accessed, 09/13/09.

[82] Kevin Schultz. Java vms compared. http://java.dzone.com/articles/java-vms-compared. Last accessed, 09/13/09.

[83] SlugOS community. Slugos. http://www.nslu2-linux.org/wiki/SlugOS/HomePage. Last accessed, 04/10/09.

[84] SOF Developer Team. Service oriented framework. http://sof.tiddlyspot.com/. Last accessed, 04/18/09.

[85] Hiromitsu Sumino, Norihiro Ishikawa, Shingo Murakami, Takeshi Kato, and Johan Hjelm. Pucc architecture, protocols and applications. In *Consumer Communications and Networking Conference, 2007. CCNC 2007. 4th IEEE*, pages 788–792. IEEE Computer Society, 2007.

[86] Sun. Java 2 platform, micro edition - data sheet. http://www.sun.com/aboutsun/media/presskits/ctia2004/J2ME.pdf. Last accessed, 04/10/09.

[87] Sun. Jini. http://java.sun.com/developer/products/jini/. Last accessed, 04/10/09.

[88] Sun. Overview of java 2 platform, micro edition (j2me). JavaWireless-2nd.book Page 7. Last accessed, 04/10/09.

[89] Suncoast Linux Users Group. Historia de slugos. http://www.suncoastlug.org/history.php. Last accessed, 04/10/09.

[90] Daniel B. Suthers. Heyu - x10 automation for linux, unix, and mac os x. http://heyu.org/. Last accessed, 04/10/09.

[91] Taiyaki Developer Team. taiyaki x10 more easier! http://code.google.com/p/taiyaki/. Last accessed, 04/10/09.

[92] Luca Tarrini and Vittorio Miori. Home automation technologies and standards. http://hats.isti.cnr.it/. Last accessed, 04/10/09.

[93] Eiji Tokunaga, Hiroo Ishikawa, Makoto Kurahashi, Yasunobu Morimoto, and Tatsuo Nakajima. A framework for connecting home computing middleware. In *ICDCS Workshops*, pages 765–770. IEEE Computer Society, 2002.

[94] Trasteandounpoco. Acceder al bus knx con interfaz usb en nslu2. http://trasteandounpoco.blogspot.com/2009/02/acceder-al-bus-knx-con-interfaz-usb-en.html. Last accessed, 04/10/09.

[95] Trasteandounpoco. Instalar knxweb 0.6 en nslu2 (unslung). http://trasteandounpoco.blogspot.com/2009/02/instalar-knxweb-06-en-nslu2-unslung.html. Last accessed, 04/10/09.

[96] UAM Departamento Ingeniería en Informática. Inetract - an intelligent environment. http://odisea.ii.uam.es/. Last accessed, 04/10/09.

[97] USB Implementers Forum, Inc. Usb 3.0 specification. http://www.usb.org/developers/docs/. Last accessed, 04/10/09.

[98] Leonardo Vidal and Diego Fontanarossa. Descubrimiento de servicios en redes heterogeneas. Proyecto de Grado de la carrera de Ingeniería en Computación, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Montevideo, República Oriental del Uruguay. Julio de 2008.

[99] Wikipedia. Actuadores. http://es.wikipedia.org/wiki/Actuador. Last accessed, 04/12/09.

[100] Wikipedia. Automatic private internet protocol addressing. http://es.wikipedia.org/wiki/APIPA. Last accessed, 04/19/09.

[101] Wikipedia. Avahi (software). http://en.wikipedia.org/wiki/Avahi_(software). Last accessed, 04/19/09.

[102] Wikipedia. Bluetooth description. http://es.wikipedia.org/wiki/Bluetooth. Last accessed, 04/10/09.

[103] Wikipedia. Definición middleware. http://es.wikipedia.org/wiki/Middleware. Last accessed, 04/18/09.

[104] Wikipedia. Dlna. http://en.wikipedia.org/wiki/Digital_Living_Network_Alliance. Last accessed, 04/10/09.

[105] Wikipedia. Domótica. http://es.wikipedia.org/wiki/Domótica. Last accessed, 04/22/09.

[106] Wikipedia. Efecto Coriolis. http://es.wikipedia.org/wiki/Efecto_Coriolis. Last accessed, 04/22/09.

[107] Wikipedia. Efecto Hall. http://es.wikipedia.org/wiki/Efecto_Hall. Last accessed, 04/22/09.

[108] Wikipedia. I2c. http://es.wikipedia.org/wiki/I^2C. Last accessed, 04/10/09.

[109] Wikipedia. Ieee 1394. http://es.wikipedia.org/wiki/IEEE_1394. Last accessed, 04/10/09.

[110] Wikipedia. Knx. http://en.wikipedia.org/wiki/KNX. Last accessed, 04/10/09.

[111] Wikipedia. Knx eib. http://es.wikipedia.org/wiki/Bus. Last accessed, 04/10/09.

[112] Wikipedia. Lonworks. http://en.wikipedia.org/wiki/Lonworks. Last accessed, 04/10/09.

[113] Wikipedia. Openwebnet. http://es.wikipedia.org/wiki/Openwebnet. Last accessed, 04/10/09.

[114] Wikipedia. Openwrt. http://es.wikipedia.org/wiki/OpenWrt. Last accessed, 04/10/09.

[115] Wikipedia. Perl. http://en.wikipedia.org/wiki/Perl. Last accessed, 04/18/09.

[116] Wikipedia. Php. http://es.wikipedia.org/wiki/.php. Last accessed, 04/18/09.

[117] Wikipedia. Rs232. http://es.wikipedia.org/wiki/COM1. Last accessed, 04/10/09.

[118] Wikipedia. Sensores. http://es.wikipedia.org/wiki/Sensor. Last accessed, 04/11/09.

[119] Wikipedia. Spi. http://es.wikipedia.org/wiki/Serial_Peripheral_Interface. Last accessed, 04/10/09.

[120] Wikipedia. Upnp. http://es.wikipedia.org/wiki/UPnP. Last accessed, 04/10/09.

[121] Wikipedia. Usb. http://es.wikipedia.org/wiki/USB. Last accessed, 04/10/09.

[122] Wikipedia. Windows ce. http://en.wikipedia.org/wiki/Windows_CE. Last accessed, 04/10/09.

[123] Wikipedia. X10. http://es.wikipedia.org/wiki/X10. Last accessed, 04/10/09.

[124] Wikipedia. Zeroconfig. http://es.wikipedia.org/wiki/Zeroconf. Last accessed, 04/18/09.

[125] Xataka. Usb 3.0 a fondo. http://www.xataka.com/accesorios/usb-30-a-fondo. Last accessed, 04/10/09.

[126] Xataka. Zigbee. http://www.xataka.com/tag/zigbee. Last accessed, 04/10/09.

[127] Open Building Information Xchange. oBIX sitio oficial. http://www.obix.org/default.htm. Last accessed, 04/22/09.

[128] Theodore B. Zahariadis. *Home Networking Technologies and Standards.* Artech House, 2003.

Glosario

ADC Analog-to-Digital Converter

AJAX Asynchronous JavaScript And XML

API Application Programming Interface

APIPA Automatic Private IP Addressing

AWT Abstract Window Toolkit

BPEL Business Process Execution Language

BSD Berkley Software Distribution

CDC Connected Device Configuration

CEBus Consumer Electronic Bus

CLDC Connected, Limited Device Configuration

CLR Common Language Runtime

CPU Central Process Unit

DAC Digital-to-Analog Converter

DC Direct Current

DCE Data Communication Equipment

DHG Domotic House Gateway

DLNA Digital Living Network Alliance

DMP Digital Media Player

DMS Digital Media Server

DOG Domotic OSGi Gateway

domoML Domotics Markup Language

DSP Digital Signal Processor

DSS Diagrama de Secuencia de Sistema

DTE Data Terminal Equipment

ECF Eclipse Communication Framework Project

ECHOnet Energy Conservation and Home Network

EHS European Home Systems

EIA Electronic Industries Alliance

EIB European Installation Bus

eLua Embedded Lua

EMI ElectroMagnetic interference

EMI Electromagnetic Interference

EOS Embedded Operating System

HATS Home Area Network

HAVi Home Audio Video interoperavility

HGI Home Gateway Initiative

HomePNA Home Phoneline Networking

HTTP Hypertext Transfer Protocol

IDE Integrated development environment

IEEE Institute of Electrical and Electronics Engineers

IIS Internet Information Services

IP Internet Protocol

IrDA Infrared Data Association

ITEA Istituto Trentino Edilizia Abitativa

J2EE Java 2 Platform, Enterprise Editio

J2ME Java 2 Micro Edition

J2SE Java 2 Standar Edition

JAR Java Archive

JCP Java Community Process

JNI Java Native Interface

JSR Java Specification Request

JVM Java Virtual Machine

KO Kernel Object

LDAP Lightweight Directory Access Protocol

LDR Light-Dependent Resistor

LED Light Emitting Diodes

LOC Lines Of Code

LuaVM Lua Virtual Machine

LVDT Linear variable differential transformer

M2M Middleware-to-Middleware

MAC-1 Message Authentication Code

MAC Media Access Control

MD5 Message-Digest Algorithm 5

MIB Management Information Bases

MIDP Mobile Information Device Profile

MIPS Microprocessor without Interlocked Pipeline Stages

NAS Network-attached storage

NIO Non blocking Input Output

NTC Negative Temperature Coefficient

OLPC One Laptop per Child, es una empresa que desarrolla laptops para niños que es muy utilzada en Uruguay.

OO Object Oriented

OBR OSGi Bundle Reposiroty

OSGI Open Services Gateway Initiative

OSI Open System Interconnection

OWL Ontology Web Language

P2P Peer to peer

P2P Peer-To-Peer

PAN Personal Area Network

PHP Personal Home Page

PIC Programmable Integrated Circuit

PTC Positive Temperature Coefficient

PUCC P2P Universal Computing Consortium

PVM Python Virtual Machine

RC Release candidate (RC) o versión candidata de puesta en producción.El número representa el número de versión candidata.

RF Radio Frecuencia

RMI Remote Method Invocation

RTD Resistance Temperature Detector

RTOS Real-Time operating system

RVDT Rotatory Variable Differential Transformer

SCS Sistema Cableado Simplificado

SDP Service Discovery Protocol

SeSODA Semanticaly Enabled SODA

SLP Service Location Protocol

SMS Short Message Service

SNMP Simple Network Management Protocol

SOAP Simple Object Access Protocol

SOA Service-Oriented Architecture

SODA Service Oriented Device Architecture

SOF Service Oriented Framework

SO Shared Object

SPI Serial Peripheral Interface

SSDP Simple Service Discovery Protocol

SSL Secure Socket Layer

TCP Transmission-Control-Protocol

TM TechManager

UDP User Datagram Protocol

UHNM Universal Home Network Middleware

UPnP Universal Plug and Play

URI Uniform Resource Identifier

URL Uniform Resource Locator

USB Universal Serial Bus

UTP Unshielded Twisted Pair

VAN Vehicle Area Network

VESA Video Electronics Standards Association

VM Virtual Machine

VSG Virtual Service Gateway

WAP Wireless Application Protocol

WPAN Wireless Personal Area Network

WSCDL Web Services Choreography Description Language

WS Web Services

XML Extensible Markup Language

Apéndices

Apéndice A

Sensores y Actuadores

En este apéndice se presenta la definición de sensores en el marco de la domótica, los distintos tipos de sensores, los actuadores y también sus tipos.

A.1. Sensores

Dentro del contexto de la automatización, un sensor es un dispositivo capaz de transformar un estímulo físico en una señal eléctrica. Los sensores son un subconjunto de los transductores (dispositivo capaz de transforman una magnitud en otra). Los sensores se pueden clasificar en digitales o analógicos:

- Los sensores digitales transforman sus datos de tal forma que pueden ser procesados directamente por microcontroladores o DSP (*Digital Signal Processor*), luego de un pre-procesamiento o adaptación (filtrado, amplificación).
- Los sensores analógicos, transforman sus datos a medidas que necesariamente deben pasar por un conversor analógico-digital ADC (*Analog-to-Digital Converter*) antes de ser interpretados.

Los sensores[50] son un sub conjunto de los transductores (dispositivo capaz de transforman una magnitud en otra).

A.1.1. Tipos de Sensores

Existe una gran variedad de sensores para medir diversas magnitudes físicas. Una magnitud física puede ser medida por varios sensores distintos, de forma directa o indirecta a través de un transductor.

A continuación se enumeran algunas magnitudes físicas y transductores capaces de medirlas.

- Temperatura: Comúnmente la temperatura es medida de forma analógica con un termómetro que es un sensor analógico de esta magnitud, sin embargo también se utilizan otro tipo de sensores:
  - Termopares: utilizan la diferencia de tensión (medida en volts, V) generada por los metales a distintas temperaturas para medir esta

magnitud. Esto se debe a que la cantidad de electrones libres depende de la temperatura del medio.

- Resistivos: RTD (*Resistance Temperature Detector*) y termisores (se clasifican en 2 tipos NTC: *Negative Temperature Coefficient* y PTC: *Positive Temperature Coefficient*): miden la resistencia eléctrica (medida en Ohms, Ω) de los materiales (magnitud que depende de la temperatura).
- Semiconductores: basada en la las propiedades conductivas de los diodos comportándose como un aislante o un conductor dependiendo de la temperatura, mide la diferencia de corriente (en amperes, A).

- Humedad: Se denomina humedad a la cantidad de vapor de agua en el aire. Se mide con sensores:
  - Capacitivos: en este caso la magnitud humedad se traduce en la capacidad (medida en faradios F) del condensador.
  - Resistivos: la humedad también afecta la resistencia eléctrica de ciertos materiales.
- Presión: La presión es la fuerza ejercida, para medirla hay sensores:
  - Resistivos, se utilizan membranas que hacen variar el valor de resistencias cuando se las somete a presión.
  - Piezoeléctricos (Piezo-Cerámicos/Multicapa), son sensores que entregan una señal eléctrica cuando se los estimula con una presión.
- Posición: Esta magnitud está estrechamente asociada con la velocidad. Ya que calculando la velocidad en la que se desplaza un objeto se puede dar una ubicación relativa. La posición se puede medir en forma lineal o angular.
  - Existen sensores inductivos, es decir que inducen un voltaje:
    - Generador síncrono: es un transductor que al aplicársele una velocidad angular entrega un voltaje inducido (transforma energía mecánica en energía eléctrica).
    - RVDT (*Rotatory Variable Differential Transformer*) y LVDT (*Linear Variable Differential Transformer*) ambos dada una variación en la posición (el primero linear y el segundo angular) retornan una variación de voltaje.
    - Inductosyns Lineales y Inductosyn Rotatorios (ídem anterior).
  - Existen sensores que varían sus resistencia de acuerdo a los cambios de ubicación:
    - Potenciómetros: se utilizan potenciómetros particulares que dada una entrada DC (*Direct Current*) constante la salida es proporcional al ángulo.
  - Otros sensores están basados en principios magnéticos:
    - Magneto-resistencias: son dispositivos cuya capacidad resistiva varía según el campo magnético que se le aplique.

- Brújula Electrónica: se utiliza la polarización de la tierra para el posicionamiento y navegación mediante variaciones magnéticas.
- efecto Hall: existen sensores basados en este fenómeno físico (dice que si existe una corriente a la cual se le aplica un campo magnético perpendicular, entonces se genera un voltaje perpendicular a ambas).

- Por último hay sensores hechos en base a principios ópticos:
  - Foto-interruptores de barrera: se coloca un emisor infrarrojo y un foto-transistor enfrentados pero separados, se detecta cuando el haz de luz es cortado.
  - Foto-interruptores reflectivos: se colocan un emisor y un receptor infrarrojos en un mismo plano, el receptor detecta la luz reflejada (no es directa) por el emisor.
  - Encoders ópticos: es un transductor que convierte tanto la distancia lineal como angular a un código binario resultado de la recepción de la luz reflejada por un emisor. Generalmente son armados con Foto-interruptores y discos concéntricos.

- Sensores de Movimiento (Velocidad y Aceleración)
  Varios de los sensores anteriormente vistos sirven para medir esta magnitud: electromecánicos, piezo-eléctricos, capacitivos y usando el efecto Hall [107]. A continuación se detallan los que aún no han sido explicados.

  - Giróscopo: para realizar medidas de velocidad angulares, mide la velocidad con la que gira sobre su propio eje, mide cambios de inclinación o cambios de dirección angular, están basados en el efecto Coriolis (describe la aceleración de un objeto sobre otro en rotación, ver "*Efecto Coriolis*"[106]).

- Luz: es una oscilación electromagnética, con una longitud de onda perceptible por el ojo humano. Para medir se tiene:

  - foto-diodo, es un sensor optoelectrónico que genera un pequeña corriente eléctrica que depende directamente de la luminosidad incidente.
  - Foto-resistencia, es un sensor resistivo o LDR, la resistencia entre sus polos está inversamente relacionada a la luz recibida.
  - Foto-transistor, es un transistor sensible a la luz.

En el cuadro A.1 se clasifican los transductores de acuerdo a las magnitudes que miden y la unidad de la magnitud resultante (salida).

Existen otras magnitudes que también son medibles como ser el caudal que tiene un río, la corriente que pasa por un circuito, la presencia o ausencia de personas en un determinado lugar, la proximidad de las mismas, los ruidos o propiedades químicas de una solución, presencia de humo, entre varios más. Se puede seguir profundizando en la referencia "Sensores"[118].

Magnitud	Transductor	Salida
Temperatura	Termopares Resistivos Semiconductores	Tensión (V) Resistencia (Ω) Corriente (A)
Humedad	Capacitivos Resistivos	Capacidad (F) Resistencia (Ω)
Presión	Resistivos Piezoeléctricos	Resistencia (Ω) Tensión (V)
Posición	Inductivos Resistivos Magnéticos Ópticos	Tensión (V) Resistencia (Ω) Resistencia (Ω) \ Tensión (V) \Otros Cierre o Apertura del circuito \ Código Binario
Movimiento	Giróscopo	Frecuencia (Hz)
Luz	Foto-diodo Foto-resistencia Foto-transistor	Corriente (A) Resistencia (Ω) Cierre o Apertura del circuito

Cuadro A.1: Transductores según magnitud que miden.

A.2. Actuadores

En el artículo "*Actuadores*"[99], se denominan actuador a aquellos dispositivos capaces de generar una fuerza y afectar el ambiente mediante un proceso automatizado a partir de líquidos, energía eléctrica o gaseosa.

A.2.1. Tipos de Actuadores

Un actuador es un dispositivo capaz de realizar un cambio en su entorno, por analogía con los sensores, hacen variar alguna magnitud física. Los más conocidos son los utilizados en robótica (principalmente para movimientos o desplazamientos). Un simple LED o una resistencia pueden ser considerados actuadores siempre y cuando la acción deseada sea la emisión de luz o calor respectivamente.

Salvo actuadores digitales, es necesario introducir un DAC (*Digital-to-Analog Converter*) para que el actuador pueda interpretar el comando recibido.

A continuación se enumeran los distintos tipos de actuadores según su funcionamiento.

- Hidráulico: Son aquellos que funcionan en base a fluidos a presión.
  - Cilindro hidráulico: permite ejercer presión para empujar o contraer.
  - Motor hidráulico: mediante la presión genera un movimiento rotatorio.
- Neumáticos: Son aquellos que transforman el aire comprimido en trabajo mecánico. Esencialmente funcionan del mismo modo que los hidráulicos,

pero suelen tener otros fines. En esta categoría se encuentran los fuelles, diafragmas y músculos artificiales de hule.

- Eléctricos: Son todos aquellos que utilizan la energía eléctrica para modificar alguna magnitud física. Estos están más íntimamente ligados al proyecto en cuestión. De todas maneras, se pueden utilizar combinaciones como la activación de un pistón hidráulico desde un actuador eléctrico.

  - Motores, son aquellos que provocan un desplazamiento, al ser utilizados en robótica hay una amplia gama de acuerdo a su fin. Pueden ser Rotativos (Motores DC, Steppers o Servomotores, sincromotor, resolver) o Lineales. También existe otro tipo de motores que son los músculos artificiales que emulan los músculos biológicos y alambres musculares [99].
  - Únicamente a modo de completar el análisis se citan los ejemplos de actuadores utilizados en la referencia "*Abstract virtual reality for teaching domotics*"[54] que se centra en el estudio de un ambiente domótico: sistemas de calefacción, aire acondicionado, luminarias, apertura y cierra persianas, alarmas, etc.

Apéndice B

Estándares y *middlewares*

En el este apéndice se encuentra el relevamiento y una breve descripción de estándares y *middlewares* relacionados con la domótica: Home Audio Video interoperavility, IEEE 1394, RS232, Universal Serial Bus, Inter-Integrated Circuit, Serial Peripheral Interface, Bluetooth, ZigBee, Consumer Electronic Bus, Digital Living Network Alliance, KNX, LonWorks, OpenWebNet, X10, Universal Plug and Play, Open Services Gateway Initiative, Service Oriented Framework, P2P Universal Computing Consortium, Zeroconf, Service Location Protocol y Jini entre otros de menor importancia.

