

Universidad Autónoma de Baja California
Facultad de Ciencias Químicas e Ingeniería



Plataforma de Software para el Desarrollo de
Aplicaciones en el Lenguaje Java en una Red
Inalámbrica de Sensores Experimental

TESIS

que para obtener el grado de
Doctor en Ciencias

Presenta:

CARLOS ANTONIO COTA APODACA

Director de Tesis

Dr. Leocundo Aguilar Noriega

Tijuana B.C.

Enero 2016

Universidad Autónoma de Baja California
FACULTAD DE CIENCIAS QUÍMICAS E INGENIERÍA
COORDINACIÓN DE POSGRADO E INVESTIGACIÓN

FOLIO No. 167

Tijuana, B. C., a 11 de enero de 2016

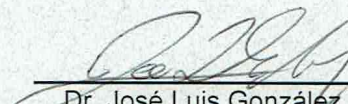
C. CARLOS ANTONIO COTA APODACA
Pasante de: Doctor en Ciencias
Presente

El tema de trabajo y/o tesis para su examen profesional, en la
Opción TESIS

Es propuesto, por el C. Dr. Leocundo Aguilar Noriega
quien será el responsable de la calidad de trabajo que usted presente, referido al
tema Plataforma de Software de Desarrollo de Aplicaciones para una Red Inalámbrica
de Sensores Experimental.

el cual deberá usted desarrollar, de acuerdo con el siguiente orden:

- I.- INTRODUCCIÓN
- II.- LAS REDES INALÁMBRICAS DE SENSORES
- III.- LA PLATAFORMA LISANDRA
- IV.- DESARROLLO DE SOFTWARE EN WSN
- V.- HERRAMIENTAS DE DESARROLLO EN LOS NODOS WSN
- VI.- LA MAQUINA VIRTUAL DE JAVA
- VII.- EL LENGUAJE JAVA PARA LAS WSN
- VIII.- PLATAFORMA DE SOFTWARE PARA EL DESARROLLO BASADO EN JAVA
- IX.- USO DE LA PLATAFORMA
- X.- CONCLUSIONES


Dr. José Luis González Vázquez
Sub-Director Secretario




Dr. Leocundo Aguilar Noriega
Director de Tesis


Dr. Luis Enrique Palafox Maestre
Director

Agradecimientos

A mis padres y hermanos, por su valioso apoyo para lograr esta meta. Su compañía siempre ha sido crucial y más en los de falta de salud.

A mi tía Fátima Estela Armenta Vega, siempre agradeceré todo lo que me ayudó y más esa entrega en sanarme.

A mis sobrinos, por ser parte de mi vida; cada uno de ustedes ha enriquecido mi vida, enseñándome la simplicidad de las cosas y recordándome el niño, que junto con mis hermanos, un día fui.

A mis maestros; la formación como profesional e investigador se las deberé siempre a ustedes. En especial a mi director de Tesis, el Doctor Leocundo Aguilar Noriega, por su confianza siempre expresada en su valioso apoyo durante mi formación, desde la licenciatura hasta la culminación de mi trabajo doctoral. Aquí no concluye mi instrucción, aún me falta mucho por desarrollar, así como aprender, por lo tanto ustedes siempre serán mis profesores.

Resumen

En esta tesis se presenta una plataforma de software de desarrollo de aplicaciones para una red inalámbrica de sensores (WSN) experimental. La idea central de esta plataforma es facilitar la programación en el procesamiento y transmisión de la información adquirida por un nodo WSN. Esta plataforma utiliza Java como lenguaje de programación, ayudado por un Framework para el desarrollo de aplicaciones y un middleware como el entorno de ejecución. El middleware implementa las clases pertinentes al nodo WSN existentes en la biblioteca estándar de Java, a fin de aprovechar las características de este lenguaje. En cuanto a la intercomunicación de los nodos WSN, se hace una abstracción haciendo uso de los sockets Berkeley, aunque la implementación está lejos de ser al nivel del de las redes de PC.

También proporciona ejemplos de la codificación en Java, muestra cómo su adopción facilita la programación del nodo WSN y cómo hacer uso de esta plataforma.

Por último, se ilustra una posible ampliación del Framework de desarrollo de aplicaciones Java de acuerdo a los cuatro ámbitos donde se puede también aplicar las redes de sensores inalámbricas.

Abstract

This thesis presents an application software development platform for an experimental wireless sensor network (WSN). The central idea of this platform is to make easy the programming of the processing and transmission of the acquired information by a WSN-node. This platform uses the Java as the programming language, helped by an application development Framework and a middleware as the run-time engine. The middleware implements the relevant classes for WSN-node, which exists in the Java standard library, and with this, we can exploit the features of this language. Regarding of the WSN-nodes intercommunication, it abstracts making use of the Berkeley Sockets, although the implementation is far from the level used in the PC networks.

Also provides examples of the coding in Java, shows how its adoption makes easy the programming of the WSN-node. And how to use this platform (the Java application development Framework and the network abstraction).

Finally, is illustrated possible extensions of the Java application development Framework according to the four scopes where can be also applied.

Índice general

	Pag.
Agradecimientos	III
Resumen	IV
Abstract	V
Índice General	VI
Lista de Figuras	XI
Capítulo	
1 Introducción	1
1.1 Motivación	4
1.2 Planteamiento del problema	5
1.3 Propuesta de Solución	6
1.4 Objetivos	7
1.4.1 Objetivo General	7
1.4.2 Objetivos particulares	7
1.5 Metodología	8
1.6 Contenido de la Tesis	8
2 Las Redes de Sensores Inalámbricas	11
2.1 Arquitectura generalizada del nodo WSN	12
2.1.1 La unidad de procesamiento	14
2.1.2 La Memoria	16
2.1.3 Dispositivo de comunicación	17
2.1.4 Sensores y Actuadores	18
2.1.5 Fuente de poder del nodo WSN	19
2.2 Ejemplos de nodos WSN	19
2.2.1 Mica Mote	19

2.2.2	EYES	20
2.2.3	BTnodes	21
2.2.4	Scatterweb	22
2.2.5	Soluciones Comerciales	22
2.3	Topologías de las redes de sensores inalámbricas.	23
2.4	Arquitecturas de las Redes Inalámbricas de Sensores.	24
2.5	Características de las Redes Inalámbricas de Sensores.	25
2.6	Aplicaciones de las WSN	26
3	La plataforma LiSANDRA	30
3.1	Arquitectura	30
3.1.1	El nodo S	31
3.1.2	El nodo SR	33
3.1.3	El nodo G	34
3.2	Software	35
4	Desarrollo de Software en WSN	36
4.1	Desarrollo basado en Middleware	38
4.1.1	Máquinas virtuales.	40
4.1.2	Programación Modular.	41
4.1.3	Bases de datos.	42
4.1.4	Lanzadera de Aplicaciones.	43
4.1.5	Middleware Orientadas a Mensajes.	43
4.2	Desarrollo basado en tecnologías de agentes.	44
4.3	Desarrollo Basado en Sistemas Operativos.	49
4.4	Desarrollo Basado en Sistemas de Servicios (SOA).	50
5	Herramientas de desarrollo en los nodo WSN	51
5.1	IDE para TinyOS/nesc.	52
5.1.1	TinyDT.	54
5.1.2	TinyOSIDE.	55

5.1.3	TOSDev.	55
5.1.4	YETI.	56
5.1.5	VIPTOS.	57
5.1.6	GRATIS.	57
5.2	IDE de Sistemas Embebidos.	57
6	La máquina virtual de Java	61
6.1	Definición	62
6.1.1	La máquina de Turing	62
6.1.2	La JVM como máquina de Turing	64
6.1.3	Estados de la JVM	66
6.2	El lenguaje de programación Java	70
6.3	La estructura del <i>classfile</i>	71
6.4	Instrucciones de la JVM	73
6.4.1	De almacenamiento	73
6.4.2	Aritméticas	73
6.4.3	Creación y manipulación de objetos	75
6.4.4	De manipulación de pila	75
6.4.5	Transferencia de control de programa	75
6.4.6	Invocación de métodos y retorno	76
7	Java en las Redes Inalámbricas de Sensores	77
7.1	Características de la programación basada en Java	79
7.1.1	Simplicidad	79
7.1.2	Programación orientada a objetos	80
7.1.3	Rigidez de tipo de dato	81
7.1.4	Manejo de Excepciones	82
7.1.5	Comprobación de limites de arreglos	83
7.1.6	Administración automática de memoria	83
7.1.7	Independencia de plataforma	84

7.1.8	Enriquecido conjunto de APIs estandarizados	85
7.1.9	Compacto	85
7.1.10	Multihilo	86
7.1.11	Seguridad	87
7.2	Costos en el uso del lenguaje Java	87
7.3	Reducción de costos en el uso del lenguaje Java	88
8	Implementación de la plataforma para desarrollo basado en Java	90
8.1	JavELiNA.	91
8.2	MiniVM.	92
8.2.1	Arquitectura de MiniVM.	93
8.2.2	Planificador Multi-hilo.	96
8.2.3	Mapeo de memoria RAM.	100
8.2.4	Manejo de Eventos.	101
8.2.5	Optimización de MiniVM.	103
8.2.6	Manejo de Excepciones.	106
8.3	Framework de aplicación.	107
8.3.1	La <i>clase</i> LLED	107
8.3.2	La <i>clase</i> LSwitch	108
8.3.3	Las <i>clases</i> LHumiditySnr , LTemperatureSnr y LLightSnr	109
8.3.4	La <i>clase</i> LTimer y LTimerTask	110
8.3.5	La <i>clase</i> LPinout	110
8.3.6	La <i>clase</i> LADC	111
8.3.7	La comunicación.	112
9	Uso de la Framework	115
9.1	Requisitos.	115
9.1.1	Configuración.	116
9.1.2	La herramienta <i>mixClass</i>	116
9.2	Código de Ejemplos.	119

9.2.1	Uso de los LEDs.	119
9.2.2	Uso de Lswtich.	120
9.2.3	Haciendo uso de la comunicación UART.	120
9.2.4	Uso de Threads, Sensores y Sockets.	121
10	Conclusiones.	123
10.0.5	Trabajo Futuro.	124
	Referencias	126

Lista de Figuras

1.1	Elementos de una WSN.	2
1.2	Diversas plataformas de nodos WSN.	4
2.1	Diagrama de bloques del Nodo-WSN	13
2.2	Plataforma Mica.	20
2.3	Plataforma eyesIFX.	21
2.4	Plataforma BNode.	21
2.5	Plataforma ScatterWeb.	22
2.6	Topologías usadas en las WSN.	23
2.7	Grupos de aplicaciones de las WSN.	27
3.1	Nodo S.	32
3.2	Diagrama de Flujo para parametrizar el funcionamiento del nodo S.	32
3.3	Nodo SR.	33
3.4	Nodo G.	34
4.1	Ciclo de vida de las WSN.	37
4.2	Ciclo de diseño de Software para nodos Sensores.	38
4.3	Componentes comunes de las Middleware para WSN.	39
4.4	Tipos de Middleware en base a su modelo de programación	40
4.5	Elementos que integran un nodo WSN.	45
4.6	Modelo de agente estático en WSN.	47
4.7	Modelo de agente móvil en WSN.	48
5.1	Proceso automatizado de compilación de aplicaciones.	53
5.2	Editando la aplicación Blink en TinyDT.	54

5.3	Entorno de desarrollo TOSDev.	55
5.4	Entorno YETI. 1) Navegador con el listado de todos los archivos del proyecto. 2) Ventana principal mostrando el editor con el archivo abierto. 3) Esquema del archivo abierto mostrando su estructura. 4) Opciones del automatizado Make. 5) Panel multipropósito.	56
5.5	Plataforma de desarrollo AVR Studio.	58
5.6	Plataforma de desarrollo MSP430 IAR.	59
6.1	Micro-Estados del bytecode <i>iadd</i>	67
6.2	Micro-Estados del bytecode <i>if_icmpge</i>	69
8.1	Ciclo <i>For</i> expresado en el código fuente de Java y su correspondiente <i>bytecode</i>	93
8.2	Arquitectura de MiniVM.	94
8.3	Equivalencia del <i>bytecode</i> de Java.	96
8.4	Bloque de contexto del Thread.	97
8.5	Cambio de contexto entre Threads.	98
8.6	Diagrama de transición de estados.	99
8.7	Mapeo de la memoria RAM.	101
8.8	Mecanismo para el manejo de interrupciones por el módulo de Interpretación: (a) diagrama de flujo para el registro de eventos, (b) diagrama de flujo para el manejo de la ocurrencia de un evento.	102
8.9	Medición de ciclos de UCP.	104
8.10	Módulo de carga de <i>classfile</i> optimizando la referencia a los métodos.	105
8.11	Clase LLED	108
8.12	Clase Lswitch	109
8.13	El paquete Sensors	109
8.14	Las <i>clases</i> LTimer y LTimerTask para el manejo de Temporizadores.	110
8.15	La <i>clase</i> LPinout	111
8.16	La <i>clase</i> LADC	112

8.17	La <i>clase</i> LUSART	113
8.18	Las <i>clases</i> ServerSocket y Socket	114
9.1	Proceso de desarrollo de aplicaciones usando Java y el Framework.	117
9.2	Proceso de serialización del <i>classfile</i>	118
9.3	Aplicacion SimpleLEDRed.java	119
9.4	Implementación de la <i>clase</i> Lswitch	120
9.5	EchoUart para ejemplificar el uso de la comunicación UART.	121
9.6	Aplicación que muestra el uso de <i>sockets</i> , <i>threads</i> y <i>sensors</i>	122

Capítulo 1

Introducción

El surgimiento de las redes inalámbricas de sensores (WSN por su siglas en inglés) es uno de los más recientes desarrollos en los que convergen el área de la micro electrónica y comunicaciones. Estas redes están compuestas por varias decenas, y en algunos casos, centenas de nodos capaces de obtener información del medio en el que se encuentra o actuar en función de la información adquirida (WSAN) comunicándola de forma inalámbrica entre ellos o a un nodo colector. Además están integrados al espacio físico como el que podemos encontrar en fábricas, transporte, casas entre otras.

Este tipo de redes son una nueva clase de sistemas distribuidos, pero con una función totalmente distinta a la de la mayoría. En vez de trabajar con datos capturados por el usuario, las WSN trabajan a partir de datos adquiridos por cada uno de los elementos que la integran. Permitiendo establecer una conexión de los fenómenos del mundo físico con los sistemas de información existentes, razón por la cual ha capturado la atención e imaginación de investigadores, generando un gran abanico de ideas, desde la protección ambiental hasta las aplicaciones militares (donde vio su nacimiento).

La infraestructura de las plataformas para las redes inalámbricas de sensores comprende sensores, procesadores y elementos de comunicación inalámbrica que dan al administrador la capacidad de instrumentar, observar y reaccionar a eventos y fenómenos en un ambiente específico. Dentro de esta existen cuatro componentes básicos (Ver Figura 1.1):

- *Nodo sensor* (1), compuesto por un procesador, un grupo de sensores y un transmisor inalámbrico.
- *Comunicación inalámbrica* (2), generalmente haciendo uso de un canal libre y protocolos

no estandarizados.

- *Puerta de enlace* (3), para la interconexión de la WSN a una red de tipo estandarizada como WiFi.
- *Estación de monitoreo y administración* (4), una terminal con elementos que permiten controlar, configurar y administrar la red.

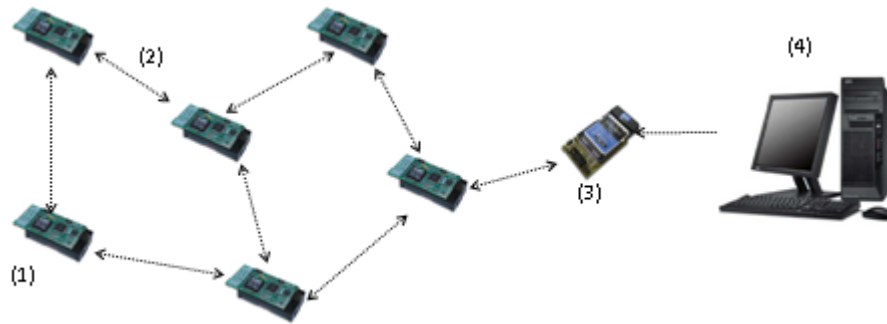


Figura 1.1 Elementos de una WSN.

Existe una variada configuración de procesadores y radio transmisores interconectados, creando una extensa diversidad de plataformas de redes de sensores inalámbricos tales como: Mica [1], telosb [2], Tmote Sky [3], Tmote Invent [4], Sentilla [5], TinyNode [6], entre otras. Debido a esto aún no ha sido posible establecer estándares que permita su desarrollo acelerado. Aun así hay organizaciones internacionales como la IEEE, la Internet Engineering Task Force e International Society of Automation que proveen de referencias completas para soluciones verticales cubriendo desde una hasta todas las capas con sus respectivos protocolos, sin descartar que también hay varios mecanismos propietarios y especificaciones no estándares [7]. Algunos estándares predominantes usados en las WSN son: WirelessHart, ISA100, IEEE 1451 y ZigBee [8].

Así mismo, las plataformas para las WSN están diseñadas alrededor de microcontroladores como el TI MSP430 [9] (usado en telosb y Tmote Sky entre otras) o de Atmel ATmega 128L

[10] (en el que se basan las plataformas Mica2, BTNode, etc.). Estos procesadores están diseñados para el bajo consumo de energía y para una amplia gama de aplicaciones empotradas, pero fundamentalmente basadas en motores de cómputo de propósito general, monolíticos y que no se adaptan muy bien al manejo de eventos. Esto es un impedimento para el tipo de carga de trabajo que existe en las redes de sensores inalámbricos.

Este modo de trabajo implica que mientras los procesadores están en el modo de bajo consumo de energía, se deshabilita el procesador entero y es reactivado en la próxima ocurrencia de una interrupción. De esta forma se limita su uso a un patrón dominado por interrupciones en vez de aplicaciones basadas en eventos. Además, la naturaleza monolítica de esta arquitectura también excluye la granularidad del ciclo de trabajo para cada uno de los componentes del procesador. A pesar de esto, uno de los mayores retos en las WSN es el ser capaces de producir arquitecturas de nodos sensores pequeños y de muy bajo costo.

Hoy en día, la situación comercial con respecto al hardware se ve en el incremento del número de pequeñas compañías produciendo plataformas, haciéndolo comparable con la revolución informática de las computadoras personales en los 1970s. Sin embargo, muchos de estos nodos aún están en experimentación y desarrollo (principalmente en lo que respecta a su software) [11].

Por último, estas plataformas al igual que una computadora, requieren de ser programadas para realizar la tarea en la que están embebidas. Las personas implicadas en la realización de esta tarea deben de disponer de conocimientos en el área de electrónica, desarrollo de interfaces para microcontroladores, circuitos digitales, arquitectura de software, lenguajes de programación, entre otros. Esta es una limitante que impide a personas que pudiesen hacer uso de esta tecnología, como medio auxiliar, para la realización de estudios científicos o relacionados con la productividad (generalmente hablamos de individuos especializados en áreas muy distintas de las de electrónica o computación) puedan desarrollar sus propias aplicaciones de software para aprovechar esta tecnología.

El presente trabajo, es una propuesta para la adopción del lenguaje Java como lenguaje de programación para el nodo en las Redes Inalámbricas de Sensores (WSN por sus siglas

en inglés), auxiliado por un conjunto de clases para el manejo de los recursos existentes en la plataforma; logrando con esto que el programador tenga una abstracción completa de la complejidad que pudiese envolver el conocimiento del hardware.

La idea principal es permitir solo el enfocarse en el desarrollo de la aplicación, desde una perspectiva de software realizando una(s) tarea(s). Esta propuesta consta también de una middleware como una capa de firmware sobre la que se soporta la implementación de la aplicación (desarrollada en el lenguaje Java) y la operación básica del nodo dentro de la red inalámbrica de sensores.

1.1 Motivación

Las redes de sensores inalámbricas están compuestas por decenas o cientos de dispositivos pequeños autónomos y distribuidos geográficamente, llamados *nodos sensores*. Estos nodos (Ver Figura 1.2):

- Integran sensores para realizar mediciones.
- Están restringidos en energía, capacidad de cómputo y memoria.
- Hacen uso intensivo de la radio con enlaces de reducido ancho de banda (para comunicación) y CPU (para procesamiento).

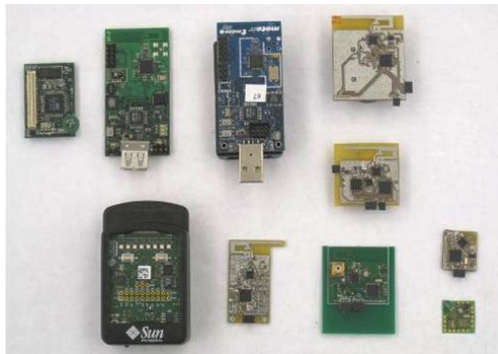


Figura 1.2 Diversas plataformas de nodos WSN.

Además, están incrustados en entornos donde deben de ser programados y para alcanzar el éxito comercial en su uso se depende de la disponibilidad de herramientas y metodologías que existan para el desarrollo de software sumado a la abstracción de los detalles de bajo nivel tanto en el despliegue como la operatividad de la red. Para esto, existen opciones que requieren desde conocimientos específicos con respecto al hardware (como lo sería la interconexión existente entre puertos y unidad de procesamiento o comunicación a nivel hardware entre procesador y unidad de comunicación RF), hasta la completa adopción de un sistema operativo embebido, adoptando mecanismos nada flexibles para la programación y ejecución de la aplicación, que de manera idéntica, requiere de un aprendizaje sobre el mismo.

Para ambas situaciones se describe la necesidad de conocer la plataforma (ya sea en hardware o Sistema operativo) de manera previa antes de poder aplicar esta tecnología en algún campo deseado.

Esta situación no es conflicto para un profesional en el área eléctrica-electrónica o computacional. Sin embargo, la gran cantidad de aplicaciones a ser realizadas para las WSN deberán ser programadas por profesionistas ajenos a las áreas descritas anteriormente. Como pudiera ser un biólogo realizando monitoreo del flujo del ciclo del carbono en un habitat. En este escenario, para el biólogo, son conocidos los aspectos que envuelve la interacción entre los organismos y el medio ambiente, pero la adopción de una WSN para el monitoreo, implicaría problemas al tener que aprender a usar estas plataformas desde los aspectos de hardware hasta el del sistema operativo adoptado. También la comunicación entre nodos es realizada por protocolos no estandarizados y poco documentados que implican, de acuerdo a la plataforma adoptada, configurarse para el envío de datos a través de la WSN.

1.2 Planteamiento del problema

El desarrollo de aplicaciones para las WSN requiere de un amplio conocimiento en arquitecturas de hardware o sistemas operativos embebidos. Esto desalienta en gran medida la

popularización de las WSN en áreas que no involucren a un profesional en eléctrica-electrónica. Debido a que para la programación de estas plataformas, se requiere del aprendizaje de lenguajes de programación muy distintos a los utilizados de manera común para PC; y en caso de existir, la ausencia de librerías que auxilien o den soporte en el desarrollo de software complica la programación de los mismos, utilizando tiempo del desarrollo de la aplicación en la programación de estas.

Por otra parte, dada la ausencia de mecanismos de control para la aplicación, esta tiene acceso directo a los recursos existentes en el hardware y en caso de realizar alguna operación incorrecta hace inestable el funcionamiento de la plataforma WSN.

Por último, para la programación de la intercomunicación de los nodos WSN, se requiere conocer los mecanismos y protocolos manejados por la plataforma. Esto complica todavía más la programación, cuando se requiere de comunicar datos entre los distintos nodos.

1.3 Propuesta de Solución

La *programación* del nodo WSN es inevitable, ante esto, es conveniente la adopción de un lenguaje de programación que tenga una sintaxis “más natural” haciendo de su curva de aprendizaje más corta. De esta forma el *integrador* (persona que implementa la WSN) requiere de poco tiempo para adaptarse a él y enfoca su esfuerzo en el desarrollo de la aplicación. Para esto, se **propone** el lenguaje de programación Java, idóneo por su facilidad de aprendizaje.

En cuanto al *correcto cumplimiento de los requerimientos y rapidez* en la programación del nodo, es **propuesto** un Framework de aplicación donde se tienen las clases que manejan los recursos de hardware existentes en la plataforma de sensores. De forma tal que el integrador solo ocupa hacer uso de sus métodos y propiedades para acceder a los recursos que desee en la aplicación. Con esto puede abstraerse sobre la complejidad del hardware y software involucrado, enfocándose exclusivamente en la lógica de la aplicación.

Por otro lado, para la nula seguridad existente al estarse ejecutando la aplicación directa-

mente en el hardware y los riesgos que esto conlleva. Es también **propuesta** una middleware, como capa intermedia entre la aplicación y el hardware existente. Siendo esta la encargada de administrar los recursos de hardware y el (los) proceso(s) existente(s) en ejecución. Además de verificar y ejecutar el bytecode de Java, garantizando la correcta operación del nodo WSN.

Por último, para la abstracción de la comunicación entre los nodos de la WSN, se **propone** hacer uso de BSD-Socket donde se percibe a la red, desde la perspectiva de programación, como flujos de datos que se abren y cierran entre dos nodos. Es claro que en los mecanismos mas bajos es adoptado un protocolo, pero que el *integrador* no requiere conocer pues desde la perspectiva de los sockets se observa a la red como conexiones que se crean entre los nodos para transmitir datos entre ellos.

1.4 Objetivos

1.4.1 Objetivo General

- El modelado de un conjunto de herramientas para la programación, depuración y producción de aplicaciones basadas en Java (como lenguaje de programación), para dispositivos empotrados; bajo el marco de trabajo de una red hibrida de sensores inalámbricos y sus restricciones generales (comunicación inalámbrica, energía limitada, espacio de memoria reducido, entre otras).

1.4.2 Objetivos particulares

- Modelado de instrumentos para programación de aplicaciones basadas en Java para las plataformas Nodo-SR.
- Modelo de una middleware para la abstracción y ejecución de la aplicación desarrollada en el lenguaje Java.

- Modelo de métodos nativos para facilitar el desarrollo de aplicaciones en el Nodo-SR.
- Proponer metodologías de desarrollo de aplicaciones de computacion ubicua aplicadas en el aula universitaria.

1.5 Metodología

La programación de las plataformas inalámbricas de sensores requieren una gran experiencia en el desarrollo de sistemas embebidos. Para poder lograr que su programación sea sencilla es importante simplificar la programación de los mismos. Esto implica un análisis del hardware involucrado en la plataforma, el tipo de aplicaciones que se realizan en ellas, agruparlos como servicios disponibles dentro de una capa de software intermedia y un entorno de ejecución seguro para la aplicación.

Un mecanismo ideal para lograr estos objetivos es el desarrollo de una máquina virtual de Java. Para esto, se parte del estudio de su composición y adaptación a la plataforma, concluyendo en la implementación y desarrollo de una Framework de Desarrollo que agilice la programación de aplicaciones en las WSN.