B.1. Bluetooth

Bluetooth fue publicado en el año 1999, es una estándar industrial para Redes Inalámbricas de Área Personal (WPANs del inglés *Wireless Personal Area Network*) que posibilita la transmisión de voz y datos entre diferentes dispositivos mediante un enlace por radiofrecuencia segura. Utiliza un rango de frecuencia en 2,4GHz, este rango de frecuencia se puede utilizar en cualquier parte del globo sin restricciones legales [102]. Dado que fue diseñado para PAN (*Personal Area Network*) sólo alcanza unos pocos metros. La versión actual de Bluetooth es la 2.1.

Las redes formadas por dispositivos Bluetooth se llaman Piconet[75], las cuales pueden tener entre 2 y 8 dispositivos interconectados. Las redes están formadas por un dispositivo que actúa como *master* y los otros dispositivos actúan como *slave*.

Se puede profundizar en "*Core Specification v2.1 + EDR*"[7].

B.2. *Consumer Electronic Bus*

Consumer Electronic Bus (CEBus)[13], nacido en el año 1984, es un estándar vigente en los Estados Unidos, también conocido como EIA600. Define una serie de estándares eléctricos y protocolos de comunicación para trasmitir información sobre el *bus*.

Es independiente del medio físico por lo que puede ser utilizado tanto sobre red eléctrica, par trenzado de cobre o UTP (*Unshielded Twisted Pair*), fibra óptica u otro medio de transmisión ver figura B.1 obtenida de "*Intelligent*

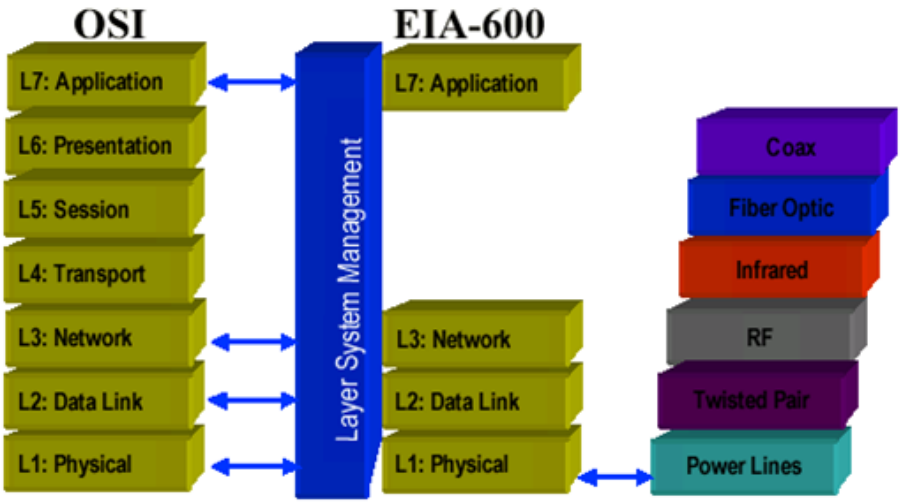


Figura B.1: Capas de CEBus.

Environments" [11]. En todos los medios físicos, la información de control y datos se transmite a la misma tasa de transferencia, 8000b/s. Sin embargo también se permiten canales para transmitir audio o vídeo u otra funcionalidad que requiera mayor velocidad de transmisión.

El estándar define un *stack* bastante completo y desarrollado, comparable con el modelo OSI(Open System Interconnection), como se muestra en al figura B.1 extraída de "*Intelligent Environments*" [11].

B.3. *Digital Living Network Alliance*

Digital Living Network Alliance (DLNA) es un estándar definido en la industria en el año 2004 para la interacción entre dispositivos basado en gestión de contenidos. Sus mayores virtudes son su fácil configuración y su versatilidad.

Este estándar o protocolo utiliza un subconjunto de UPnP (*Universal Plug and Play*) permitiendo localizar y enlazar los dispositivos compatibles en la red local como se muestra en la figura B.2, extraída de la página oficial de DLNA [17]. Debido al uso de UPnP corre únicamente sobre redes IP (*Internet Protocol*). Dentro del protocolo se definen 2 tipos de dispositivos (esto no impide que un solo dispositivo implemente ambos), estos son:

- DMP (*Digital Media Player*)[104] - donde se reproducen los contenidos. Estos son los clientes, por ejemplo televisores o equipos de audio.

- DMS (*Digital Media Server*)[104] - donde se guardan los contenidos y a los cuales se les puede pedir los diferentes contenidos. Estos actúan como servidor.

Figura B.2: Interacción entre los diferentes dispositivos DLNA.

B.4. *Home Audio Video interoperavility*

Home Audio Video interoperavility (HAVi)[35] es un estándar definido en el año 1998 por ocho de las empresas más importantes de la industria audio-visual, entre las cuales se encuentran Mitsubishi, RCA y Sony entre otros.

El estándar define la interoperabilidad entre dispositivos de audio y vídeo permitiendo de forma *plug-and-play,* interconectar cualquier dispositivo que lo soporte.

Además describe una arquitectura centralizada donde el *backbone* es IEEE1394 (por más detalles, ver "*A Framework for Connecting Home Computing Middleware*" [93]).

B.5. IEEE 1394

IEEE 1394 (conocido como *FireWire* por Apple Inc. y como i.Link por Sony)[109] es un estándar multiplataforma para entrada/salida de datos en serie a gran velocidad. Actualmente hay varias versiones, siendo la velocidad con la cual trabajan su mayor diferencia:

- FireWire 400 tiene un ancho de banda de 400Mb/s, lanzado en 1995.
- FireWire 800 tiene una velocidades de 768.5Mb/s, lanzado en 2000.

Suele utilizarse para conectar dispositivos digitales como cámaras digitales, discos externos y cámaras de video a la computadoras. El estándar define una dirección única de dispositivo, llamada "IEEE EUI-64 exclusivo (una extensión de las direcciones MAC Ethernet de 48-bit)" lo que permite interconectar hasta 63 dispositivos al mismo tiempo. Además su formato permite realizar conexiones P2P(*Peer-To-Peer*) consumiendo pocos recursos en los dispositivos involucrados.

B.6. *Inter-Integrated Circuit*

Inter-Integrated Circuit (I^2C)[108] es un bus de datos de comunicación serie desarrollado por Philips en 1992, el cual es usado en la industria para interconectar microcontroladores y sus periféricos en sistemas integrados. Tiene una velocidad de 100 Kb/s en modo estándar aunque también permite velocidades de 3.4 Mb/s.

Es un protocolo en el cual los dispositivos toman los roles de *Master* o *Slave*. El modo *Master* es el que controla la señalización del canal y mantiene las comunicaciones entre los dispositivos por lo cual necesariamente debe contar con mayor poder de procesamiento que el *Slave*.

B.7. Jini

Jini también llamado Apache River[87], fue desarrollado por Sun Microsystems en el año 1998, define una arquitectura de sistemas distribuidos basado íntegramente en Java que permite a los sistemas descubrir y publicar servicios. Requiere que cada aparato corra una JVM o esté asociado con otro aparato que pueda correr una JVM para intercomunicarse.

Según el *paper* "*A Framework for Connecting Home Computing Middleware*"[93], Jini es un buen *middleware* para integrar servicios basados en Java.

B.8. KNX

KNX está estandarizado por Konnex Association desde el año 1999. Es un estándar definido a nivel Europeo que se establece bajo el techo de tres de los principales sistemas de domótica usados en Europa: EIB (*European Installation Bus*), EHS(*European Home Systems*), y BatiBus [95]. Este estándar ha sido adoptado por varias empresas que han desarrollado productos completamente operables entre sí [110] y han constituido la más amplia oferta de dispositivos para la automatización.

Konnex se basa en un arquitectura distribuida constituida de una red de dispositivos que interactúan entre sí mediante una serie de protocolos y señales que están establecidos en el estándar.

Utiliza medios de comunicación variados[29]: principalmente el UTP, la red eléctrica (*power line*) y más recientemente RF (radio frecuencia, 868MHz).

Konnex determina tres formas de implementar los dispositivos con lo cual la configuración de los mismos varía según cada caso [29]:

- *Automatic Mode*, es una configuración *plug-and-play*, la cual permite una rápida instalación y los consumidores finales pueden por sí mismos agregar más dispositivos a la red domótica.

- *Easy Mode*, dispone de una serie de perfiles de funcionamiento que permiten efectuar la configuración de un modo más sencillo.

- *System Mode,* es el más completo pero requiere de un técnico especializado para su configuración el cual accede mediante una computadora.

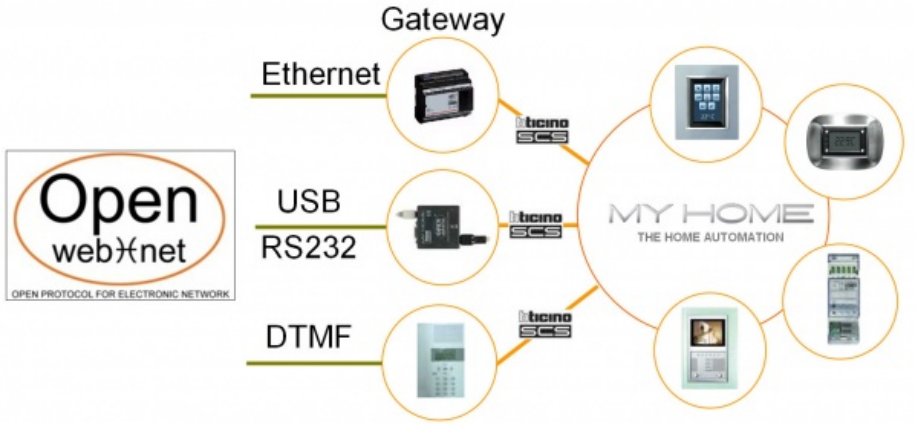


Figura B.3: OpenWebNet *gateways*.

B.9. LonWorks

Este estándar define una plataforma especialmente diseñada para abordar el rendimiento, la fiabilidad, la instalación, el mantenimiento y las necesidades de aplicaciones de control. La plataforma se basa en un protocolo creado por Echelon Corporation en el año 1999[112], que luego se transformó en LonMarks International, para la creación de redes independientes de medios físicos como UTP, la red eléctrica, fibra óptica y RF. Es un estándar popular dentro de la domótica.

Se trata de un estándar abierto con arquitectura distribuida por lo que no es necesario pagar ningún *royality* para su utilización [20]. Garantiza la interoperabilidad de los dispositivos a través de la estandarización de protocolo de comunicación llamado *LongTalk*. Cada dispositivo utiliza un chip llamado *Neuro Chip* el cual tiene tres procesadores, dos dedicado a la implementación del protocolo y otro genérico.

Los dispositivos en LongWorks se pueden clasificar en dominios y en grupos lo cual permite una mejor comunicación entre los mismos. Éstos cuentan con una dirección física llamado *Neuron ID*, que es único en todo el mundo. Otra dirección de dispositivo que es asignada cuando se agrega a una red. Por último, hay una dirección de grupo para cuando se quiere utilizar *Multicast* y direcciones de *Broadcast* [19].

B.10. OpenWebNet

En el año 2000, BTicino desarrolla un protocolo de comunicación llamado OpenWebNet. Este nace para permitir la interacción con todas las funciones disponibles en el sistema domótico MyHome (ver E.2.1) basado en el *bus* SCS (originalmente "Sistema Cablaggio Semplificato" o Sistema Cableado Simplificado).

Originalmente OpenWebNet proveía *gateways* de SCS con Ethernet, USB, RS232, y DTMF (*Dual-Tone Multi-Frequency*) con lo cual permite que sea independiente del medio físico, como se muestra en la figura B.3.

La evolución reciente permite utilizar el protocolo OpenWebNet[113] para interactuar con KNX y DMX (Digital MultipleX) mediante los *gateway* adecuados.

B.11. Open Services Gateway Initiative

Open Services Gateway Initiative (OSGi) es una plataforma creada en el año 1999, que permite tener varias aplicaciones que brinden servicios que interactúen entre sí.

La plataforma OSGi se ejecuta sobre Java, por lo que requiere una JVM. Si bien esto presenta una gran desventaja, el objetivo es poder implementar servicios basados en esta plataforma sobre versiones livianas o recortadas de JVM.

Se puede ver una implementación para bajos recursos de OSGi en el sitio "*Overview of a OSGI implementation*"[28].

La arquitectura de OSGi está constituida por tres grandes entidades:

- Proveedor de servicios y de redes.
- Servicio de *gateway*.
- Control de redes internas.

Las aplicaciones que se desarrollan como proveedores de servicios que se quieran incluir en la plataforma se implementan en forma de *bundle* y corren en forma independiente con respecto al resto de las aplicaciones que se encuentran dentro de la plataforma. Los servicios se pueden utilizar de forma dinámica ya que existen diferentes *class loaders* que cargan las aplicaciones en diferentes *sandboxs* y permite registrar y publicar los servicios que cada aplicación quiera compartir.

OSGi permite que las aplicaciones cooperen entre sí, dentro y fuera de la plataforma, a través de interfaces declaradas en archivos de configuración.

Para profundizar más sobre OSGi ir a la referencias "*Home Networking Technologies and Standards*"[128], "*OSGi Pagina oficial*"[66] y/o apéndice G.7.

B.11.1. Service Oriented Framework

En el año 2008 aparece *Service Oriented Framework* (SOF), un *framework* de desarrollo hecho en C++ estándar que permite desarrollar orientado a servicios independientemente del sistema operativo de base. La API de SOF es muy similar a la de OSGi. En este documento se ha incluido dentro de OSGi dado que este *framework* conceptualmente es lo mismo que OSGi con la gran diferencia de que gracias a estar escrito en C++, no corre sobre una máquina virtual por lo que, tanto el *footprint.* como el *overhead* de todo el sistema son menores que en las implementaciones sobre Java. Para profundizar ir a la referencia "*Service Oriented Framework*"[84].

Si bien esta plataforma se proclama independiente del sistema operativo, al momento del relevamiento del estado del arte, aún no se había migrado para que corriera en Linux. Esta información fue provista por uno de los, desarrolladores de SOF, ver `http://sourceforge.net/projects/sof/forums/forum/783059/topic/3439800`.

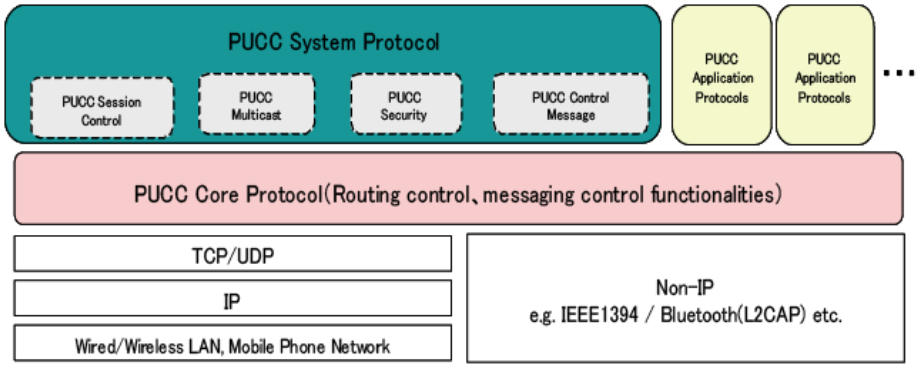


Figura B.4: *Stack* del protocolo de la plataforma de servicio PUCC.

B.12. P2P Universal Computing Consortium

P2P Universal Computing Consortium [70] es un consorcio establecido en el año 2005 que busca realizar un estándar para la comunicación de dispositivos en redes heterogéneas mediante su propio *framework*.

PUCC apunta a facilitar el trabajo de implementar nuevas aplicaciones que utilicen diversos protocolos. Para poder realizarlo, PUCC propone utilizar un *gateway*, el cual interconecta los diferentes protocolos y dispositivos.

Propone una capa P2P *networking* sobre todas las tecnologías implicadas con el fin de establecer una red heterogénea de comunicación. En esta capa permite comunicar redes IP con redes que no son IP (como ser ZigBee, X10, IrDA, etc.) de forma transparente. Además de esto, al ser P2P la plataforma PUCC prevé la capacidad de ofrecer interacción de forma descentralizada, sin necesidad de contar con un servidor centralizado el cual suele ser un punto crítico del sistema.

Esta diseñado para provee las siguientes funcionalidades:

- Descubrimiento de dispositivos y servicios.
- Enrutamiento dinámico.
- Invocación de funcionalidades de servicios mediante *comunicaciones ad-hoc*.

El *stack* de la plataforma de servicios descrita por PUCC consta de tres grandes partes: *Core Protocol*, *System Protocol* y *Application Protocol* el cual se muestra en la figura B.4 obtenida de "Invitation to PUCC technology"[69].

B.13. RS232

RS-232 (también conocido como EIA: Electronic Industries Alliance RS-232, EIA RS232 o EIA232)[117] es una interfaz que designa una norma para el intercambio serie de datos binarios entre un DTE (*Data Terminal Equipment*, equipo terminal de datos) y un DCE (*Data Communication Equipment*, equipo de comunicación de datos).

Es uno de los protocolos más antiguos ya que fue creado en el año 1962 por EIA, y en año 1969 define su revisión C. También es uno de los protocolos más

simples. Está cayendo en desuso para los usuarios comunes (las PC y *laptops* ya no traen dicha interfaz), pero es muy utilizado en la comunicación entre microcontroladores ya que muchos incluyen módulos especializados para este tipo de conexión.

B.14. *Serial Peripheral Interface*

Serial Peripheral Interface (SPI)[119] es un estándar que define un *bus* de comunicación principalmente utilizado para la transferencia de información entre circuitos integrados en equipos electrónicos. Permite controlar casi cualquier dispositivo electrónico digital que acepte un flujo de bits serie regulado por un reloj.

La comunicación es *Full-Duplex* lo cual permite mayores velocidades que I^2C, pero no tiene control de flujo por *hardware* ni cuenta con señales de asentimiento.

B.15. *Service Location Protocol*

En el año 1999 aparece *Service Location Protocol* (SLP)[98] que es un mecanismo de descubrimiento de servicios que puede funcionar tanto en forma distribuida como con un directorio centralizado.

Provee un marco de trabajo flexible para otorgar a los *hosts* acceso a información acerca de la existencia, ubicación y configuración de servicios de la red. Para esto el solicitante indica el tipo de servicio deseado y un conjunto de atributos que describen el servicio. Basándose en esta descripción, SLP retorna la dirección de red del servicio al solicitante.

B.16. *Universal Plug and Play*

Universal plug and play (UPnP) es una evolución a finales de los años 90 del estándar inicial de Microsoft que extiende el modelo de periféricos *Plug and Play*.

UPnP está desarrollado sobre el *stack* TCP/IP y tecnologías Web, incluyendo IP, TCP, UDP, HTTP, SOAP y XML. Los dispositivos puede dinámicamente unirse o abandonar redes, obtienen una dirección IP, transmiten sus servicios y cualidades a demanda, y descubren la presencia y las prestaciones de otros dispositivos [120]. Es un mecanismo de descubrimiento de servicios distribuido.

Los servicios en UPnP se describen mediante listas de funcionalidades que el dispositivo provee y una lista donde se muestra el estado del servicio en tiempo de ejecución. El servicio publica actualizaciones cuando su estado cambia y además permite que otros dispositivos puede suscribirse para recibir esta información.

UPnP usa *Simple Service Discovery Protocol* (SSDP) para el descubrimiento de servicios. Este protocolo anuncia la presencia de un dispositivo a otros y descubre otros dispositivos o servicios. SSDP usa HTTP sobre *multicast* y *unicast* UDP, conocidos como HTTPMU y HTTPU, respectivamente. Para seguir profundizando se recomienda ver las referencias: "*Descubrimiento de Servicios en Redes Heterogéneas*" [98] y "*Intelligent Environments*"[11].

B.17. *Universal Serial Bus*

Universal Serial Bus o Conductor Universal en Serie[121], abreviado comúnmente como USB, es un estándar que se utiliza para interconectar dispositivos. Fue creado en el año 1996 por siete empresas: IBM, Intel, Northern Telecom, Compaq, Microsoft, Digital Equipment Corporation y NEC.

Actualmente se encuentra en la versión 2.0 (que lleva casi 8 años en el mercado), la cual puede llegar en teoría a velocidades de 480Mb/s con transmisión de datos de forma uni-direccional (o *simplex*).

Intel está desarrollando la versión 3.0 que permitiría, entre otros, velocidades de 4.8Gb/s [97] y tráfico bidireccional (*dúplex*) gracias a la incorporación de una nueva linea de transmisión (ver por más información la referencia "*USB 3.0 a fondo*"[125]).

B.18. X10

Es un estándar creado en USA en el año 1975[123], pensado para comandar a distancia los interruptores.Soporta hasta 256 dispositivos máximo en una sola red. Es un protocolo de comunicación ampliamente usado en domótica por su bajo costo, instalación simple y su fácil gestión.

Originalmente este estándar comenzó soportando únicamente comunicación sobre sobre líneas eléctricas (*power* LAN). Si bien este medio de comunicación es el más usado por su facilidad de instalación (*no-new-wires*, que no requiere un tendido extra) y precio, hoy en día también utiliza otros medios.

EIB y luego Konnex [111] son considerados una evolución de este protocolo ya que permiten mejor *performance* y más adaptabilidad.

Actualmente este estándar es muy difundido a nivel global[29], por lo que resulta muy sencillo conseguir dispositivos que implementen este protocolo y además los costos asociados a la instalación y a la compra de los dispositivos son muy bajos.

B.19. Zeroconf

"Zeroconf se refiere a un conjunto de técnicas que permiten crear automáticamente una red IP operable, sin la necesidad de configuración especial o servidores dedicados" [124]. Es un mecanismo de descubrimiento de servicios del tipo totalmente distribuido.

La forma que tienen los dispositivos que quiere anunciar algún servicio es registrarlo localmente en el demonio que implementa el protocolo, con lo que queda pronto para responder pedidos. Estos servicios son descubiertos utilizando *multicast*.

Zeroconf comenzó en el año 2003 con dos grandes implementaciones, *Rendezvous* por parte de Apple Computer, *software* que fue luego renombrado a *Bonjour* y APIPA (*Automatic Private IP Addressing*) desarrollado por Microsoft. Bonjour es la solución de Descubrimiento de Servicios usada en los sistemas operativos de Apple para programas como iTunes, iPhoto, iChat, etc. Es un software *open-source* y está portado a Linux, Solaris, FreeBSD y Windows. Sin embargo APIPA[100] fue desarrollado originalmente para Windows y está

integrado a el desde la versión Windows 98 hacia adelante, siendo popular en este sistema operativo.

Además existe otra implementación llamada Avahi [101] que se basa en Bonjour, la cual también es *open-source*, es el sistema de descubrimiento de servicios que viene instalado en las máquinas OLPC (*One Laptop Per Child*). Por más información ver "*Descubrimiento de Servicios en Redes Heterogéneas*" [98].

B.20. ZigBee

ZigBee es un estándar de comunicación basado en IEEE 802.15.4 definido en el año 2003 para redes inalámbricas WPAN, creado por ZigBee Alliance. La velocidad de transmisión es de entre 20 kB/s y 250 kB/s y alcanza distancias de hasta 75 m.

Se puede implementar este protocolo con dispositivos de muy bajo consumo (un sensor que ZigBee con 2 pilas AA puede dura entre 6 meses y 2 años, según "[98]"[14]), tiene bajo costo y baja tasa de transferencia. Se perfila para ser el protocolo por excelencia en el ámbito hogareño. Se ha empezado a usar por las grandes empresas para sustituir el infrarrojo en la interacción con los equipos [126].

B.21. Otros

A continuación se hace una breve referencia a otros estándares y *middlewares* existentes pero menos difundidos.

B.21.1. Salutation

Es una arquitectura para buscar, descubrir y acceder a servicios de información, ver "*Intelligent Environments*" [11].

B.21.2. Konark

Es un mecanismo de descubrimiento de servicios para redes *ad-hoc* basado en mensajes SOAP. Por más información, ver referencia "*Descubrimiento de Servicios en Redes Heterogeneas*"[98].

B.21.3. *Home Gateway Initiative*

Home Gateway Initiative (HGI) es un foro de telcos (*telephone / telecomunication company*), vendedores y productores para definir especificaciones de los *Home Gateway*, los cuales tienen que tener determinadas interfaces para poder interactuar con los servicios que las telcos disponen. En general son interfaces LAN y WLAN [36].

B.21.4. VESA *Home Network*

El VESA *Home Network Committee* ha probado desarrollar redes, dispositivos, gestión y controles de redes interoperables basándose en las

especificaciones existentes para crean un estándar único de interoperabilidad [56].

B.21.5. *Home Phoneline Networking Alliance*

Home Phoneline Networking Alliance (HomePNA) desarrolla una estándar de red para distribuir información sobre cableado Coaxial y lineas telefónicas ya existentes en la mayoría de los hogares [37].

B.21.6. *Energy Conservation and Home Network*

Energy Conservation and Home Network (ECHOnet), prevé un estándar para utilizar los cableados existentes, modelos de desarrollos para vendedores de dispositivos e interconectar *middlewares* de forma económica [21].

B.21.7. Open Building Information Xchange

oBIX es un protocolo basado en WS y en un estándar propio de XML para facilitar el intercambio de información en el campo de la domótica [127].

Apéndice C

Sistemas Operativos Embebidos

En este apéndice se describe brevemente algunos EOS, en primer lugar cinco distribuciones de Linux: Openmoko, OpenWrt, SlugOS, μClinux, OpenDomo y luego ThreadX, Windows Embedded CE, QNX Neutrino NTOS y BSD.

C.1. Linux

En esta sección se describen algunas distribuciones de Linux para sistemas embebidos[47]. Lejos de ser un análisis exhaustivo, se detalla solamente los más populares y/o los que están más cerca de sistemas de domótica o de control.

C.1.1. Openmoko

Es una distribución[63] *open-source* de Linux basado en la versión 2.6 del *kernel*. Su aparición fue anunciada en el año 2006. Está especialmente desarrollada para celulares, lo que determina muchas de sus características.

C.1.2. OpenWrt

Es una distribución[114] *open-source* de Linux para dispositivos embebidos que apareció en el año 2006 con su distribución Whiterussian. Inicialmente fue creada para el *router*, modelo Linksys WRT54G, pero rápidamente fue soportando otros dispositivos.

En vez de crear un *firmware* estático, el cual ya trae todos los componentes estándares[64], permite utilizar una *suite* de paquetes para poder agregar los que sean necesarios para el uso que se le va a otorgar. El tamaño varía según los paquetes instalados, pero puede ocupar como mínimo 1 MB.

C.1.3. SlugOS

Se trata de otra distribución[83] *open-source* de Linux para sistemas embebidos inicialmente diseñadas para NAS(*Network-attached storage*). Sus comienzos se remontan al año 1997 [89].

C.1.4. μClinux

Original fue derivado de la versión de Linux 2.0 lanzada en el año 1998, para microcontroladores sin *Memory Management Units* (*MMUs*). El mismo incluye interesantes bibliotecas y *toolchains* para *kernel* 2.0, 2.4 y 2.6 [58]. Es *open-source* y tiene un *footprint* de 600 kB [24].

C.1.5. OpenDomo

OpenDomo es una distribución *open-source* de Linux que nace en el año 2006, especialmente desarrollada (activamente) para el control domótico.

Busca resolver la comunicación con los dispositivos mediante protocolos de capa 4 o superior en el modelo OSI. De esta forma no queda atado al canal de comunicación. Para manejar los dispositivos que no soportan comunicación IP es necesario desarrollar un adaptador o *driver*. Ejemplos de estos dispositivos pueden ser: controladores X10, *dongles* infrarrojo o Bluetooth instalados en uno o varios agentes OpenDomo, entre otros.

Estos adaptadores (desarrollados específicamente para cada *middleware*) consisten en un dispositivo físico accesible, el controlador pertinente y una fina capa lógica (habitualmente *scripts*) que adapta el controlador a la sintaxis OpenDomo.

Con ello se consigue una capa homogénea basada en IP, compatible con tecnologías como SSL (*Secure Socket Layer*), que permite abstraer y coordinar todos los elementos interconectados dentro del hogar digital.

En cuanto a su orientación a bajos recursos, tiene como importante limitante un *footprint* de 6,7 MB.

Esta información fue provista por Oriol Palenzuela, desarrollador de OpenDomo y el sitio oficial "*OpenDomo Seguridad y domótica libre*"[61].

Se probó dicha distribución en una máquina virtual, pero no la comunicación con los distintos *middlewares*.

C.2. ThreadX

ThreadX es un RTOS (*Real-Time Operating System*), lanzado al mercado en el año 1980 [22], el cual tiene una huella pequeña (2 kB), es compatible con una gran gama de microprocesadores, provee manejo de *threads, application timers* y cola de mensajes[33].Por lo cual tiene prestaciones muy interesantes en comparación a su pequeño *footprint*.

Esta distribución es paga, pero al comprarla te distribuyen el código fuente con *royality free*.

C.3. Windows Embedded CE

Sistema operativo de Microsoft para sistemas embebidos. La versión 1.0 vio la luz en el año 1997. Soporta múltiples arquitecturas y según Microsoft[51] su huella es muy pequeña, es de entre 300 kB y 700 kB. La compra de la licencia incluye el código fuente del *kernel* como *shared-souces*, no todo el código es distribuido [122].

Sistema operativo	*Kernel*		*Footprint*	Código fuente
	Tipo	**Arquitectura**		
FreeBSD	BSD	híbrido	5 MB	*open-source*
QNX Neutrinos NTOS	-	*Real-time microkernel*	-	libre para uso no comercial
OpenDomo	Linux	modular	6,7 MB	*open-source*
Openmoko	Linux	modular	-	*open-source*
OpenWrt	Linux	modular	1 MB	*open-source*
SlugOS	Linux	modular	-	*open-source*
ThreadX	-	*pico kernel*	2 kB	*royality free*
Windows Embedded CE	-	monolítico / híbrido	300 kB	*shared-source*
μClinux	Linux	modular	600 kB	*open-source*

Nota: El símbolo "-", significa que no se han encontrado referencias al respecto.

Cuadro C.1: Sistemas operativos y sus características.

C.4. QNX Neutrino NTOS

Es un sistema operativos con arquitectura de *microkernel*[78]. La primera versión es del año 1982. Puede correr sobre una amplia gama de arquitecturas entre ellas: MIPS, PowerPC, SH-4, ARM, StrongArm, XScale y x86. Desde finales del año 2007 su código está disponible para uso no comercial.

C.5. BSD

BSD(Berkley Software Distribution) es un sistema operativo *open-source,* desarrollado por Berkley, que originalmente fue una extensión del sistema operativo UNIX. Dio origen a varios sistemas operativos, entre ellos: SunOS, FreeBSD, NetBSD, OpenBSD y Mac OS X.

FreeBSD[30], empezó en el año 1993 y se trata de un sistema operativo basado en BSD Lite, distribución de BSD. Actualmente se encuentra disponible para distintas arquitecturas: amd64, i386, ia64, pc98, sparc64, ARM, MIPS y PowerPC. A modo de referencia, la versión 2.1 tiene un *footprint* de 5 MB.

C.6. Cuadro Comparativo

En el cuadro C.1 se muestra información sobre el *kernel,* el *footprint* que tiene cada EOS y la disponibilidad del código fuente. Esta información fue obtenida de las referencias de cada uno de los sistemas anteriores.

Apéndice D

Lenguajes

En este apéndice se describen los lenguajes de programación relevados en el estado del arte: C/C++, Python, Perl, PHP, Lua y Java los cuales fueron tenidos en cuenta para la decisión de qué lenguaje usar en el proyecto.

D.1. C/C++

Tal como lo conocemos hoy, ANSI/ISO C/C++, recién se estandariza en el año 1998, sin embargo C aparece en el año 1972 como sucesor del lenguaje B. Al ser un lenguajes que existe hace muchos años, tiene y tuvo una gran difusión, existen compiladores para la mayoría de las arquitecturas. Es un lenguaje compilado, por lo que los objetos binarios pueden correr directamente sobre el CPU sin necesidad de un intérprete o una VM. Permite programar de forma estructural y embeber código assembler. C es un subconjunto de C++ el cual le agrega el paradigma de orientación a objetos u OO (*Object Oriented*).

D.2. Python

Lenguaje que aparece en el año 1992, corre sobre una máquina virtual de Python o PVM (*Python Virtual Machine*). Los fuentes cuya extensión es `.py` se traducen a *bytecode* con extensión `.pyc` que es interpretado por la PVM. Para evitar el retrabajo de traducción al ejecutar un fuente Python, si existe su correspondiente .pyc y está actualizado, entonces se ejecuta directamente el *bytecode*.

Hay implementaciones de la PVM para Windows, Linux/Unix, Mac OS X, OS/2, Amiga, Palm Handhelds, y celulares Nokia. No sólo hay diversidad en cuanto a las arquitecturas soportadas sino que también existen muchas implementaciones con distintas características, como velocidad de procesamiento, uso del *stack*, y traductores que traducen Python al CLR (*Common Language Runtime*) de .net o a Java *bytecode*.

Es muy flexible ya que se puede programar de forma estructural, OO y de forma funcional [77].

Deeply Embedded Python[45] es una implementación de PVM cuya huella es menor a 200 kB.

D.3. Perl

Es un lenguaje de programación versátil[115], interpretado, que aparece en el año 1987. Soporta OO, programación estructurada, funcional y genérica. El intérprete tiene una huella de 1 MB por lo que no cualquier dispositivos de bajos recursos lo puede soportar.

D.4. PHP

Originalmente llamado *Personal Home Page*[116], aparece en el año 1994. Es un lenguaje interpretado que puede utilizarse para programar en forma estructurada, *scripting*, y OO. Comúnmente es usado para programar páginas web dinámicas y *side-server-scripting*.

Para reducir el *overhead* de la compilación en tiempo de ejecución (*runtime*) de los *scripts* por el PHP *engine*, PHP permite la compilación en tiempo de codificación (antes del *deploy*) así como lo hacen otros lenguajes (es el caso de C). En este sentido también existen optimizadores capaces de reducir el tiempo de ejecución de los códigos PHP compilados.

Otro enfoque es usar aceleradores de PHP que reducen el tiempo de compilación en *runtime* de los *scripts*.

D.5. Lua

Es un lenguaje interpretado muy poderoso creado en el año 1993. Permite programación estructurada, OO, programación funcional y *scripting*. Fue diseñado para embeberse en otros lenguajes, además corre en cualquier plataforma. Lua se compila y genera *bytecode* de Lua que corre sobre la maquina virtual LuaVM (*Lua Virtual Machine*), este proceso generalmente se realiza en *runtime* aunque se puede hacer antes del *deploy* para mejorar la eficiencia.

El interprete de eLua (*Embedded Lua*, ver [23]) tiene una huella (sin el *parser* ni el *lexer*) de 64 kB de acuerdo a la referencia "*Lua footprint*" [34].

Existen compiladores como LuaJIT[71] que traducen el Lua *bytecode* a lenguaje de máquina (x86). La gran ventaja es que no hay *overhead* de interpretación. Sin embargo, no es compatible con cualquier arquitectura.

D.6. Java

Fue creado por SUN Microsystems en el año 1995. Es un lenguaje orientado a objetos, e interpretado. No obstante hay implementaciones de JVM en *hardware* (Java *processor*); algunas de las más conocidas son: picoJava, aJ100, Cjip, Komodo, FemtoJava, ARM926EJ-S, Java Optimized Processor para FPGAs, SHARP, jHISC y ObjectCore.

También existen distintas ediciones y plataformas de Java con sus respectivas prestaciones de acuerdo al uso que se le quiera dar:

- J2EE (*Java 2 Platform, Enterprise Edition*).
- J2SE (*Java 2 Platform, Standard Edition*).

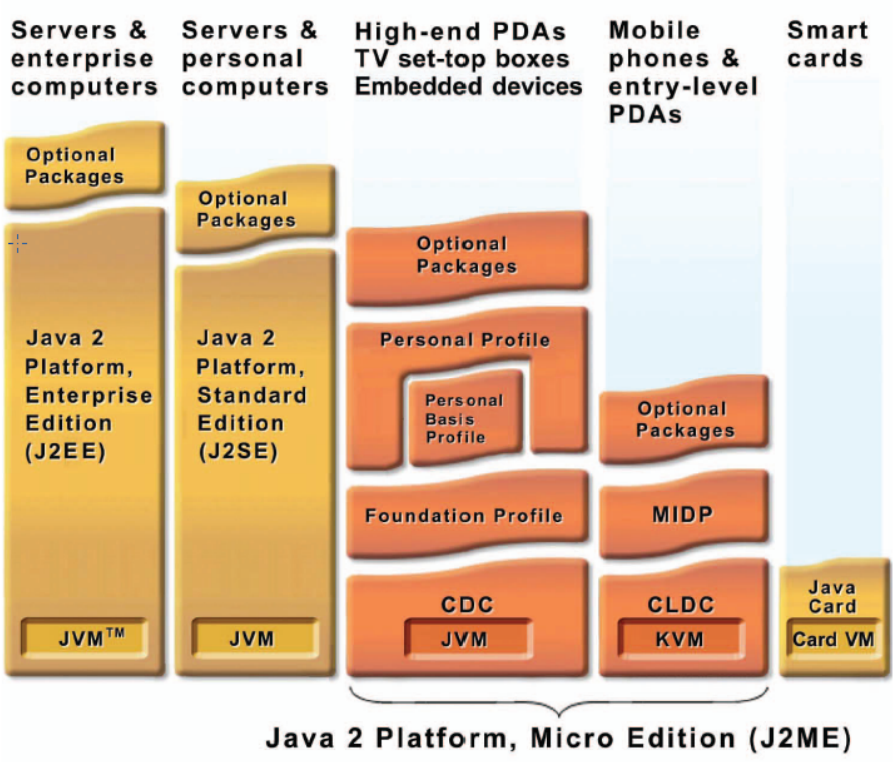


Figura D.1: Plataformas de Java y mercados.

- J2ME (*Java 2 Platform, Micro Edition*).
- Java Card.

J2ME es la edición orientada a dispositivos embebidos como se puede ver en la figura D.1 de la referencia "*Data Sheet*"[86].

J2ME apunta a 2 objetivos:

- *High-end*: etiquetados como CDC (*Connected Device Configuration*) cuyo costo de memoria es de entre 2 MB y 4 MB.
- *Low-end*: etiquetados como CLDC (*Connected, Limited Device Configuration*) cuyo costo de memoria es de 128 kB a 256 kB.

Según la referencia "*Overview of Java 2 Platform, Micro Edition (J2ME)*"[88], la linea entre ambos es difusa y con el objetivo de reducir costos la mayor parte de la industria *Wireless* elije usar CLDC y MIDP (*Mobile Information Device Profile*).

Para conocer el consumo exacto de recursos es necesario saber exactamente los requerimientos de bibliotecas y los detalles del *hardware* ya que todas las ediciones son personalizables.

Apéndice E

Sistemas Existentes

En este apéndice se describen los distintos sistemas existentes en el mercado para domótica. Los mismos se clasificaron en libres y comerciales. Se describen los siguientes: MisterHouse, Heyu, DomoticaCasera, LinKNX , Taiyaki y Sistema Casa.

E.1. Libres

Los sistemas libres son los que tienen una implementación abierta y disponible. Esto permite que se pueda desarrollar en base a los mismos.

E.1.1. MisterHouse

Es un programa de automatización de hogar creado en el año 2005, *open-source,* programado en Perl. Puede correr tanto en Windows 95/98/NT/2k/XP como en la mayoría de las distribuciones basadas en Unix incluyendo Mac OSX.

Alguna de las prestaciones que brinda son[60]:

1. Control por voz e interfaz web [62].
2. Trabaja únicamente con dispositivos X10.
3. Permite definir grupos para los dispositivos y asociar varios dispositivos a una misma dirección.
4. Permite programar eventos.
5. Trabajo con actuadores y sensores binarios (*on/off*) y *dimmers* con algunas limitaciones [53].

Testeo:

1. instalado en una VM con Windows.
2. Interfaz web testeada.

E.1.2. Heyu

Es un programa para controlar en forma remota distintos dispositivos a través de la red eléctrica (*power line*), su versión 1.0 vio la luz en el año 2005 [90].

Fue desarrollado para Linux aunque actualmente también funciona sobre Mac OS X (Darwin), FreeBSD, NetBSD, OpenBSD, SunOS/Solaris, SCO Unix, AIX, NextStep, y OSF.

Prestaciones:

1. Trabaja únicamente con dispositivos X10.

2. Los dispositivos puede ser *on/off y dimmer*.

E.1.3. DomoticaCasera

Control remoto vía *WAP* y web, hecha en PHP. Utiliza apache como servidor web y MySql como base de datos.

Contempla algunos aspectos de seguridad como encriptación mediante SHA-1 y técnicas para prevenir SQL *injection*. Utiliza *hardware* propio cuya especificación (y manual de construcción) están en disponibles [18].

Descripción:

1. Solo funciona en Windows con altos requerimientos en *hardware*.

2. Trabaja únicamente sobre protocolo propio usando el puerto paralelo.

3. Sólo permite *on/off*.

E.1.4. LinKNX

El objetivo de este proyecto, que comenzó en el año 2007, es proveer un sistema domótico basado en el estándar KNX utilizando la menor cantidad de recursos posibles. Para esto utiliza un *router* WiFi WRT54GS con OpenWrt [46].

Posee 2 módulos, el de control y el de diseño. Ambos permiten el acceso web. Mientras que el primero permite el monitoreo y control de dispositivos, el segundo permite armar el mapa del hogar/oficina/etc, especificando la cantidad de pisos, ubicación y tipos de sensores/actuadores.

En la página referenciada en "linkKNX Demo"[68] corre un simulador donde es posible probar ambos módulos.

Descripción:

1. Usa el protocolo KNX, puede trabajar tanto sobre IP, USB, o RS232 [94].

2. Permite *on/off y dimmer,* control de temperatura, captura de fotos (imagines jpeg).

3. Interfaz web liviana que está hecha en PHP y AJAX(*Asynchronous JavaScript And XML*) [95].

E.1.5. Taiyaki

Es un proyecto creado en el año 2008 que permite el control de dispositivos X10 mediante una interfaz web que es accesible desde un *browser* común y desde IPhone.

Utiliza *Java X10 library (J.Peterson's X10 API 1.0.1)* para resolver el acceso al medio [91].

E.2. Comerciales

Existen cuantiosos sistemas comerciales, como ser: MyHome BTicino, Crestron, Home PAC XP8741 y ByMe Vimar que a continuación se describen resaltando características de interés.