1.6 Contenido de la Tesis

El contenido de este trabajo se encuentra conformado por 9 capítulos adicionales a este capítulo (capítulo 1) y que se explican brevemente a continuación.

- **Capítulo 2:** Se analiza la arquitectura genérica que conforma la plataforma para redes inalámbricas de sensores, identificando cada uno de los componentes en hardware y software. Estos componentes son: la unidad de procesamiento, la memoria, el módulo de comunicación, sensores, actuadores y fuente de poder. También estos componentes son ejemplificados mencionando los más populares en su uso. Además de su composición, se enlistan algunas de las plataformas disponibles en el mercado. Por último, algunas

generalidades sobre las topologías, arquitecturas, características y aplicaciones de las WSN.

- **Capítulo 3:** Se describe la plataforma LiSANDRA con sus diversos elementos: el *nodo S*, el *nodo SR* y el *nodo G* formando una arquitectura específica con sus diversos componentes, haciendo especial énfasis en el nodo SR ya que es en este donde se implementa la plataforma de Desarrollo. Se describe también el software existente dentro de la plataforma LiSANDRA.
- **Capítulo 4:** Se enlistan los distintos paradigmas de desarrollo de aplicaciones para redes inalámbricas de sensores. Estos paradigmas son: **basado en middleware**, como la programación modular, base de datos, lanzadera de aplicaciones y orientadas a mensajes; **basado en agentes**, **sistemas operativos** y **basado en servicios**. En cada uno de estos paradigmas se ejemplifica con tecnologías implementadas y se hace un análisis de cada una de estas perspectivas en el ámbito de uso y desarrollo de software para las redes inalámbricas de sensores.
- **Capítulo 5:** Se hace referencia a las distintas herramientas de software existentes para la programación del nodo de la red inalámbrica de sensores. La mayoría de estas herramientas están sobre el ambiente del sistema operativo *TinyOS*; a excepción de las utilizadas para la programación de microcontroladores (se enlistan las de la familia MSP430 de Texas Instruments y ATmega de Atmel).
- **Capítulo 6:** Se hace un análisis de la composición de la máquina virtual de Java como una máquina de Turing, la complejidad existente en los estados internos de la misma a fin de comprender su implementación como middleware en el nodo-SR. Por otro lado se hace un análisis del lenguaje de programación Java, la estructura del archivo *classfile* y el conjunto de instrucciones que forman parte de la *JVM*.
- **Capítulo 7:** Se hace un análisis de las características idóneas del uso del lenguaje de programación Java en las redes inalámbricas de sensores. Se argumenta la razón por la

cuál cada una de las características listadas de Java son benéficas dentro del ámbito de las redes inalámbricas de sensores. Por último se contrasta el costo del uso del lenguaje de programación Java y como se puede reducir estos “costos”.

- **Capítulo 8:** Se describe a detalle la implementación de la plataforma de software para el desarrollo de aplicaciones basada en Java para la red inalámbrica de sensores experimental (llamada JavELiNA) y cada uno de los elementos que la componen como: la middleware (llamada MiniVM), el Framework de Aplicación y la comunicación basada en Berkeley Sockets.
- **Capítulo 9:** Se describen los requisitos, la configuración del equipo de desarrollo y se ejemplifica el uso de la Framework de aplicación con aplicaciones demostrativas.
- **Capítulo 10:** Se detallan las conclusiones del trabajo presentado, así como el trabajo futuro a ser realizado con esta plataforma de software para el desarrollo de aplicaciones en el lenguaje Java en una red de sensores experimental.

Capítulo 2

Las Redes de Sensores Inalámbricas

Las redes inalámbricas de sensores, WSN por sus siglas en inglés, enlazan los fenómenos físicos con el mundo digital, capturando los fenómenos del mundo real y convirtiéndolos en datos que pueden ser almacenados y procesados; al mismo tiempo pueden ejercer acción, si tiene actuadores, sobre el entorno en el que se encuentra. Estas redes están integradas por numeroso dispositivos llamados *nodos sensores*, embebidos en un ambiente donde pueden ayudar a prevenir fallas catastróficas en estructuras, conservar recursos naturales, incrementar la productividad, aumentar la seguridad. Además de que permiten nuevas aplicaciones tecnológicas, como aplicaciones conscientes del contexto o casas inteligentes.

Este nuevo tipo de redes han surgido como consecuencia del avance en la tecnología de integración a gran escala (VLSI), sistemas microelectromecánicos y comunicaciones inalámbricas. Para ilustrar esto, el impresionante avance en materia de semiconductores nos ha llevado a producir microprocesadores con alta capacidad de procesamiento al mismo tiempo que su tamaño se reduce. Esta reducción en los microprocesadores y sensores habilitan el desarrollo de plataformas pequeñas, de bajo consumo, sensores, actuadores y controladores baratos [12].

También, para el área de informática, estas redes inalámbricas de sensores son una nueva clase de sistemas distribuidos, pero con una función totalmente distinta a la de la mayoría. La forma más común de procesar la información ha sido en dispositivos computacionales de propósito general, que van desde las mainframes a laptops modernas o computadoras de mano como teléfonos inteligentes. Dado que en la mayoría de sus aplicaciones, pueden ser las aplicaciones de oficina, estos dispositivos computacionales son mayormente usados para procesar la información centrada en el ser humano que hace uso del sistema pero para nada

relacionados con el ambiente físico. Mientras que para las WSN, el entorno físico es el foco de atención. Es decir, en vez de trabajar con datos capturados por el usuario y procesados en dispositivos de computo de propósito general, las WSN trabajan a partir de datos adquiridos por cada uno de los elementos que las integran. En otras palabras, las WSN se integran dentro del sistema físico y no requieren la interacción humana para trabajar.

Dado que estos sistemas integrados son un concepto conocido y utilizado durante mucho tiempo en las ciencias de la ingeniería (las estimaciones dicen que hasta el 98% de los procesadores se utilizan en un contexto incrustado [13]) su impacto en la vida diaria ha ido creciendo rápidamente, de ahí que las redes inalámbricas de sensores son parte de ese crecimiento acelerado y cada día se pueden apreciar más como parte de nuestro entorno como parte de la visión de *ambiente inteligente*. Donde varios dispositivos diferentes estarán obteniendo y procesando información de diferentes fuentes para controlar los procesos físicos e interactuando con los seres humanos.

Actualmente, las WSN, son consideradas una de las más importantes tecnologías de este siglo. Se distinguen de las comunicaciones inalámbricas tradicionales como lo serían los sistemas celulares y redes Ad Hoc (MANET) por tener características únicas, por ejemplo, un nivel denso de despliegue, poca confiabilidad de los nodos sensores, restricciones de energía, computación y restricciones de almacenamiento [14] [15]. Además, las WSN son redes poderosas y susceptibles a soportar una gran cantidad de diferentes aplicaciones del mundo real; son también un reto en el área de la investigación y solución de problemas en ingeniería por la increíble flexibilidad existente en ellas debido principalmente a que no existe un único conjunto de requisitos que clasifique claramente a todas las WSN y por ello no hay una única solución que abarque por completo el campo de su diseño.

2.1 Arquitectura generalizada del nodo WSN

La composición de una red inalámbrica de sensores está constituida de forma primaria por nodos, llamados *nodos sensores*. Estos tienen la solución a los requisitos que forman

parte de los requerimientos dados por la aplicación de la red. Generalmente deben de ser pequeños, baratos, eficientes en el uso de energía, equipados con los sensores correctos y los recursos suficientes de memoria y procesamiento obviamente también con la adecuada facilidades de comunicación.

Un nodo sensor básicamente comprende 5 componentes principales (ver Figura 2.1):

- **Unidad de procesamiento.** Un controlador para procesar todos los datos relevantes y la ejecución del código de programa.
- **Memoria.** Tanto para el almacenamiento de programas como los datos intermedios.
- **Sensores y actuadores.** Para la interfaz con el mundo físico.
- **Transreceptor.** Para el envío y recepción de información sobre un canal inalámbrico.
- **Fuente de poder.** Como usualmente una conexión a línea física de poder no esta disponible, se requiere una fuente de poder que típicamente es dada por baterías o del ambiente por medio de celdas solares.

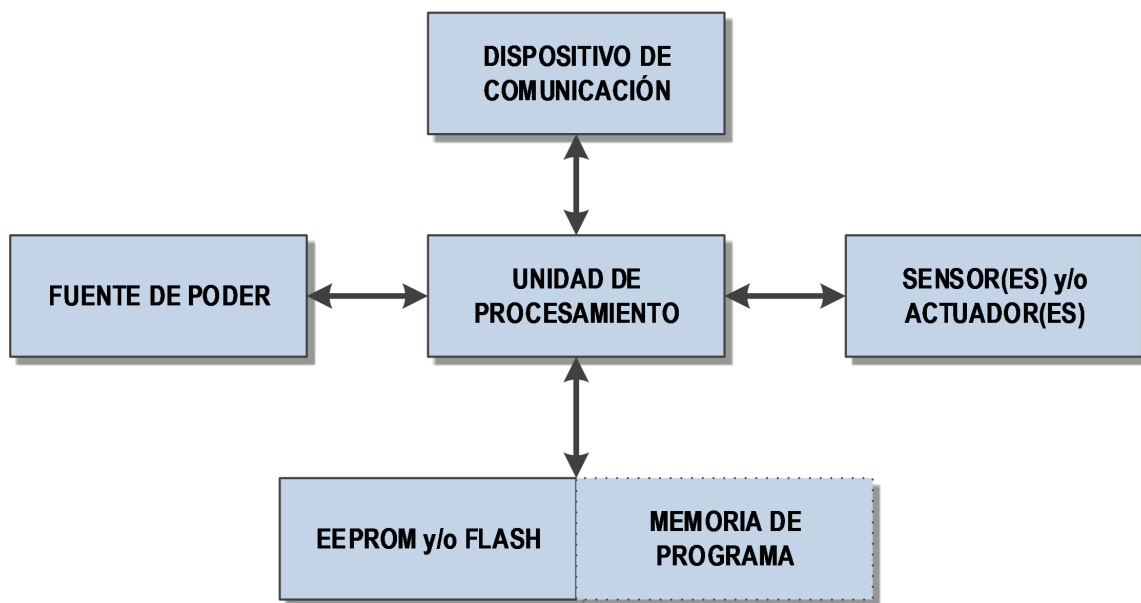


Figura 2.1 Diagrama de bloques del Nodo-WSN

Cada uno de estos componentes tienen que operar de manera balanceada entre el consumo mínimo necesario y el cumplimiento completo de las tareas. Como ejemplo, tanto el transreceptor como el microcontrolador deben de ser apagados tanto como sea posible y entrar en modo de operación cuando una interrupción dada por algún sensor o evento requieran la operación de estos recursos, inclusive cuando se detecta la recepción de alguna transmisión [16].

2.1.1 La unidad de procesamiento

La unidad de procesamiento es el centro básico de un nodo WSN. Este colecta datos de los sensores, los procesa y decide si los envía. Además de recibir datos de otros sensores o realizar alguna actividad a través de los actuadores. Por otra parte, esta ejecutando varios programas que pueden ser de procesamiento crítico, protocolo de comunicación o programas de aplicación. En otras palabras, es la unidad central de todo procesamiento a realizar dentro del nodo WSN.

Dada la gran variedad de tareas a ser procesadas se puede elegir de una diversidad de arquitecturas de procesadores que se evalúan en los valores de flexibilidad, desempeño, eficiencia, gasto de energía y costo. Una solución común es usar procesadores de propósito general, idénticos a los usados en los equipos de escritorio. Estos suelen tener un consumo excesivo de energía en contra parte, existen otros especialmente diseñados para los sistemas embebidos llamados *microcontroladores* cuyas características son loables; además de ser hechos para desarrollo de sistemas embebidos, es la flexibilidad de conectarlos con otros dispositivos (que pudiesen ser sensores), su conjunto de instrucciones son manejables en procesamiento crítico de señales y su bajo consumo de poder; esto hace de ellos los candidatos idóneos para este tipo de plataformas ya que también incluyen internamente memoria para programas. Una característica muy importante de los microcontroladores, sobre todo en los de última generación, es la capacidad de poder reducir su modo de consumo significativamente apagando varios de los periféricos que tiene y entrando en *modo de hibernación*; claro que eso depende directamente del microcontrolador que se este usando.

Existe también el caso de *procesadores digitales de señales (DSPs)* y *FPGAs*, pero en el caso de los primeros, fueron desarrollados específicamente con el objetivo de procesar grandes arreglos vectoriales de datos. Tal como es el caso de aplicaciones de procesamiento de señales, dentro de las WSN pueden ser usados para extraer los datos digitales de alguna comunicación y es ahí donde son usadas de forma exitosa, pero para el procesamiento dentro del WSN (en las tareas), los requisitos de procesamiento suelen ser mucho más modestos. Por otra parte, los FPGAs pueden ser reprogramados (o reconfigurados) para adaptarse a los cambios en los requisitos; sin embargo, esto puede tomar tiempo y energía (no es muy práctico hacerlo con la misma frecuencia con la que se haría en un microcontrolador). En resumen, las opciones finales solo serían viables para aplicaciones donde el trabajo de los nodos sensores no sean cambiantes, pero hasta el momento se busca que la tecnología de las WSN sea flexible en cuanto a la ejecución de requerimientos [17].

Asumiendo que la tendencia en el uso de microcontroladores como la unidad central de procesamiento se conserve, los más usados son:

- **Intel StrongARM** [18]. Es para las WSN, un procesador de alta gama ya que este principalmente es aplicado en dispositivos de manos como *Personal Digital Assistant* (PDA) u otro tipo de computadoras de bolsillo. Esta familia es una arquitectura de 32-bit tipo RISC con velocidad de reloj por arriba de los 206 MHz.
- **Texas Instruments MSP 430** [9]. Texas Instruments tiene una familia completa de microcontroladores bajo la firma MSP430. A manera distinta del StrongARM, este está desarrollado para aplicaciones embebidas. Su arquitectura de 16-bit tipo RISC tiene baja velocidad de reloj (arriba de los 4 MHz) pero con una gran posibilidad de interconectar diversos dispositivos a él y un conjunto de instrucciones sencillas. Sus características varían en la cantidad de RAM (de 2 a 10 KB), cantidad de convertidores analógicos digitales (de 12-bit) y el manejo de reloj para tiempo real, pero sus modos de operación consumen algunos mili amperios. En gran medida, este microcontrolador es suficiente para las tareas típicas de computación de los nodos WSN (con la excepción

de la comunicación vía radio).

- **Atmel ATmega** [10]. Es un microcontrolador de 8-bit también creado para su uso en aplicaciones embebidas y equipado con interfaces externas relevantes para periféricos comunes. Sus características varían de acuerdo al derivado de la familia que se elija que van desde 1 KB a 128 KB memoria FLASH y de 32 bytes a 4 KB RAM. También su modo de operación es de bajo consumo (menos de 10 mili amperios).
- **Microchip PIC** [19]. Es un microcontrolador (de 8-bit o 16-bit) muy usado en los sistemas embebidos, por lo mismo contienen elementos, como lo son CAN y RF, usados en diversas aplicaciones donde se está empezando a utilizar las WSN. Existe mucha variedad de ellos y su consumo de energía es menor a los 8 mili amperios.

2.1.2 La Memoria

La memoria es el componente más sencillo. Evidentemente, están las del tipo de acceso aleatorio (RAM) usadas idóneamente para el almacenamiento intermedio de la lectura dada por los sensores, procesos en ejecución o paquetes siendo recibidos de otros nodos. Aunque este tipo de memoria es muy rápida en su lectura/escritura no sirve para el almacenamiento de información persistente ya que su contenido se pierde cuando se desconecta de la fuente de energía. Es por esto que las instrucciones de programa deben ser almacenadas en alguna memoria con mayor permanencia como lo son las de tipo ROM (por sus siglas en inglés *Read Memory Only*). Actualmente se ha popularizado el uso de memorias del tipo EEPROM (de las siglas en inglés *Electrically Erasable Programmable Read-Only Memory*) o FLASH (similar a las EEPROM pero que también puede guardar o borrar datos en cualquier momento). Determinar de forma correcta el tamaño de la memoria puede ser crucial con respecto a los costos y el consumo total del nodo WSN, esto depende directamente de los requisitos dados por la aplicación de la WSN.

2.1.3 Dispositivo de comunicación

El dispositivo de comunicación es usado para intercambiar datos entre nodos individuales. En algunos casos, la comunicación vía cable es el método elegido y frecuentemente aplicado en varias configuraciones de redes de sensores (usando comunicación del tipo BUS como PROFIBUS [20], LON [21], CAN [22] u otros). Aunque estas redes no son del tipo WSN, son un excelente antecedente de las mismas ya que son personalizadas a partir de componentes ya estandarizados y listos para usarse (*off-the-shelf*).

Para el caso de las comunicaciones inalámbricas (que dan forma a las WSN), las cosas son más interesantes. La primera elección a realizar es el medio de transmisión, que actualmente comprende desde radio frecuencias, comunicaciones ópticas, ultrasonido hasta inductancia magnética. De estas posibles opciones, la radio frecuencia es la más relevante y la que mejor se adapta a la mayoría de los requisitos de las aplicaciones WSN; este provee un largo rango de alcance y alta tasa de transferencia de datos, además de no requerir una línea de visión directa y menor cantidad de errores en transmisión. Es por eso que se atiende la comunicación basada en RF y el transceptor como el dispositivo de comunicación principal en esta sección.

Para una red inalámbrica práctica, el sistema de RF tiene que ser elegida cuidadosamente, pero por el momento sólo es necesario saber que las WSN típicamente usan las bandas de frecuencia de 433 MHz a 2.4 GHz.

En la comunicación actual en las WSN es necesario que el nodo WSN tenga tanto el transmisor como el receptor. Cuya principal tarea es convertir un bit o una secuencia de bits enviados por el microcontrolador en ondas de radio. Para propósitos prácticos, este es usualmente llamado *transceptor*. Usualmente, una comunicación semi-duplex es realizada al momento de transmitir o recibir, lo cual resulta impracticable en la mayoría de los casos.

Existen una gran variedad de transceptores de bajo costo disponibles en el mercado y el costo de decisión está basado de acuerdo a las tareas que realizará y las características que posee. Algunas de estas características son:

- **Servicios a las capas superiores.**

- Consumo de energía.
- Frecuencia portadora y canalización.
- Múltiples cambios de estado para el consumo de energía.
- Taza de transmisión.
- Modulación.
- Codificación.

Usualmente hay una familia entera de transceptores que manejan distintas restricciones regulatorias, de Europa y Norte America, en la frecuencia portadora. Actualmente la serie de productos mas populares son: el RFM TR 1001 [23], Chipcon CC 1000 y CC2400 (los primeros en cumplir con el estándar IEEE 802.15.4) [24] y el Infineon TDA525x [25] por nombrar algunos.

2.1.4 Sensores y Actuadores

Existe una vasta variedad de sensores y actuadores que crean una gran diversidad de aplicaciones y solo es posible dimensionarla revisando cuales son los que pueden ser usados en las WSN.

Los sensores pueden ser organizados en tres categorías [26]:

- Sensores omnidireccionales pasivos.
- Sensores de haz estrecho.
- Sensores Activos.

En la práctica, estos sensores estan dispersos en distintas formas y peculiaridades individuales. Obviamente incluye presición, dependencia, consumo de energía, costo y tamaño. Una asunción hecha ocasionalmente en la literatura [27], [28] es que cada nodo WSN tiene una área de cobertura para la cual tiene un grado de confiabilidad y exactitud su observación.

Por otra parte, los actuadores son casi tan diversos como lo son los sensores, sin embargo al efecto en el diseño de una WSN son más simples. Por principio, todos los nodos sensores pueden encender o apagar un relevador o asignar un valor de alguna forma. No hay mucho que decir respecto a ellos, tan solo que es una buena práctica en la mayoría de las aplicaciones de los sistemas embebidos parrear un actuador con un sensor controlador siguiendo el principio de: "nunca confíes en un actuador" [29].

2.1.5 Fuente de poder del nodo WSN

Para los nodos WSN, la fuente de poder es un componente crucial del sistema. Este tiene esencialmente dos aspectos: el primero, almacenar energía y proveerla cuando se requiera; segundo, intentar recuperar la energía perdida de alguna fuente de alimentación externa.

El almacenaje de energía convencionalmente es haciendo uso de baterías. A modo de orientación, una batería normal AA almacena entre 2.2-2,5 Ah con 1.5 V. El diseño de las baterías es una ciencia e industria en sí misma y las técnicas para la recuperación de energía ha llamado la atención a los investigadores. Para revisar este punto quizás es más indicado hacer una búsqueda dentro de las diversas referencias [30], [31], [32] y [33].

2.2 Ejemplos de nodos WSN

Existen un número muy variado de nodos WSN disponibles para su uso dentro de las redes de sensores inalámbricos ya sea con fines de investigación o implementación; dependiendo del escenario de aplicación y los diferentes requisitos con los que tiene que cumplir [34].

2.2.1 Mica Mote

A finales de los 90s, una familia entera de nodos fueron desarrollados para proyectos de investigación por la Universidad de California en Berkeley, con la colaboración parcial de Intel. Esta familia es conocida como *motes Mica* y esta poblada de varias versiones (Mica,

Mica2, Mica2Dot) [1], ver Figura 2.2. Esta plataforma esta disponible de forma comercial via la compañía Crossbow. El sistema operativo usado comunmente es el TinyOS.

Todas las plataformas estan soportadas por un microcontrolador de la familia Atmel, un sistema de radio simple (usualmente el TR 1000) y varios conectores hacia el exterior. Permite agregarle otras tarjetas de sensores como pudiesen ser de biometria o humedad.



Figura 2.2 Plataforma Mica.

2.2.2 EYES

Estos nodos fueron desarrollados por la compañía Infineon para el contexto de la Unión Europea, bajo el auspicio del proyecto *Redes de Sensores con eficiencia de energía* o *EYES* por sus siglas en inglés (Figura 2.3). Este está equipado con el microcontrolador MSP 430 de la compañía Texas Instruments, un radio modem de Infineon TDA 5250 y con una interfaz USB para conectarse a la PC.



Figura 2.3 Plataforma eyesIFX.

2.2.3 BTnodes

Los BTnode [123] fueron desarrollados por el ETH Zürich como parte de varios proyectos de investigación (Figura 2.4). Está basado en el microcontrolador Atmel ATmega 1281, 64 + 180kB RAM y 128 kB de memoria FLASH. A diferencia de la mayoría de los otros nodos (con excepción de los desarrollados por Intel) usa la comunicación via Bluetooth.



Figura 2.4 Plataforma BTnode.

2.2.4 Scatterweb

Esta plataforma, ver Figura 2.5, fue desarrollada por el grupo Computer Systems & Telematics de la universidad Freie Berlin. Esta es una familia entera de nodos, que van desde nodos estandarizados (por estar basados en el microcontrolador MSP 430) hasta los servidores Web embebidos, los cuales vienen equipados con un amplio rango de posibilidades de interconexión.



Figura 2.5 Plataforma ScatterWeb.

2.2.5 Soluciones Comerciales

Aparte de las soluciones presentadas en las secciones anteriores (que son de indole academico) ya existe algunos nodos totalmente comerciales que incluyen su propia caja de empaque, certificación entre otras cosas. Por mencionar algunas, ember (www.ember.com) o Millenial (www.millenial.net); pero el mercado es más dinámico de lo que se puede escribir acerca de ellas haciendo probable el surgimiento de más productos comerciales y disponibles a corto plazo.

2.3 Topologías de las redes de sensores inalámbricas.

El desarrollo y uso de las redes de sensores inalámbricas han hecho que las topologías tradicionales de redes tomen nuevas direcciones; principalmente porque este tipo de redes requieren de alternativas que reduzca el costo y la complejidad mientras se mejora la confiabilidad en ellas. Las WSN están típicamente organizadas en una de las tres topologías siguientes (ver Figura 2.6):

- En una *topología estrella*, cuando cada nodo se conecta directamente al gateway (a).
- En una *topología de árbol*, cuando cada nodo se conecta a su nodo inmediato superior y este a la puerta de enlace. Los datos en esta topología son ruteados desde el nodo jerárquicamente más bajo hasta la puerta de enlace (b).
- Finalmente, a fin de incrementar la confiabilidad, la *topología malla* caracteriza a los nodos para que se puedan conectar con varios nodos en la red y pasar los datos a través de los nodos más confiables (c).

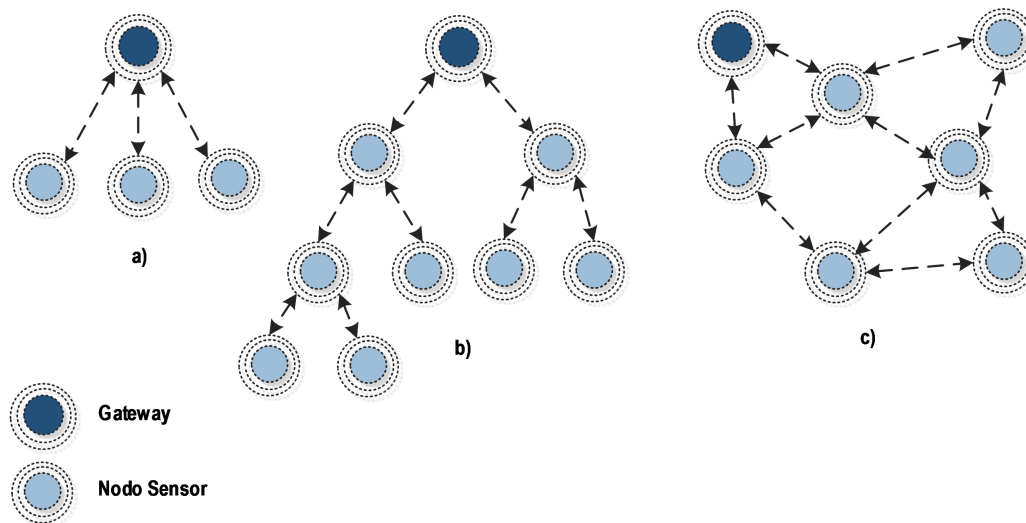


Figura 2.6 Topologías usadas en las WSN.

Por otra parte, es necesario denotar tres fases relacionadas en el mantenimiento y cambios en la topología [14]:

- La fase de *planeación y despliegue*.
- La fase *posterior a la implementación*.
- La fase de *redistribución de nodos adicionales*.

2.4 Arquitecturas de las Redes Inalámbricas de Sensores.

Existen diversos reportes técnicos [35], donde se presentan las arquitecturas de las WSN a la luz de las diferencias principales en los escenarios de aplicaciones y la tecnología de comunicación subyacente. La arquitectura de las WSN será drásticamente diferente tanto en la relación con un único nodo y la red como un todo. Una amplia gama de arquitecturas de nodos de sensores se han presentado hasta el día de hoy, pero con un principio general de diseño en todos ellos, centrandose en los siguientes objetivos: eficiencia en el uso de energía, tamaño pequeño y bajo coste. Hay otros reportes técnicos [36], donde la arquitectura es presentada desde el punto de vista de las características de la WSN. Ya que que éstas son diseñadas para una aplicación específica, no existe la necesidad de adaptarse a una arquitectura establecida, sino más bien a un conjunto de carateristicas, siendo esto usado como punto de partida. Por ejemplo una arquitectura centrada en datos.