Únicamente a modo de mostrar la gran cantidad de sistemas, se menciona otros sistemas comerciales existentes: HomeSeer, XTension, Indigo, DomusTechABB, Hoasis e Nehos BPT, Vantage, Chorus Gewiss (adopta la tecnología bus EIB, conforme al estándar europeo EN50090), Atmosphera BM, Domino e Contatto Duemmegi, Semplice Urmet y PICnet Sinthesi entre otros.

E.2.1. MyHome BTicino:

Nacido de la evolución del *bus* SCS , fue introducido en el mercado en el año 1998. Es un sistema modular orientado a la domótica con un atención al aspecto estético y abierto a una constate innovación.

Se trata de un *bus* propietario basado en UTP, que no solamente se utiliza para alimentar los dispositivos sino también para enviar los comandos a los mismos.

Si bien el sistema es propietario, cuenta con una serie de interfaces que permiten la interconexión con otros *middlewares*. Además recientemente se puso a la venta un dispositivo que contiene un lenguaje *open-source* que permite realizar programas que interactúen con MyHome [29].

Por más información sobre la evolución del protocolo ver OpenWebNet B.10.

E.2.2. Creston

Se basa en la red "cresnet", donde se conectan todo tipo de dispositivos: *touchpanels*, *PDA´*s, telecomandos, etc. Es una arquitectura centralizada [29].

E.2.3. Home PAC XP8741

Sistema desarrollado para arquitectura x86, tienen integrado un Windows Embedded que permite controlar diferentes tipos de protocolos por medio de *slots* [6].

E.2.4. ByMe Vimar

Sistema basado en Konnex, brinda: automatización, anti-intrusión, control de temperatura, videocitofonía (sistema de audio y video cerrados) y telegestión [29].

E.2.5. Sistema Casa

Es un sistema italiano que opera sobre Windows XP basado en dispositivos X10. Brinda una interfaz *touch-screen* para el control amigable del hogar y una interfaz consola, para la línea económica [10].

Apéndice F

Trabajos Relacionados

En este apéndice se describen los trabajos académicos relacionados a la domótica, los cuales fueron analizados durante el relevamiento del estado del arte:

- *The Role of Web Services at Home* [3].
- *INTERACT-DDM* [49].
- *DOG* [8].
- *PUCC Architecture* [85].
- *A Framework for Connecting Home Computing Middleware* [40].
- *Domotic House Gateway* [73].
- *Una implementación de un sistema de control domótico basada en servicios web* [72].
- *DomoNet* [52].
- *Dynamic discovery and semantic reasoning for next generation intelligent environments* [48].
- *A Generic Gateway for Testing Heterogeneous Components in Acceptance Testing Tools* [15].
- *A Framework for Connecting Home Computing Middleware* [93].
- *UHMN*[55].

Al final del apéndice, se presenta un cuadro que muestra la disponibilidad del código fuente de estos trabajos.

F.1. The Role of the Web Services at Home

El caso de estudio de este trabajo fue denominado ITEA(debido a las iniciales del *Istituto Trentino Edilizia Abitativa*, Instituto Trentino de vivienda pública)

[3], que clasifica las distintas soluciones para redes domóticas según cuatro atributos de calidad: apertura, escalabilidad, heterogeneidad y topología.

Presenta una clasificación interesante de los distintos escenarios domóticos denominados S1 a S4:

S1: *Bus* simple.
Existe un *bus* al cual se conectan todos los dispositivos. Tiene baja escalabilidad y baja heterogeneidad, por ejemplo sistema que interactúa únicamente con dispositivos X10.

S2: Centralizado, cerrado.
Se introduce un *gateway* en la arquitectura de forma de permitir control remoto o un proveedor de servicios remoto.
El protocolo es cerrado, típicamente usado por empresas para mantener el monopolio: LonWorks (de Echelon B.9), MyHome (de BTicinoE.2.1), Sistema Casa (E.2.5), etc.
Las soluciones de este tipo son cerradas, relativamente escalables, y con baja heterogeneidad.

S3: Jerarquía basada en servidor, abierto.
La tendencia actual es tener un sistema abierto que respete algún estándar. Algunos sensores y actuadores se comunican con un controlador quien a su vez se comunica con el *gateway*, otros dispositivos más complejos que se comunican directamente con el *gateway*.
El estándar para la comunicación entre el controlador central y sus clientes, según este *paper,* es usar WS
Está arquitectura es abierta, permite alta heterogeneidad y son escalables (considerando los dispositivos). Sin embargo el servidor central es un cuello de botella que puede afectar la confiabilidad siendo también el único punto de falla del sistema.

S4: Arquitectura *web-service*, P2P.
Para atacar las falencias del escenario anterior se construye una arquitectura descentralizada P2P, en este caso el servidor no es el único punto de acceso al mundo exterior ya que los dispositivos se pueden comunicar por intermedio de otras redes. Existen soluciones de este tipo (P2P) sin WS como Jini B.7 y HAViB.4.

En la figura G.6 se muestra la clasificación de los escenarios según la apertura y escalabilidad/heterogeneidad, hecha en el *paper* "*The Role of Web Services at Home*"[3].

Argumenta que WS es la mejor opción para obtener dichas propiedades en instalaciones con suficiente capacidad computacional. En la figura F.2 se detalla el *stack* para WS propuesto para domótica en el *paper* "*A Framework for Connecting Home Computing Middleware, Distributed Computing Systems Workshops*"[3]:

Además implementa un caso de estudio de un escenario S3.

F.2. Proyecto de Sistema de control domótico

Es un un proyecto de fin de carrera de la universidad La Salle Bonanova (Universidad Ramón Llull) de Barcelona. El sistema de control domótico,

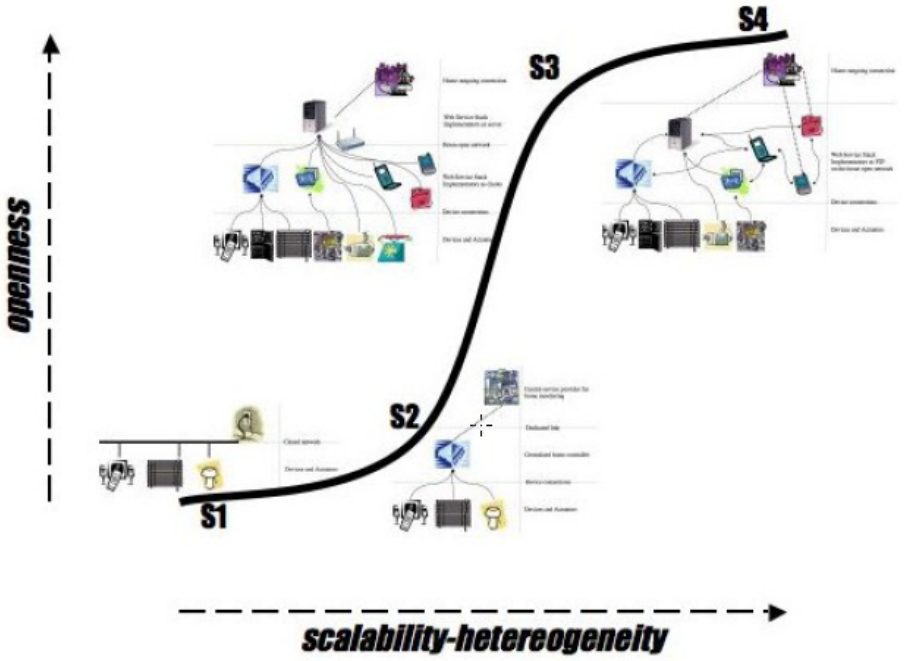


Figura F.1: Evolución de la tecnología domótica.

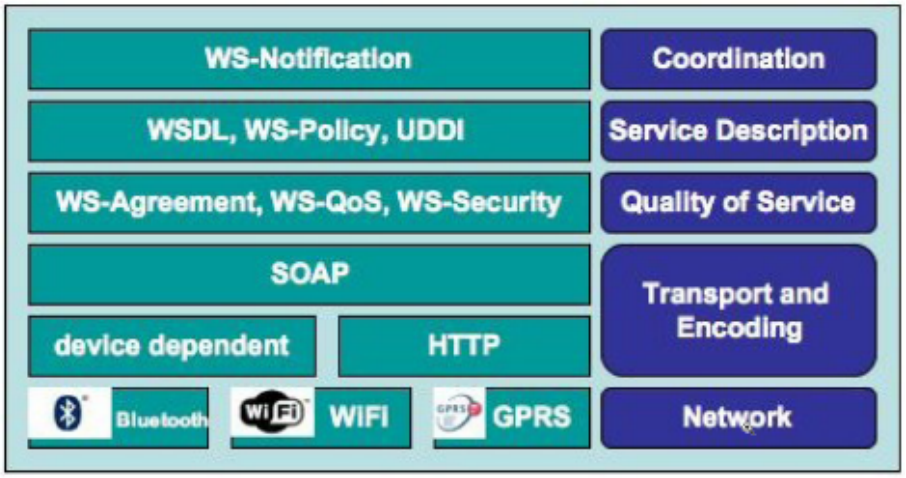


Figura F.2: Implementación del *Stack* de *Web Service* para domótica.

como interfaz móvil que se presenta en este artículo, fue valorada por el tribunal calificador con Matrícula de Honor en el año 2002 por la innovación y adaptabilidad del sistema, el cual ha tenido mucha aceptación en las publicaciones y empresas del sector domótico y electrónico [4].

El proyecto consiste en control con placas que se comunican a la PC a través de RS232; la implementación del prototipo fue hecho en Windows, con IIS(Internet Information Services), y la interfaz de acceso es WAP(Wireless Application Protocol, comúnmente usado para celulares), web y control vía SMS(Short Message Service).

El sistema consta de 2 bloques [38]:

- PC-Servidor.
- Bloque de Domótica.

Este sistema dio lugar a otros sistemas como el "Sistema de riego inteligente" [5]. Básicamente es el sistema "Proyecto de Sistema de control domótico" aplicado en el contexto de riego. Por lo tanto tiene las mismas características que dicho sistema, salvo por el Bloque de Domótica, el cual se remplaza por el Bloque de control Agrario.

F.3. InterAct

El *paper* "*An intelligent environment*"[96], presenta una solución para integrar dispositivos domésticos. La arquitectura propuesta se basa en el estándar de gestión de redes TCP/IP: SNMP(*Simple Network Management Protocol*) y MIB(*Management Information Bases*) de esta forma se logra una definición flexible de los dispositivos y configuración dinámica.

Se basa en 3 principios:

1. Los MIBs controlan las meta-definiciones de los dispositivos.
2. Existe un único *gateway* el cual presenta la interfaz para las aplicaciones.
3. Los dispositivos tienen la autonomía suficiente para realizar tareas simples sin la gestión de la aplicación centralizada.

SNMP presenta varias ventajas frente a otros protocolos:

- Es un protocolo simple y es posible implementarlo en dispositivos con bajo consume de procesador (a diferencia de los WS).
- Los WS solo funcionan a demanda (*request-response*), y no son capaces de enviar notificaciones a los clientes, a diferencia de los agentes SNMP.
- SNMP usa datagramas UDP como mecanismo de transporte, que es adecuado para una red cerrada con alta confiabilidad.
- Las interacciones con el sistema son principalmente por intervención humana y van a suceder con baja frecuencia (asume el *paper*), por lo que el mecanismo de *polling* utilizado por SNMP no es un cuello de botella en cuanto al ancho de banda utilizado.

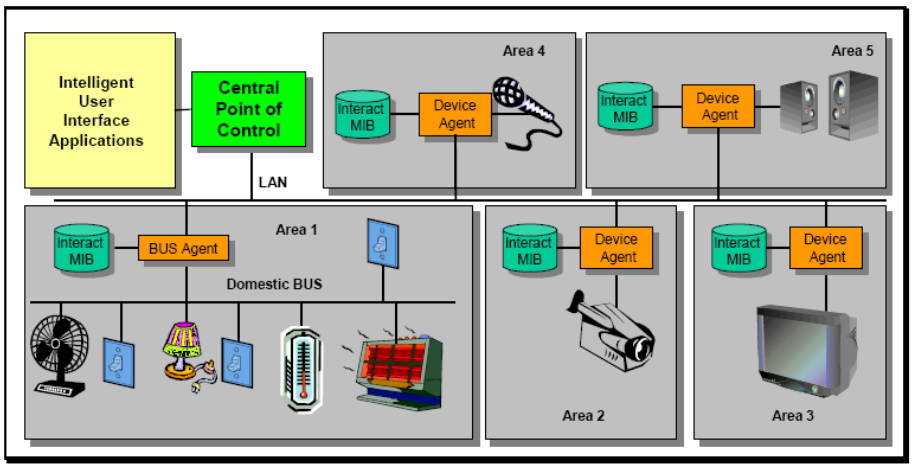


Figura F.3: Arquitectura de InterAct.

- SNMP permite la incorporación de dispositivos más complejos de gestión de red.

Existen 3 componentes: los dispositivos, los agentes y el punto central de control.

- Los dispositivos y agentes se conectan a la LAN (*backbone* de comunicación de la red domótica).
- Cada agente puede manejar uno o varios dispositivos (grupo). Cada agente representa un área a la cual se le asigna una IP.
- El punto central de control interactúa con los agentes para consultar o modificar los dispositivos.

Esta arquitectura se puede ver sintetizada en la figura F.3 obtenida del *paper* "*A Solution for the Integration of Domestic Devices on Network Management Platforms*"[49].

F.4. Domotic OSGi Gateway

DOG(Domotic OSGi Gateway) [74] es un *gateway* basado en OSGi para poder mostrar diferentes redes domóticas como si fuera una sola, es independiente de las tecnologías de los sistemas de automatización del hogar.

La adopción de un marco normativo como OSGi, y de sofisticadas técnicas de modelado derivadas de la comunidad de investigación de web semántica, DOG permite ir más allá de la simple automatización y apoyar el razonamiento basado en la inteligencia dentro de los ambientes domésticos, mediante la utilización de dispositivos automáticos que permiten comandos de validación entre definiciones de escenarios entre distintas redes [8].

DOG proporciona los elementos básicos para apoyar la evolución de los sistemas actuales, para crear sistemas denominadas "Entornos Domóticos Inteligentes", donde dispositivos heterogéneos y sistemas domóticos se coordinan para comportarse como un sistema único, inteligente y activo [32].

Se caracteriza principalmente por:

1. Utilizar ontologías para hacer chequeos semánticos de los comandos ejecutados.

2. Usar el framework OSGi y por lo tanto Java.

Si bien parece requerir de importantes prestaciones de *hardware*, debido a las ontologías, esto le presenta un valor agregado semántico interesante.

F.5. P2P Universal Computing Consortium

Hay varios trabajos basados en esta tecnología. Ver entre ellos los *papers*: "*PUCC Architecture, Protocols and Applications*"[85], y "*Home Appliance Control Using Heterogeneous Sensor Networks, Consumer Communications and Networking* "[40].

Algunas de las grandes ventajas de usar P2P son:

1. Descubrimiento de recursos.

2. Compartir recursos.

3. Balance de carga.

El *paper* "*PUCC Architecture, Protocols and Applications*"[85] argumenta que P2P es la mejor tecnología para las redes omnipresentes (*ubiquitous networking*) ya que además de las características propias de P2P, permite comunicación punto a punto y soporte para numerosos dispositivos.

Asemeja la aplicación de PUCC a una solución domótica e implementa un prototipo.

En la figura F.4 se muestra el carácter distribuido de la arquitectura de PUCC, la imagen fue obtenida del *paper* "*PUCC Architecture, Protocols and Applications*"[85].

El *paper* "*A Framework for Connecting Home Computing Middleware, Distributed Computing Systems Workshops*"[40] se basa en PUCC para desarrollar una solución domótica enfocada en la detección de eventos por medio de un *gateway* de sensores. También utiliza la detección de eventos compuestos para construir dicho sistema. La detección de eventos compuestos es un trabajo colaborativo de varios dispositivos.

En la figura F.5 obtenida del *paper* "*Home Appliance Control Using Heterogeneous Sensor Networks, Consumer Communications and Networking* "[40], se muestran las distintas capas de PUCC y su relación con el modelo OSI.

F.6. Domotic House Gateway

Analiza las distintas alternativas para instalaciones domóticas y presenta una clara heterogeneidad de tecnologías donde ninguna predomina sobre las otras. Luego presenta las distintas soluciones para la interoperabilidad entre diversos protocolos que necesitan elementos externos para interactuar [73].

Define una arquitectura para resolver la heterogeneidad, donde las aplicaciones de control de *hardware* se comunican con las aplicaciones de gestión

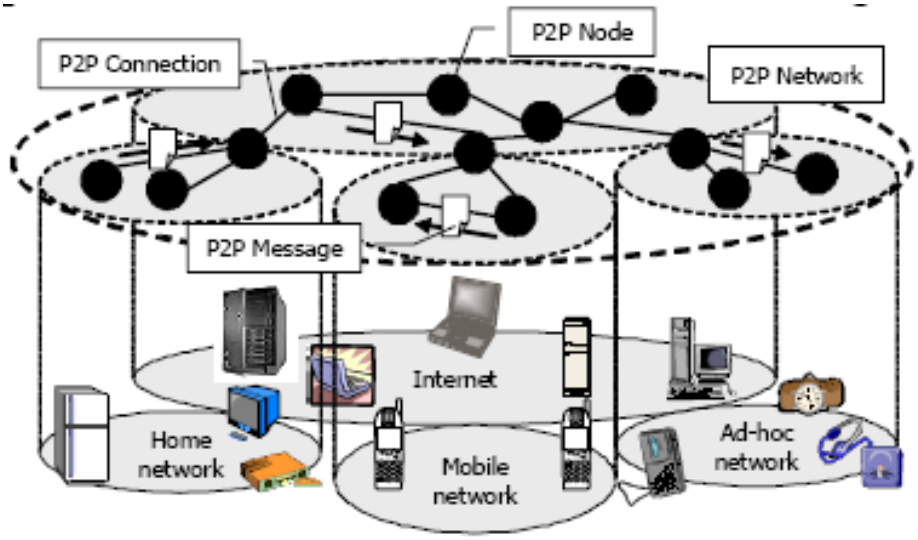


Figura F.4: Arquitectura de PUCC.

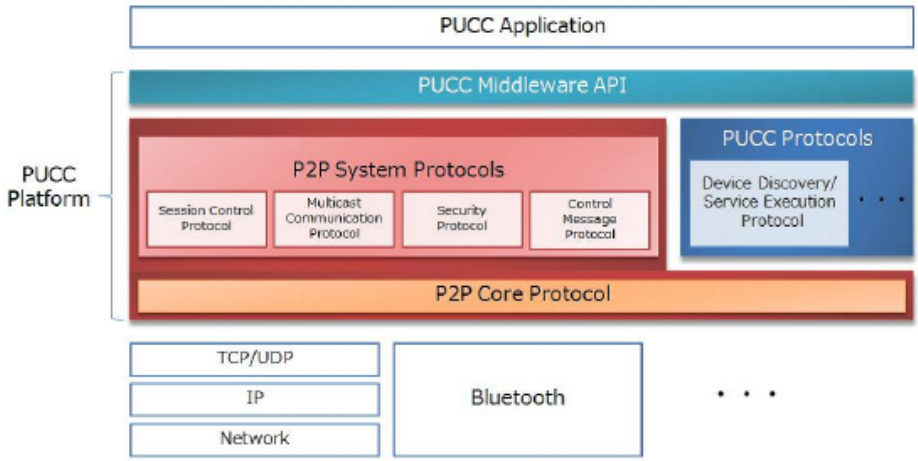


Figura F.5: Capas de PUCC.

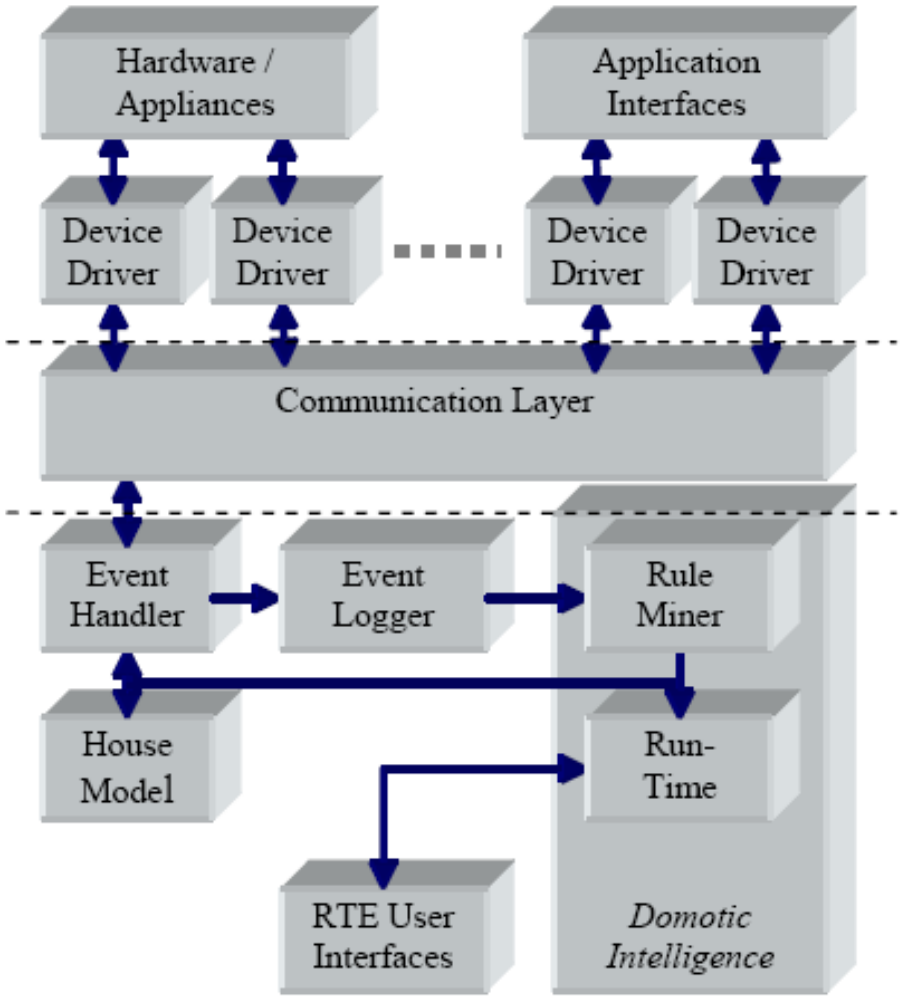


Figura F.6: Arquitectura de DHG.

a través de la capa de comunicación, donde al haber una llamada se dispara una interacción de eventos. Se entiende por evento, un suceso lógico o físico.