Existen seis aspectos a considerar para definir la arquitectura de una WSN [37]: principios de diseño, arquitectura funcional, arquitectura de programación, arquitectura de protocolos, arquitectura del sistema de soporte y arquitectura física. Debido a esto se puede concluir que la situación al dia de hoy en las WSN, es que mientras ninguno de estos seis niveles de arquitectura en la red se resuelvan o incluso queden claramente establecidas, la gran mayoría de los estudios fallaran en categorizar la arquitectura.

2.5 Características de las Redes Inalámbricas de Sensores.

Las *redes inalámbricas de sensores*, tienen restricciones en los *nodos sensores*, que son de tamaño pequeño equipados de sensores, microprocesadores empujados y transreceptores de radio. Comparada con las redes inalámbricas tradicionales, las WSN tiene las siguientes restricciones y características únicas:

- *Denso despliegue de nodos.* Los nodos sensores suelen estar densamente desplegados en el área de interés. Este número puede ser mucho más elevado que las de las redes MANET.
- *Alimentados por baterías.* Usualmente los nodos en este tipo de red son alimentados con baterías. En la mayoría de las situaciones estan desplegados en lugares ásperos o hostiles, donde es difícil cambiar o recargar las baterías.
- *Severas restricciones en energía, almacenamiento y computo.* Están altamente limitados en estas capacidades.
- *Autoconfigurables.* Los nodos sensores son usualmente desplegados de forma aleatoria sin un plan cuidadoso. Una vez desplegados los nodos tienen autonomía para configurarse dentro de una red comunicándose.
- *Específicos a una aplicación.* Las redes son hechas para una aplicación específica, al cambiar los requerimientos de diseño de la red, ésta cambia con ella.
- *Nodos sensores poco confiables.* Usualmente éstos operan de forma autónoma y sin vigilancia en lo individual, además que son propensos a daños y fallas en donde se encuentran operando.
- *Cambios frecuentes en la topología.* La topología cambia continuamente dado los nodos que se van dañando dentro de la red.

- *Sin identificador global.* Dado el gran número de sensores, usualmente es imposible construir una identificación para direccionamiento global porque este introduce una gran carga de trabajo.
- *Patrón de comunicación de muchos a uno.* En la mayoría de las redes de sensores, los datos obtenidos por los sensores son enviados a un nodo colector.
- *Redundancia de datos.* Dada la distribución densa es posible que un dato sea leído por más de un sensor.

Estas restricciones y características únicas presentan muchas de los nuevos retos en el diseño de las redes de sensores.

2.6 Aplicaciones de las WSN

Las WSN tienen significativas ventajas sobre el mismo tipo de redes pero alámbricas [38]. Éstos no solo pueden reducir significativamente el costo y tiempo de desarrollo, sino también ser aplicadas a cualquier ambiente, sobretodo en aquellos donde las redes de sensores alámbricas son imposibles de poner. Hay una gran variedad de áreas donde desplegarse y éstas solo se ven limitadas por la imaginación, sin embargo a grandes rasgos las aplicaciones se pueden agrupar en cuatro tipos (ver Figura 2.7).

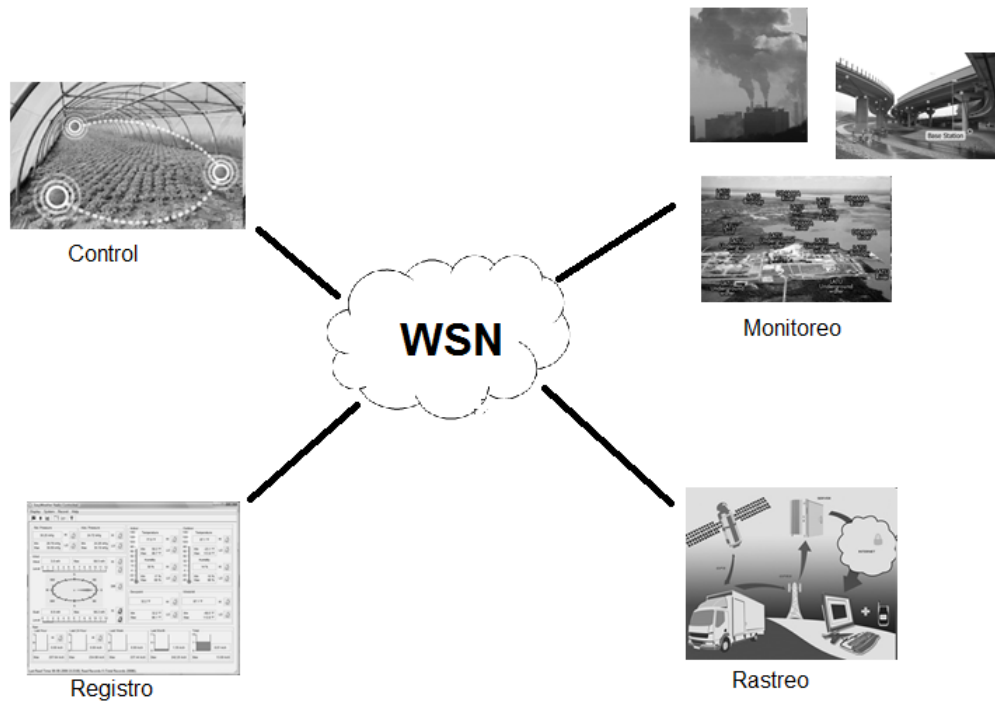


Figura 2.7 Grupos de aplicaciones de las WSN.

Algunos ejemplos explícitos de las aplicaciones de las redes de sensores inalámbricos son:

- *Monitoreo ambiental.* En esta los sensores son usados para monitorear una variedad de parámetros ambientales o condiciones.
 - Monitoreo de habitat. Los sensores pueden ser usados para monitorar las condiciones de los animales salvajes o plantas en habitat salvaje, dando a conocer bien los parámetros ambientales de los habitat [39].
 - Monitoreo de la calidad de aire o agua. Los sensores pueden ser desplegados para monitoreo del agua en una planta hidroquímica o el monitoreo de los contaminantes de aire en una ciudad.
 - Monitoreo de escenarios peligrosos. Los sensores pueden monitoriar localidades biológicas o químicas peligrosas como lo serían una planta química o un campo de batalla.

- Monitoreo de desastres. Los sensores pueden ser densamente desplegados en una región para detectar desastres naturales.
- *Aplicaciones Militares.* La idea central de este tipo de redes nace en el campo militar como parte del comando, control y sistemas de inteligencia [14]. Algunas aplicaciones específicas son:
 - *Monitoreo de escenario de batalla.* Los sensores desplegados en el campo de batalla monitorean la presencia de fuerzas y vehículos, seguir sus movimientos permitiendo el seguimiento cercano de los enemigos.
 - *Protección de objetivos.* Pueden ser desplegados alrededor de objetivos sensibles como plantas nucleares, puentes estratégicos, líneas de gas o combustible, centros de comunicaciones y campos militares con propósito de protección.
 - *Guía inteligente.* Los sensores pueden ir montados en vehiculos automatizados como tanques, aeronaves, submarinos, misiles o torpedos, para guiar a estos a través de diversos obstáculos.
- *Cuidados de la salud.* Las WSN pueden ser usadas para monitorear y dar seguimiento a pacientes o personas de edad avanzada con propósitos de cuidados de salud [40]. Algunos ejemplos concretos son:
 - *Monitorear el comportamiento de pacientes.* Los sensores pueden ser desplegados en la casa del paciente para monitorear su comportamiento. Por ejemplo, puede alertar a los doctores en caso de que el paciente recaiga y requiera atención inmediata.
 - *Monitoreo medico.* Sensores en vestimenta o usables que pueden ser integrados en la ropa del individuo formando una red inalámbrica (WBAN) para monitorear signos vitales [41].
- *Control de procesos industriales.* En este ramo pueden ser usados para monitoreo del proceso de manufactura o las condiciones del equipo de manufactura. Para ilustrar

esto, las WSN pueden ser instrumentos en los procesos de producción o ensamblaje para control.

- *Seguridad y vigilancia.* Las redes de sensores pueden ser usadas en muchas aplicaciones de seguridad y vigilancia. Por ejemplo, sensores de sonido, video o de cualquier otro tipo pueden ser utilizados en edificios, aeropuertos, metro u otro tipo de instalaciones para vigilancia de eventos.
- *Casas inteligentes.* Son usadas para proveer ambientes más convenientes e inteligentes [42].

Capítulo 3

La plataforma LiSANDRA

La Universidad Autónoma de Baja California (UABC), junto con el Centro de Investigación Científica y de Educación Superior de Ensenada (CICESE), percibieron el desarrollo de las redes inalámbricas de sensores como una oportunidad de desarrollo, aplicación, uso y difusión en esta tecnología. Trabajos en el área de investigación y difusión de esta tecnología dieron como resultado el diseño y la implementación de una plataforma WSN, llamada LiSANDRA (por sus siglas en inglés de *Light-weight Sensor and Actuator Network with Distance Range Auto-adjustable*) [43] y [44].

Esta red es experimental, a fin de tener una plataforma donde se pueda dar conocimientos y experiencia, además de asistir en el proceso de enseñanza-aprendizaje, para el entrenamiento de los estudiantes, permitiendo proveer de habilidades y conocimientos en este tipo de tecnología. Experiencias como estas pueden ser evidenciadas [45], [46], donde después fueron modificadas a fin de ser una plataforma comercial para las WSN adaptandose a aplicaciones específicas.

3.1 Arquitectura

LiSANDRA tiene básicamente una arquitectura jerárquica, también llamada *arquitectura de niveles*. En esta, los nodos sensores forman una jerarquía en la cual un nodo realiza un conjunto de tareas en nombre de un conjunto de nodos por debajo de su nivel. Esta arquitectura contrasta con las usadas comúnmente para las WSN que son de *arquitectura plana*, donde solo se tiene un único nivel y todos los sensores son iguales y homogéneos en forma y función. La razón principal por la que posee esta arquitectura es para soportar la

relación en cuestión de costo-efectividad, ya que este tipo de arquitectura reduce el costo de los nodos sensores ubicando los recursos en donde pueden ser efectivamente empleados. Existen otras posibles arquitecturas, pero no soportadas por LiSANDRA [47].

La red LiSANDRA esta compuesta de tres tipos de nodos:

- El **Nodo S**, este es el nodo más básico del que se compone la red. Su única función es sensar y transmitir pequeños paquetes de datos sobre su lectura a un solo nodo por encima de su jerarquía.
- El **Nodo SR**, que tiene previamente asociado un grupo de nodos S formando un *cluster*, tiene la función de sensar y rutear los paquetes, propios y los de por debajo de él, a través de la red con otros nodos de su misma jerarquía.
- El **Nodo G**, es el más alto en la jerarquía de nodos y opera como una puerta de enlace entre la red WSN y una red Wi-Fi 802.11b/g. Este nodo es el más costoso y para fines de reducir costos puede ser retirado de la red y usar un nodo SR como gateway. Esto, claro está, conectándolo al puerto serie de una PC y ejecutando un software específico; inclusive con esto se puede lograr dar conectividad a las redes tradicionales alámbricas.

3.1.1 El nodo S

El nodo S es el elemento más simple dentro de la red LiSANDRA (ver Figura 3.1). El tamaño de este nodo es de 1.65" x 1.15" y usa una batería de Litio de 3v con capacidad de 225 mAh como unidad de energía. En su PCB se tiene el microcontrolador ATmega88P que opera a 2 MIPS y cuenta con 8 KB de memoria tipo Flash para el programa y 1 KB de memoria RAM. Además en la unidad de sensado posee sensores de luz, temperatura y humedad (existentes también en el nodo SR); cuenta también con líneas libres para interfaz con otros sensores y actuadores. Para la transmisión se tiene un módulo que sólo transmite el TXM-418-LR (compatible con módulo de RF que usan los nodos SR).



Figura 3.1 Nodo S.

Este nodo pertenece a la capa inferior de la red, por lo que es el de menor complejidad. Su modo de operación es parametrizado (ver Figura 3.2) mediante una secuencia de tareas requeridas para llevar a cabo la correcta función. La cual básicamente consiste periódicamente sensor y transmitir información a un nodo SR al cual previamente fue asociado.

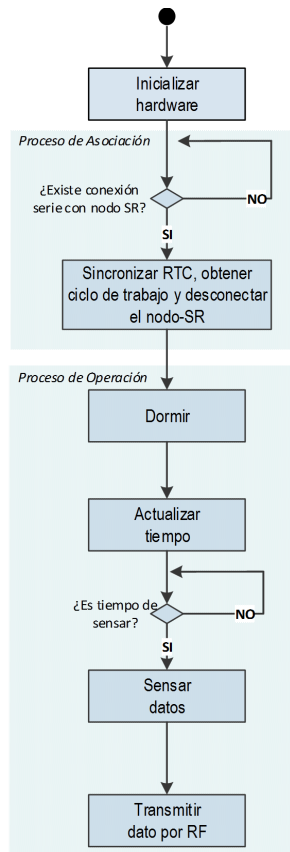


Figura 3.2 Diagrama de Flujo para parametrizar el funcionamiento del nodo S.

Dada su simplicidad de operación y recursos de hardware existentes no requiere de la plataforma de desarrollo propuesta, ya que las tareas más complejas a ser realizadas se delegan a la siguiente capa en la jerarquía, compuesta por el nodo SR.

3.1.2 El nodo SR

Este nodo, ver Figura 3.3, está construido en un PCB de 1.25" x 2.50" de doble cara. Como fuente de poder utiliza dos baterías recargables AA. Su unidad de procesamiento es el microcontrolador ATmega644PV con 64 KB de memoria FLASH y 4 KB de memoria de datos (RAM) operando a una velocidad de 2 MIPS. Además utiliza un oscilador de reloj externo de 32 KHz que sirve como base para un RTC (*Real-Time Clock*) y una memoria serie externa de 1 MB que sirve como almacenamiento de datos.

La unidad de sensado está compuesta por un sensor de temperatura SHT11 y un sensor de luz ambiental TSL2550T. Sin embargo, en el diseño de la plataforma se consideraron líneas de E/S que pueden ser utilizadas como puertos de expansión a interfaces de actuadores o de otro tipo de sensores de acuerdo a los requisitos particulares de la aplicación.

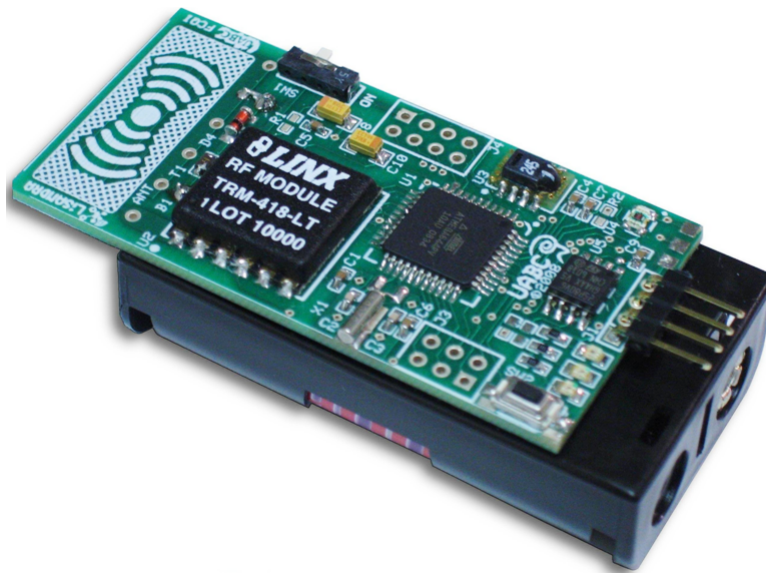


Figura 3.3 Nodo SR.

En la unidad de comunicación se maneja el transreceptor TRM-418-LT con antena embibida en el PCB. También se tiene una sección de comunicación alámbrica en el estándar RS232 con voltajes TTL para comunicarse con el puerto serie de una PC para bajar programas y/o cargar datos almacenados en la memoria serie.

3.1.3 El nodo G

La función del nodo G es la de servir de puerta de enlace entre la red basada en LiSANDRA y una red estándar como la 802.11 b/g. Para esto se utiliza como base un nodo SR con un módulo Wi-Fi conectado al puerto de expansión (vease Figura 3.4)

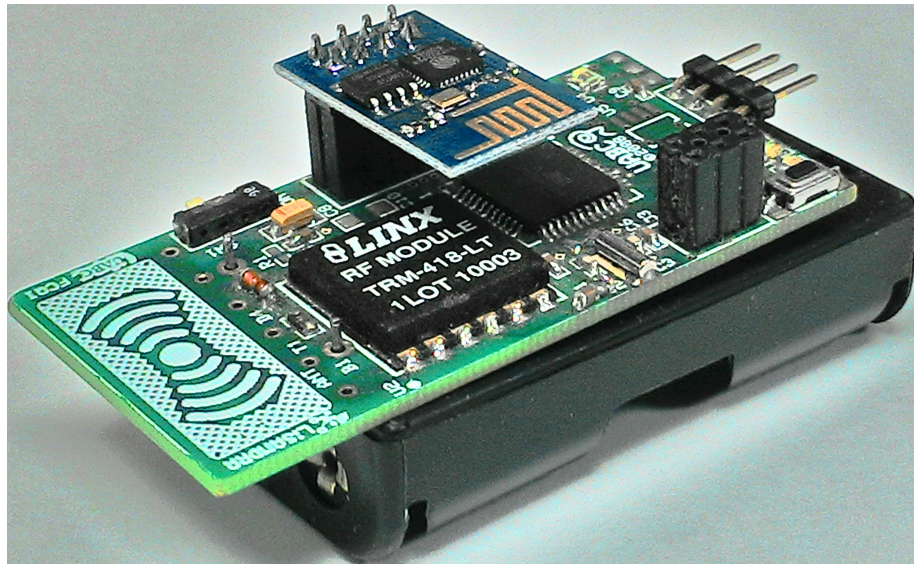


Figura 3.4 Nodo G.

Este nodo posee el software requerido para realizar las operaciones de direccionamiento del tráfico interno hacia el exterior y viceversa permitiendo que la red LiSANDRA se integre a una red mayor y otorgar servicios proporcionados por los nodos SR.

3.2 Software

En software es del tipo monolítico compuesto por la aplicación y los subsistemas que controlan las unidades de hardware. Estos subsistemas son el procesamiento de la comunicación y el controlador para la comunicación y sensores; todos estos subsistemas están encapsulados dentro de un núcleo de sistema de tiempo real llamado *LEARN* [48].

Sin embargo, uno de los objetivos de este trabajo de tesis, es elevar el nivel de abstracción del hardware y la programación de aplicaciones en las redes de sensores haciendo uso de un lenguaje de alto nivel. Para esto, solo se mantiene el software de arranque y el núcleo del sistema.

Capítulo 4

Desarrollo de Software en WSN

El desarrollo de software para las WSN es difícil porque su hardware consiste de dispositivos embebidos pequeños, frágiles y de recursos limitados que se comunican sobre enlaces de reducido ancho de banda (Ver Capítulo 2). Además de que, por la existencia de una gran diversidad de plataformas no estandarizadas [49], no es posible establecer una metodología única en el desarrollo de aplicaciones ya que éstas son del tipo Ad Hoc y están orientadas a resolver el problema dentro de un contexto en específico.

No obstante en el rubro del despliegue de las WSN se pueden identificar fases genéricas en su ciclo de vida (Ver Figura 4.1) como siguientes:

- La etapa de *planeación*, usualmente involucra la inspección del área de despliegue y la selección del posicionamiento de los sensores de tal forma que cumpla con el objetivo previsto.
- La etapa de *despliegue*, los nodos sensores continuamente envían su calidad de la conexión inalámbrica y su enrutamiento a la estación base.
- La etapa de *monitoreo*, el usuario se interesa principalmente en los valores leídos por la red de sensores.
- La etapa de *control*, la aplicación usada para controlar la WSN también puede ser usada para el envío de comandos a la red. Estos comandos pueden decirle a la red de dispositivos que se detengan en el envío de mensajes, incrementar el tiempo entre mensajes o reiniciar la red.

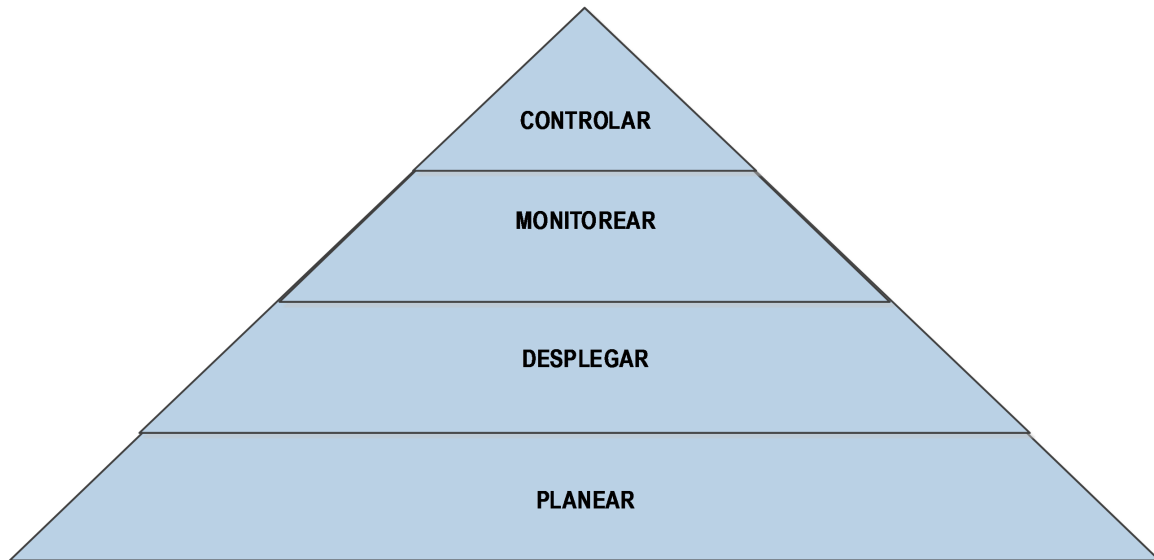


Figura 4.1 Ciclo de vida de las WSN.

El recurso humano de este ciclo de vida se puede agrupar en investigadores y usuarios. Los investigadores están usualmente envueltos en las fases de planeación y el despliegue de las WSN, mientras que los usuarios finales solo están interesados en el monitoreo y control de las mismas.

Otra propuesta interesante para las fases de despliegue de las WSN puede ser encontrado en [50]. Donde todos los componentes de esta arquitectura son representados por bloques (Ver Figura 4.2).

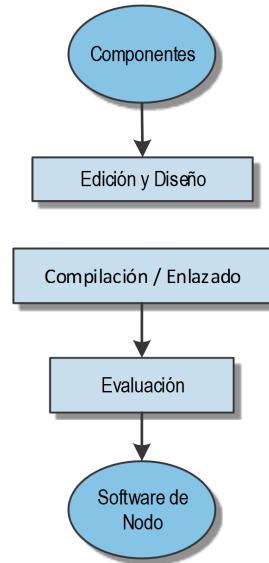


Figura 4.2 Ciclo de diseño de Software para nodos Sensores.

Ante el desafío que presenta el desarrollo de aplicaciones en WSN se han desarrollado soluciones parciales, incompletas o incompatibles entre sí. Al mismo tiempo el concepto de reusabilidad de software se ha convertido en parte esencial del desarrollo en otros entornos distribuidos.

Un ejemplo claro es la definición de arquitecturas middleware, que son diseñadas teniendo en mente los principios de modelado de la ingeniería del software como dogma para proporcionar sistemas reusables que puedan ser compuestos, configurados e instalados para crear sistemas robustos [51].

4.1 Desarrollo basado en Middleware

La middleware es considerada una capa que ayuda al programador a desarrollar variadas aplicaciones. Este ofrece la abstracción necesaria entre el hardware, sistema operativo, baterías y la aplicación. El objetivo de la capa middleware es proveer:

- Interfaces apropiadas para diversas aplicaciones.

- Un entorno de ejecución que soporte y coordine múltiples aplicaciones.
- Mecanismos para lograr un uso adaptativo y eficiente de los recursos del sistema.

Además, permite la reutilización de código con sus servicios, como podría la actualización de datos de tal forma que el programador puede desarrollar y ejecutar la aplicación sin entrar en conflicto con la complejidad y uso de funciones tediosas, también ayuda en la administración y adaptación en la infraestructura de la red (Ver Figura 4.3) [52].

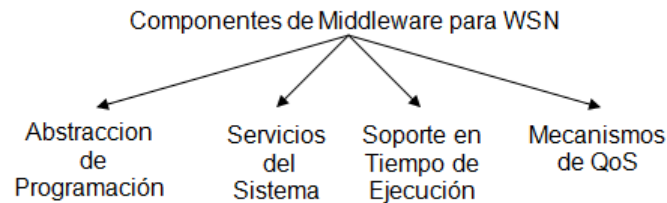


Figura 4.3 Componentes comunes de las Middleware para WSN.

Las middleware han sido a menudo utilizadas por los sistemas tradicionales como un puente entre los sistemas operativos y las aplicaciones. Aunque las middleware tradicionales existentes (como DCOM, CORBA) no son adecuadas para los requerimientos especiales de las WSN (como memoria, energía y poder de cómputo), el desarrollo de una se hace evidente por las restricciones que este tipo de redes presentan. En general esta middleware debe de ser simple, ligera, fácil de ser implementada, portable, estandarizada, poseer soporte para la coordinación de múltiples tareas y facilitar el desarrollo de aplicaciones.

Diferentes tipos de middleware han sido creados desde que se dio el auge de las WSN, éstas se pueden seleccionar y clasificar tomando los modelos de programación que usan (ver Figura 4.4).

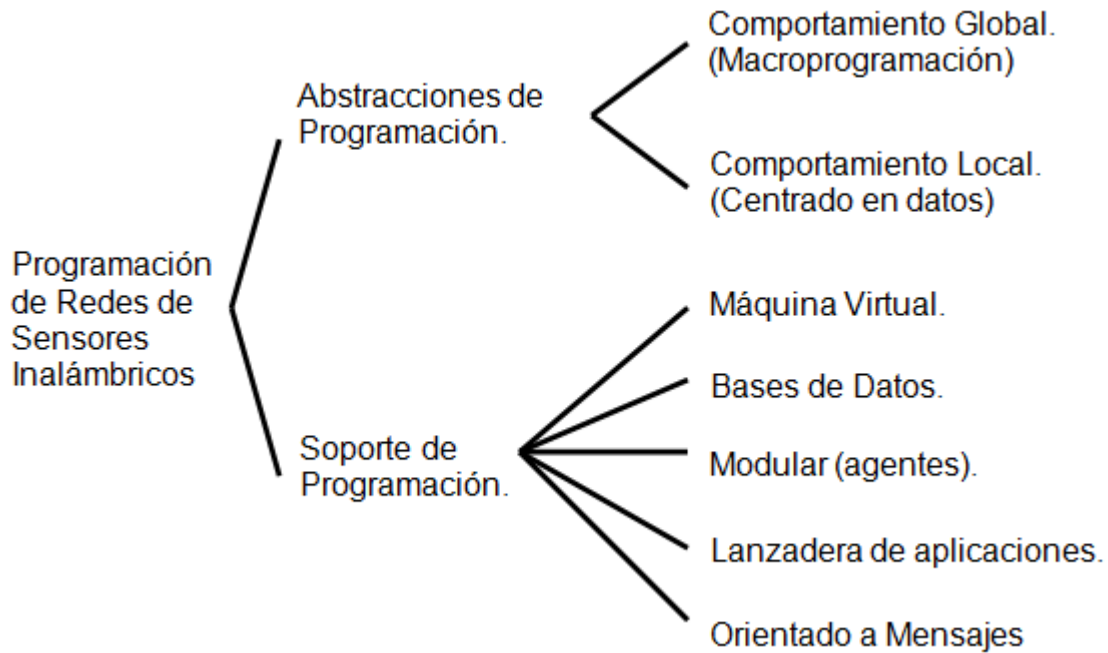


Figura 4.4 Tipos de Middleware en base a su modelo de programación

4.1.1 Máquinas virtuales.

Dentro del enfoque a las máquinas virtuales (MV) se encuentran también los intérpretes y agentes móviles. Su principal característica es la flexibilidad que permite a los desarrolladores escribir aplicaciones divididas en pequeños módulos, los cuales son inyectados y distribuidos a través de la red por el sistema, haciendo uso de algoritmos diseñados e interpretados por la MV. Estos algoritmos minimizan al máximo el consumo de la energía además del uso de recursos. Sin embargo, la tecnología es compleja y las instrucciones introducen un gasto extra de recursos [53].

Algunos de los enfoques hechos en este campo son:

- **Maté** (Levis y Culler, 2002). Ofrece una mejor interacción y adaptación a la variabilidad de las redes de sensores mediante el soporte a la apertura y escalabilidad. Hace uso de protocolos de red y parámetros actualizados para el envío de active messages

como consecuencia de la introducción de algún nuevo modulo. Sin embargo, la interpretación de sus instrucciones introduce un gasto de recursos que la hace ineficiente para aplicaciones complejas [54].

- **Squawk** (Simon et al. 2006). Máquina virtual escrita en Java que corre sin un sistema operativo en la plataforma de redes de sensores. Esta provee de una API que permite a los desarrolladores escribir aplicaciones para las WSN. Esta API es una extensión del GCF. El principal problema con esta máquina virtual es que solo ha sido aplicada al Sun SPOT (Sun Small Programmable Object), un dispositivo creado para experimentar en el desarrollo de aplicaciones para las redes de sensores y actuadores inalámbricas [55].

4.1.2 Programación Modular.

Este tipo de programación facilita la movilidad, inyección y distribución de código através de la red llevando consigo la modularidad de la aplicación. Cuando se envían pequeños módulos de una aplicación completa es requerida menos cantidad de energía.

Algunos desarrollos son:

- **Impala**(Liu y Martonosi 2003). Hace uso de un modelo de arquitectura apropiada que provee a la aplicación de adaptabilidad en tiempo de ejecución ofreciendo seguridad contra un inoportuno error de programación. El soporte para la movilidad, apertura y escalabilidad es realizada mediante el cambio entre diferentes protocolos y modos de operación en función de las aplicaciones y condiciones de la red [56]. Sin embargo, ésta solo es capaz de ser ejecutada en las Pocket PC iPAQ de HP/Compaq.
- **Smart Messages Project (Smart Messages)**. Hace una propuesta de un modelo distribuido llamado “Computación Cooperativa” en la cual las unidades migradas son llamadas *Smart Messages*.

4.1.3 Bases de datos.

Esta perspectiva observa a la red como un sistema virtual de bases de datos, ofrece una interfaz sencilla de usar que permite al usuario extraer datos de su interés con la edición de búsquedas en la red de sensores. No obstante esta solución no soporta aplicaciones de tiempo real, ya que solo proporciona resultados aproximados y la detección de las relaciones espacio-temporal entre los acontecimientos no es posible. Los trabajos presentados en este rubro son:

- **Cougar** (COUGAR). Establece una innovadora dimensión en el campo de la investigación de middleware para WSN asumiendo una aproximación de estas a la bases de datos en la cual, los datos de cada sensor forman una base de datos relacional. La administración de la WSN es realizada en forma de búsquedas al estilo del lenguaje SQL. Además caracteriza a un sistema de base de datos de sensores que contiene datos almacenados, lecturas de los sensores y búsquedas en los sensores [57].
- **TinyDB** (Madden et al. 2003). Es un sistema de procesamiento de búsquedas para retirar información de una red de sensores. Esta usa TinyOS como sistema operativo, el cual obliga al usuario a escribir código de C para sistemas embebidos al extraer los datos del sensor. Sin embargo, TinyDB simplifica la extracción de datos dando una interfaz al estilo de SQL [58].
- **SINA** (Shen et al. 2001). Siglas de “System Information Networking Architecture” y modela a la red como un conjunto sustancial de objetos distribuidos. El sistema esta basado en clusters y el núcleo de este es una base de datos para su consulta y monitoreo. En su totalidad la red de sensores es un conjunto de hojas de datos. Cada hoja de datos lógica contiene celdas que caracterizan los atributos de un nodo sensor. Cada celda es exclusiva y cada sensor contiene una hoja de datos [59].
- **DSWare** (Li et al. 2003). Es una solución tipo base de datos que esta basada en la detección de eventos. El sistema soporta la flexibilidad haciendo medias de grupos y

un almacenamiento central confiable, también soporta aplicaciones de tiempo real y reduce el gasto de recursos en la transmisión. Además provee de una interfaz tipo SQL para registrar y cancelar eventos [58].

4.1.4 Lanzadera de Aplicaciones.

Este enfoque establece un innovador aspecto en la investigación de las middleware al complementar una arquitectura que cumple con una pila de protocolos de red, permitiendo a los programadores ajustar la red de acuerdo a sus requerimientos exactos. Esta provee una ventaja en el QoS ya que la aplicación determina las operaciones de administración en la red.

- **Milan** (Murphy y Heinzelman 2002). Son las siglas para *Middleware Linking Applications and Networks*, y es usada en aplicaciones que afectan a la red por completo. En ésta se definen dos clases de aplicaciones, las lanzaderas de aplicaciones de datos (colectar y analizar datos) y aplicaciones basadas en el estado (en la cual los requisitos podrían cambiar con los datos recibidos). Es instalada en cada sensor de la red, en donde recibe información acerca de aplicaciones y términos referente a sus requisitos QoS. Esta plataforma recibe la siguiente información para su operación: 1) Variables de interés para la aplicación, 2) La QoS requerida para cada variable y 3) El nivel del QoS que los datos de cada sensor o conjunto de estos puede proporcionar para cada variable [60].

4.1.5 Middleware Orientadas a Mensajes.

Esta solución es esencialmente un modelo de comunicación de una red distribuida de sensores. El sistema facilita el intercambio de mensajes entre nodos y el nodo *sink* por medio de mecanismos de suscripción publica. Este modelo soporta la comunicación asíncrona, haciendo móviles las combinaciones entre el emisor y el posible receptor.

- **MIRES** (Souto et al. 2004). Implementa un modelo de programación basado en componentes usando *active messages* para publicar su infraestructura de comunicación como una suscripción. Incluye: 1) un componente núcleo el cual publica la suscripción del servicio, 2) un componente de enrutamiento, y 3) un servicio de agregación de datos que permite al usuario indicar como obtener los datos entre otras cosas. Este componente esta construido sobre TinyOS y hace uso de C embebido para programar [61].
- **SensorBus** (Ribeiro et al. 2005). Se basa en el paradigma de suscripción y publicación. Permite el libre intercambio de mecanismos de comunicación entre los nodos sensores, los cuales por turnos permite más de un mecanismo de comunicación al direccionar los requerimientos de un gran número de aplicaciones [62].

4.2 Desarrollo basado en tecnologías de agentes.

El término agente involucra una gran variedad de definiciones que incluyen:

- *"Un agente es todo aquello que puede percibir su entorno a través de sensores y actuar sobre el medio que lo rodea haciendo uso de actuadores"* (Russell y Norving 1995).
- *"Los agentes inteligentes realizan tres funciones: percibir las condiciones dinámicas del ambiente, actuar en lo que concierne a esas condiciones e interpretar la información recibida. Ellos resuelven problemas, muestran interfaces y determinan acciones"* (Hayes-Roth 1995).

Basándonos en estas definiciones es posible considerar a los agentes como entidades dentro de un ambiente que están preparados para sentir y actuar, y por lo tanto calificados para comunicar y colaborar con otras entidades, como las humanas.

De acuerdo a las características de los agentes, se puede definir a un sistema multiagente como un sistema distribuido formado por agentes capaces de cooperar con otros y compartir

recursos a fin de completar una tarea común o individual. Por esta razón, ellos deben de ser capaces de interactuar en un entorno común y poseer habilidades de comunicación, negociación y coordinación. El objetivo de una plataforma multiagente es ser una plataforma autónoma.

Las características principales de un programa tradicional que difiere de un programa basado en agentes es:

- **Autonomía.** Los agentes hacen sus propias decisiones basándose en sus propios objetivos.
- Cada **agente** tiene su propio **thread** ejecutándose. Sin embargo, un programa tradicional tiene solamente un flujo de instrucciones para todo el sistema.

Los agentes requieren de ciertas plataformas que les proporcionen ciertos servicios como identificación, movilidad, comunicación y otras más para su ejecución. Existen varias especificaciones propuestas para las plataformas de agentes de diferentes grupos como: KAoS [63], OMG [64], FIPA [65], etc.

Un agente recibe señales o información de su entorno a través de algunos sensores. Estas señales serán procesadas por su software, en el cual se decidirá cual acción se tomara haciendo uso de los actuadores. El modelo de agentes presenta muchas similitudes con el modelo actual de sensores que aparece en la literatura de las WSN (Ver Figura 4.5)

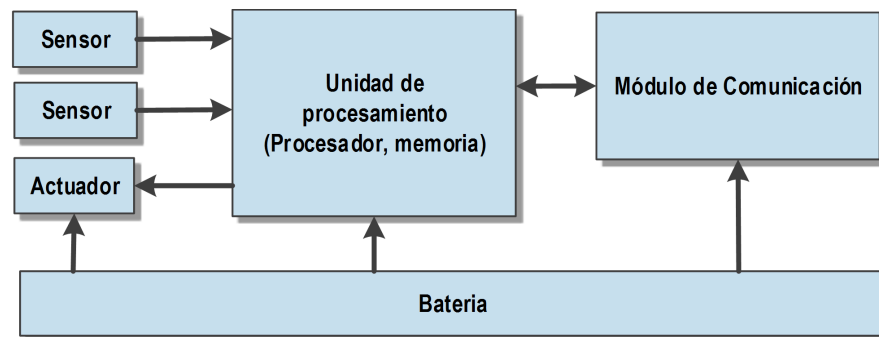


Figura 4.5 Elementos que integran un nodo WSN.

Como se puede observar los nodos WSN poseen sensores que entregan información el entorno, mientras la unidad de procesamiento administra esa información (ya se la obtiene el o de otro nodo de la red) y realiza una acción en base a ese procesamiento.

Desde la perspectiva presentada en el modelo de agentes parece haber una aplicación natural en el campo de los nodos WSN, al menos en la parte de sus componentes. Inclusive hay una propuesta hecha (Lesser et al. 2003) de una aplicación de redes de sensores que consiste de un modelo de agentes en la cual cada nodo sensor es controlado por un agente.

Las características que coinciden entre las aplicaciones basadas en el modelo de agentes en las redes de sensores inalámbricos son:

- Reactividad.
- Capacidad de trabajo asíncrona.
- Autonomía.
- Orientados a objetivos.
- Capacidad de comunicación.
- Colaboración y cooperación.

En resumen la aplicabilidad de los agentes en las WSN viene en las similitudes entre sus objetivos.

Algunas formas de implementar el modelo de agentes a las WSN son:

- **Agentes estáticos..** De acuerdo a la propuesta hecha por Lesser et al. (2003) la WSN se organiza en grupos con un cierto numero de sensores por sector. Un número usual es de ocho por sector, considerando máximo 10 y mínimo 5. Cada agente deberá conocer el tamaño de su sector ya que los agentes determinaran el comportamiento de los objetos en el. Los agentes deben saber si un objeto pertenece o no a su sector (Ver Figura 4.6).

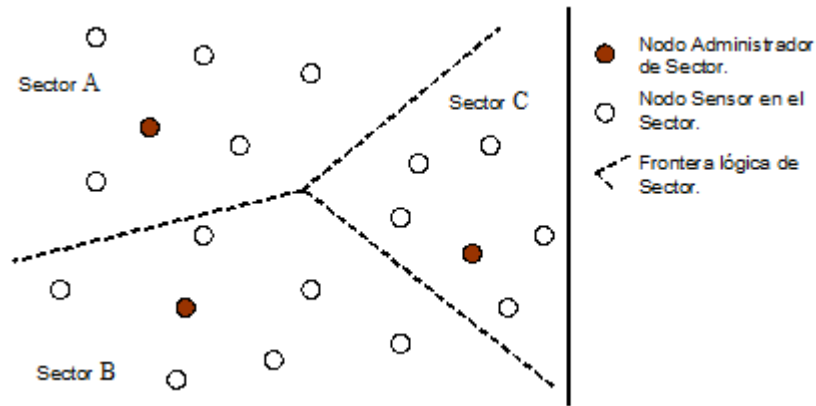


Figura 4.6 Modelo de agente estático en WSN.

Dentro de cada sector, los agentes son especialistas en tareas de rastreo y exploración de la red. Cada sector tiene un administrador de ese sector que se encarga de determinar nuevos objetivos, asignar roles y almacenar y comunicar información de los sensores locales como parte de un directorio de servicios. Además, el administrador actúa como un concentrador, facilitando el intercambio de datos entre las partes interesadas y la población general de agentes [66].

- **Agentes móviles.** De acuerdo a la propuesta hecha por Szumel et al. (2005), las redes futuras tendrán probablemente miles o millones de nodos heterogéneos. Esto dificultará la escalabilidad de las soluciones para las limitaciones de las WSN en la energía y su limitado ancho de banda. Por lo tanto, los agentes pueden ser usados, estando presentes en cualquier nodo, procesando datos y reduciendo las interacciones entre los nodos ya que esta actividad consume una gran cantidad de energía (Ver Figura 4.7).

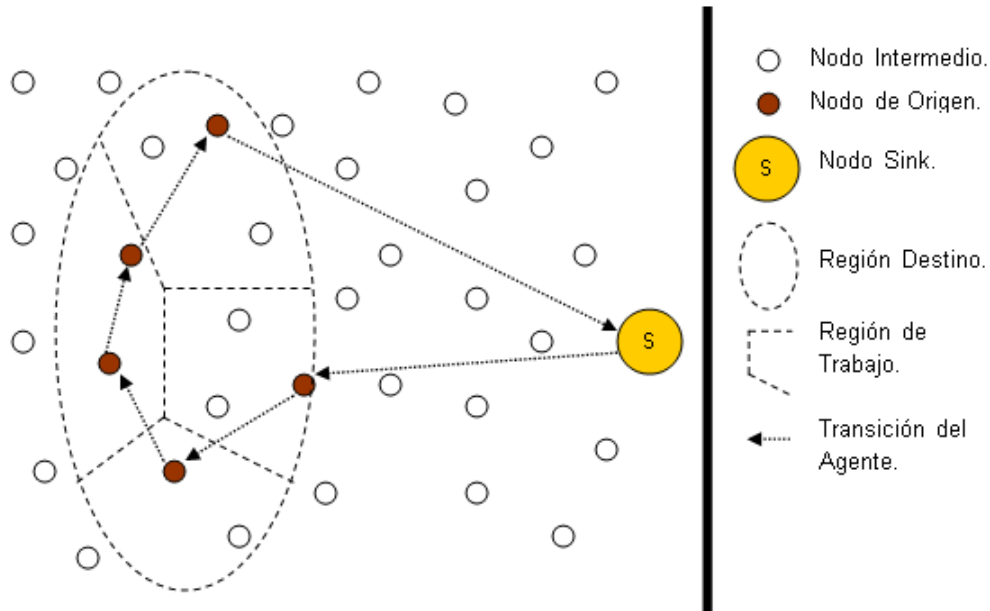


Figura 4.7 Modelo de agente móvil en WSN.

Las ventajas de usar soluciones basadas en agentes incluyen el soporte para múltiples usuarios, la programación incremental y un uso eficiente de los recursos. Los agentes solo utilizarían los recursos del nodo que visitan como su capacidad de procesamiento y de transmisión de datos. Se generan nuevos agentes solo en los nodos implicados en la detección del nuevo objeto. La mayor ventaja puede ser observada al momento de que la red debe de ser reorientada a una nueva tarea, usando agentes solo requeriríamos reprogramar algunos nodos [67].

Hay un trabajo publicado por Fok et al. (2005) que describe Agilla, una middleware que corre en TinyOS y provee un entorno para la ejecución de agentes móviles en las WSN. Esta simplifica la flexibilidad de la red, simplifica el desarrollo de la aplicación, y hace uso de una máquina virtual (llamada Maté). El estado de la ejecución del agente es almacenado en un espacio de memoria compartido dentro de la memoria del nodo.

4.3 Desarrollo Basado en Sistemas Operativos.

El desarrollo de aplicaciones para PC no hubiera alcanzado su auge (en el gran número de aplicaciones, paradigmas de desarrollo y lenguajes de programación) si no hubiesen evolucionando los sistemas operativos a lo que hoy en día son. Debido a que este provee el entorno dentro del cual los programas son ejecutados, además de los servicios que este sistema proporciona y las facilidades que otorga tanto a usuarios como programadores con el uso de sus componentes e interconexiones.

En contraste con el desarrollo de software en la PC, los sistemas operativos en las redes de sensores consisten esencialmente en una biblioteca de funciones enlazadas al código de la aplicación para producir una imagen ejecutable en el nodo (Sistema monolítico). Usualmente, éste soporta el uso de un lenguaje de programación, típicamente C o un dialecto específico para las WSN (p.e. NesC para TinyOS [68] o ATaG para Data-Driven ATaG Runtime [69]), además de un módulo de comunicación de bajo nivel para la interconexión de los mismos.

La utilización del lenguaje C y uso de Sockets en la PC se asemeja al conjunto de abstracciones básicas de programación proporcionadas por los sistemas operativos en las WSN [70].

Además de esto, hay que tener presente que el sistema operativo actúa de forma aislada en un nodo administrando sus procesos y recursos en él.

El sistema operativo es clave para el desempeño de los ambientes de computación distribuida de las redes de sensores inalámbricos. Hay algunas otras características agregadas de éstos; tales como: protección, memoria virtual, planificador de tareas, entre otros, que le dan confiabilidad a los sistemas de las WSN y faciliten el desarrollo de software complejo. Algunos ejemplos son: Contiki [71], Squawk [55], ParticleVM [72], Bertha [73], NutOS [74].

4.4 Desarrollo Basado en Sistemas de Servicios (SOA).

Una arquitectura orientada a servicios es esencialmente una colección de servicios. Estos servicios se comunican entre ellos y esta comunicación puede involucrar simplemente dos o más servicios que se coordinan para alguna actividad. Esto involucra la necesidad de una interconexión entre los distintos servicios [75].

Dentro de esta arquitectura se definen tres roles: el que solicita el servicio, el que provee el servicio y el que registra el servicio; y tres simples operaciones: publicar, encontrar y enlazar. Estos roles y operaciones que se explican por si mismos y pueden ser accesadas en varias ocasiones por cualquier entidad [76].

En resumen, el desarrollo de aplicaciones en este tipo de sistemas involucra a la iteracción de roles y operaciones de servicios ya existentes.

En las WSN este tipo de desarrollo implica que los nodos proveen datos encapsulados en servicios, permitir la programación en gran escala y la introducción de nuevos conceptos basados en enrutamiento centrado en servicios separando la lógica de la aplicación del enrutamiento. Sin embargo, las “buenas prácticas” que permite la programación de los sistemas de computación distribuida no pueden ser reutilizadas en las WSN (por su comportamiento dinámico, continuo cambio de enrutamiento y movilidad de los nodos) y por lo tanto terminan siendo una adaptación acorde al dominio específico del despliegue de las WSN.

Algunos ejemplos de este tipo de arquitecturas (aunque no necesariamente están implementadas en nodos de WSN) son: OASIS [77], TinySOA [78], Servilla [79], Refective [80], Flexible [81].

Capítulo 5

Herramientas de desarrollo en los nodo WSN

El éxito comercial en la aplicación de las WSN depende en la disponibilidad de herramientas y metodologías que existan para el desarrollo de software, la abstracción de los detalles de bajo nivel, y el despliegue y operabilidad de la red.

Por dar un ejemplo, para las PC, en lugar de las WSN, puede usarse el ambiente Eclipse [82], Visual Studio [83], CodeGear [84], y otras herramientas de desarrollo que ofrecen varios beneficios a los desarrolladores. Ilustrando esto, si se tiene un ambiente de desarrollo/programación para las WSN en el cual la sintaxis fuese corregida, puede ser compilado por un programa y probado en una máquina virtual que simule el hardware de un mote o un conjunto de motes interconectados. De tener lo anterior para el desarrollo de la aplicación que es cargada al mote, el desarrollo del software se simplificaría y el éxito de las WSN sería más sencillo de lograr.

Desde que surgió el interés de los investigadores en las WSN hasta el día de hoy, no existen herramientas completas y consolidadas que ayude en el desarrollo de software para las WSN. Por lo tanto resulta interesante el diálogo que se ha abierto para el desarrollo de una plataforma compuesta por Frameworks extensibles, herramientas, ayudada por la capacidad de probar en máquinas virtuales y entornos de ejecución para la construcción, despliegue y administración de software para las WSN.

Aunque hay algunas propuestas interesantes sobre el diseño de software para las WSN, esta sigue siendo un área donde se puede hacer más aportaciones valiosas; sobre todo si tratan de la generalización y la definición de metodologías para la creación de software confiable.

Una propuesta interesante ya comentada (Capítulo 4 de Blumenthal et al. (2003)), donde hay un ciclo propuesto de diseño de software para las WSN. Una adaptación al modelo de ciclo de vida para el desarrollo en WSN esta en la Figura x.1, donde a las partes que integran este modelo se le llama componentes de desarrollo.

Los desafíos en materia de la programación, monitoreo, gestión y resolución de problemas en las WSN van en aumento y con ello los desafíos para las herramientas correspondientes de desarrollo [85].

5.1 IDE para TinyOS/nesc.

TinyOS es un sistema operativo de núcleo multitarea reducido, útil para pequeños dispositivos como los nodos de las WSN [68]. Este sistema operativo funciona a partir de eventos producidos que llaman a su vez a funciones. El entorno de desarrollo soporta directamente la programación de diferentes microprocesadores y permite programar cada tipo con un único identificador para diferenciarlo, o lo que es lo mismo se puede compilar en diferentes plataformas cambiando el atributo *plataforma*.

El sistema operativo, su biblioteca de funciones y aplicaciones, esta escrito en nesc, una versión de C que fue diseñada para programar sistemas embebidos. En nesc, los programas están compuestos por componentes que se enlazan para formar un programa completo.

Muchas de las investigaciones realizadas en el área de las WSN han sido basadas en el sistema operativo TinyOS, creado por la Universidad de California Berkeley, pero hoy en día un proyecto público de Sourceforge incluye a varios grupos de investigadores.

Actualmente para compilar los programas de TinyOS requieres de:

- *Editor de texto*, como notepad, WordPad o jEdit.
- Un *entorno de línea de comandos*, como Cygwin [86].

Estos componentes del IDE es uno de los intentos por integrar estos dos procesos individuales dentro de un solo componente.

El proceso de programación en este ambiente (Ver Figura 5.1) es automatizado por la herramienta make y el código fuente esta en nesC.

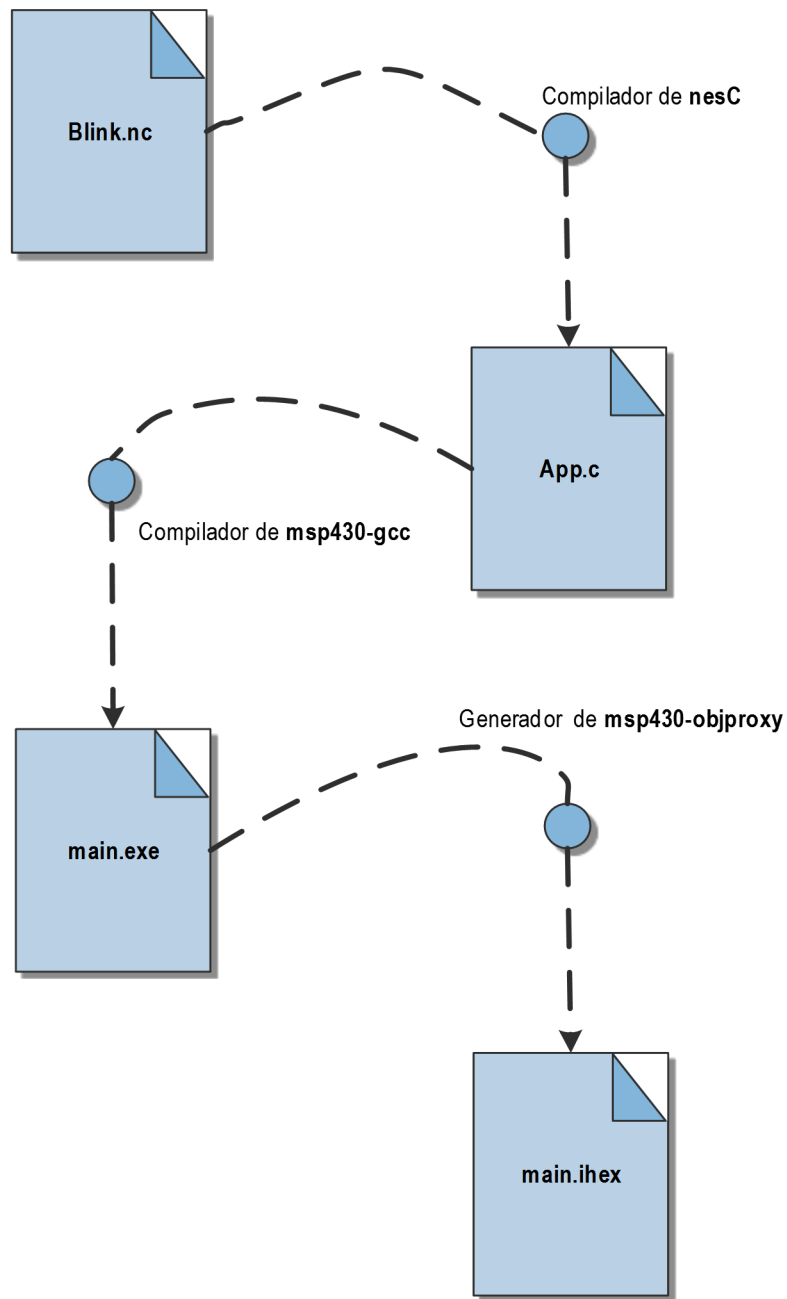


Figura 5.1 Proceso automatizado de compilación de aplicaciones.