En la figura F.6, se muestra el modelo de eventos de DHG(Domotic House Gateway) y la arquitectura que resuelve la comunicación entre los dispositivos de *hardware* y la aplicación de control.

F.7. Una implementación de un sistema de control domótico basada en servicios web

El foco de este proyecto es la heterogeneidad de tecnologías de base y la interacción persona-computadora, optan por usar WS para la interoperabilidad remota: "Los servicios web o *web services* son una metodología que permiten intercomunicar dos sistemas remotos a través de la web, manteniendo el estado

de los objetos en la transferencia." Está implementado en C# [72].

Se clasifica como categoría S3 de "The role of Web Services at home"[3], si bien no se asegura la interoperabilidad.

F.8. DomoNet

En la página de "*Home Automation Technologies and Standards*"[92] se presenta el proyecto HATS(*Home Area Network*), apunta a la computación penetrante (*pervasive computing*) implementando:

1. Definición del lenguaje de marcas domoML(*Domotics Markup Language*) que permite el intercambio de información entre distintos electrodomésticos controlados por distintos sistemas y tecnologías de base.

2. Este lenguaje permite definir el modelo lógico de cada dispositivo, el cual tiene que ser compartido y accesible por todos los participantes del sistema.

3. Tecnología semántica web, la cual es necesaria para que todos los dispositivos puedan manejar la información del modelo de terceros.

La arquitectura de HATS consiste en un *middleware* llamado domoNet es una arquitectura SOA necesaria para la computación penetrante e implementa integración dinámica del modelo que permite una abstracción de los servicios y dispositivos.

Los pilares de esta infraestructura son WS, *Semantic web* y *Ubiquitous Computing technologies*.

En la figura F.7 extraída del *paper* "*DomoNet: a framework and a prototype for interoperability of domotic middlewares based on XML and Web Services*"[52] se ilustra la arquitectura de DomoNet.

Hay un DeviceWS por cada tipo de dispositivo: Lights WS, ClockWS...

Y un TM(*TechManager*) por cada tecnología de base. Los TM se registran en el WS que le compete.

Cada TM se comunica con otro *middleware* a través de los *DeviceWS* por medio de DomoML

A su vez, cada TM consiste en tres elementos:

1. *TechServer*, TCP server que siguiendo las políticas de registro de los TM, es visible a través de su *DeviceWS* y puede conectarse a otros *middlewares* que estén en DomoNet.

2. *DeviceController* se encarga del control específico de cada un *middleware*.

3. *Translator* su funcionalidad es traducir el protocolo específico a DomoNet.

Los *RealDevices* representan los dispositivos conectados al *middleware* manejado por cada TM y los *VirtualDevices* son la representación de los dispositivos conectados a otros TM. Los clientes conectados al *middleware* pueden comunicarse con los *Virtual Devices* de forma transparente, estos mensajes son interceptados por el *Device Controller* quien traduce las señales a mensajes de DomoNet (DomoML) y los pasa al *DeviceWS* correspondiente. En la figura F.8,

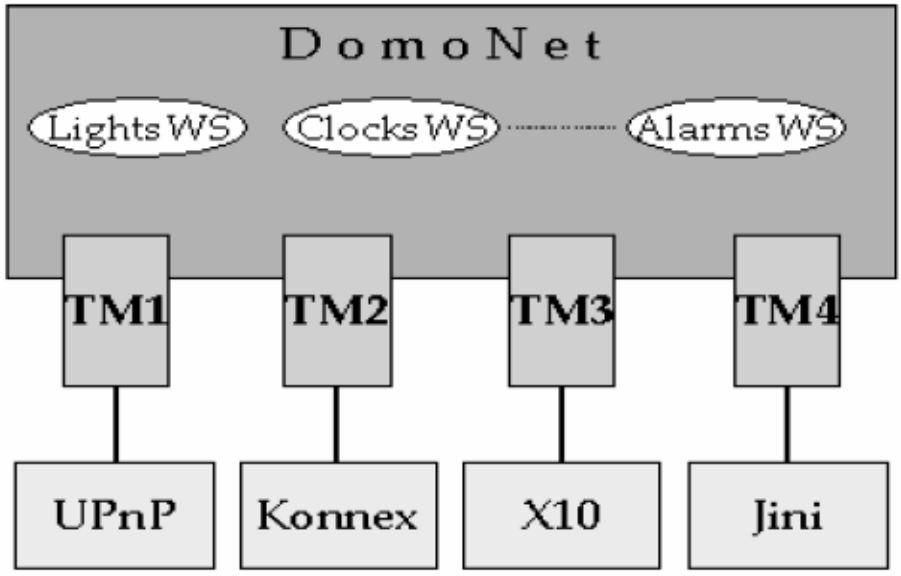


Figura F.7: Arquitectura de DomoNet.

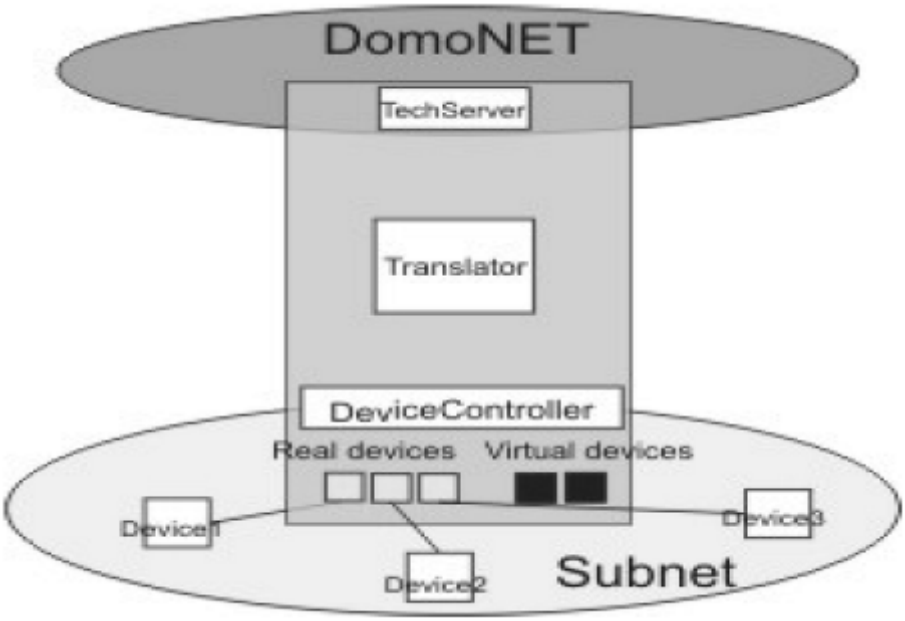


Figura F.8: Estructura lógica del *TechManager*.

se puede ver los elementos de los TM, y la pertenencia de dichos elementos a las redes DomoNET y la Subnet específica del *middleware* que se está controlando.

La función principal de los *DeviceWS* es enrutar mensajes. Dentro de DomoNet, cada dispositivo tiene una dirección única formada por IP y puerto TCP del TM correspondiente a la tecnología del dispositivo y una dirección específica que lo identifica dentro del *middleware*.

La interfaz de los *DeviceWS* permite:

1. Registro de los TM.
2. Notificar cambios topológicos.
3. Ejecutar comandos.
4. Descubrimiento de TM y servicios.

De este artículo se resalta la abstracción del manejo de cada *middleware* específico.

F.9. Dynamic discovery and semantic reasoning for next generation intelligent environments

En el *paper* "*Dynamic discovery and semantic reasoning for next generation intelligent environments*"[48] se desarrolla una solución denominada SmartLab que apunta a mejorar las prestaciones de OSGi con:

1. Descubrimiento dinámico de dispositivos.
2. Monitoreo de los servicios semánticos distribuidos.
3. Modelado del contexto semántico.
4. Apunta a la provisión de servicios inteligentes.

Explica que el auge de OSGi como base del desarrollo domótico (tanto de investigación como en la industria) para redes heterogéneas es debido a que:

1. Provee un ambiente de ejecución propio.
2. Tiene una interfaz capaz de :
   a) Descubrimiento.
   b) Cooperación dinámica de dispositivos heterogéneas.
   c) Conexión externa (control remoto del sistema).
   d) Herramientas de diagnóstico.
   e) Herramientas de gestión.

Por otra parte, también resalta sus limitaciones:

1. Arquitectura centralizada.
2. Requiere cooperación con el protocolo de descubrimiento de redes para presentar un nuevo servicio.

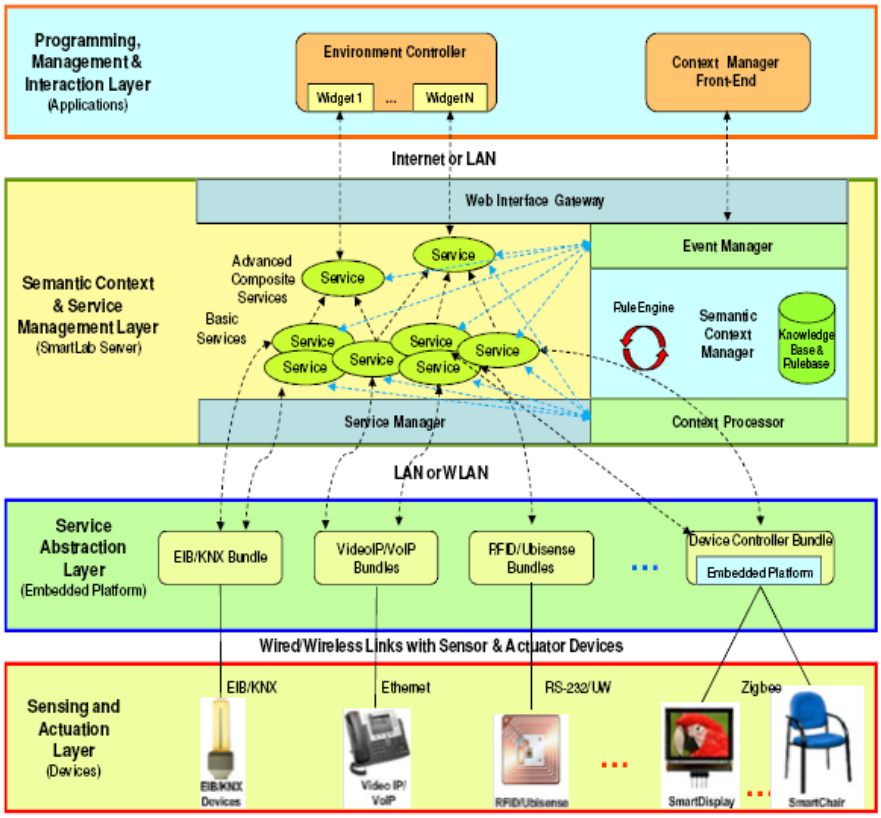


Figura F.9: Arquitectura en capas de SmartLab.

Para paliar esta última limitación con características semánticas, agregan la capacidad de gestión por medio de ontologías semánticas basadas en OWL(*Ontology Web Language*) creando bases de conocimiento. Gracias a la información semántica de los servicios, permite la interacción de servicios sin el previo conocimiento de interfaces.

SmartLab *Semantic Middleware* está dividido en cuatro capas:

1. *Layer 1:* Actuadores y sensores.

2. *Layer 2*: Abstracción del servicio.

3. *Layer 3*: Modelado del contexto semántico y Gestión de servicio.

4. *Layer 4*: Programación de servicio, Gestión de la interacción.

En la figura F.9, extraída del *paper* "*Dynamic discovery and semantic reasoning for next generation intelligent environments*"[48] se presentan estas cuatro capas junto con sus componentes.

Cada servicio de la tercer capa de SmartLab (Modelado del contexto semántico y Gestión de servicio) presenta la interfaz definida a continuación:

```
public interface ISmartlabService
{

    public String getOntology();
    public String getIndividual();
    public String getRules();
    public String[] getEventsToRegister();
    public void startUpdatingContext();
    public void stopUpdatingContext();
    public String getInterface();

}
```

En el *paper* "*A Framework for Connecting Home Computing Middleware, Distributed Computing Systems Workshops*"[48] se explica cada función. Se puede ver que siempre devuelve *String* (en caso de tener retorno), como forma de no perder genericidad en cuanto a la interfaz. Los tres primeros métodos están relacionadas con la funcionalidad semántica, los segundos tres directamente con los servicios de OSGi y los *bundles* y el último método es una forma de acceder a una interfaz genérica.

F.10. A Generic Gateway for Testing Heterogeneous Components in Acceptance Testing Tools

En el paper "*A Generic Gateway for Testing Heterogeneous Components in Acceptance Testing Tools.*" [15] se presenta un *gateway* para testeo de redes y plataformas heterogéneas. Para resolver el problema, descrito en el *paper*, se utiliza ingeniería orientada a servicios, para la cual se mencionan distintas tecnologías: WS y OSGi. En este trabajo se elige OSGi para abordar la interoperabilidad entre el *gateway* y cada red específica.

"Interoperability and middleware technologies: an overview"

En este *paper*, se entiende como *middleware* un *software* que es capaz de conectar otros componentes de *software* con distintas tecnologías de desarrollo, interfaces de red y plataformas. Luego compara distintos *middleware* que presentamos en forma tabular para una mejor comprensión.

Primero muestra algunos ejemplos de *middleware* heterogéneos, donde ninguno prevalece sobre el resto. Cada unos tiene sus propias características (HAVi, desarrollado para *multimedia*, restringido para redes FireWire IEEE 1394; LonWorks, Konnex, X10, específicos para seguridad y automatización de hogar, HomePNA, etc).

El cuadro F.1 es una elaboración propia en base a la información del paper "*A Generic Gateway for Testing Heterogeneous Components in Acceptance Testing Tools.*"[15] donde se comparan distintas tecnologías de *middleware*.

Middleware	**Heterogeneidad**
CORBA	Si bien logra lo deseado, es un estándar muy complejo y dificulta su implementación en dispositivos de bajos recursos. No es muy usado en la industria. En este se ha trabajado en *minimum* CORBA.
Jini	Todos los dispositivos de base deben soportar correr JVM y deben respetar las interfaces de Java.
OSGi	Herramienta para SOA. Iniciativa basada en Java para la provisión de servicios, se usa tanto en dispositivos con bajos recursos así como también de base para grandes aplicaciones (Eclipse 3.0 *runtime*). OSGi funciona con varias tecnologías de base : xDSL, Cable *módems*, satélite y LAN como HAVi, HomePNA, HomeRF, USB, IEEE 1394 Firewire. Además es compatible con servicios de descubrimiento como Jini y UPnP. El trabajo explica claramente las diferentes tecnologías utilizadas para la interoperabilidad de *middlewares*, y presenta en detalle su implementación usando OSGi.
UPnP	Necesita Microsoft OS (estos es lo que dice el paper, es un estándar abierto).
Web services	Herramienta para SOA, los paquetes SOAP están basados en mensajes muy pesados con un importante *overhead* y la necesidad de un pre-procesamiento lo que no se corresponde con la capacidad de sistemas embebidos. Para paliar el problema del consumo de recursos, se ha desarrollado mensajes SOAP en formato binario (como *binary web services*). Otro enfoque es el de DomoNet donde se usan WS en la capa superior y en las capaz inferiores se usan tecnologías específicas para cada *middleware*. Distribuyendo el procesamiento y centralizando los WS se hace posible el uso de WS para la interoperabilidad de varios dispositivos con bajos recursos. De todos modos cada vez que se agrega un nuevo *middleware* es necesario agregar la implementación del mismo.

Cuadro F.1: Comparación de *middlewares*, derivada de la información.

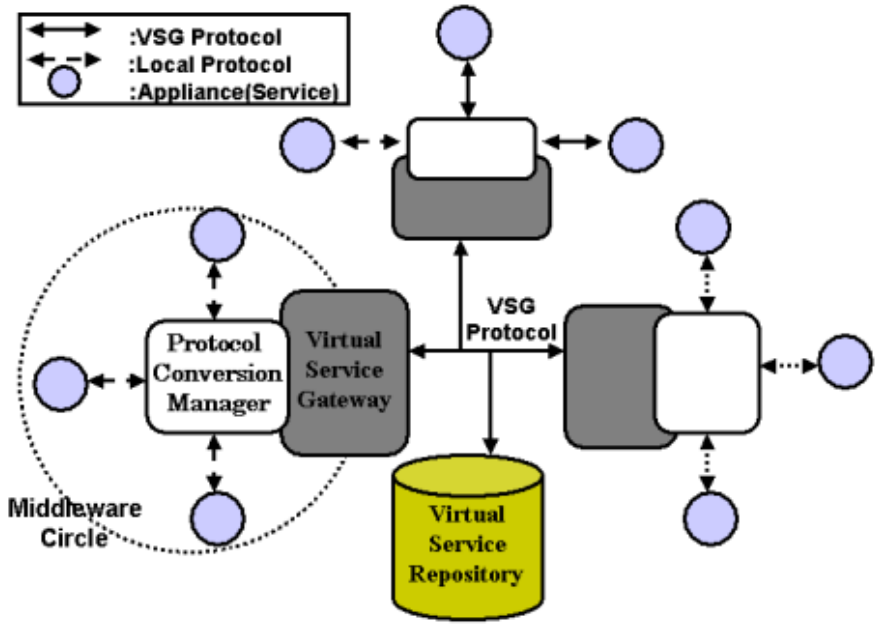


Figura F.10: Arquitectura del *framework*.

F.11. A Framework for Connecting Home Computing Middleware

Presenta un *framework* para interconectar *middleware* heterogéneos y comenta alguno de los distintos *middlewares*. Como primer paso de la interconexión, muestra los *bridges* como el de Philips, Sony y Sun entre HAVi y Jini. Esta solución es muy poco escalable ya que es necesario un *bridge* por cada par de tecnologías que se quieren interconectar [93].

Presenta una arquitectura bien dividida. (Ver figura F.10.)

Esta arquitectura tiene 3 componentes:

1. *Virtual Service Gateway* (VSG): conecta 2 *middlewares* distintos.
2. *Protocol Conversion Manager*: convierte el protocolo especifico en el protocolo del VSG.
3. *Virtual Service Repository*: guarda el lugar y funciones del servicio.

Si bien no especifica una restricción sobre el protocolo de comunicación del VSG, en el prototipo usan mensajes SOAP. Dicho prototipo está hecho sobre Linux usando Java.

F.12. Universal Home Network Middleware

Este proyecto presenta la arquitectura Universal Home Network Middleware o UHNM que permite la interoperabilidad entre *middlewares* heterogéneos, con

una abstracción de alto nivel y sin configuración [55]. La arquitectura UHNM esta compuesta por [44]:

1. Un adaptador: convierte el protocolo del *sub-middleware* en el protocolo de comunicación de UHNM y los componentes UHNM.
2. *Event Manager* (EM): entrega los eventos a otros componentes, si otro componente quiere ser notificado de un evento debe suscribirse al EM.
3. *Configuration Manager* (CM): *directory service* sin configuración previa requerida.
4. *Resource Manager* (RM): Reserva y libera recursos de la red, mantiene el estado de los recursos.
5. *Virtual Proxy* (VP) representa los servicios de un dispositivo dentro de un *sub-middleware*.
6. *Device Manager* (DM): responsable por instalar y desinstalar VP.
7. *Service Manager* (SM): invoca los servicios y brinda la infraestructura para dinámicamente crear eventos compuestos en forma dinámica.

El *paper* explica la interrelación de los componentes pero no detalla el formato de los mensajes UHNM y desarrollan un prototipo en Java sobre Linux.

F.13. Disponibilidad del Código

En el cuadro F.2 se presenta para cada trabajo, si su código está disponible o no.

Paper	**Implementación disponible**
The Role of Web Services at Home [3]	no
INTERACT-DDM [49]	no
DOG [8]	si
PUCC Architecture [85]	Especificación de PUCC para socios
Home Appliance Control Using Heterogeneous Sensor Networks [40]	Especificación de PUCC para socios
Domotic House Gateway [73]	no
Una implementación de un sistema de control domótico basada en servicios web [72]	no
DomoNet [52]	si
Dynamic discovery and semantic reasoning for next generation intelligent environments [48]	no
A Generic Gateway for Testing Heterogeneous Components in Acceptance Testing Tools [15]	no
A Framework for Connecting Home Computing Middleware [93]	no
UHMN [55]	no

Cuadro F.2: Disponibilidad del código de los distintos trabajos.