5.1.1 TinyDT.

Es un plugin para la plataforma Eclipse que implementa un IDE para el desarrollo en TinyOS/nesc [87]. La característica mas importante que distingue a TinyDT de otros IDE de TinyOS es que, TinyDT tiene un analizador interno que reconstruye una representación en memoria de la aplicación que se esta desarrollando en nesc, visor de jerarquía de componentes, el cableado, interfaces y la documentación (al estilo de JavaDoc) de nesc.

Este plugin implementa varias características como el código de navegación y la terminación automática de código. Además de resaltado de sintaxis, soporte para plataformas múltiples y placas de sensores, el apoyo automático de construcción, el apoyo del equipo de desarrollo y el soporte para múltiples fuentes de TinyOS (Ver Figura 5.2).

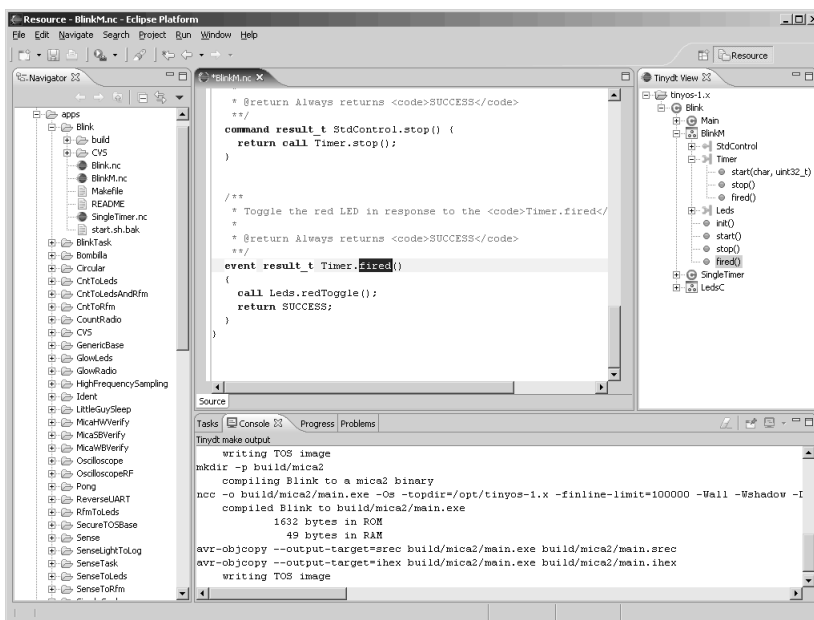


Figura 5.2 Editando la aplicación Blink en TinyDT.

Haciendo referencia al modelo de desarrollo de la Figura 5. TinyDT tiene participación en las fases de: Administración de proyecto, Edición, Compilación y el Desarrollo en equipos (CVS).

5.1.2 TinyOSIDE.

Un IDE para el desarrollo, prueba y evaluación de las aplicaciones desarrolladas para las WSN usando TinyOS. Este también es un Plugin desarrollado por Richard Tynan, que requiere de la instalación de TinyOS y Eclipse. Tiene participación en las fases de Administración de proyecto, Edición, Compilación y desarrollo en equipos (CVS) [88].

5.1.3 TOSDev.

Es una herramienta de desarrollo integrada para TinyOS incluyendo el alambrado gráfico con edición de código fuente. Liberando a los desarrolladores de la molestia de escribir manualmente los archivos de cableado, mientras que da un entorno de desarrollo integrado [89]. Fue desarrollado para mantener una abstracción real de los cableados del código fuente. TOSDev también incluye varias características comunes en muchos IDEs, como el término de escritura automatizada de los comandos/eventos y el despliegue de prototipos (Ver Figura 5.3).

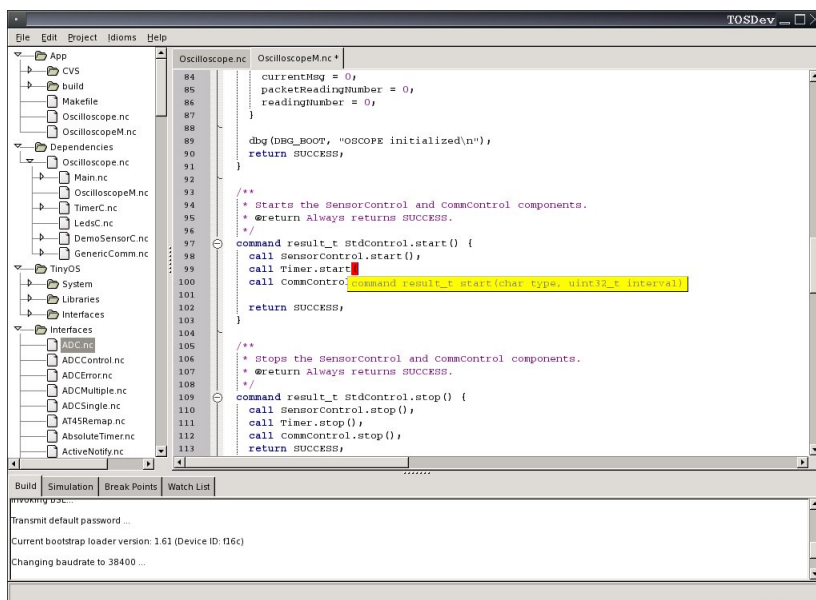


Figura 5.3 Entorno de desarrollo TOSDev.

Esta plataforma de desarrollo corre en ambientes de Windows, Linux y Mac OSX. Soporta el desarrollo, depuración y despliegue de aplicaciones de TinyOS.

5.1.4 YETI.

Es una herramienta que provee elementos para un desarrollo eficiente y conveniente tanto para usuarios experimentados como recién iniciados [90]. Este Plugin funciona para la plataforma Eclipse haciendo uso de varios de sus componentes como: su editor, su sistema de persistencia y su cliente CVS (Ver Figura 5.4). Consiste de dos Plug-ins:

- *System*. Contiene toda la funcionalidad del ambiente de desarrollo y las herramientas para el de desarrollo orientado a TinyOS. Como se aprecia en la Figura x.4, una vez instalado YETI ofrece una nueva perspectiva; y el
- *TinyOS Environment Wrapper*. Da acceso a la instalación del TinyOS.

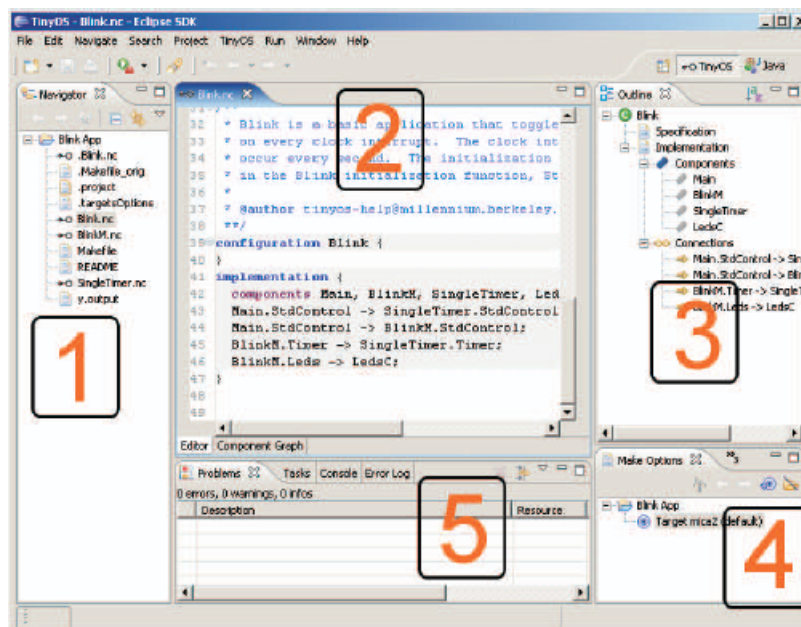


Figura 5.4 Entorno YETI. 1) Navegador con el listado de todos los archivos del proyecto. 2) Ventana principal mostrando el editor con el archivo abierto. 3) Esquema del archivo abierto mostrando su estructura. 4) Opciones del automatizado Make. 5) Panel multipropósito.

5.1.5 VIPTOS.

Es un entorno gráfico de desarrollo integrado y simulación para WSN basadas en TinyOS. Provee de un puente entre el desarrollo basado en VisualSense y nesC/TinyOS/TOSSIM permitiendo el desarrollo en un ambiente gráfico y una simulación al nivel de interrupciones de los programas actuales de TinyOS, con simulación de la red al nivel de paquetes, mientras permite a los desarrolladores usar el modelo de computación disponible para Ptolemy II [91].

5.1.6 GRATIS.

Es un entorno de desarrollo completamente funcional para el modelado, generación de código, verificación y análisis para el desarrollo de aplicaciones de TinyOS. La tecnología detrás de GRATIS es el modelo de computación integrado, específicamente el *Generic Modeling Environment* (GME). El sistema operativo y las interfaces de sus componentes usados junto con sus interdependencias son capturadas en un ambiente gráfico y el código introducido es el “pegamento” que mantiene junta a la aplicación y a los componentes del sistema operativo [92].

5.2 IDE de Sistemas Embebidos.

Los nodos sensores de las WSN son una plataforma embebida, con un sistema operativo para sistemas embebidos (Contiki, Mantis, SOS, etc.), así en cierto sentido cuando se trabaja con ellos las IDE para los sistemas embebidos vienen a embonar de forma natural como herramientas de desarrollo para las WSN.

Las herramientas más populares son:

- **AVR Studio** [93]. Es un IDE para la programación y depuración de aplicaciones en AVR en Windows. Tiene un administrador de proyectos, editor de archivos y simulador del microcontrolador. Además hace interfaz con emuladores In-Circuit y plataformas

de desarrollo disponibles para la familia de microcontroladores AVR de 8 bits RISC (Ver Figura 5.5).

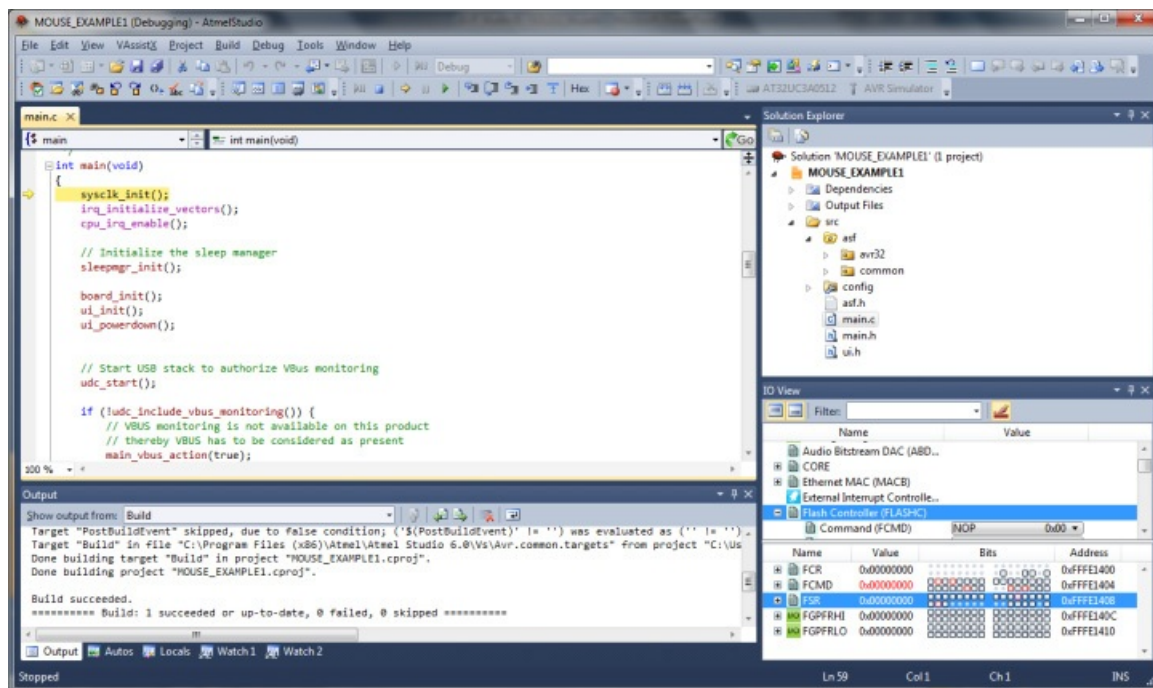


Figura 5.5 Plataforma de desarrollo AVR Studio.

Esto resulta interesante ya que varias de las plataformas de WSN están basadas en esta familia de controladores, una de las más populares es de la familia de MOTES mica.

- **MSP430 IAR** [94]. Este IDE para Sistemas embebidos está disponible para un gran número de microprocesadores y microcontroladores desde 8 a 32 bits. Lo cual permite a los desarrolladores moverse en el mismo ambiente de desarrollo independientemente de que la plataforma destino sea o no la misma a la anterior. Tiene una interfaz gráfica que facilita su uso de manera rápida e implementa al máximo las capacidades de herencia de código (Ver Figura 5.6).

Lo interesante en esta plataforma de trabajo es que ofrece un entorno de trabajo donde

están todas las herramientas de desarrollo integradas como:

- Administrador de proyecto.
- Un editor.
- Un compilador con alto índice de optimización de código (MSP430 IAR C/C++).
- Un ensamblador (MSP430 IAR Assembler).
- Un versátil enlazador (IAR XLINK).
- Una biblioteca extensa (IAR XAR y IAR XLIB).
- Un depurador (IAR C-SPY).
- Utilería de línea de comando.

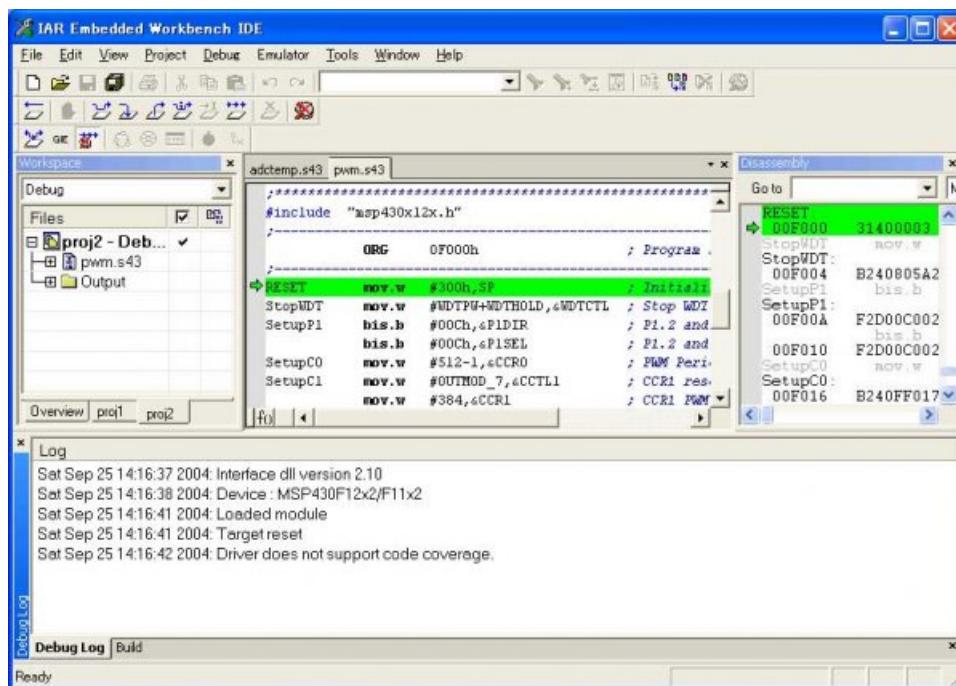


Figura 5.6 Plataforma de desarrollo MSP430 IAR.

Si se usan MOTES telos o similares es probable que se requiera de esta plataforma para desarrollar.

- **MSPGCC Toolchain.** Este es un grupo de herramientas de programación y desarrollo para los procesadores msp430. El único detalle es que todas son utilizadas desde la línea de comandos. Actualmente, los usuarios prefieren un *IDE*, sin embargo se puede instalar y adaptar uno de manera separada. Es importante denotar que esta herramienta es de licencia *open source* y además ha sido usada para desarrollar proyectos completos [96].

Capítulo 6

La máquina virtual de Java

La máquina virtual de Java, *JVM* por sus siglas en inglés, es la piedra angular en la tecnología de las plataformas que hacen uso de Java. Es la responsable de la independencia entre hardware y software, el tamaño pequeño del código compilado y la habilidad de proteger a los usuarios y sistemas de códigos maliciosos.

La JVM es una máquina de cómputo abstracto. Al igual que una computadora real, tiene un conjunto de instrucciones y manipula varias áreas de memoria en la ejecución de éstas [97]. La idea de implementar un lenguaje de programación que haga uso de una máquina virtual es muy conocida. Un buen ejemplo de qué tan amplia ha sido esta técnica es con la máquina de P-Code de la UCSD [98].

El primer prototipo implementado de la máquina virtual de Java, hecha por la extinta compañía Sun Microsystems, emulaba el conjunto de instrucciones de la JVM en software ejecutándose dentro de un Asistente Personal Digital (PDA). En la actualidad, las implementaciones de la JVM, con sus componentes, Java 2 SDK y Java 2 Runtime Environment, emulan en conjunto la JVM para computadoras con sistemas operativos Win32 y Solaris de formas mas sofisticadas. Sin embargo, la JVM no asume ninguna tecnología particular para su implementación, sistema operativo anfitrión o arquitectura de hardware, inclusive puede no ser necesariamente interpretado, puede ser implementado al compilar su código para las instrucciones específicas de un CPU o trabajarlo como microcódigo.

Es importante resaltar que la JVM no conoce absolutamente nada sobre el lenguaje de programación Java, sino más bien de un formato binario particular llamado *classfile*. Dentro de este formato binario (generalmente es un archivo) estan las instrucciones en el código de la JVM (*bytecodes*) y una tabla simbólica, además de otra información auxiliar.

Para garantizar la seguridad, la máquina virtual de Java impone formatos duros y limitaciones estructurales en el código del *classfile*. Sin embargo, cualquier lenguaje con la funcionalidad expresada en términos de un formato válido de *classfile* puede ser ejecutado por una JVM.

6.1 Definición

Para comprender la máquina virtual de Java, es necesario primero aprender que cuando se habla de ella puede estar refiriendo a tres cosas diferentes. Estas pueden ser:

- una especificación abstracta,
- una implementación concreta, o
- una instancia de las dos primeras ejecutándose.

La especificación abstracta es un concepto, descrito a detalle en [97], las implementaciones concretas pueden existir en diferentes plataformas y de distintos desarrolladores, ya sea inclusive como una combinación de hardware y software [99]. La instancia de la especificación implementada y ejecutándose suele ser un servicio ejecutando una aplicación de Java. Pero para la definición formal de la máquina virtual de Java se puede hacer uso de modelos desarrollados por las ciencias de la computación y en especial en uno llamado **máquina de Turing**.

6.1.1 La máquina de Turing

Creado en 1936 por Alan Turing, muy similar a un autómata finito, pero con un ilimitado tamaño de memoria. Es hasta ahora la aproximación más acertada de una computadora de propósito general, este modelo puede hacer todo lo que una computadora real puede hacer. Sin embargo, incluso una máquina de Turing no puede resolver ciertos tipos de problemas. En un sentido real, estos problemas están más allá de los límites de la computación teórica.

Las diferencias esenciales entre los autómatas finitos y la máquina de Turing son:

1. Puede leer/escribir en una cinta.
2. La cabeza lectora de la cinta puede mover en las dos direcciones (izquierda o derecha) la cinta.
3. La cinta es infinita (generalmente referida como una memoria ilimitada).
4. Los estados especiales de aceptación o rechazo toman efecto inmediato.

El corazón de la definición de la **máquina de Turing** es la función de transición (δ) porque es donde se establece como la máquina va de un estado a otro. Para la máquina de Turing, δ toma la forma de $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$. Esto es, cuando la máquina esta en cierto estado q y la cabeza lectora esta sobre una locación de la cinta que contiene un símbolo a , y si $\delta(q,a) = (r,b,L)$, la máquina escribe el simbolo b remplazando el a y se transita al estado r . El tercer componente es L o R e indica si la cabeza lectora se movio a la izquierda o la derecha despues de escribir [100].

En resumen, una máquina de Turing es una 7-tupla $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, donde Q, Σ, Γ son conjuntos finitos y

1. Q es el conjunto de estados,
2. Σ es el alfabeto de entrada que no contiene el simbolo de **espacio en blanco** $\sqcup$
3. Γ es la cinta con el alfabeto, donde $\sqcup \in \Gamma$ y $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$ como su función de transición,
5. $q_0 \in Q$ como el estado inicial,
6. $q_{accept} \in Q$ es el estado de aceptación, y
7. $q_{reject} \in Q$ es el estado de rechazo, donde $q_{reject} \neq q_{accept}$.

La máquina de Turing opera de la forma siguiente. Supongamos una máquina M , inicialmente recibe como entrada $w = w_1w_2...w_n \in \Sigma^*$ en el extremo izquierdo n cuadros de la cinta y el resto de la cinta esta en blanco. La cabeza comienza en el cuadro del extremo izquierdo de la cinta. Note que Σ no contiene el símbolo de espacio en blanco, al aparecer este marcaría el final de la cinta de entrada. Una vez que M ha iniciado, el cómputo procede de acuerdo a como esta escrito en las reglas en la función de transición. Si M siempre intenta mover la cabeza de lectura hacia la izquierda fuera de la línea de la izquierda de la cinta, la cabeza mantiene la misma posición, hasta que la función de transición indique L . La computación continuará hasta que entre en un estado de aceptación o rechazo, es ahí cuando se detiene. Si esto nunca ocurre, M seguirá para siempre computando.

Cuando una máquina de Turing computa, ocurren cambios en el estado actual, el contenido de la cinta en la locación en que se encuentra y la posición de la cabeza lectora. El conjunto de estos tres nuevos valores son llamados *configuración* de la máquina de Turing. Las configuraciones son representadas de manera especial. Para un estado q y dos cadenas u y v sobre la cinta del alfabeto Σ nosotros escribimos $u q v$ para la configuración donde el estado actual es q , el contenido de la localidad actual en la cinta es uv y la posición de la cabeza de lectura es el primer símbolo de v .

Por otra parte, a la colección de cadenas que una máquina de Turing (M) acepta es llamada **lenguaje de la máquina** (lenguaje de la máquina M) o **Lenguaje reconocido por la máquina** y es denotado como $L(M)$.

Cuando inicia una máquina de Turing en una entrada, tres resultados son posibles. La máquina podría *aceptar*, *rechazar* o conservarse en un *bucle*. Entiéndase por bucle que la máquina simplemente no se detiene, inclusive podría suponer un comportamiento simple o complejo que nunca lleve a un estado de detención.

6.1.2 La JVM como máquina de Turing

La especificación del lenguaje Java (Capítulo 6.2) establece que el lenguaje de programación Java es un lenguaje con **Gramática de Libre Contexto**. De esto se puede inferir que

el lenguaje de la máquina virtual de Java esta definido como:

$$L(M_{java}) = \{ w \mid w \text{ que puede ser generado por el CFG para Java en la especificación del Lenguaje Java} \}$$

Además de que

$$L(M_{java}) = \{ w \mid \text{un compilador de Java puede construir un arbol de analisis para } w \}$$

Este tipo de lenguajes son reconocidos por gramáticas de libre contexto, en donde las reglas de producción son de la forma $N \rightarrow (N \cup \Sigma)^*$. Además estos son reconocidos por un automata de pila *Pushdown Automata*. En este punto es necesario recordar que las máquinas de Turing se diferencian de los autómatas de pila en que la primera tiene una memoria que puede ser accesada en cualquier dirección, mientras que el autómeta de pila solo puede accesar el último elemento ingresado en la pila.

Se puede dar una descripción formal de cualquier sistema haciendo referencia a una máquina de Turing y especificando cada uno de sus siete componentes. Para esto, definire la JVM como una máquina de Turing, introduciendo a M_{java} , tal que reconozca el lenguaje dado por $L(M_{java})$ tal que $M_{java} = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, donde Q, Σ, Γ son conjuntos finitos y

1. Q es el conjunto de estados generados por las instrucciones de la *JVM* (aritméticas, control de programa, manipulación de pila y de modelo de objetos).
2. Σ es el alfabeto de entrada, con el *bytecode* de java, como instrucciones de la JVM y que no contiene el simbolo de **espacio en blanco** $\sqcup$
3. Γ es la cinta con el alfabeto, generalmente denotado como *memoria de programa* donde $\sqcup \in \Gamma$ y $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ como su función de transición denotado por el tipo de instrucción a la que pertenece,

5. $q_0 \in Q$ como el estado inicial e indicativo del *magic number* (0xCAFEBAFE),
6. $q_{accept} \in Q$ es el estado de aceptación para un *bytecode* valido, y
7. $q_{reject} \in Q$ es el estado de rechazo, donde $q_{reject} \neq q_{accept}$.

Aunque el modelo de Turing es una descripción muy cercana a una computadora real, esta no es la adecuada para describir completamente el funcionamiento de la JVM. Por ejemplo, la cinta de símbolos en la máquina de Turing puede ser recorrida en cualquier dirección (izquierda o derecha) y solo una posición; mientras que en la JVM, durante la lectura del *bytecode* (símbolo de la cinta), puede encontrarse con una instrucción que implique un salto de muchas localidades en cualquier dirección.

Por otra parte, el diagrama de transición de estados de la JVM (uno por cada posible *bytecode*) implica la ejecución de múltiples microinstrucciones para “satisfacer” la estado en él. La transición por ende no es tan simple como sería en el modelo de Turing, donde solo basta plasmar la dirección en la que se recorre la cinta, el simbolo que se lee y el que se escribe en la cinta; mientras que en la JVM implicaría una serie de “cambios” a nivel máquina en varios elementos internos (pila, contador de programa, apuntador a variables locales, etc.). Para comprender esto es necesario describir algunos de los estados de la JVM.

6.1.3 Estados de la JVM

Como ya ha sido comentado en la sección anterior, cuando se compila un programa en Java, el compilador genera un archivo en el formato de *classfile*. Dicho archivo contiene toda la información del programa (constantes, descriptores, métodos, etc.) además de las instrucciones en el *bytecode* de Java. Estas instrucciones, como también se ha comentado, implica la ejecución de un variable número de microinstrucciones dentro de la máquina.

Para ilustrar esto, analizemos los micro-estados de algunas instrucciones como *iadd* usada para la suma de dos números enteros e *if_cmpge* para el salto condicional dentro del programa.

El *bytecode* *iadd* es la instrucción usada por la JVM para la suma de dos números enteros. El indicativo de la instrucción esta dado por el valor *0x60* dentro “la cinta de simbolos” e

implica que los dos valores insertados en la pila de la máquina virtual son los valores a ser sumados.

Para “cumplir” con este estado, la JVM saca dos valores del tope de la pila, *value1* y *value2*, que deben ser enteros, los suma y el resultado (*result*) lo mete a la pila (ver Figura 6.1).

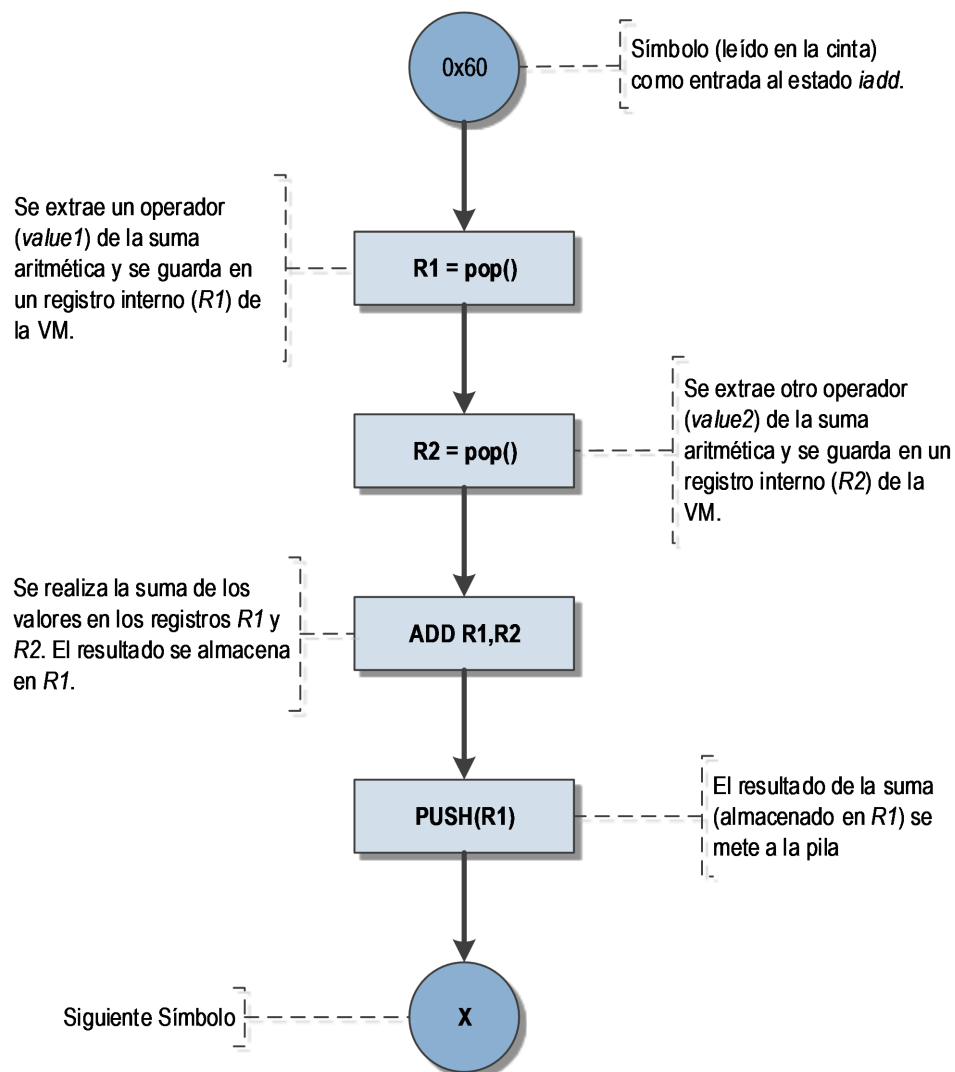


Figura 6.1 Micro-Estados del bytecode *iadd*.

En otras palabras, el bytecode *iadd* es el resumen de la operación:

$$\mathbf{StackTop} \Leftarrow S_{R1}(\mathit{StackTop}) + S_{R2}(\mathit{StackTop} - 1)$$

Donde el registro *StackTop*, que apunta al tope de la pila, señala el resultado de la suma de los dos valores extraídos de la pila.

Por otra lado, la instrucción *if_cmpge*, usada para el salto condicionado del programa (*Program Counter*) en caso de que un operando (*value1*) sea mayor o igual al otro (*value2*), es un ejemplo de como el movimiento de la “cinta” en la JVM puede no ser de una sola localidad. Este bytecode está señalizado por el símbolo *0xa2* en “la cinta de símbolos” e implica que para su ejecución los valores a ser comparados están en los dos espacios del tope de la pila. Para esto, la máquina extrae dos valores de la pila, *value1* y *value2*, que deben de ser números enteros. Si *value1* es mayor a *value2*, la JVM forma un número signado con los 16-bit siguientes en “la cinta” y lo suma al registro *contador de programa* (PC) modificando la localidad de la siguiente instrucción (símbolo) a ser leído. De lo contrario, si es menor, se omiten los dos símbolos consecutivos en la “cinta” y se continúa con la ejecución del *bytecode* (ver Figura 6.2).

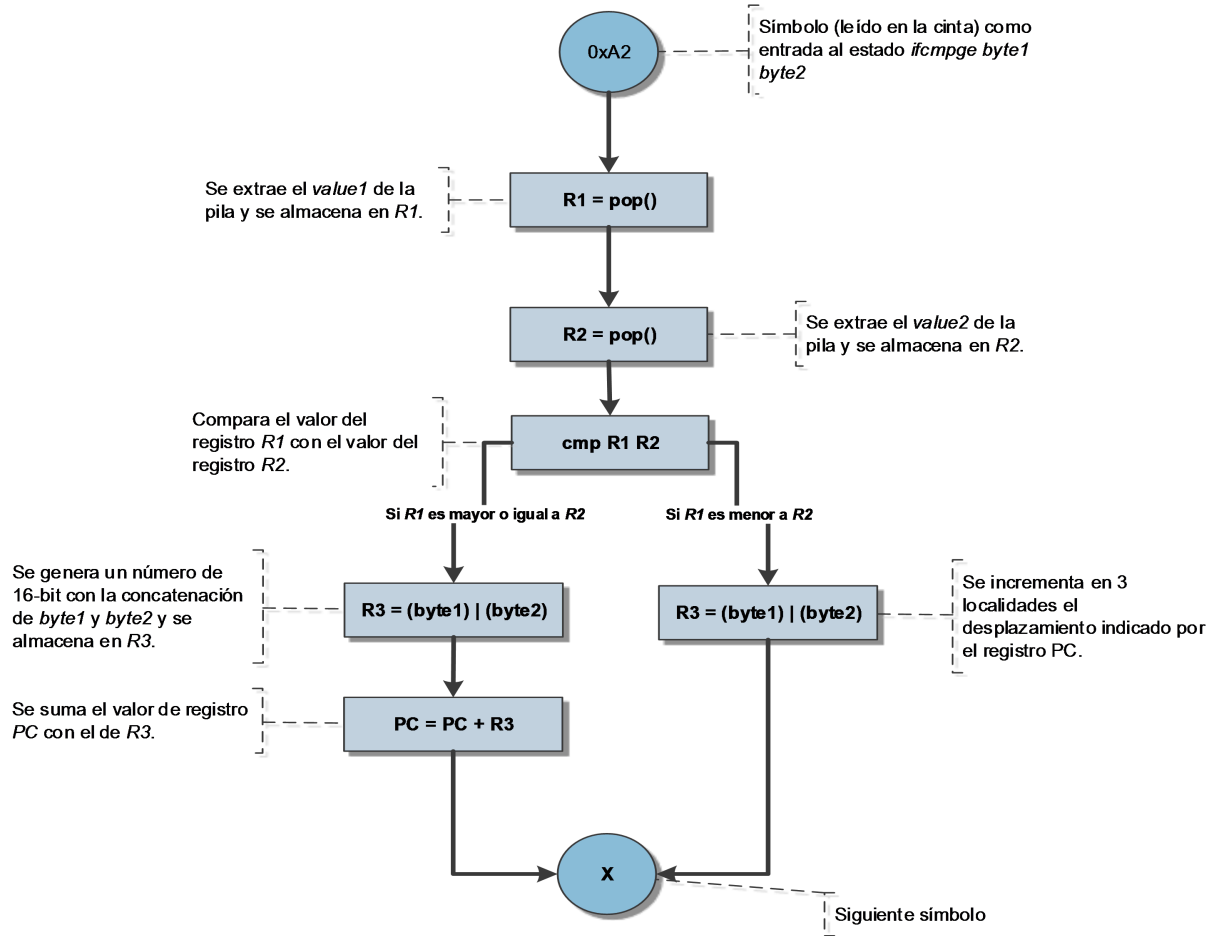


Figura 6.2 Micro-Estados del bytecode *if_icmpge*.

Agrupando todo el conjunto de operaciones que realizaría la JVM para satisfacer la transición a este estado quedarían de la siguiente forma:

$$S_{R1} \Leftarrow (StackTop)$$

$$S_{R2} \Leftarrow (StackTop - 1)$$

Donde los dos valores existentes en el tope de la pila, son almacenados en los registros *R1* y

$R2$. Para despues, realizar la comparacion entre estos dos valores de la forma

$$R_{FLAGS} \Leftarrow cmp R1 R2$$

Dependiendo de las banderas que fuesen activadas, el registro PC (contador de programa) es modificado. Si $R1$ es mayor o igual a $R2$, estaría dado por

$$S_{R3} \Leftarrow PC+1|PC+2$$

$$PC \Leftarrow PC + S_{R3}$$

De lo contrario por

$$PC \Leftarrow PC + 3$$

En resumen, cada uno de los *bytecode* de Java requieren de un variable número de micro-estados que la componen. Sería extremadamente largo explicar a detalle como se compone cada uno de ellos, pero una excelente referencia para comprenderla se puede encontrar en la especificación de la máquina virtual de Java [97].

6.2 El lenguaje de programación Java

Este lenguaje de programación fue originalmente llamado *Oak* y diseñado para ser usado en aplicaciones embebidas de consumo por James Gosling [101]. Despues de varios años de experimentar con el lenguaje y con contribuciones significativas por Ed Frank, Patrick Naughton, Jonathan Payne y Chris Warth este fué para ser usado en Internet.

El lenguaje Java es de propósito general, concurrente, basado en programación orientada a objetos y espécificamente diseñado para tener las menores dependencias posibles, por lo que tiene la filosofía “*compile once, run anywhere*”. Fue diseñado para ser lo suficientemente simple que muchos programadores puedan lograr fluidez en el idioma. Tiene mucha relacion en varios aspectos a C++, pero con un numero de ellos omitidos y algunas otras ideas de

otros lenguajes incluidos.

Actualmente sus desarrolladores y varios profesionistas del área consideran que ya es un lenguaje muy maduro y listo para su aplicación como lenguaje de programación en varios ámbitos. Sin embargo, también advierten que necesitara evolucionar en algunos aspectos en los años venideros cuidando la compatibilidad con las aplicaciones existentes. Es decir, los compiladores y sistemas deberán de ser capaces de soportar distintas versiones simultaneamente y con completa compatibilidad.

Es importante recordar que el lenguaje de programación Java es un lenguaje con *Gramática Libre de Contexto*. Aunque la especificación del lenguaje incluye a tres:

1. Gramática léxica. Define reglas gramaticales que se parecen a la definición de las reglas de gramática libre de contexto de Chomsky, pero interpretada de manera diferente. Si se interpretara como una verdadera gramática libre de contexto y luego de que la gramática es muy ambigua. Por ejemplo, la definición estandar de “ifthen” podría ser interpretada como (*keyword*) “if” seguido por (*keyword*) “then”, o como un (*identifier*) “ifthen”, etc.
2. Gramática Sintáctica. Esta es presentada en varios capitulos de la especificación [101]. Es descrita como “muy similar a la gramática LALR pero mas leíble”. Esto significa que se necesita más de una señal de búsqueda hacia delante para analizar y los autores esperaban que con este se describiese el mismo idioma que la gramática que se utiliza para construir analizadores actuales.
3. Gramática LALR. Esta es una gramática no tan leible como la primera, pero se ha comprobado mecánicamente sólo requieren una única señal de búsqueda hacia adelante.

6.3 La estructura del *classfile*

El *classfile* es un formato binario muy preciso de los programas de Java. Cada *clase* en Java es representado por uno de estos en una completa descripción. Por lo mismo, no se

puede poner más de una clase dentro de esta estructura. La definición precisa de la estructura *classfile* permite que cualquier clase sea cargada e interpretada de manera correcta por cualquier JVM, sin importar que sistema produce la clase o que sistema operativo aloja la JVM.

Tipo	Nombre	Cantidad
u4	magic	1
u2	minor_version	1
u2	major_verson	1
u2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interface_count	1
u2	interfaces	interface_count
u2	fields_count	1
fields_info	fields	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attributes_count	1
attributes_info	attributes	attributes_count

La estructura *classfile* contiene todo lo que la JVM necesita conocer acerca de una clase de Java o interfaz.

6.4 Instrucciones de la JVM

Una instrucción de la JVM consiste de un *opcode* que especifica la operación a ser desarrollada, seguida por cero o más operandos requeridos por la instrucción. Es necesario resaltar que dentro del conjunto de opcodes se han reservado tres para su uso para en la implementación de la JVM. Dos de estos son los numeros 254 (0xFE) y 255 (0xFF) con los nemónicos *impdep1* y *impdep2* respectivamente. Estas instrucciones son para proveer de “back doors” o trampas para una funcionalidad especifica implementada en software y hardware respectivamente. El tercer opcode reservado es el número 202 (0xCA) como un nemónico de *breakpoint* y como un intento para ser usado por depuradores al implementar “breakpoints”. Aunque estos opcodes han sido reservados, estos solo deben aparecer dentro de la implementación de la JVM y no aparecer en un *classfile* valido. El conjunto de instrucciones se pueden agrupar de acuerdo al tipo de operación que realiza.

6.4.1 De almacenamiento

Las instrucciones de almacenamiento transfieren valores entre variables locales y la pila. Estas son:

- Almacenar el valor de una variable local dentro de la pila: *iload*, *iload_ < n >*, *lload*, *lload_ < n >*, *fload*, *fload_ < n >*, *dload*, *dload_ < n >*, *aload*, *aload_ < n >*.
- Almacenar un valor de la pila en una variable local: *istore*, *istore_ < n >*, *lstore*, *lstore_ < n >*, *fstore*, *fstore_ < n >*, *dstore*, *dstore_ < n >*, *astore*, *astore_ < n >*.
- Almacenar una constante dentro de la pila: *bipush*, *sipush*, *ldc*, *ldc_w*, *ldc2_w*, *aconst_null*, *iconst_m1*, *i >*, *lconst_ < l >*, *fconst_ < f >*, *dconst_ < d >*

6.4.2 Aritméticas

Las instrucciones aritméticas calculan el resultado de una operación típica tomando los operandos de la pila y metiendo el resultado de esta operación en la misma. Existen básica-

mente dos tipos de instrucciones aritméticas: las que operan con valores enteros (*integers*) y las que operan con variable de punto flotante (*float*). Para cada uno de esos tipos la JVM tiene instrucciones especializadas en los dos tipos numericos. Es importante resaltar que no hay soporte directo para operaciones enteras en valores del tipo *byte*, *short* y *char* o en del tipo *boolean*; todas estas operaciones son cubiertas por instrucciones que manejen el tipo *int*. Tambien recordar que las instrucciones para los de tipo entero y de punto flotante varian en el comportamiento ante un *desborde* y *división por cero*. Las instrucciones aritméticas se componen por:

- Suma: *iadd*, *ladd*, *fadd*, *dadd*.
- Sustracción: *isub*, *lsub*, *fsub*, *dsub*.
- Multiplicación: *imul*, *lmul*, *fmul*, *dmul*.
- División: *idiv*, *ldiv*, *fdiv*, *ddiv*.
- Módulo: *irem*, *lrem*, *frem*, *drem*.
- Negación: *ineg*, *lneg*, *fneg*, *dneg*.
- Corrimiento: *ishl*, *ishr*, *iushr*, *lshl*, *lshr*, *lushr*.
- OR lógico: *ior*, *lor*.
- AND lógico: *iand*, *land*.
- XOR lógico: *ixor*, *lxor*.
- Incremento en variable local: *iinc*.
- Comparación: *dcmprg*, *dcmpl*, *fcmpg*, *fcmpl*, *lcmp*.

6.4.3 Creación y manipulación de objetos

Aunque ambas instancias de clase y las matrices son objetos, la JVM crea y manipula instancias de clases y matrices utilizando distintos conjuntos de instrucciones:

- Crear una nueva instancia de clase: *new*.
- Crear un nuevo arreglo: *newarray*, *anewarray*, *multianewarray*.
- Accesar campos de las clases (campos **static**, conocidos como variables de la clase) y campos de las instancias de la clase (campos **non-static**, conocidos como variables de instancia): *getfield*, *putfield*, *getstatic*, *putstatic*.
- Cargar un arreglo de componentes dentro de la pila: *baload*, *caload*, *saload*, *iaload*, *laload*, *faload*, *daload*, *aaload*.
- Almacenar un valor del operando de la pila como un arreglo de componente: *bastore*, *castore*, *sastore*, *iastore*, *lastore*, *fastore*, *dastore*, *aastore*.
- Obtener la longitud de un arreglo: *arraylength*.
- Verificar las propiedades de la instancia de una clase o arreglo: *instanceof*, *checkcast*.

6.4.4 De manipulación de pila

En la manipulación de pila se tiene instrucciones que realizan esto de manera directa: *pop*, *pop2*, *dup*, *dup2*, *dup_x1*, *dup2_x2*, *dup2_x2*, *swap*.

6.4.5 Transferencia de control de programa

Las instrucciones de control de programa condicionados o no condicionados provocan que la JVM continúe la ejecución de las instrucciones en una que este indicada por la transferencia de control. Estas pueden ser:

- Saltos condicionales: *ifeq, iflt, ifle, ifne, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmpgt, if_icmple, if_icmpge, if_acmpeq, if_acmpne*.
- Saltos condicionales compuestos: *tableswitch, lookupswitch*.
- Saltos no condicionales: *goto, goto_w, jsr, jsr_w, ret*.

6.4.6 Invocación de métodos y retorno

Las instrucciones para este tipo de acción para invocación son: *invokevirtual, invokeinterface, invokespecial* e *invokestatic*. Las instrucciones para retorno (dependiendo del tipo de dato) son: *ireturn, lreturn, freturn, dreturn* y *areturn*.

Capítulo 7

Java en las Redes Inalámbricas de Sensores

Las redes inalámbricas de sensores son un tipo de sistema embebido, el cuál opera con recursos muy limitados, en comparación a los que pudiésemos encontrar en una computadora personal. Y al igual que todo tipo de sistema, requiere del desarrollo de un software para su operación. Este sistema, puede ser desde un simple control de apagado/encendido hasta el sistema de control en una ciudad inteligente y la posibilidad de soporte para su ejecución es una enorme variedad de hardware, sistemas operativos y estilos de programación.

A pesar de esa variedad tienen elementos comunes. Por ejemplo, estos sistemas son desarrollados para sistemas basados en arquitecturas de uni-procesador y sistemas operativos.

En lo que respecta al software, debe de ser escrito para ser analizable, con un control de flujo simple y sin reserva dinámica de memoria; esto implica un análisis formal de el software, de no tenerlo lo hace impredecible. Por último, es muy común que en este tipo de sistemas los programadores elijan lenguajes de programación que son subconjuntos del lenguaje C o Ada y para algunos aventureros C++.

La elección de estos lenguajes de programación funciona bien, pero el éxito en la programación de los sistemas dependerá grandemente de la experiencia que tenga el programador para desarrollar este tipo de software y como se menciona en [102] "... el tamaño del código del software se ha estado incrementando en miles de lineas de instrucciones y factores como la productividad, resusabilidad y la disposición de personal capacitado a despertado el interés en Java como una alternativa a los lenguajes de bajo nivel y motivado al desarrollo de extensiones del mismo...".

Esto debido más que nada, a la necesidad de simplificar la codificación del software y la no disposición de personal capacitado. Prácticamente, con la adopción de Java se obtiene acceso a una creciente comunidad de programadores que acostumbrados a desarrollar para una máquina virtual pueden desarrollar diversos tipos de sistemas para ambientes que solo requieren poseer como ambiente de ejecución una máquina virtual de Java.

A pesar de esto, se tiene que reconocer que el éxito de la introducción de Java como lenguaje de programación, ha sido muy limitado en los sistemas embebidos de tiempo real suaves y desastroso en los sistemas de tiempo real duro por múltiples razones. En un estudio hecho en el proyecto *Golden Gate* por la NASA/GPL se concluyó que el tiempo de desempeño de Java fué entre 10 a 100 veces mas lento que C. Conclusiones, que también fueron confirmadas por un estudio independiente, en el que se indica la región del sistema hecha para la administración de la memoria era difícil de usar y propensa a errores. Esto fue resuelto después por investigadores de la IBM en su trabajo publicado aquí [103], al colocar un mecanismo colector de basura (*garbage collection*) en tiempo real para la máquina virtual siendo ahí donde Java ganó la verdadera atención.

Por otra lado, algunos requerimientos pertinentes también a las redes inalámbricas de sensores, como la actualización dinámica de código, manejo en redes, interoperabilidad, corto lanzamiento al mercado, entre otras listadas en [105]. Fomentan la necesidad de un lenguaje de programación independiente de plataforma. Con el lenguaje de programación Java, al tener el programa compilado en el *bytecode de java*, se puede realizar cada una de las tareas listadas anteriormente. Básicamente fue creado en función de dar respuestas ha esto.

No hay que dejar de lado que la sencillez en la programación de los sistemas es importante. Dado el gran avance que hay hasta estos momentos en el desarrollo de la computación, el desarrollo del software, no debe de ser exclusivo de profesionales en el área y como se comenta en [106] “...el uso de aproximaciones de código más ligero en los lenguajes de programación de alto nivel rinden en un desarrollo más rápido de aplicaciones que los demás lenguajes de sistemas...”. Más adelante en la citada referencia se comenta también, que la facilidad de programación ya no debe ocupar lo que antes eran meses de entrenamiento, si no darle

entrada a los que llamaríamos *programadores casuales*, estos profesionales, hacen uso de la tecnología, a la que pueden programar, para realizar tareas profesionales. Por ejemplo, podríamos tener a un Biólogo recolectando información de un determinado ecosistema o a un arquitecto vigilando el estrés que sufre una estructura.

Al momento, todos los esfuerzos con respecto a la tecnología de Java, se han centrado en optimizarlo y adecuarlo en base a las nuevas oportunidades que oferta la tecnología reciente. Estos esfuerzos han llevado a la creación de sistemas embebidos de bajo costo cercanos a la viabilidad comercial, tanto es así, que empiezan a desarrollarse algunos productos habilitados para el uso del lenguaje Java como en [107] con tarjetas inteligentes programables en Java.

Por último, en lo que respecta en el campo de formación de profesionales, algunas universidades adoptan su uso dentro de sus programas de estudio. Esto involucra la enseñanza de la programación de sistemas embebidos haciendo uso del lenguaje Java, a fin de desarrollar experimentos con protocolos y algoritmos dentro de las posibilidades de los sistemas embebidos como se muestra aquí [108]. En cierta forma, con esto se reconoce el potencial que se tiene al desarrollar profesionales en el uso de la misma.

La adopción del lenguaje Java no es por motivo de boga. Para comprender mejor las razones para su estudio e implementación como lenguaje de sistemas embebidos se tienen que analizar a detalle cada una de las características que ofrece como tal y la manera en la que esto acelera enormemente la programación de los mismos.

7.1 Características de la programación basada en Java

7.1.1 Simplicidad

Aún muchos de los sistemas embebidos siguen siendo desarrollados en el lenguaje C e inclusive C++. Ya que escribirlos en él, es más productivo que hacerlo en el lenguaje ensamblador, pero falla al momento de la programación libre de errores. Esto es debido más que nada por sus mecanismos de administración de memoria, mediante la cuál los

programadores reservan y liberan la memoria.

Por otra parte, C++ es muy rico en características y complejo que lo hace difícil de aprender; dificultando el encontrar y retener los desarrolladores de sistemas embebidos que son buenos en su uso. Pero con lo que respecta al lenguaje Java, es simple de aprender, además hace uso de una sintaxis muy parecida a la de los lenguajes de programación, como por ejemplo, la usada en C (C++). Sin embargo, muchos de los constructos complejos han sido removidos. Aparentemente, sin reducir la utilidad del lenguaje.

Un buen ejemplo de la complejidad eliminada del lenguaje son las interfaces, además de la herencia múltiple (aunque es cuestionado su uso dentro de los sistemas embebidos por algunos desarrolladores). También el manejo de múltiples interfaces, que es casi igual de poderoso que la herencia múltiple. Sin embargo, con esto se evita problemas como podría ser el heredar mas de una vez de la misma clase.

Sumado a esto, el estandar de APIs que provee el lenguaje, puede incrementar enormemente su complejidad de aprendizaje, pero dado que no todas son adecuadas para su uso y estas se reducen a unas cuantas que involucran el control del hardware existente en la plataforma. En vez de hacer complejo su aprendizaje lo simplifica enormemente.

Para finalizar, el lenguaje Java es tan sencillo de aprender que su comunidad de usuarios sigue creciendo rápidamente. Lo que se puede entender en que es sencillo encontrar profesionales calificados en el uso del lenguaje Java. Por lo tanto, otros interesados pueden participar mas fácilmente en el desarrollo y aseguramiento de la calidad en proyectos de software como lo explican en [109].

7.1.2 Programación orientada a objetos

La programación orientada a objetos se ha convertido en el paradigma predominante en la creación de software. Este paradigma acelera de manera significativa el ciclo de desarrollo. Dentro de las redes inalámbricas de sensores, es de gran practicidad su adopción ya que este tipo de sistemas trabajan adquiriendo datos del mundo real tal y como el paradigma orientado a objetos abstrae las cosas del mundo real en forma de objetos y clases.