Apéndice G

Framework OSGi

En este apéndice se presentan las principales características de OSGi y las posibilidades que brinda el uso de este *framework.*

El *framework* OSGi es el centro de la especificación de la plataforma de servicios. Provee un entorno de trabajo ("*framework*") de desarrollo y ejecución de forma "segura" basada en Java.

Se entiende por "segura" la resolución de dependencias de servicios, habilitación o inhabilitación de ejecución de un servicio sobre el *framework.* Dentro de este se encuentra definida una API donde , cada *bundle* brinda servicios, que independientemente de su estado, permite que otros *bundles* o servicios consuman. Por todas estas características el desarrollo es ágil y el *deploy* de servicios es transparente ya que el *framework* se encarga de gestionar el ciclo de vida del mismo.

Cabe notar, que las dependencias son débiles ya que un *bundles* depende de las interfaces que publican otros *bundles* y no dependen de una implementación concreta.

OSGi permite bajar desde repositorios, instalar y actualizar los *bundles* utilizados en la aplicación, como también sacar *bundles* (que durante un tiempo no fueron utilizados) de forma dinámica y escalable dentro del entorno de la aplicación sin necesidad de volver a compilar el sistema. Para lograr esto utiliza las ventajas de Java: independencia de plataforma y carga de código de forma dinámica, que permite que el *framework* maneje las dependencias entre los servicios y los *bundles* utilizados pudiendo realizar todas las opciones descritas.

Existen repositorios de *bundles* llamados OBR donde se pueden buscar implementaciones de servicios requeridos. Los más conocidos son los de la página oficial de OSGi [66] y los de las diversas implementaciones específicas del OSGi.

Las funcionalidades del *framework* son divididas en capas, como se muestra en la figura G.1, las cuales se detallan a continuación:

- Security Layer
- Module Layer
- Life Cycle Layer
- Service Layer

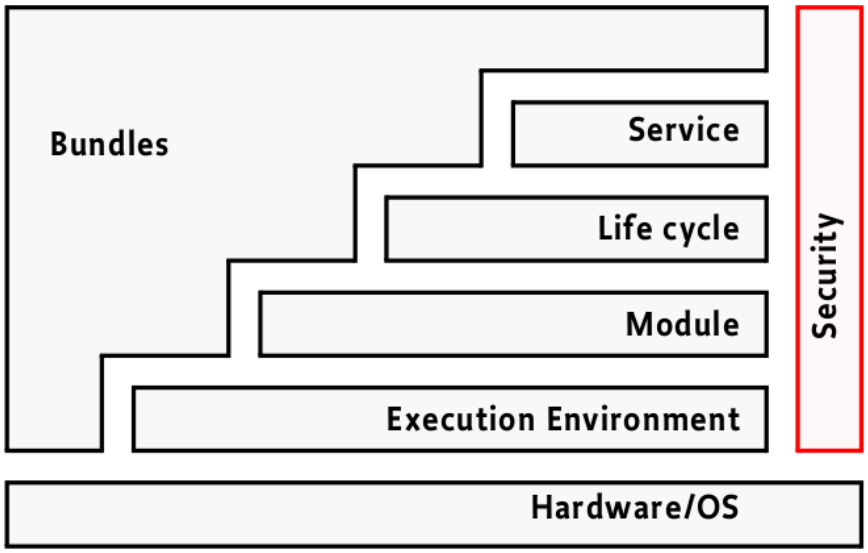


Figura G.1: Arquitectura en capas de OSGi.

G.1. Security Layer

La capa de seguridad o *security layer*, es una capa opcional, que esta basada en la seguridad provista por *Java 2 security*. Esta implementa una arquitectura de *sandbox* donde existen (dinámicamente) 2 tipos de códigos: seguros y restringidos. Esta distinción se hace en base a las políticas de seguridad. Los códigos restringidos corren sobre el *sandbox* donde el acceso a los recursos del sistema operativo están limitados. Por más información ver: *http://Java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-specTOC.fm.html* .

La capa *Security Layer*, complementa su capa inferior con una amplia gama de mejoras en restricciones y cubre muchos huecos que *Java 2 security* no contempla. Las características que brinda esta capa son:

- Autenticación de código: permite autenticar el código mediante 2 formas diferentes: por ubicación del *bundle* o por la firma de los mismos.
- Brinda API de seguridad: los *bundles* pueden implementar los mismos *stubs* que provee *Java 2 security,* con lo cual permite chequear los permisos y decidir con qué privilegios se puede ejecutar un *bundle* según las políticas definidas.

G.2. Module Layer

La plataforma estándar de Java provee un limitado soporte para empaquetar, hacer *deploy* y validar aplicaciones y componentes. Por lo tanto varios proyectos basados en Java, como JBoss, NetBeans y OSGi han creado diferentes *module layers* que se especializan en la carga de clases, empaquetados (llamados *bundles* en el caso de OSGi) y validación de aplicaciones y componentes.

G.2.1. Bundles *OSGi*

OSGi define la unidad de modularización llamada *bundle*. A su vez también lo define como la única forma de poder realizar los *deploys* de las aplicaciones.

Un *bundle* es un archivo JAR (Java Archive), que contiene clases de Java y otros recursos (como puede ser archivos de configuración, imágenes, etc), que en conjunto puede proveer funcionalidades por sí mismo.

Además cada *bundle* contiene, un archivo MANIFEST.MF (como cualquier JAR) que provee información del *bundle*, este es utilizado por el *framework* para poder cargarlo y ejecutarlo correctamente. La información que se puede incluir en el MANIFEST esta dividida en secciones que se listan a continuación (para tener una descripción de cada campo ir a la referencia "*A Framework for Connecting Home Computing Middleware, Distributed Computing Systems Workshops*"[67]):

- Bundle-ActivationPolicy
- Bundle-Activator
- Bundle-Category
- Bundle-Classpath
- Bundle-ContactAddress
- Bundle-Copyright
- Bundle-Descripción
- Bundle-DocURL
- Bundle-Localization
- Bundle-ManifestVersion
- Bundle-Name
- Bundle-NativeCode
- Bundle-RequiredExecutionEnvironment
- Bundle-SymbolicName
- Bundle-UpdateLocation
- Bundle-Vendor
- Bundle-Version
- DynamicImport-Package
- Export-Package
- Export-Service
- Fragment-Host
- Import-Package
- Import-Service
- Require-Bundle

G.3. Life Cycle Layer

La capa de ciclo de vida o *Life cycle layer* provee una API para controlar la seguridad y las operaciones relativas al ciclo de vida de los *bundle objects*. Para esto se basa en las capas: *Module Layer* y *Security Layer*.

Provee una API que cubre todas las transiciones de estados de los *bundles*: instalación, arranque, parada, actualización, desinstalación y monitorización. Permite analizar el estado en el cual se encuentra el *framework* y gestionarlo de forma remota. Es responsabilidad de esta capa, permitir que la API se pueda usar en un *sandbox* y la especificación (opcional) de permisos con un alto grado de granularidad.

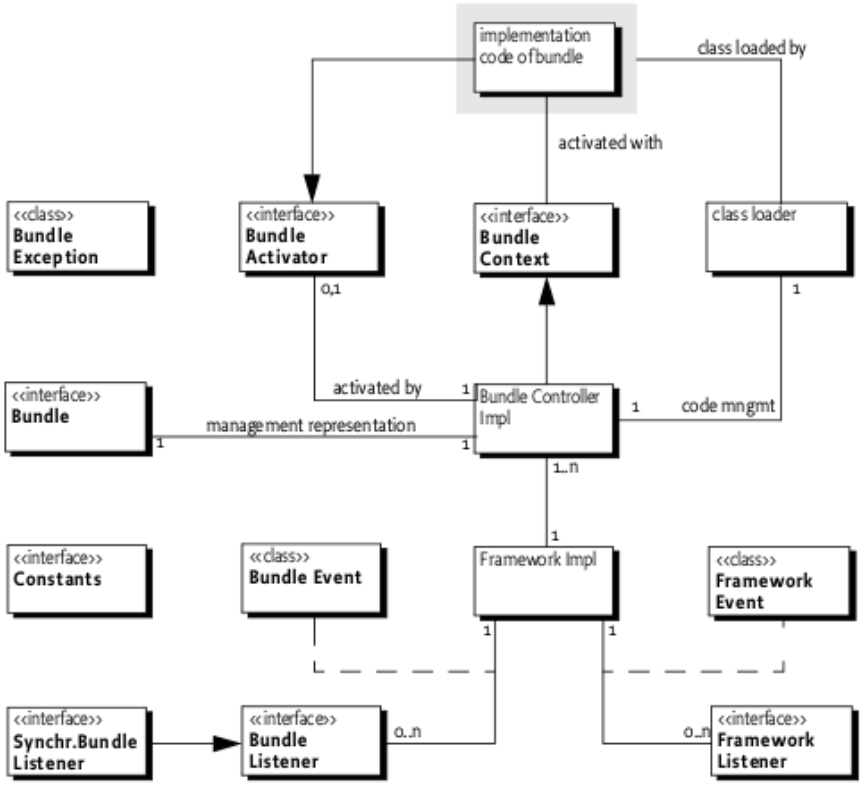


Figura G.2: Clases relacionadas con el ciclo de vida de los *bundles*.

Un *bundle* es una unidad de *servicios* instalado dentro del *framework*. Cada *bundle* tiene su propio *bundle context* que tiene los parámetros de ejecución (contexto) de cada *bundle*. El *framework* pasa el *bundle context* a un *Bundle Activator* cada vez que el *bundle* es activado o desactivado.

El *bundle activator*, es una interfaz que implementa una clase dentro del *bundle* para que el mismo se active o desactive.

A su vez dentro de la capa los *bundle* pueden tener eventos, *bundle event*, las cuales son señales en el ciclo de vida de un *bundle* que permite a otro *bundle*, de forma sincrónica, mediante un *Framewok Listener* recibir los eventos que este genera.

Por último, el *framework* es representado mediante un *System Bundle* el cual permite interactuar directamente con el mismo.

En la figura G.2, se puede observar la principales clases involucradas en el ciclo de vida de los *bundles* y las relaciones entre sí.

G.3.1. Bundle Object

El *bundle objects* es la representación lógica del *bundle* dentro del *framework*. El *Management Agent*, que también es un *bundle*, gestiona el ciclo de vida de cada *bundle* a través del *bundle object*.

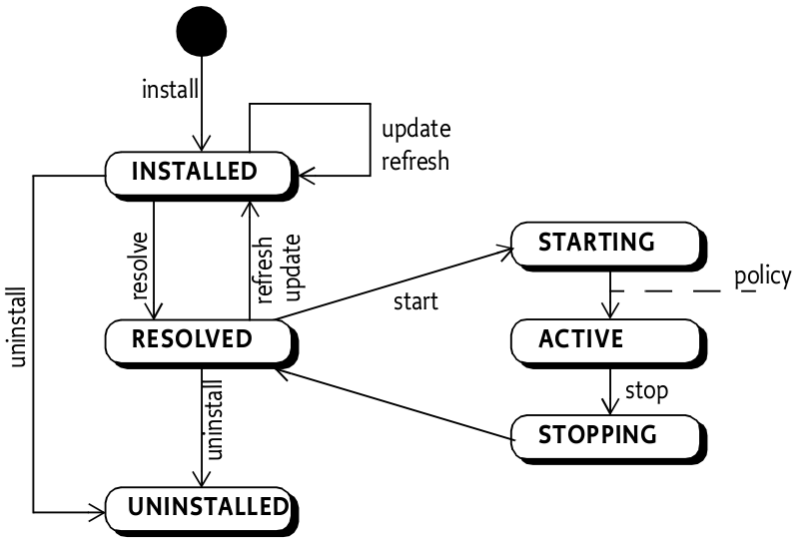


Figura G.3: Transiciones de estados para los *bundles*.

G.3.1.1. Identificación de bundles

Hay varias formas de poder identificar un *bundle* dentro del *framework*, las cuales varían según el alcance que se le quiera dar:

Bundle Identify: es un número que el *framework* le asigna al *bundle* cuando este es instalado y permanece durante todo el ciclo de vida del mismo.

Bundle location: es una URL asignada por el *Management agent,* durante la instalación, por lo general es la ubicación del JAR, sin embargo esto no es obligatorio. La URL es única y no cambia cuando el *bundle* es actualizado.

Bundle Simbolic Name: es un nombre asignado por el desarrollador, que junto con la versión son una identificación global única.

G.3.1.2. Estados de los bundles

Durante su ciclo de vida, un *bundle* pasa por varios estados, cada uno determina las acciones y propiedades que los mismos pueden tener. En la figura G.3 se muestra los 6 estados posibles en los que se puede encontrarse un *bundle* y la transiciones de estados.

En detalle, los posibles estados de un *bundle* son:

Installed: el *bundle* fue instalado en el *framework* pero aún no se puede utilizar debido a dependencias no satisfechas.

Resolved: todas las clases de Java que necesita el *bundle* están disponibles, este estado indica que puede ejecutarse o que fue parado.

Starting: indica que fue llamado el método *BundleActivatorBundle Activator.start*, pero todavía no se a terminado de ejecutar la función. Si el *bundle*

contiene políticas de activación, el mismo se mantiene en este estado hasta que no se active en la forma que dicta la política.

Active: el *bundle* fue activado exitosamente y está corriendo.

Stopping: el *bundle* esta siendo parado, se ejecutó la función *BundleActivator.stop.*

Uninstalled: el *bundle* fue desinstalado del *framework*, de este estado no puede pasar a ningún otro, es un estado pozo.

G.3.1.3. System Bundle

System bundle o *"bundle* del sistema", es la representación del *framework* dentro de la plataforma. Mediante el *system bundle*, el *framework* registra servicios que pueden ser utilizados por otros *bundles.* Sin embargo difiere en algunos aspectos con los *bundles* "comunes":

- Identificación especial:
  - Tiene asignado el identificar 0 como su *Bundle Identify.*
  - Su *Bundle location* es *"System Bundle".*
  - Tiene un *Bundle Simbolic Name* distinto para cada versión, pero es un alias del nombre: *system.bundle.*
- El ciclo de vida del *bundle* no puede ser manejado como un *bundle* normal.

G.3.2. Events

Como ya se comentó, el *Life Cycle layer* soporta eventos, estos se pueden separar en 2 grandes tipos:

- *BundleEvent,* son eventos que reportan cambios en el ciclo de vida de un *bundle.*
- *FrameworkEvent*, son eventos que reportan errores y cambios en el *framework.*

Igualmente los eventos se pueden extender permitiendo ampliar sus posibilidades, las cuales se dejan para futuras implementaciones.

G.3.2.1. Listeners

Para cada tipo de evento hay una interfaz definida para un *Listener* quién recibe la notificación del evento (sincrónico u asincrónico).

G.4. Service Layer

La *Service Layer* o "capa de servicios" define un modelo colaborativo, dinámico, integrado con la capa de *Life Cycle,* que permite publicar, buscar y ligar los servicios.

Los servicios son objetos Java que implementan una o más interfaces, registrados en el *service registry.* Los *bundles* pueden registrar, buscar o recibir notificaciones de los cambios de estados de lo servicios.

Las características principales que provee esta capa son las siguientes:

Collaborative: la capa provee mecanismos para la publicación, descubrimiento y *bind* entre servicios sin necesidad de un conocimiento a priori de los *bundles.*

Dynamic: el mecanismo de servicios se adapta a los cambios en el mundo exterior y en las estructuras internas, dando estabilidad a los servicios.

Secure: permite restringir accesos de servicios.

Reflective: provee un acceso total a los estados del *Service Layer.*

Versioning: provee mecanismos que hacen posible la adaptación al dinamismo de los servicios y los *bundles* que evolucionan con el tiempo.

Persistent Identifier: provee mecanismos a los *bundles* para que puedan rastrear servicios entre reinicios del *framework.*

En la figura G.4, se puede ver el diagrama de clases para esta capa, la cual muestra la interdependencias.

G.4.1. Services

En la plataforma OSGi, los *bundles* son desarrollados alrededor de una cooperación de servicios disponibles a través de un registro de servicios. Los servicios dentro de la plataforma, son definidos de forma semántica mediante los *service interface* e implementadas a través de los *service objects.*

Los *service objects* pertenecen y corren dentro de los *bundles.* El *bundle* debe registrar los *service objects* en el *framework service registration*, para que las funcionalidades de los servicios estén disponibles para otros *bundles* dentro del *framework.*

Los *service interfaces* son la especificación de los métodos públicos de los servicios. En la prctica los desarrolladores crean los *service objects* que implementen el *service interfaces* y registran las interfaces de servicios en el *framework.* Una vez que los *bundles* registran los *service interfaces* bajo un nombre, los servicios asociados pueden obtener estos servicios publicados, referenciados por el nombre.

En general, los servicios registrados son referenciados mediante el *ServiceReference.* Esto evita crear dependencias de servicios dinámicos entre los *bundles,* en particular, cuando un servicio necesita conocer si existe otro servicio pero no necesariamente una implementación concreta.

Cada servicio registrado en el *framework* tiene un único *ServiceRegistration object* y puede haber más de un *ServiceReference* que se refieren a él. Los *ServiceReference* exponen las propiedades de registro, el *Service Object* e incluye un serie de interfaces de servicios que ellos implementan. El *ServiceReference object* se usa para adquirir el *service object* que implementa la interfaz de servicio que se desea obtener.

Los *bundles* registran los servicios en el *framework* cuando es llamada una de las siguientes funciones:

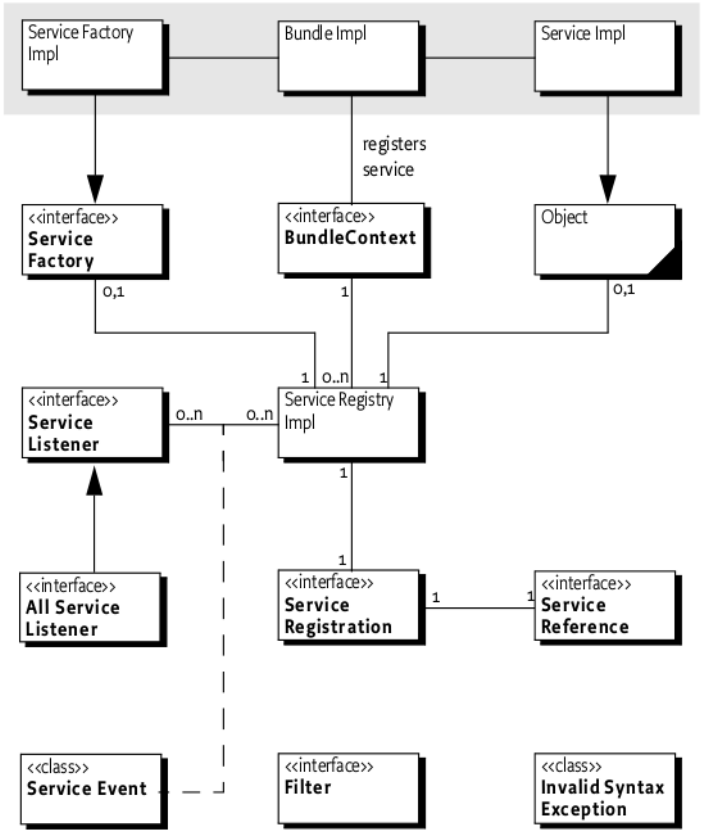


Figura G.4: Diagrama de clases de *Service Layer*.

- `registerService(String,Object,Dictionary)` para registrar un servicio bajo una sola interfaz.
- `registerService(String[],Object,Dictionary)` para registrar un servicio bajo más de una interfaz.

G.5. Framework API

La API está definida en el paquete *org.osgi.framework* que consta de las siguientes interfaces:

AdminPermission Un *bundle* que implementa esta interfaz, define su comportamiento ante el acceso desde los invocadores a operaciones administrativas privilegiadas e información sensible.

AllServiceListener Los *ServiceListener objects*, únicamente reciben los *ServiceEvent* de los *bundles* que registraron el servicio y que su ruta de paquetes es igual. *AllServiceListener* es la interfaz de un *ServiceEvent listener* que no filtra los eventos de acuerdo al paquete del *bundle*.

Bundle Son los *bundles* instalados dentro del *framework*, los *bundles objects* son los puntos de acceso al ciclo de vida del *bundle*.

BundleActivator Personaliza el comportamiento del *bundle* ante el evento de START y STOP.

BundleContext Provee el contexto de ejecución del *bundle* dentro del *framework*. Es válido únicamente en los estados STARTING, ACTIVE y STOPPING.

BundleEvent Evento del *framework* que refleja un cambio en el ciclo de vida de un *bundle*.

BundleException Refleja que ocurrió un problema en un cambio correspondiente al ciclo de vida de un *bundle*.

BundleListener Permite recibir la información de un *BundleEvent*.

BundlePermission Define la autoridad que tiene un *bundle* de proveer o consumir servicios de otros *bundles*.

Configurable Permite configurar los servicios, esto es usado por los *bundles* que requieren en una configuración previa.

Constants Define las constantes referentes al *framework*.

Filter Permite filtrar *ServiceReferences* de acuerdo a un *string* que especifica la semántica del mismo. Las condiciones deben ser escritas en LDAP (*Lightweight Directory Access Protocol*) ver RFC 1960: *http://tools.ietf.org/html/rfc1960*.

FrameworkEvent Describe un evento del *framework*, estos son recibidos por los *FrameworkListener*.

FrameworkListener Es la interfaz mediante la cual, el *framework* le permite a los desarrolladores suscribirse a sus eventos. Estos son distribuidos de forma asincrónica.

FrameworkUtil Contiene el método para crear objetos *Filter*.

InvalidSyntaxException Permite indicar que la condición de un filtro no fue sintácticamente bien especificado.

PackagePermission Permite definir los permisos que tiene un *bundle* para importar o exportar paquetes.

ServiceEvent Describe un evento del *framework*, referente a un cambio en el ciclo de vida de un servicio.

ServiceFactory Le brinda el control sobre la generación y destrucción de *ServiceReference*.

ServiceListener Permite captar los *ServiceEvent* de forma sincrónica.

ServicePermission Define los permisos que tiene un *bundle* para publicar o consumir un servicio.

ServiceReference Es la referencia a un servicio. Un objeto *ServiceReference* puede ser utilizado por varios *bundles* y permite inspeccionar las examine las propiedades del servicio, así como consumirlo.

ServiceRegistration Es la referencia a un servicio registrado, permite acceder al *ServiceReference*, modificar las propiedades del servicio, y desregistrarse.

SynchronousBundleListener Permite recibir *BundleEvent* de forma sincrónica a diferencia del *BundleListener*.

Version Identificación de versión de paquetes y *bundle*s.

Esta especificación se puede ver en "*OSGi Service Platform Core Specification*"[67].

G.6. Evolución del framework

En el año 1998 se crea JCP (*Java Community Process*) el proceso mediante el cual los interesado pueden sugerir o pedir cambios o mejoras a la plataforma Java, estos se especifican mediante JSRs (*Java Specification Request*). OSGi nació en el año 1999 como el JSR 8 y está disponible en "*Open Services Gateway Specification*"[39].

La especificación tuvo una rápida evolución, en Mayo del año 2000 aparece OSGi Release 1 (R1), en Octubre del año 2001 la especificación R2, en Marzo del año 2003 la R3, en Octubre del año 2005 aparece la especificación del núcleo de R4 y en Setiembre del 2006 con JSR 232 de por medio, aparece la especificación de OSGi Movil, en Mayo del año 2007 con JSR 291 de por medio aparece R4.1 y actualmente se esta debatiendo la especificación R4.2. En la figura G.5 se pueden ver los *releases,* antes mencionados, ordenados cronológicamente.

Figura G.5: Evolución del *framework* OSGi.

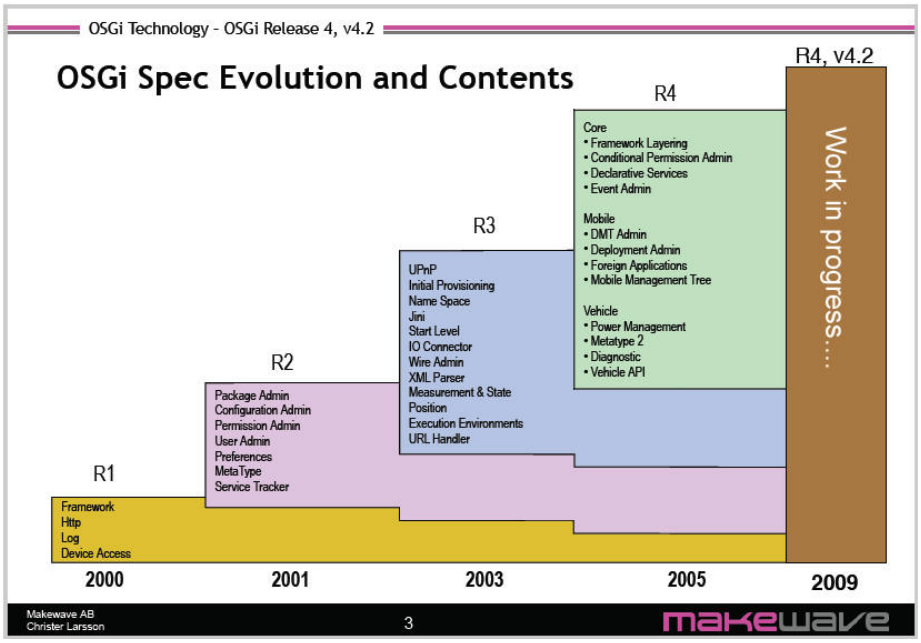


Figura G.6: Evolución de la especificación de OSGi y su contenido.

En G.6 se puede ver una imagen extraída de "*OSGi Release 4, version 4.2*"[42], dónde se muestra la evolución del contenido de la especificación de OSGi.

G.7. *Frameworks* en el mercado

Dentro del mercado existen varios frameworks disponibles, en la mayoría de los casos, son de código libre pero también se encuentran algunos privativos. Se comenta las diferencias y cualidades de los frameworks más conocidos.

Según la referencia "*OSGi Pagina oficial*" [66] los implementaciones certificadas de la especificación R4 son:

- Makewave Knopflerfish Pro 2.0
- ProSyst Software mBedded Server 6.0
- Eclipse Equinox 3.2
- Samsung OSGi R4 Solution
- HitachiSoft SuperJ Engine Framework

Sin embargo, existen varias implementaciones más que las antes mencionadas, en especial de *releases* menos recientes. Algunas de las otras implementaciones disponibles son:

- Oscar / Apache Felix
- Newton Framework
- Concierge

En el artículo "*Concierge: a service platform for resource-constrained devices*" [80] se evalúan las implementaciones más adecuadas para dispositivos embebidos de bajos recursos: Oscar, Knopflerfish y Concierge.

En este artículo se comentan los resultados de pruebas de performance comparativa de estos 3 y Concierge resultó ser mejor que el resto en casi todas las métricas. En especial se destaca: menor consumo de *heap*, menor *footprint* y menor tiempo de ejecución para determinados *benchmarks*.

Actualmente Concierge no tiene implementación de OSGi especificación R4, pero cumple con R3.

Apéndice H

Soluciones Similares

Luego de ser tomada la decisión de usar OSGi como *framework* de base para este proyecto, se analizaron cuatro artículos de domótica basadas en dicha tecnología:

- *Enhancing Residential Gateways: A Semantic OSGi Platform* [16].
- *Pervasive Service Composition in the Home Network* [9].
- *Context-Aware Service Composition for Mobile Network Environments* [43].
- *An OSGi-Based Semantic Service-Oriented Device Architecture* [31].

H.1. Enhancing Residential Gateways: A Semantic OSGi Platform

En el *paper* "*Enhancing residential gateways: a Semantic OSGi platform*"[16] se presentan las ventajas de usar OSGi como plataforma SOA sobre la cual construir los sistemas domóticos, ya que potencia la interacción entre servicios. Sin embargo, marca como desventaja el acoplamiento de interfaces entre servicios, la selección de servicios en base a las propiedades que este define y la invocación de métodos precisa la firma de los mismos.

Se plantea transformar OSGi en una plataforma de descubrimiento de servicios semántica, pudiendo diferenciar servicios sinónimos y homónimos. Aquí se cita el problema de descubrimiento de servicios en WS y como conclusión de este análisis, se recomienda el uso de ontologías. Estas permiten explicitar de forma compartida, una conceptualización del problema, que a su vez, permite seleccionar el servicio más adecuado.

La solución propuesta en este artículo fue modificar el servicio de registro de OSGi, integrando una clasificación semántica mediante ontologías, descripción semántica de las propiedades e información sobre las invocaciones.

H.2. Pervasive Service Composition in the Home Network

En el *paper* "*Pervasive Service Composition in the Home Network*"[9] se presenta una infraestructura para el desarrollo de *pervasive services* mediante un modelo de descubrimiento y publicación de servicios entre nodos distribuidos en redes heterogéneas. La solución planteada intenta ocultar las características propias de cada *middleware*.

Si bien en este artículo se mencionan los motores de inferencia para el descubrimiento semántico sobre ontologías, adoptan el descubrimiento de servicios estático en tiempo de desarrollo.

Identifican dos tipos de componentes: *Discovery Base Drivers,* que son los encargados de cada *middleware,* y *Refining Drivers,* cuya función es dar un grado de abstracción mayor.

Presenta dos formas dinámicas de invocación de métodos: *Java reflection* y generación de *bytecode*. Si bien la segunda es más eficiente, la primera es mucho más simple y fácil de comprender.

Eligen el *framework* OSGi ya que este les brinda capacidades de *plug-and-play* de los distintos componentes. En la solución planteada se destaca: la publicación automática de servicios, manejo uniforme de los servicios locales y remotos, adaptación automática a los nuevos servicios, *ranking* dinámico y *switch* automático de servicios.

H.3. Context-Aware Service Composition for Mobile Network Environments

El artículo "*Context-Aware Service Composition for Mobile Network Environments, Ubiquitous Intelligence and Computing*"[43] se centra en la composición de servicios en redes dinámicas de equipos móviles. Si bien auguran el rápido crecimiento de OSGi en los últimos años, marcan la necesidad de resolver el problema de composición de servicios. Una de las características de esta solución, al igual que en "*Pervasive Service Composition in the Home Network*"[9] es que provee *ranking* dinámico de servicios, criterio que se usa para la composición de los mismos. Para este trabajo se toman en consideración pequeñas redes PAN, VAN (Vehicle Area Network) y redes hogareñas.

H.4. An OSGi-Based Semantic Service-Oriented Device Architecture

En el *paper "An OSGi-Based Semantic Service-Oriented Device Architecture*"[31] se resaltan los problemas de heterogeneidad entre datos y mensajes que intercambian los sistemas interoperantes. Afirman qué, a nivel semántico, está bastante maduro si se usa WS para la comunicación y en particular si se consideran los estándares para WSCDL (*Web Services Choreography Description Language*) y BPEL (*Business Process Execution Language*). Destaca OSGi como el *framework* más maduro que soporta sistemas con arquitectura SODA (*Service Oriented Device Architecture*) tradicional.

Este trabajo apunta a construir un sistema SeSODA (*Semanticaly Enabled SODA*) para lo cual se basa fuertemente en el descubrimiento de servicios provisto por OSGi usando LDAP.

Apéndice I

Getting Started

En este apéndice se explica lo mínimo que se debe implementar en un sistema de domótica funcional utilizando la API propuesta como solución para este proyecto.

Se presenta un caso simple de ejemplo, dónde se quiere implementar un solo *TechManager* para una tecnología específica, un *Proxy* el cual publique todos los servicios del *TechManager* de forma pública. Se habilitará la seguridad de forma que sólo el *Proxy* generado podrá comunicarse con el mismo.

I.1. TechManager

Para poder crear un *TechManager* hay que seguir los siguientes pasos:

- Crear un *bundle* nuevo.
- En el MANIFEST importar los siguientes paquetes:
  - `uy.edu.fing.domo.api.exception`
  - `uy.edu.fing.domo.api.security`
  - `uy.edu.fing.domo.api.techmanager`
  - `uy.edu.fing.domo.api.util`
  - `org.osgi.util.tracker`
- Implementar el *TechManagerCustom* de forma tal, que extienda *TechManagerDomo.*
  - En el constructor implementar lo necesario para conectarse con el *middleware* específico.
  - En la función `isUsingSecurity(...)`devolver `true` para que el servicio chequee seguridad.
  - En la función `getDefaultPropertiesForServices(...)` incluir este código:

```
Properties properties =new Properties();
properties.put(
    DomoApiConstant.DOMO_API_PROXY_INSECURE,
    Boolean.TRUE);
return properties;
```

Esto hace que todos los servicios publicados sean encapsulados por el *Proxy* que se va a definir.

- En las funciones `hasNewIAuthenticationChecker(...)` y `hasRemoveIAuthenticationChecker(...)` dejar en blanco la implementación.
- En la función `getAllPrivateServices(...)` retornar un `Map<IServiceSecure, Properties>` donde los *IServiceSecure* sean las implementaciones las clases concretas de *ServiceDomoSecure* que implementan cada servicio deseado. Las *properties* pueden estar en `null`, ya que no es necesario definir ninguna propiedad específica para este caso de prueba.

- Implementar *ServiceDomoSecureCustom* tal que extienda *ServiceDomoSecure.*
  - En la función `isUsingSecurity(...)` devolver `true` para que el servicio chequee seguridad.
  - En el constructor pasar los parámetros necesarios para saber con qué dispositivo se mapea cada uno de los *ServiceDomoSecureCustom.*
  - En la función `getInternalMethods(...)` devolver los métodos definidos para cada uno de los servicios.
  - En la función `internalExecuteMethod(...)` definir cómo se ejecutan cada uno de los métodos devueltos en la función anterior según la tecnología utilizada.

- Implementar el *Activator* de *bundle TechManagerActivatorCustom* tal que extienda *TechManagerActivator.*
  - En la función `getTechManager(...)` devolver la instancia del *TechManagerCustom* (la forma de obtener el *Bundle Context* es utilizar `getContext(...)`).
  - En la función `getTechManagerProperties(...)` retornar `null`.
  - Las funciones `startOtherThingsTechManagerActivator(...)` y `stopOtherThingsTechManagerActivator(...)` dejarlas en blanco.

I.2. Proxy

Para crear el *Proxy* es necesario:

- Crear un *bundle* nuevo.
- En el MANIFEST importar los siguientes paquetes:

- *uy.edu.fing.domo.api.exception*
- *uy.edu.fing.domo.api.proxy*
- *uy.edu.fing.domo.api.security*
- *uy.edu.fing.domo.api.techmanager*
- *uy.edu.fing.domo.api.util*

- Implementar el *ProxyDomoCustom* tal que extienda *ProxyDomo*.

```
ProxyNew(BundleContext context) {
super(context);
}
void hasSetToken(){
    if (getTechManagers().size()>0){

       for (ITechManager t:getTechManagers())

         t.publishAllServices(this.getToken());
    }
}
Properties
getDefaultPropertiesForWrapper() {
return null;
}
boolean
useSameProperiesForWrapperFromServiceSecure() {
    return true;
    }
boolean
wrapperAllServiceDomoSecure() {
return true;
}

void hasChangeITechManager(
    ITechManager service) {}
void hasChangeIServiceSecure(
    IServiceSecure service) {}
void hasNewIServiceSecure(
    IServiceSecure service) {}

void hasNewITechManager(
    ITechManager service) {
    if (this.getToken()!=null)

       service.publishAllServices(this.getToken());
}
```

```
void hasRemoveIServiceSecure(
    IServiceSecure service) {}
void hasRemoveITechManager(
    ITechManager service) {}
```

- En la función `getDefaultPropertiesForWrapper(...)`, como esto es un ejemplo y no se quiere poner ninguna etiqueta devolver `null`.
- En la función `useSameProperiesForWrapperFromServiceSecure(...)`, devolver `true`.
- En la función `wrapperAllServiceDomoSecure(...)` devolver `true`. De esta forma todos los servicios publicados por el *TechManager* serán encapsulados y vueltos a ser publicados como *IServiceDomo*.

I.3. Firmas

En lo que a las firmas respecta, existen 3 etapas: la generación de la firma, el uso de la firma (firmado de archivos) y por último la verificación.

Para generar una nueva firma, si ya no se cuenta con una ejecutar este comando:

```
keytool -genkey -alias duke
-keypass dukekeypasswd
```

Por más información de cómo utilizar este comando, ver: "man keytool"

Para firmar los *bundles* ejecutar para cada archivo el siguiente comando:

```
jarsigner -keystore [keyStoreLocation]
-storepass [KeyStorePassword] -keypass [KeyPassword]
-signedjar [nombreBundeFirmado].jar [nombreBundle].jar
[nombreAliasKeyStore]
```

I.4. Ejecución

Para poder correr este ejemplo en OSGi tienen que estar los siguientes *bundles* instalados: *EventAdmin* y *ServiceTracker*. Con ello un archivo de inicialización de ejemplo sería de la siguiente manera:

```
-init
-istart shell-1.0.0.RC2.jar
-istart service-tracker-1.0.0.RC2.jar
-istart event-admin-1.0.0.RC2.jar
-istart ApiS_1.0.0.jar
-install ProxyNewS_1.0.0.jar
-install TechManagerNewS_1.0.0.jar
```

I.5. R-OSGi

Para usar R-OSGi es necesario definir una serie de propiedades, ya sea en archivo de inicialización del *framework* OSGi (por defecto `init.xargs`) o como propiedad de la JVM (con `-D<propiedad>`).

- SLP:
  - puerto para la comunicación *multicast*: `net.slp.port=<puerto>`
  - interfaz de salida para la comunicación: `net.slp.interfaces=<dirección IP>`
- R-OSGi
  - puerto para la comunicación punto a punto: `ch.ethz.iks.r_osgi.port=<port>`
  - dirección IP del host: `ch.ethz.iks.r_osgi.ip=<dirección IP>`

Apéndice J

Circuito

En este apéndice se presenta la solución de *hardware* que fue necesario implementar para mostrar la factibilidad del caso de estudio con el *middleware* Usb4All.

J.1. Componentes

Para construir la placa se necesitan los siguientes elementos:

- 1 socket DIL de 8 patas
- 1 conector tira doble 40 pins
- 2 LEDs (1 rojos, 1 verdes)
- 5 resistencias 10 KΩ
- 1 transistor NPN C547C W79s
- 1 fotoresistencia (LDR)
- 1 amplificador operacional lm358n dil8
- 1 relé at1rc
- 8 conector tira doble 2 pins

J.2. Esquemático Electrónico

En la figura J.1 se presenta el diseño esquemático electrónico de del circuito que se construyó.

J.3. Diseño de Pistas

En la imagen J.2 se presenta el diseño de pistas derivado del esquemático electrónico.

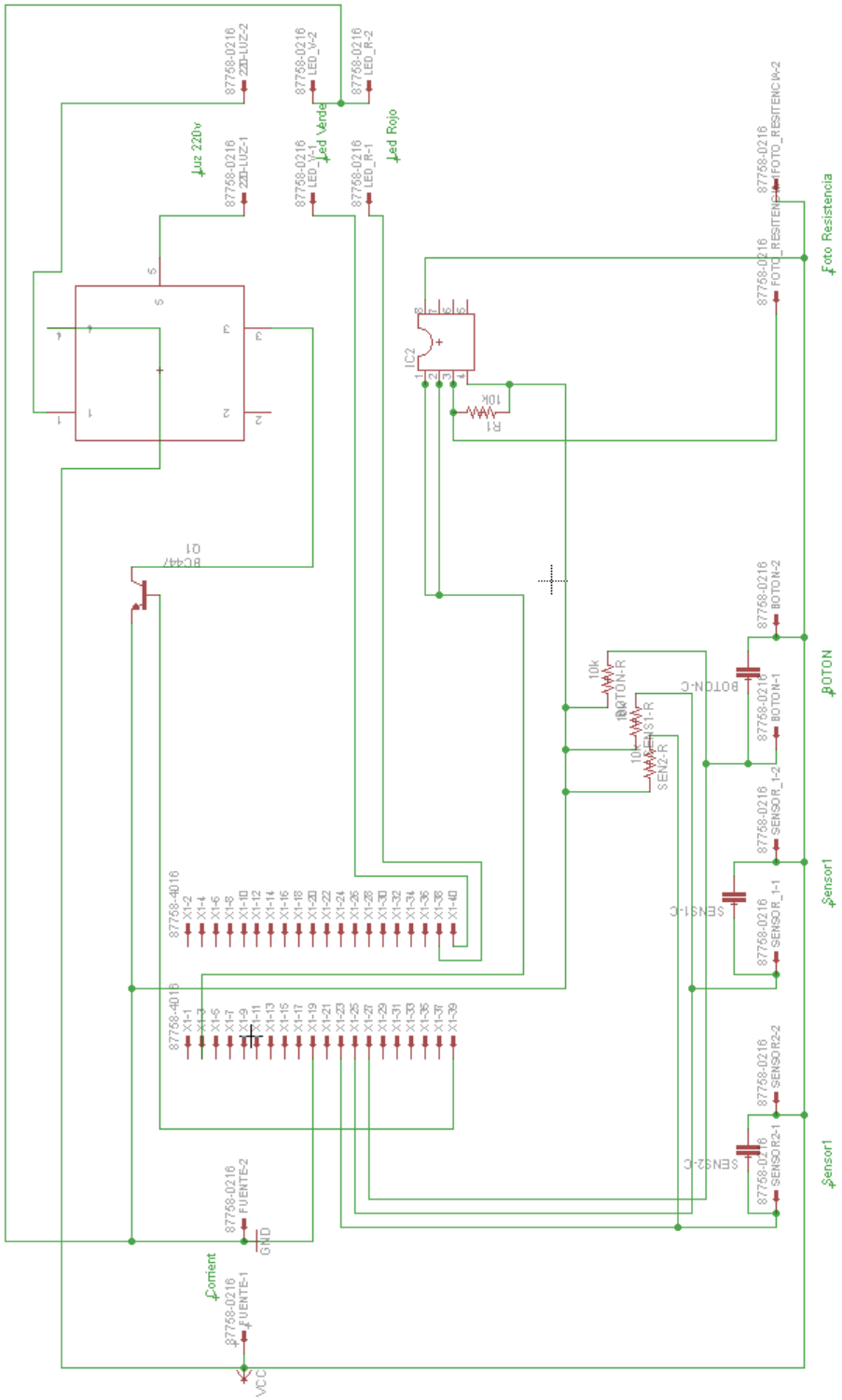


Figura J.1: Diseño esquemático.