Sin embargo, las razones más importantes por las cuales es beneficioso en las redes de sensores son:

- **Se abstrae el sensor en clases.** Los objetos encapsulan y esconden los detalles acerca de la implementación y diseño de datos. Mucha de la información a ser manejada por el algoritmo de procesamiento en la red de sensores tienden a estar estrechamente relacionada con la disposición del hardware. Esto hace que sea difícil de reutilizar el código y, finalmente, se ralentiza cualquier proceso de desarrollo. Mediante el uso de las clases abstractas genéricas de sensores y con algoritmos genéricos, por ejemplo en los datos de los sensores, se puede desarrollar y unir a una instancia de hardware concreta en una etapa tardía, incluso en tiempo de ejecución. En niveles más abstractos se puede utilizar el polimorfismo a través de una interfaz genérica.
- **Serialización.** La encapsulación de los datos es valiosa cuando se trata de comunicarlos. Estableciendo un esquema como lo sería en un objeto es suficiente para serializar los datos del sensor con el código de manera conjunta y transmitirlo a otra parte en la red. Esto es debido a que todo está dentro de un solo objeto. El sistema puede claramente separar la sección de datos y el código reduciendo el sobrecargado del sistema receptor al mínimo.

7.1.3 Rigidez de tipo de dato

Una de las ventajas adicionales de la programación y procesamiento de datos en el lenguaje Java sobre otros lenguajes de programación imperativos es su rigidez en el tipo de dato. Esto es muy importante en los ambientes de programación para microcontroladores, ya que no hay protección forzada de memoria o necesariamente un sistema operativo.

La ocurrencia de una violación en el tipo de datos dentro del código del usuario puede tener un efecto negativo en la estabilidad dentro del sistema. Esto ha sido plasmado en el proyecto SPIN 7 [104], donde se demuestra que la rigidez en los tipos de datos permiten la ejecución del código con privilegios de sistema.

Dentro del lenguaje Java se tiene *rigidez en el tipo de dato*. Esta rigidez es implementada mediante el compilador del lenguaje, en el cuál, se puede verificar por muchos de los errores comúnmente realizados. Uno de estos errores puede ser el paso de argumentos de tipo erróneo, que puede llevar a la pérdida o adhesión de un signo aritmético, problemas de aritmética de apuntadores, entre otros.

De hecho, algunos investigadores afirman que debido a esta fuerte verificación de código por parte del compilador de Java, se tiene más conocimiento sobre la forma en que el código se ejecutará y por lo tanto se le puede aplicar optimizaciones más agresivas. En resumen, podría ser que el código a ejecutarse generado por el compilador de Java corra más rápido que el código generado por un compilador de C. Esta es una de las razones principales para enfocarse en el lenguaje Java en vez de la implementación de un sistema operativo robusto como entorno de ejecución.

Por último, dentro del lenguaje Java existe un conjunto básico de subsistemas críticos, por ejemplo el acceso a bus de datos, que es verificado vía el compilador de Java. Esto básicamente es una capa de abstracción muy ligera. Dejando a las capas superiores de la abstracción tareas de otro tipo como lo serían el muestreo de datos de los sensores y su conversión, las cuales pueden ser cargadas y ejecutadas dinámicamente dentro de la máquina virtual. Esto es muy ventajoso, ya que permite una arquitectura de sistema con diseño reconfigurable y adaptable.

7.1.4 Manejo de Excepciones

El diseñador de software puede definir a nivel de aplicación la excepción que desee atrapar. Para esto, el lenguaje Java ofrece constructores, que pueden ser tratados por los desarrolladores en niveles por debajo de la aplicación y de forma transparente para el desarrollador. Por supuesto, el lenguaje también ofrece constructores para el fácil manejo de excepciones al nivel definido para ese propósito. Como medida extra de seguridad, las excepciones que ocurran serán atendidas por el entorno de tiempo de ejecución (la máquina virtual).

7.1.5 Comprobación de límites de arreglos

A nivel de código se puede garantizar que no serán violados los límites de los arreglos. Los desarrolladores de software, como buenas practicas, deben de checar que los límites de los arreglos no sean rebasados. Pero en caso de que este falle, en un último momento, el estado del sistema no se corrompe.

Esto es debido a que los mecanismos de excepciones son estandarizados para cada una de estas situaciones. Además del manejo de los límites de los arreglos por parte del lenguaje, lo cuál puede ser muy práctico cuando se manejan bucles de ejecución. Inclusive, las excepciones poseen mecanismos que terminan legalmente los bucles (aunque esto no muy recomendado).

7.1.6 Administración automática de memoria

La memoria es un recurso limitado e importante dentro de los sistemas embebidos, por ello debe de ponerse especial cuidado en su uso a fin de tener mecanismos óptimos para su manejo. Esto es complicado cuando se tiene aplicaciones cuyos requerimientos cambian radicalmente durante su ejecución y la asignación estática de memoria la pueden volver ineficiente y difícil de mantener [110] [111].

Por otro lado, lidiar con el uso de la memoria utilizando elementos previamente listados a ser procesados de forma dinámica puede permitir el uso de la memoria de forma más eficiente. Sin embargo, si el sistema es *multithread* se vuelve *no deterministico* imposibilitando este mecanismo [112] [113].

En lo que respecta al uso del lenguaje Java, no existe una instrucción que realice una reserva explícita de memoria. Esto deja al programador sin acceso directo a este recurso. Para poder hacer uso de ella solamente es durante la *creación de objetos* y dejan la tarea de recuperación de la memoria al sistema operativo, quien hace apropiadamente el reclamo de la memoria.

Otros lenguajes (incluyendo C) actualmente no poseen constructos para la reserva y recuperación de memoria como parte de su lenguaje; para esto hacen uso de una biblioteca

de funciones. Pero en Java y C++, la reserva de memoria es parte del lenguaje y en Java además el colector de basura (*garbage collector*) también es parte del lenguaje, pero su construcción explícita ha sido omitida a propósito.

Desde la perspectiva del desarrollador, el manejo no explícito de memoria permite un nivel de abstracción donde el programador puede crear objetos a discreción dejando la tarea de recuperación de memoria al entorno de ejecución. Este mecanismo es uno de los ejemplos más representativos de la abstracción de hardware hecha por el lenguaje Java.

7.1.7 Independencia de plataforma

Con el incremento en la complejidad de los sistemas embebidos, los enfoques tradicionales de diseño no proveen suficiente soporte para el desarrollo de esos sistemas. Esto es debido a los problemas descritos en [114]. De hecho, los detalles de *bajo nivel* dentro del enfoque tradicional no otorgan facilidades para realizar una abstracción adecuada del hardware. En ese sentido, el diseño en los sistemas embebidos sufren de un “*efecto mariposa*”, como puede ser una omisión pequeña que podría resultar en un problema mayor.

A fin de hacer frente a este problema, los enfoques de diseño deben ofrecer abstracciones adecuadas, protegiendo el diseño de los detalles existentes en la plataforma. Para esto, los enfoques de *independencia de plataforma*, como se muestra en [115] y [116] son idóneos.

En este sentido, en la programación basada en Java, la aplicación es compilada a un lenguaje intermedio (llamado *java bytecode* o *JBC*). Esta no es una característica del lenguaje de programación. De hecho, un programa escrito en Java puede ser compilado perfectamente directamente en código nativo de cualquier procesador [117]. En principio, no es posible compilar programas hechos en C y C++ a JBC, porque Java no requiere del *JBC* para ofrecer manipulación directa de memoria.

En gran medida, esta independencia de plataforma se debe al entorno de ejecución llamado *máquina virtual de Java*, el cuál posee un intérprete y en su combinación resulta en un enorme ahorro de costos de desarrollo ya que solo se ocupa una única versión para plataformas distintas.

7.1.8 Enriquecido conjunto de APIs estandarizados

Este conjunto, incluyendo sus implementaciones (mayormente en Java) fué liberado junto con el lenguaje de programación. También es conocido como *clases estándar de Java*.

La idea principal de este conjunto, es que los desarrolladores de software pueden asumir que esta biblioteca está disponible para cualquier plataforma con capacidad de ejecutar el *JBC*. Además de que los desarrolladores pueden concentrarse en resolver los problemas reales, sin preocupación de la plataforma final donde se ejecutará o gastar tiempo en estudiar el conjunto de APIs disponibles para diferentes plataformas.

Pero una de las razones más destacables de este atributo, es que el uso de clases fuerza a que se codifique las aplicaciones de forma *modular* alentando aún más la abstracción de datos [118].

En lo que concierne al uso del lenguaje Java para los sistemas embebidos, no es posible la existencia completa de las clases estándares debido a que no todas son utilizables dentro de este ámbito. Sin embargo, un subconjunto de estas es muy apreciado dentro la misma y más aquellas que tienen que ver con aspectos del hardware como lo sería la comunicación, manejo de temporizadores, flujo de datos, hilos de ejecución, entre otras.

7.1.9 Compacto

Se ha argumentado que, incluso para sistemas embebidos basados en ROM y donde la portabilidad no es un problema, todavía tiene sentido utilizar una aplicación basada en el *JBC*, ya que los *bytecodes* son supuestamente más compacto que el código nativo [118] [119]. Sin embargo, no esta muy claro si esto es realmente el caso. Por ejemplo, una programa para calcular números Fibonacci compilado en *JBC* ocuparía 134 bytes, mientras que uno compilado para *i86* y *Sparc* tomarían entre 373 y 484 bytes respectivamente (asumiendo que las clases requeridas estuviesen también precompiladas; de otra forma, serían necesarios 417 y 540 bytes respectivamente). Sin embargo, dentro del archivo compilado de Java es necesaria una tabla con información necesaria conocida como *tabla de constantes*. Esta tabla

incrementa el tamaño de la aplicación Fibonacci compilada a 1227 bytes y la información requerida en tiempo de ejecución depende de la información simbólica que se representa. Por otra parte, este código precompilado también necesita espacio para estructura de datos que ayuden a manejar esta información -alrededor de 520 bytes para el ejemplo de Fibonacci-.

En conclusión, es que la ventaja sobre el espacio usado por los *bytecode* va de menor a mayor, pero introduce una pérdida significativa en la velocidad de ejecución.

7.1.10 Multihilo

El lenguaje de programación Java fue diseñado para dar soporte a la programación multihilo tanto en su conjunto de clases estándar como en la propia especificación de la máquina virtual de java (cuya implementación a nivel nativo no necesariamente tiene que ser cubierto por ella). Para esto, se hace uso de la clase *thread* que permite desarrollar a nivel de aplicación soluciones que tengan un enfoque en uno o varios procesadores con tiempo y recursos compartidos.

Además, esta fuertemente orientado a la reutilización de código. Esto implica que los diseñadores de las *clases de Java* que hereden de la clase *thread* deben de tener siempre en cuenta que múltiples hilos pueden desear acceder al mismo tiempo estructuras de datos internas de estas clases. Cada objeto que desee acceder al mismo tiempo tiene que ser protegido contra varios subprocesos que interfieren con cambios de los demás. Por lo tanto, los objetos Java son *sincronizados* de una forma conservadora.

Las operaciones de sincronización implica el atender e interpretar demandas, a nivel de sistema operativo, en el reclamo de acceso exclusivo a determinados recursos. Estas llamadas al sistema operativo son muy costosas. Pero una gran cantidad de tiempo puede ser ahorrado si se piensa en el hecho de que la interferencia real es muy rara.

7.1.11 Seguridad

La seguridad es otro de los beneficios principales de Java. Ambos, el lenguaje y la plataforma de Java fueron diseñados desde el principio pensando en la seguridad. La plataforma de Java permite a los usuarios trabajar con código “*inseguro*” y ejecutarlo en un ambiente en el cuál no puede causar daño la ocurrencia de una instrucción que haga referencia invalida a una dirección de memoria o recurso reservado.

Para lograr esta tarea, la aplicación compilada de Java se ejecuta en un ambiente de *máquina virtual* donde se emula un procesador real o un sistema operativo. La *máquina virtual de Java* provee un ambiente en el cuál el *JBC* puede ser ejecutado, manejar de forma automatica la ocurrencia de excepciones e inclusive la independencia de hardware.

7.2 Costos en el uso del lenguaje Java

En lo que respecta al costo de producción de dispositivos que adoptan Java a fin de poseer sus características como la sencillez de lenguaje, tipado fuerte o manejo de excepciones entre otros no implica costo alguno sobre ellos. Pero la introducción de las demás características, sí, listadas a continuación:

- **Penalidad en el desempeño.** Un ejemplo de esta penalidad está en que el lenguaje Java especifica que cada acceso a un arreglo debe de ser verificado para evitar que se sobrepasen los limites del mismo. Un coste exacto de esta acción es complicada pero imagínese el caso en que el arreglo es accesado 40 veces, esto implica que se realizó la verificación de limites 40 veces cuando quizás con una hubiese sido suficiente.
- **Requerimientos de memoria elevados.** Este requerimiento es debido a tres factores:
 - La máquina virtual de Java es una pieza relativamente grande de software. Algunas implementaciones llegan a ocupar 100 KB de memoria y a medida de que se

le incluyan más mecanismos puede incrementarse hasta 500 KB. La opción a fin de reducir esto, es el desarrollo de un subconjunto de la máquina virtual de Java.

- Un sistema completo basado en Java requiere cerca de 9 MB de biblioteca de clases. Por varias razones, esto debe de ser alojado en una memoria re-escriturable y en ambientes de redes este código es elegible para ser actualizado periódicamente.
- La administración de la memoria en Java es relativamente costosa (en términos de uso). Esto es debido a las prácticas de programación y a la acción del *garbage collector*.

Actualmente, los requerimientos minimos para una implementación completa de Java es 10 MB de ROM y 2 MB RAM.

- **Incremento en la complejidad del sistema.** El costo ligado a este punto es difícil de contabilizar de una manera genérica. Pero el escenario más simple es donde una ejecución eficiente e interfaz gráfica no es necesaria. Este escenario involucra solamente portar “el esqueleto” de la máquina virtual de Java con su intérprete a la plataforma final. Un experimentado ingeniero de software puede ocupar medio año-hombre en portar, probar y verificar la pila de la KVM proporcionada por la compañía Sun. Por otro lado, en un escenario más realista y complejo, como el presente en los dispositivos de consumo como los móviles, en donde además es requerida una interfaz grafica para aplicaciones simples, el costo del ingeniero de software llega por encima de 2 año-hombre. Sumado a esto, el mantenimiento podría costar alrededor de medio año-hombre anualmente.

7.3 Reducción de costos en el uso del lenguaje Java

La elección de una tecnología depende directamente de los cuellos de botella que se pueden encontrar durante el desarrollo de una aplicación. Por ejemplo, no tendría sentido realizar una optimización en la sincronización de hilos de ejecución en dispositivos embebidos

pequeños de los que no se espera llevar a cabo ejecución multihilo. Sin embargo, en muchos casos es conveniente escribir el *bucle de interpretación* en ensamblador por ser donde la JVM gasta mayor tiempo de su procesamiento.

Cuando se analiza la tecnología a fin de reducir los costos en el uso del lenguaje Java se pueden encontrar soluciones que involucran tanto hardware como software. A fin de enfocarnos en un único tipo de solución se tocarán exclusivamente en las soluciones en software. Estas posibles soluciones son:

- **Compiladores JIT.** Los compiladores JIT (de las siglas en inglés Just-in Time) [120] categorizados como soluciones de software genéricas para incrementar la velocidad de ejecución de Java y el coste en el incremento de uso de la memoria. Esto aún es cuestionable aunque decremente los costos en ejecución. Si la memoria es más cara que el procesador esto no tiene sentido.

Paradójicamente, son también llamados compiladores “muy tardados”, dado que ellos inician la compilación del *JBC* en el momento exacto en que el usuario o el sistema lo requieren. Esto es muy disruptivo en operaciones de tiempo real.

A fin de prevenir este comportamiento tardío y reducir el costo de la memoria de los compiladores JIT, compiladores JIT basados en perfiles fueron introducidos [121]. Este compilador, puede contener un sistema de ejecución basado en interprete. Inicialmente, todo el código es ejecutado por un interprete. Para cada llamada a un bloque de código se lleva un conteo de ocurrencias, cuando este conteo sobre pasa una medida establecida el bloque de código es compilado nativamente.

- **Subconjunto de interpretes.** Esta es una optimización que busca reducir el uso de memoria, usualmente a expensas de desempeño. Un ejemplo de esto esta en [121] donde un interprete no soporta por completo el conjunto completo de *JBC*.

Lo mismo va para la implementación de las bibliotecas estándar de Java. Generalmente, esta restringido en términos de interfaces de usuario.

Capítulo 8

Implementación de la plataforma para desarrollo basado en Java

El término de plataforma y Framework es genérico, y se hace uso de él refiriéndose en si a un “*ambiente de trabajo y ejecución*”, pero en general se habla de soluciones completas que contemplan tanto herramientas de apoyo a la construcción de aplicaciones (con ambientes de desarrollo) como a motores de ejecución. En la programación de las aplicaciones de software se involucra conocimiento y experiencia, haciendola una tarea compleja. Cuando un desarrollador se enfrenta a la programación de un sistema del tipo que no ha creado con anterioridad, tiene que reaprender muchas de las prácticas que implican este nuevo tipo de aplicación. Además de aquellos requisitos de rendimiento específicos que tienen impacto directo en el cómo y en que se desarrolla. Esto por lo general, se traduce en la introducción de errores, debido en mayor parte, a las restricciones generales del sistema mas que al manejo del lenguaje de programación generando retardos en los tiempos de entrega.

La utilización de metodologías de *Desarrollo Rápido de Aplicaciones* (R.A.D. por sus siglas en inglés) ha buscado resolver en gran medida el problema, sin embargo, estas siguen siendo herramientas genéricas que no tienen un propósito de programación específico; aceleran el desarrollo en ambientes visuales, pero no en la programación de aplicaciones embebidas. Para este tipo de aplicaciones se requiere de herramientas que auxilien en el desarrollo y pueden incluir:

- Librerías.
- Entornos de ejecución.

- Depuradores y
- Programas de auxilio para configuración.

Son en estos puntos donde la creación de una plataforma de desarrollo toma importancia. Además de las ventajas que se derivan de su uso como:

- El programador **no necesita plantearse** una estructura global de la aplicación, sino que el framework le proporciona un esqueleto que hay que “rellenar”.
- Facilita la **colaboración**. Cualquiera que haya trabajado dentro de la plataforma es sencillo que reconozca, pueda entender y modificar el código desarrollado para una aplicación; por tanto, todo lo que sea definir y estandarizar va a ahorrar tiempo y trabajo a los desarrollos colaborativos.
- Es más fácil desarrollar **herramientas** que puedan ser adaptadas a la plataforma.

Es claro que un programador puede desarrollar una aplicación sin uso de una plataforma o puede ser tan pequeña que no sea considerada necesaria. Sin embargo, a medida que la aplicación crece, un programador debe procurar seguir determinadas pautas que faciliten el trabajo de desarrollo y mantenimiento. La evolución natural es que de algún modo, se construya su “*propio framework*”.

También es importante resaltar que el uso de la plataforma implica un cierto coste inicial de aprendizaje, aunque a largo plazo se traduce en mayor facilidad al momento de desarrollar y mantener la aplicación.

8.1 JavELiNA.

La plataforma JavELiNA (por sus siglas en inglés *Java for Embedded Light Networks Applied*) esta formada por la 3-tupla:

$$F = \{M, \beta, \Phi\}$$

Donde:

1. M es un middleware (MiniVM) del tipo máquina virtual capaz de ejecutar el bytecode de Java.
2. β es el conjunto de *clases* que conforman la librería de programación para el nodo WSN, este elemento es referido como *Framework de Aplicación*.
3. Φ la pila de comunicación estandarizada, desde la perspectiva de usuario, sobre BSD-Socket.

Esta es una descripción general respecto a la composición de *JavELiNA*, ya que cada elemento de la tupla tiene su complejidad y en las siguientes secciones se describe a más detalle.

8.2 MiniVM.

Como capa intermedia, la MiniVM (middleware) llevará una completa abstracción del hardware existente en la plataforma y esta se responsabilizará por proporcionar las interfaces apropiadas, proveerá de un ambiente de ejecución y posee métodos para lograr un uso eficiente y adaptativo de los recursos del sistema. En su implementación, el acceso al hardware existente en el nodo-SR, es a través de los métodos nativos, las cuales son pequeñas piezas de software embebido dentro del código de la MiniVM usadas en el manejo de recursos y que son accesibles desde la aplicación (en Java), obviamente haciendo uso de un objeto nativo.

Desde la perspectiva de programación, es necesario recordar que esta es realizada haciendo uso del lenguaje de programación Java, el cual es compilado a un programa “de bajo nivel” a ser ejecutado por la máquina virtual y dentro de este proceso se tiene varias etapas, en las cuales, es analizado el programa de entrada realizando una revisión en búsqueda de evitar errores existentes. Si no es encontrado alguno, este será traducido. Durante la traducción, el programa es transformado de su código fuente a una serie de mnemónicos de la middleware

de Java conocidos como *bytecode* de Java y empaquetado en un formato estándar llamado *classfile* el cual es reconocido e interpretado por la MiniVM. Para ejemplificar este proceso en la Figura 8.1 es ilustrada la transformación de un ciclo *for*.

```
//código fuente en Java
...
    public void for99(){
        for(int i=0;i<99;i++);
    }
...
//traducción al bytecode
Method void for99()
0   iconst_0           //una constante cero es insertada en la pila.
1   istore_1           //el valor extraído de la pila es almacenado en la variable de índice 1.
2   goto 8             //se salta el contador de programa (PC) a la línea número 8.
5   iinc 1 1           //incrementa la variable local 1 en 1 (i++).
8   iload_i            //lee el valor actual de i e inserta su valor en la pila.
9   bipush 99          //inserta la constante 99 en la pila.
11  if_icmplt 5         //compara los dos valores en el tope de la pila y salta si es necesario (i<99).
14  return             //regresa al valor del contador de programa (PC) anterior a la llamada del método.
```

Figura 8.1 Ciclo *For* expresado en el código fuente de Java y su correspondiente *bytecode*.

Las instrucciones del *bytecode* muestran un ciclo *for* el cual manipula la pila y los valores de las variables locales. Las instrucciones *iconst*, *iload* y *bipush* insertan valores en la pila. Las instrucciones *istore* e *iinc* actualizan el valor en las variables. La instrucción *goto* e *if_cmplt* (si un valor entero es menor que) transfieren el control a la instrucción señalada en el byte siguiente. Obviamente, todo este trabajo de manejo y secuencia lógica debe ser realizada por MiniVM.

8.2.1 Arquitectura de MiniVM.

Actualmente la implementación de MiniMV es compatible con Java. Para esto consta de tres módulos especializados de software, además de ser auxiliado por un núcleo de sistema basado en LEARN [48] y llamado Java RT el cual esta enfocado en el tratamiento de Tiempo Real en Java como la administración de Threads de ejecución y métodos nativos para el manejo (de manera independiente de MiniVM) de los recursos existentes.

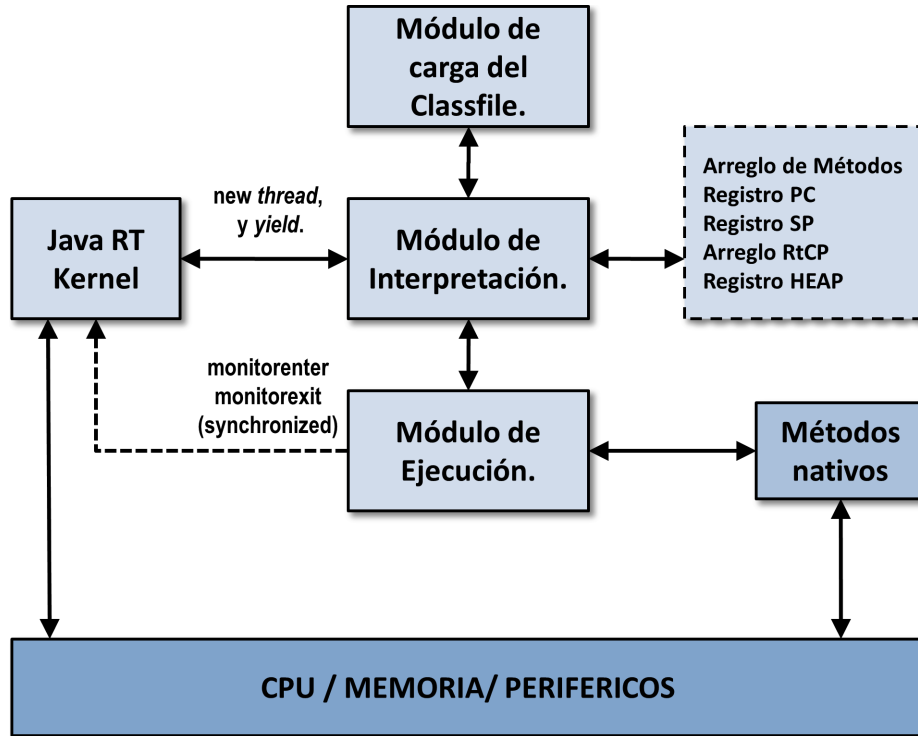


Figura 8.2 Arquitectura de MiniVM.

Estos módulos son (ver Figura 8.2): **módulo de carga del classfile** donde se realizan las tareas de verificar, preparar y resolver la información binaria existente dentro de la imagen de la aplicación y utilizar esta información para inicializar los registros internos de MiniVM, como el contador de programa (PC), apuntador de pila (SP), la tabla de constantes en tiempo de ejecución (RtCPL), el del montículo (HEAP), el hilo de ejecución principal, etcétera. El segundo, el **módulo de Interpretación** es donde la naturaleza del *bytecode* es definida. Si es una instrucción como un nuevo *thread*, *yield* o el registro de algún “escucha” (*listener*) envía la información requerida al núcleo de Java RT para que se haga cargo la tarea. De otra forma, esta es una instrucción a ser dada al módulo siguiente. Tanto el *módulo de interpretación* como el *módulo de ejecución* hacen uso de los registros internos de MiniVM para auxiliarse tanto en el acceso a la información dentro del *classfile* o para la correcta ejecución de la aplicación. El último módulo es el **módulo de ejecución** quien traduce la instrucción dada por el *bytecode* de Java al conjunto de instrucciones que implica su ejecución en el hardware.

También puede hacer uso de un conjunto de métodos nativos para simplificar operaciones complejas de hardware (como enviar datos a través del puerto serie o RF). Es necesario enfatizar que este módulo también hace llamadas al núcleo de Java RT con las instrucciones *monitorenter*, *monitorexit* y *synchronized*.

Los módulos de MiniVM trabajan juntos de manera similar a como se describe en [128]. Primero la imagen de la aplicación es leída por el módulo de carga de *classfile*, inicializando todos los elementos necesarios dentro de la máquina virtual como por ejemplo registros, el thread *main* entre otros. Incluyendo el núcleo de Java RT. Una vez inicializada MiniVM, el planificador busca el primer Thread (el cual siempre es el método *main*) cargando su contexto, una vez hecho esto le deja el trabajo al módulo de interpretación, el cuál toma el *bytecode* apuntado por el contador del programa y lo toma para su decodificación. Finalmente ya decodificada, es ejecutada por el procesador físico. En caso de que la instrucción involucre la creación de un *thread* o el método *yield*, llamada o ejecución de un temporizador el control es la núcleo de Java RT; de lo contrario, el control es regresado al módulo de interpretación para repetir este ciclo.