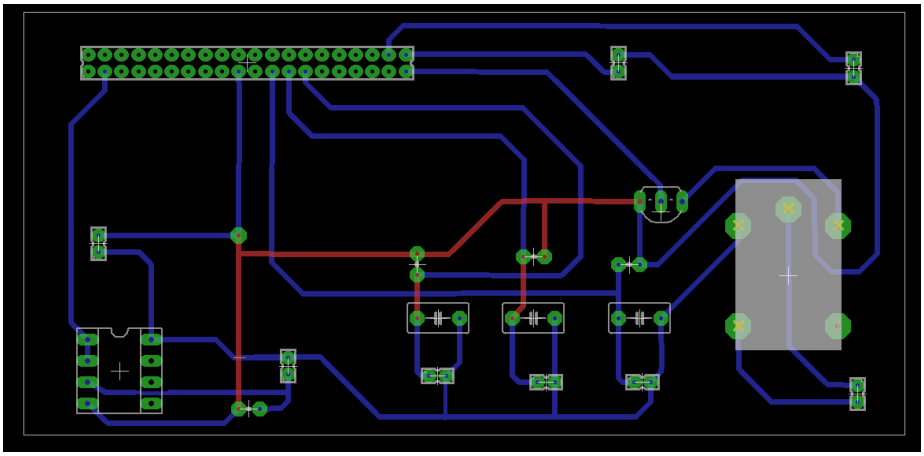


Figura J.2: Diseño de pistas.

Apéndice K

mini-HOWTO JamVM en x86

En este apéndice se detallan los pasos para hacer una instalación de JamVM y montarle arriba OSGi en una arquitectura x86.

K.1. Introducción

JamVM es únicamente la máquina virtual de Java, para correr cualquier clase Java sobre ella, es necesario tener las bibliotecas (dónde están definidas entre otras, las clases *class*, *package*, etc).

A continuación se detallan los pasos para instalar: JamVm 1.5.3 y GNU-Classpath 0.98.

K.2. Instalación

Pasos a seguir (el orden en que se siguen los pasos es importante), `[ruta]` sera la ruta donde quedará instalado:

- Instalar GNU-Classpath
  - bajar la versión 0.98 de `http://www.gnu.org/software/classpath/downloads/downloads.html`
  - resolver las dependencias: instalar sun-Java6-jdk, antlr y libtool.
  - `sh ./configure --disable-gtk-peer --disable-gconf-peer --disable-plugin --disable-examples --disable-gjdoc --disable-debug --disable-qt-peer --prefix=[ruta]`
  - `make`
  - `make install`
- Instalar jamVM
  - bajar la versión 1.5.3 de `http://sourceforge.net/projects/jamvm/files/jamvm/JamVM%201.5.3/jamvm-1.5.3.tar.gz/download`
  - `sh configure --prefix=[ruta] --with-classpath-install-dir=[ruta]`

- `make`
- `make install`

Apéndice L

mini-HOWTO JamVM en MIPS/Asus

En este anexo se detallan los pasos para hacer una instalación de JamVM en un Asus con OpenWrt.

L.1. Introducción

JamVM es únicamente la máquina virtual de Java, para correr cualquier clase Java sobre ella, es necesario tener las librerías (donde están definidas entre otras, las clases *class*, *package*, etc).

Para hacer esta instalación, el principal problema es encontrar (o fabricar) versiones compatibles entre JamVM y GNU-Classpath y que sean compatibles con la arquitectura MIPS y librerías de la placa embebida.

En la tabla L.1 se puede ver la compatibilidad entre las versiones de JamVM y GNU-Classpath.

L.2. Instalación

Para poder instalarlo y compilarlo se tiene que contar con lo siguiente:

- Los fuentes de OpenWrt (en este caso, se usó el *trunk*: `svn://svn.openwrt.org/openwrt/trunk/` que corresponde a la revisión r: 17438).
- Tener descargado el repositorio de packages `svn://svn.openwrt.org/openwrt/packages/`.

	Classpath		
JamVM	**0.97.1**	**0.97.2**	**0.98**
1.5.1	compatibles	incompatibles	incompatibles
1.5.2	incompatibles	incompatibles	incompatibles
1.5.3	incompatibles	incompatibles	compatibles

Cuadro L.1: Compatibilidad de versiones

Pasos a seguir, `[ruta]` sera la ruta donde quedará instalado:

1. Crear un softlink desde `[ruta]/packages/libs/Classpath` en `[ruta]/trunk/packages/` .
2. Ir a `[ruta]/trunk/` *(*es la ruta donde están los archivos fuentes de *OpenWrt).*
3. ejecutar make menuconfig y en el menú *"Libraries"*, marcar Classpath.
4. Ir a la ruta `[ruta]/packages/libs/Classpath` y modificar el Makefile.
   a) Cambiar *PKG_ VERSION:=0.97.2* a *PKG_ VERSION:=0.98*
   b) Cambiar el PKG_MD5SUM para la versión que se quiera compilar. (en este caso la 0.98)
      1) Para lograr esto, bajar de `ftp://ftp.gnu.org/gnu/classpath/classpath-0.98.tar.gz`.
      2) Calcular el MD5 con md5sum.
5. Ir a `[ruta]/trunk/` y ejecutar `make V=99`
6. Luego que termine de compilar, el paquete `classpath_0.98-1_brcm47xx.ipk` queda en `[ruta]/trunk/bin/packages/brcm47xx_uClibc-0.9.30.1`
7. Como la placa no cuenta con espacio de almacenamiento nativo suficiente para instalar el paquete, se necesita un dispositivo de almacenamiento externo que en caso se monta en `/mnt/sda1`.
   a) Hacer un *backup* del archivo `/etc/opkg.conf` a `/root/opkg.conf`
   b) Modificar el archivo `/etc/opkg.conf` para que quede de esta forma:
   src/gz snapshots http://downloads.openwrt.org/snapshots/trunk/brcm47xx/packages
   dest root /mnt/sda1/classpath
   dest ram /mnt/sda1/tmp
   lists_dir ext /var/opkg-lists
   option overlay_root /jffs
   option force_space
   c) Ejecutar: `opkg install [ruta_al_archivo]/classpath_0.98-1_brcm47xx.ipk`
   d) Hacer los siguiente *softlinks*:
      1) `ln -s /mnt/sda1/classpath/usr/lib/classpath /usr/lib/classpath`
      2) `ln -s /mnt/sda1/classpath/usr/share/classpath /usr/share/classpath`
   e) Restaurar el archivo `/root/opkg.conf` al lugar original.
8. Ahora instalar JamVM:

   a) Ejecutar : `opkg update && opkg -force-depends install jamvm` (`-force-depends` para que no de problemas de dependencia de paquetes.)

9. Eso es todo amigos!!! En este momento tenés JamVM instalado.
10. Salut

Apéndice M

mini-HOWTO Mika en MIPS/Asus

En este apéndice se detallan los pasos para hacer una instalación de la JVM: Mika en un *router* Asus wl500w (usado como placa embebida) con un sistema operativo OpenWrt.

M.1. Introducción

A diferencia de JamVM, Mika trae sus propio classpath, lo cual hace más simple su instalación. Sin embargo, ha probado no ser una JVM estable ni lo suficientemente madura para poder desarrollar sobre ella.

M.2. Instalación

Pasos a seguir:

- Descargar del svn de Mika (`svn checkout svn://svn.k-embedded-java.com/Mika/trunk`) la revisión 895.
- Configurar *toolchain* de OpenWrt en el archivo `Configuration/cpu/mips`.
- Arreglar problema de compilación con la función strcmp: en el archivo `core-vm/src/vm/wonky.c` comentar dicha función.
- optimizaciones:
  - En el archivo Jamrules, comentar "`CCFLAGS += -ggdb ;` "
  - En el archivo `Configuration/ant/openwrt`, deshabilitar la información de *debug*: "`JAVA_DEBUG=false`"
- Asegurarse de tener: sun Java-1.6-jre, ant, jikes, awt, jam, gcc
- Compilar con: `ant -DPLATFORM=openwrt`

M.3. Bugs

Durante las pruebas realizadas se han encontrado varios *bugs* que a continuación se detallan.

- Problema de compilación en MIPS (se solucionó comentando la función `strcmp(...)` ver `http://forums.k-embedded-Java.com/forum/index.php?webtag=GENERAL&msg=8.1`).
- JNI sólo compila con gcc no con g++ (ver `http://forums.k-embedded-Java.com/forum/discussion.php?webtag=GENERAL&msg=10.1`).
- Problema al cerrar *sockets* (ver `http://forums.k-embedded-Java.com/forum/index.php?webtag=GENERAL`).

Apéndice N

Gestión

Considerando como inicio del proyecto, la primera reunión (el 26/02/2009) y como fin de proyecto una semana después de la aprobación del trabajo por parte de los tutores (10/02/2009),se dedicaron casi 12 meses (50 semanas), de forma sostenida e intensa, prácticamente siempre en conjunto dividiendo las tareas y con ayuda mutua.

En este apéndice se plantean los detalles de gestión del proyecto y los resultados obtenidos en base a ciertas métricas.

N.1. Trabajo

Las herramientas utilizadas para el proyecto fueron:

- SVN para el control de versiones: `http://subversion.tigris.org/`.
- GoogleDocs para compartir los artículos descargados: `http://docs.google.com/` y en particular, Google *spreadsheets* como sistema de *tickets,* para anotar las tareas pendientes.
- Lyx como procesador de documentos `http://www.lyx.org/`.
- Eclipse con *plug-in* para OSGi de Knopflerfish y Concierge como IDE de desarrollo.
- MPLab para programar el *firmware*: `http://www.microchip.com/`
- Eagle Layout editor para el diseño de esquemáticos y PCBs: `http://www.cadsoftusa.com/`.

N.2. Métricas

A continuación se presentan las 3 métricas relevadas: horas dedicas, uso del SVN y LOC (*Lines Of Code*).

Actividad	Horas
Coordinación	24
Documentación	997
Experimentación	583
Desarrollo	284
Usb4All	285
Total	**1994**

Cuadro N.1: División de recursos por actividad.

N.2.1. Horas Hombre

Para este proyecto se dedicaron más de 1900 horas hombre efectivas divididas en coordinación, documentación, experimentación, desarrollo y tiempo dedicado a Usb4All.

Detalles por actividades:

- Coordinación: son las horas dedicadas a reuniones de proyecto con los tutores.
- Documentación: incluye la recopilación de información del estado del arte, generación de este documento y elaboración de documentos intermedios.
- Experimentación: horas asignadas a pruebas de tecnologías (OpenWrt, Mika, JamVM, OSGi, R-OSGi, JSLP, etc) y desarrollo de prototipos descartables.
- Desarrollo: son las horas dedicadas a la implementación, implantación y verificación de la API.
- Usb4All: se decidió poner este ítem separado debido a la importancia (en cuanto a horas dedicadas). Este ítem incluye, la formación, la experimentación, la adaptación, el desarrollo y la verificación de Usb4All.

En la figura N.1 se presentan los datos en forma de gráfica.

Ya que la documentación llevó casi un 50 % de la dedicación del proyecto es interesante ver cuánto se dedicó a la documentación relacionada con el estado del arte y cuánto al resto del trabajo.

En los primeros 3 meses (período donde se generó prácticamente todo el estado del arte), se dedicaron 414 horas. Sin considerar las etapas preliminares del proyecto. Se obtiene la gráfica de la figura N.2.

En esta gráfica se puede ver que el proyecto fue mayoritariamente de experimentación y documentación, donde se realizó un caso de estudio a modo de probar la factibilidad de la solución presentada.

Por otra parte, resulta interesante analizar las horas dedicadas en función del "triángulo de gestión de proyecto", donde el costo, el tiempo y el alcance determinan la calidad del mismo. En este caso, el alcance fue impuesto, el tiempo (que en este caso es simétrico al costo ya que los recursos son fijos) se fijó por los alumnos. Dado que se intentó hacer un trabajo de la mejor calidad posible, el tiempo dedicado fue algo menor al doble de el esperado. Si bien el promedio de dedicación semanal por persona, es algo menor a 20 horas ($1994/(50*2) = 19{,}95$), tomando en cuenta como base 10 meses, que es lo que dura el año

Figura N.1: Gráfico de división de recursos.

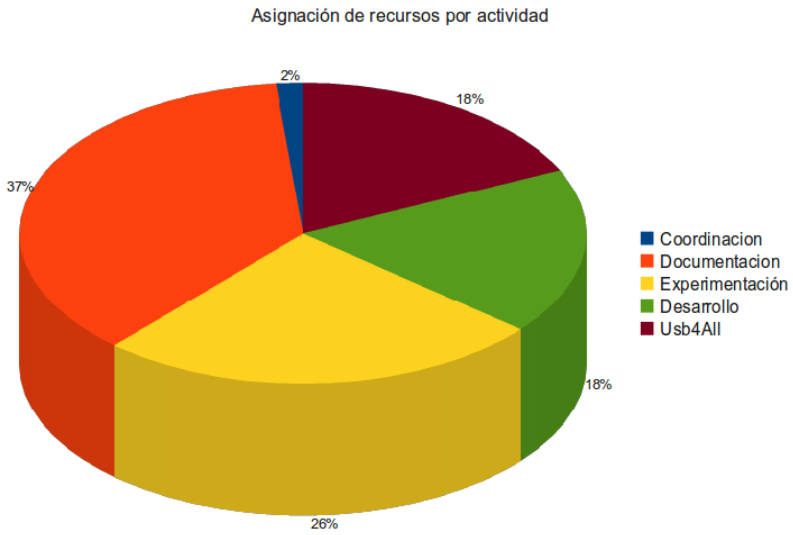


Figura N.2: Gráfico de división de recursos luego del relevamiento del estado del arte.

Proyecto	**LOC**	**LOC de *Tests***	**LOC sin *Tests***
API	1050	0	1050
HTTP	944	7	937
proxyBaño	1344	114	1230
TMUsb4all	919	250	669
TMRF	120	0	120
u4aLibusbJavaAPI	1275	741	534
R-OSGi	69	0	69
Total Java	**5721**	1112	4609
Firmware	748	0	346

Cuadro N.2: LOC por proyecto.

electivo en la facultad, el promedio de horas trabajadas es de 25 horas por persona ($1994/(40*2) = 24{,}93$).

N.2.2. Software Configuration Management (SCM)

En las 50 semanas de proyecto se registraron más de 910 revisiones, lo que da un promedio de 2 revisiones y media por cada día ($910/(50*7) = 2{,}6$). Por otra parte, se realizaron cinco *branches*, cada uno después de un hito o antes de un cambio importante.

Con respecto a las técnicas de respaldo, no solamente se uso el servidor de SVN sino que también se contó con un disco externo donde se respaldó las máquinas virtuales, binarios, imágenes del sistema operativo, entre otros.

N.2.3. LOC

El producto final (sin contar prototipos) consta de siete proyectos Java y uno en C los cuales se muestran en: N.2.

Está métrica incluye únicamente lineas con código del programa y de programas de *test* (excluye, caracteres de espaciado, saltos de lineas y comentarios).

N.2.4. Proyectos Java

Las mediciones de LOC para los proyectos Java se realizaron con la herramienta: Metrics 1.3.6 (ver `http://metrics.sourceforge.net/`).

Para tener una noción del tamaño en LOC de los proyectos, en la figura N.3 se muestran sin incluir los LOC correspondiente a clases para verificación.

Se puede ver que los tres proyectos más grandes son la API, HTTP y el proxyBaño. El proyecto HTTP debe su tamaño en gran parte a la cantidad de lineas de código utilizadas para la generación de las diferentes secciones de la página web con la gestión de usuarios de la misma.

Por su parte, el tamaño del proyecto proxyBaño está dado por la lógica del control de los baños (implementado como una máquina de estado genérica) y la composición de los servicios publicados por los *TechManagers* para así generar y publicar los servicios del estado de cada baño.

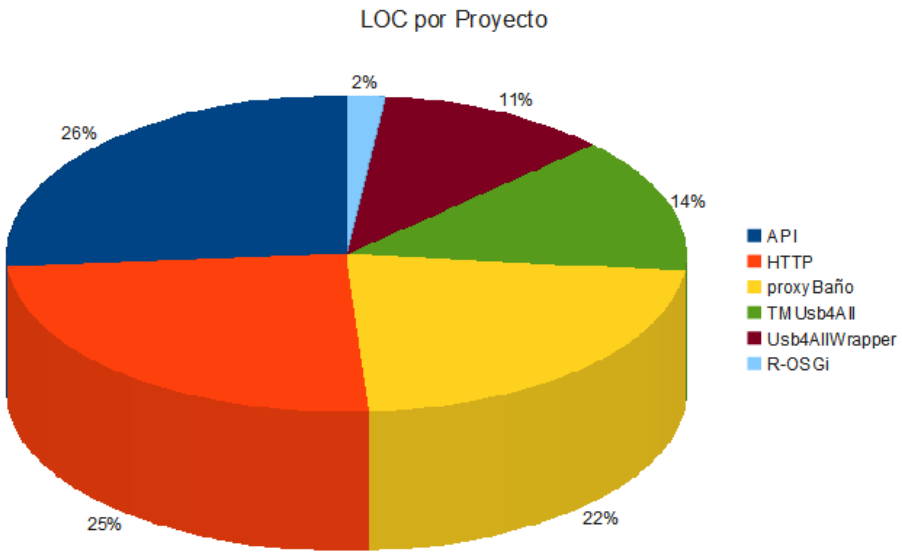


Figura N.3: LOC por Proyecto sin *tests*.

Archivo	LOC
`usr_inputs.c`	141
`usr_inputs.h`	40
`usr_outputs.c`	80
`usr_outputs.h`	46
`io_cfg.h`	39
Total	346

Cuadro N.3: LOC por archivo de *firmware*.

Por otra parte se puede ver que el proyecto para la gestión de publicación entre los diferentes nodos de la solución, utilizando R-OSGi, utiliza muy pocas lineas de código (con respecto al resto de los proyectos), por lo que se puede decir que fue muy sencillo.

N.2.5. Firmware

En esta medición se usó el programa `cloc` (ver `http://cloc.sourceforge.net/`), igual que la anterior no cuenta comentarios ni lineas en blanco.

Hay que resaltar que para el desarrollo de los diferentes módulos generados se usó plantillas ya diseñadas por los autores de USB4All. Se puede observar que el archivo con más lineas de código, también se puede decir que es el más complejo, es `usr_inputs.c`, el cual se encarga de la interacción con los diferentes sensores conectados con el PIC.

N.3. Cronograma

En esta sección se presenta y comparan el cronograma establecido en la propuesta del proyecto y el realizado.

N.3.1. Cronograma Inicial

El cronograma planteado en la propuesta del proyecto fue el siguiente:

- Abril - Junio
  - Relevamiento del estado del arte, pruebas de las alternativas disponibles.
  - Análisis y diseño de la solución planteada.
  - Documentación de las tareas realizadas
- Julio - Setiembre
  - Prototipado/desarrollo de la arquitectura diseñada.
  - Tests no funcionales al prototipo construido.
  - Documentación de las tareas realizadas.
- Octubre - Diciembre
  - Correcciones finales en los modelos/desarrollos
  - Documentación final del proyecto

N.3.2. Cronograma Realizado

El proyecto se inició antes de lo previsto (26 de febrero del 2009) en el cronograma planteado por lo que se tuvo un margen de holgura.

El cronograma realizado tuvo una gran variación en lo que respecta a la etapa de experimentación.

- Semana 1 a 9:
  - En las primeras 9 semanas se realizó el relevamiento del estado del arte casi por completo, junto con una versión inicial de su documentación. En ese tiempo también se experimentó con distintas tecnologías y se tuvo el primer acercamiento a sistemas existentes (ver E).
- Semana 10 a 28:
  - Según el cronograma propuesto, al inicio de esta etapa, quedaban 4 semanas para la experimentación de las tecnologías utilizadas. Sin embargo este tiempo no fue suficiente, por lo que la etapa de experimentación duró 19 semanas, 15 más de lo previsto.
- Semana 29 a 37:

Figura N.4: Comparación de cronogramas: programado y realizado.

- El desarrollo del prototipo se realizó en un tiempo sustancialmente menor al previsto. Hay que tener en cuenta que medir únicamente la duración, no es un buen indicador del esfuerzo. Estas fueron las semanas que se trabajó más intensamente en todo el proyecto, constatando un máximo en la semana 31 de 138 horas (promedialmente es un poco menor a 10 horas/persona por día contando semanas de 7 días).

- Semana 38 a 50:

  - No es clara la división entre esta etapa y la anterior. Se siguieron corrigiendo aspectos en cuanto a la solución planteada (frutos de un mayor tiempo de *up-time* y *testing* más intensivo) y hubo una fuerte dedicación en la documentación.

N.3.3. Comparación

En la figura N.4 se puede ver la comparación entre el cronograma planteado y el realizado. En la cual se observa que la experimentación, entendimiento y utilización de las tecnologías utilizadas fue la etapa que más se desvió de lo esperado en la propuesta.