La abstracción dada por MiniVM es completa y solamente permite el acceso a los recursos de hardware a través de sus métodos nativos. Dentro de esta su estructura interna también contiene elementos que permiten el manejo en las aplicaciones de múltiples hilos de ejecución (limitado por el espacio de memoria RAM existente) y servicios de interconexión.

Un aspecto a resaltar del uso de MiniVM o de las middleware en general es que esta protege el hardware de un acceso incorrecto o inesperado por la aplicación en ejecución, por ejemplo, refiriéndose en el hardware, cuando es posible el acceso a un registro mapeado en memoria de una manera descuidada. Por otra parte, su uso implica la introducción de una sobrecarga de código, donde una instrucción simple del *bytecode* de Java debe de ser ejecutado haciendo uso de varias instrucciones nativas de la UCP. Una explicación sencilla del proceso de interpretación y su sobrecarga de código es mostrada en la Figura 8.3; la suma de dos números enteros, en el lenguaje de programación Java, resultando en un grupo de operaciones escritas en el *bytecode* de Java y teniendo que ser convertidas de una por una a

un grupo de instrucciones nativas al procesador.

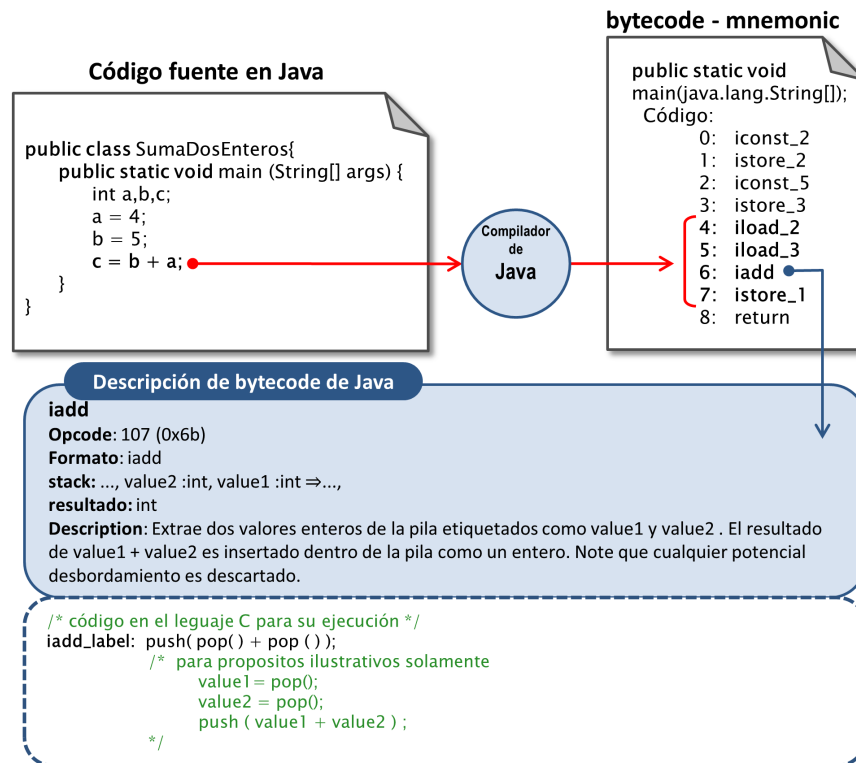


Figura 8.3 Equivalencia del *bytecode* de Java.

La sobrecarga de código introducida por el compilador de Java es en promedio 10 instrucciones en el lenguaje C por cada *bytecode*. Pero esto puede ser resuelto por diferentes técnicas como la compilación “al vuelo” o conversión a código nativo. Sin embargo, esto puede ser soportado si nosotros deseamos las características como seguridad en la aplicación y su independencia de plataforma.

8.2.2 Planificador Multi-hilo.

La programación multi-hilo (multithread) en el lenguaje Java es parte de la biblioteca estándar de programación, sin embargo, para poder hacer uso de esta es necesaria la implementación de un planificador de tareas que le de soporte. Una de las fortalezas de la adopción de este lenguaje de programación radica en poseer este recurso como parte de MiniVM, pero

con un número limitado de los mismos debido al espacio disponible dentro de la memoria RAM.

Un hilo de ejecución (thread), es una unidad básica de uso del UCP, la cuál esta esta comprendida (en su implementación dentro de MiniVM) por: un *identificador de thread* (ID), *contador de programa* (PC), *Estado*, *textitPrioridad*, el conjunto de *registros internos* de MiniVM y una *pila* todo dentro de un *bloque de contexto del thread* (TCB) (ver Figura 8.4).

Bloque de Contexto de Thread
Estado
Prioridad
Contador de Programa
Apuntador de Pila
Dato del Thread
Locales

Figura 8.4 Bloque de contexto del Thread.

El núcleo, es el encargado de realizar el intercambio entre los contextos haciendo uso de la información contenida en estos bloques de información (ver Figura 8.5) de la siguiente forma. Primero salva el contexto de MiniVM en su *bloque de contexto* correspondiente una vez agotado su tiempo de uso de la UCP, verifica el *planificador* (con la política indicada después) para extraer el contexto del siguiente thread a ser ejecutado del bloque de contextos, restaura su estado en la MiniVM y continua la ejecución del mismo mientras no se termine su tiempo de uso de la UCP.

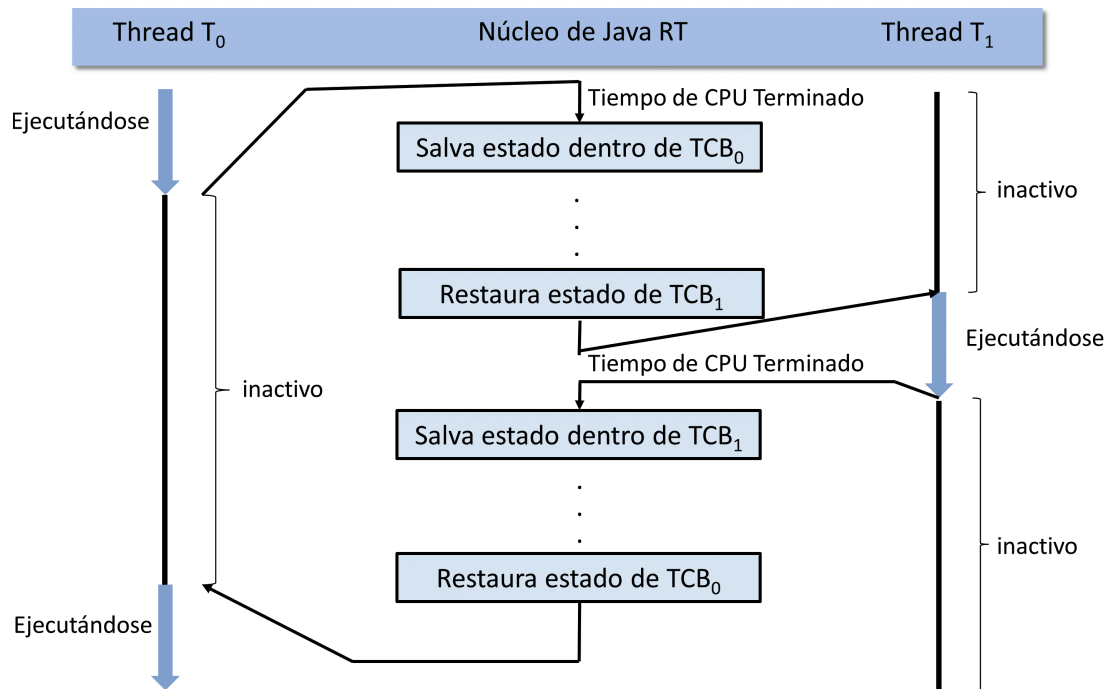


Figura 8.5 Cambio de contexto entre Threads.

La introducción del paradigma de programación multithread dentro del nodo-SR es para permitir el desarrollo de aplicaciones a partir de pequeñas piezas de código autónomo. Cada una con sus propias acciones, sus recursos de memoria y tiempo de procesador reservado, guardado dentro de la pila de MiniVM; compartiendo en algunos casos su código de programa. Estos son nativos al núcleo Java RT quien es responsable de su correcto manejo.

Dentro de la especificación de las máquinas virtuales de Java (en la que se sustenta la implementación de MiniVM) tiene la política de planificación vagamente definido, limitandose solamente a establecer que cada hilo tiene una prioridad y los hilos de mayor prioridad se ejecutarán con preferencia a los threads con prioridades más bajas. No se señala si la política de planificación debe ser preemptive o en time-slice utilizando un planificador round-robin. La decisión depende de la implementación particular que se haga de la JVM.

En el caso del planificador dentro del núcleo Java RT se hace uso de un algoritmo de planificación basado en prioridad y preemptive, lo cual significa que todos los threads tie-

nen una prioridad y la más alta será ejecutada primero. Esta prioridad es de acuerdo a la especificación de tiempo real para Java [129].

La unidad de tiempo de UCP asignada (quantum) para cada thread es fija y en base a 1 milisegundo. Dependiendo de la prioridad es asignada la cantidad de quantums. Una vez agotado su tiempo, el thread cede el uso de la UCP a otro thread de la misma o menor prioridad. Las prioridades se enumeran de la 1 (la más baja) a la 28 (la más alta) y dependiendo de esta se puede extender hasta los 200 quantum para la mayor y 20 quantum para la menor.

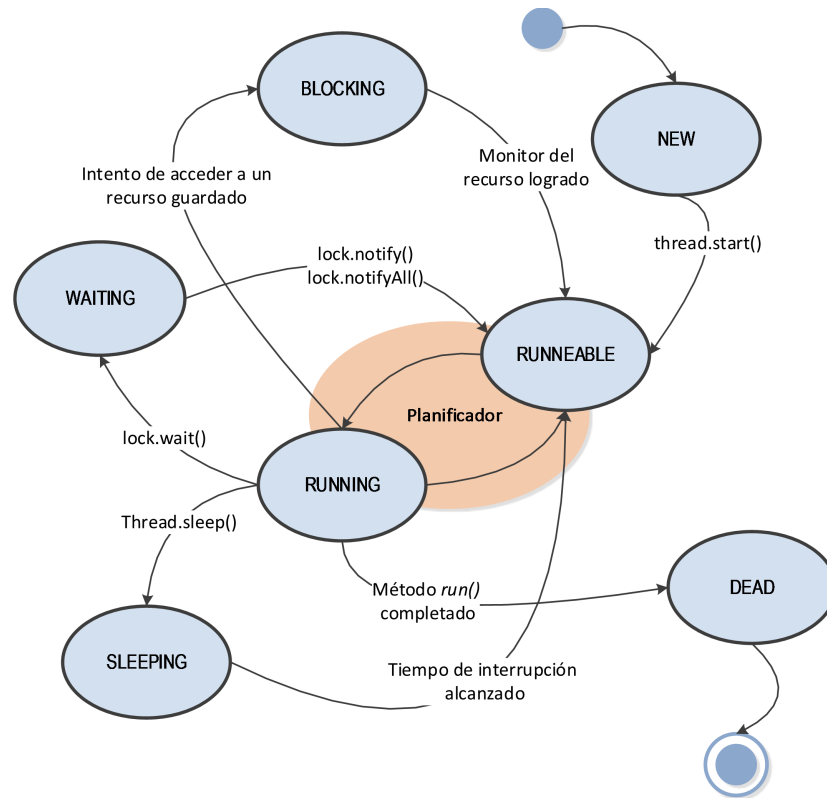


Figura 8.6 Diagrama de transición de estados.

El diagrama de estados, mostrado en la Figura 8.6, tiene la transición estándar entre los estados de un thread. Los estados *Running* y *Runnable* son los estados donde el planificador participa de manera única basándose en su tiempo de ejecución correspondiente; los demás estados (*Blocking*, *Waiting*, y *Sleeping*) son determinados por la condición de la ejecución

del thread, que puede ser la sincronización de un recurso, voluntariamente ceder el uso de la UCP o esperar por el desbordamiento de un temporizador. Un estado indeseable para el thread es el *Dead* porque es alcanzado al haber concluido la ejecución de su método *run()* y no existe un mecanismo implementado para recuperar los recursos reservados por dicho thread.

Por último, los threads que pertenecen al mismo proceso comparten los datos del proceso. Este intercambio de datos proporciona uno de los beneficios de la programación multiproceso. Sin embargo, en algunas circunstancias, cada hilo puede necesitar de su propia copia de dato, a estos se les conoce como *datos de un hilo específico*. Por ello, es necesario guardar en el programador, además de los registros del procesador y *stackTop*, el *HEAP* de datos y *VarLocs*.

8.2.3 Mapeo de memoria RAM.

La memoria RAM en el nodo-SR es uno de sus recursos más valiosos y escasos; adquiriendo con ello mucha importancia su uso óptimo por parte de MiniVM. Para esto, la memoria RAM es dividida en los segmentos genericos de: *Datos*, *Pila* y *HEAP* (ver Figura 8.7(a)).

Dentro de los segmentos, en la sección de *Datos* esta localizada la *tabla de interrupciones* que es usada para resolver las interrupciones ocurridas y los campos estáticos de la aplicación; al final de esta sección inicia el segmento de *pila* donde son creados los métodos y *frames* de los threads (descrito a mayor detalle en la Figura 8.7(b)); además de las referencias a los objetos (creados dentro del *HEAP*) y las variables locales de los métodos. Finalmente, el *HEAP*, donde es almacenado los datos y los objetos creados. Este inicia al final del segmento de la *pila* creciendo en sentido opuesto al que crece el *apuntador de pila*.

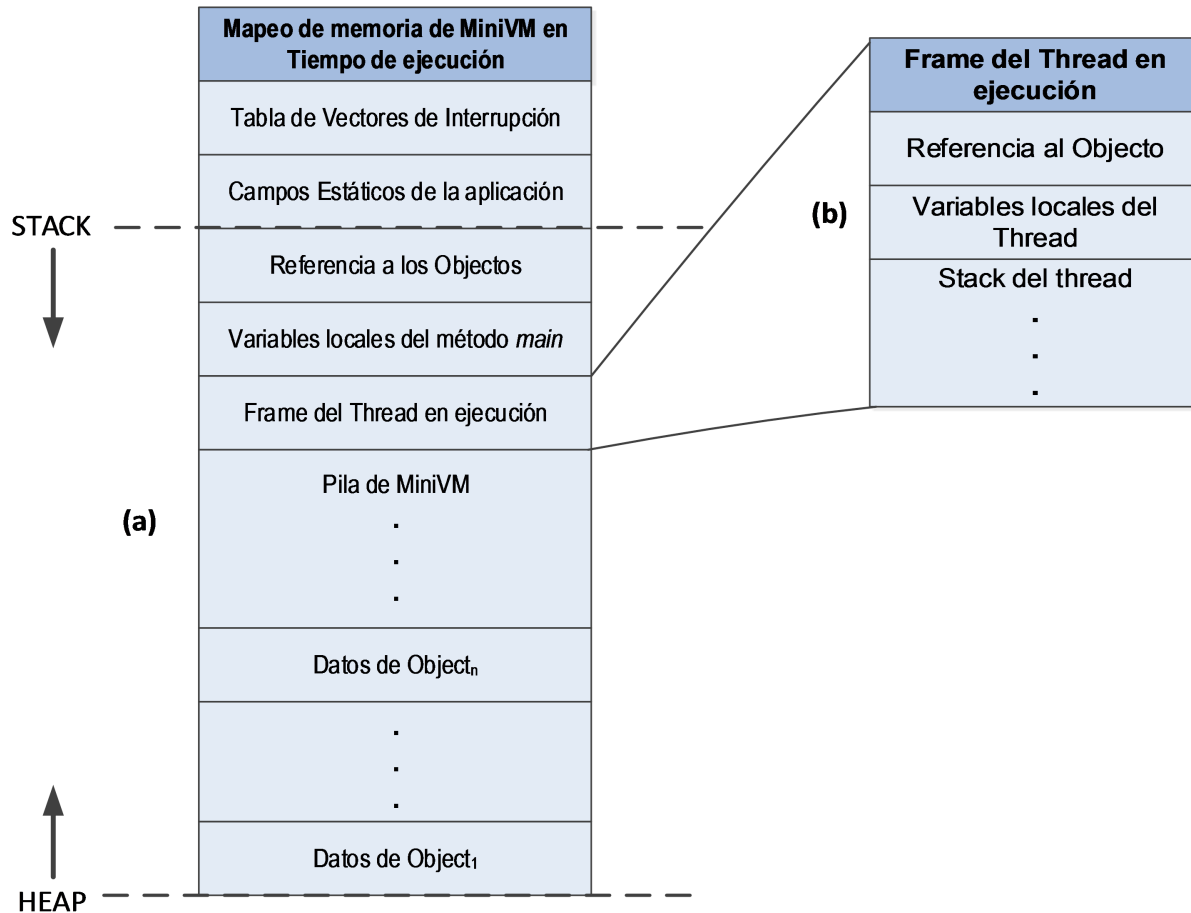


Figura 8.7 Mapeo de la memoria RAM.

Es necesario recordar que no existe un mecanismo en MiniVM para recuperar la memoria reservada en la creación de los objetos. Por esta razón, es necesario tener cuidado al momento de crearlos. El tamaño de memoria RAM requerida para la ejecución de la aplicación es de 2KB, la cual es completamente usada como *pila*.

8.2.4 Manejo de Eventos.

Dentro de la programación de aplicaciones para las WSN, los eventos son importantes y pueden ser de dos tipos. Los síncronos, quienes tienen su fuente en la expiración de temporizadores; y los asíncronos, producidos por interrupciones de software o hardware sin tener

en cuenta el tiempo de estos.

A nivel de la aplicación, esto se implementa para ambos dentro de la Framework de Desarrollo en Java; los cuáles son tratados conforme a el objeto que lo produce. Esta parte es descrita a mayor detalle en la sección del Framework de desarrollo en Java.

Por otra parte, a nivel de MiniVM, ambos eventos son similares, diferenciandose solamente donde estos son registrados. Es decir, dentro de MiniVM existen un conjunto de observadores para cada tipo de evento, el cuál es capaz de notificar a los objetos acerca de la ocurrencia del evento. Para esto, la necesidad del manejo del evento es identificada por el módulo de interpretación (véase la Figura 8.8). En la creación de cada objeto se verifica si requiere de un manejador de eventos (ver Figura 8.8(a)), si es requerida, es determinada la naturaleza del evento (puede ser un temporizador, recepción UART o interrupción de software). Una vez definida su naturaleza, el módulo de interpretación registra el código correspondiente a su manejador dentro del observador implementado para esto dentro del núcleo de Java RT. Por último, el núcleo Java RT es responsable por checar su ocurrencia.

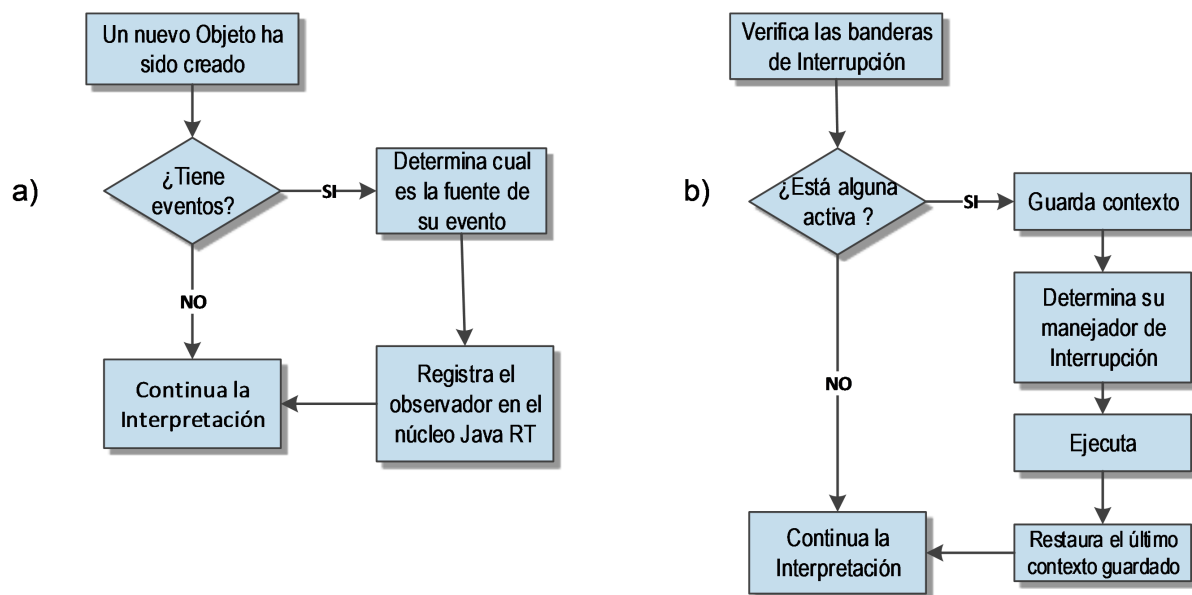


Figura 8.8 Mecanismo para el manejo de interrupciones por el módulo de Interpretación: (a) diagrama de flujo para el registro de eventos, (b) diagrama de flujo para el manejo de la ocurrencia de un evento.

El módulo de interpretación, encuesta los registros de banderas por la ocurrencia de algún nuevo evento al término de cada ciclo máquina (ver Figura 8.8(b)). Si existe alguna bandera activa, concede su ejecución al núcleo Java RT el cuál guarda el contexto, ejecuta el código para la atención de la interrupción y restaura el estado anterior a la ocurrencia de la misma. Una vez hecho esto, el módulo de interpretación continua con su trabajo.

8.2.5 Optimización de MiniVM.

Durante el desarrollo de MiniVM, se estuvo conciente de la sobrecarga de instrucciones causada en la interpretación del *bytecode* de Java. Esta interpretación es hecha por el módulo que lee la instrucción a ser ejecutada y prepara los registros internos de MiniVM para la correcta ejecución de la aplicación.

Para tener la medida exacta acerca de esta sobre carga de código, se tomaron medidas los ciclos de la UCP para algunas instrucciones. Principalmente de aquellas que llaman a métodos dentro de clases (ya sean *estáticas*, *virtuales* y *especiales*) las cuáles son complejas porque requieren de un enlazamiento tardío. Además que los *bytecode* como *iinc* (incrementa el valor de una variable entera en uno), *goto* (asigna al registro *PC* una dirección específica) y *jsr* (salta a una instrucción específica las cuáles son instrucciones fáciles de interpretar).

Los resultados antes de la optimización son (ver Figura 8.9): el *bytecode* para llamar a los métodos requieren más de 2500 ciclos. Esto es debido a que la clase y el nombre del método tienen que ser resueltos en tiempo de ejecución, cada ocasión que este ocurre dentro de la tabla de constantes se identifica el nombre de la *clase*, donde esta contenido el método. Ya resuelta la clase y el método, este se busca dentro de las referencias a métodos en tiempo de ejecución, el cual indica la dirección del método a donde el registro *PC* debe señalar (obviamente todo esto salvando antes el contexto de la MiniVM). Todo esto es hecho mientras la aplicación se esta ejecutando; de ahí la preocupación acerca de los ciclos de la UCP requeridos.

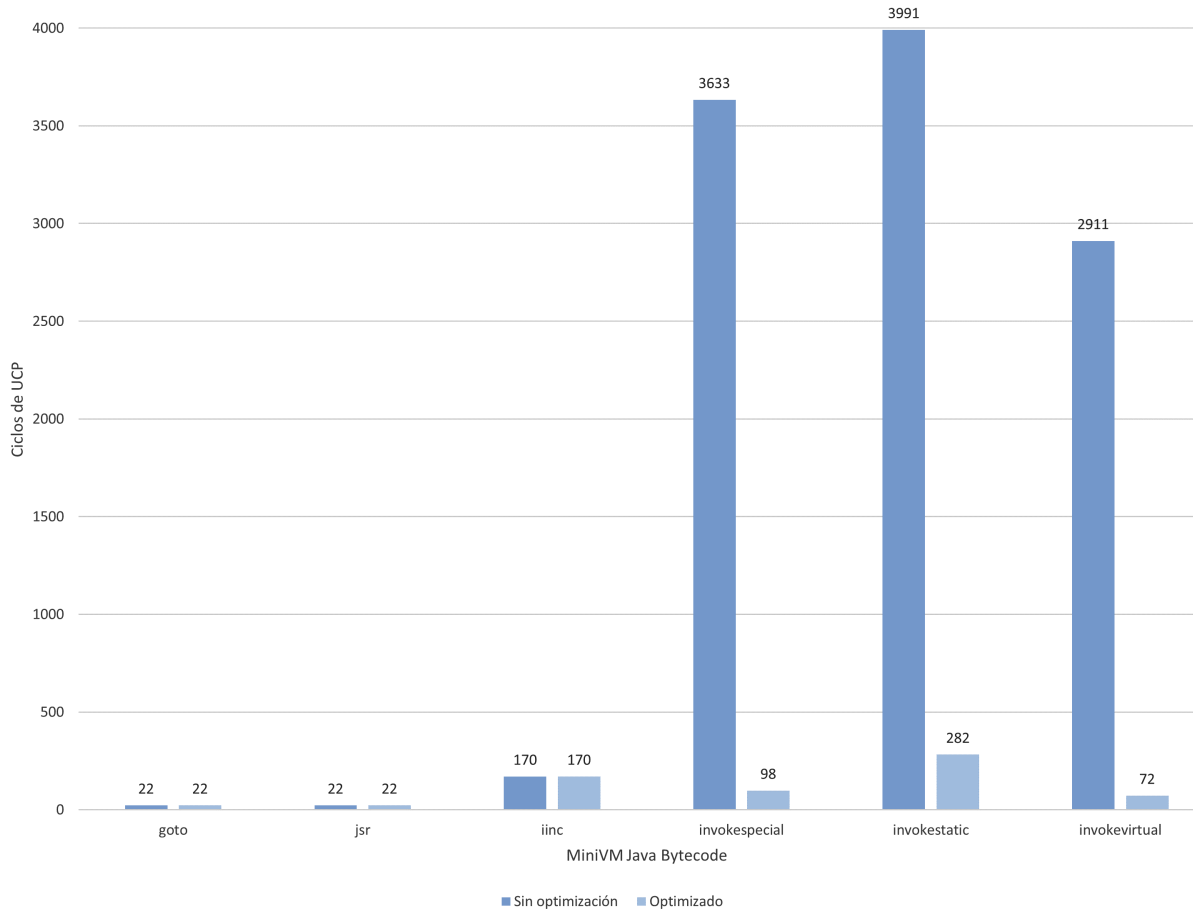


Figura 8.9 Medición de ciclos de UCP.

Para un mejor desempeño (ver Figura 8.10), dentro del módulo de inicio se resuelven los índices a las clases y sus métodos, después busca su dirección (sin importar si es nativo o de enlace tardío) y registra su referencia dentro del registro en tiempo de ejecución de la tabla de constantes. Esto hace lento el arranque de MiniVM pero hace muy eficiente la interpretación reduciendo significativamente los ciclos de UCP.

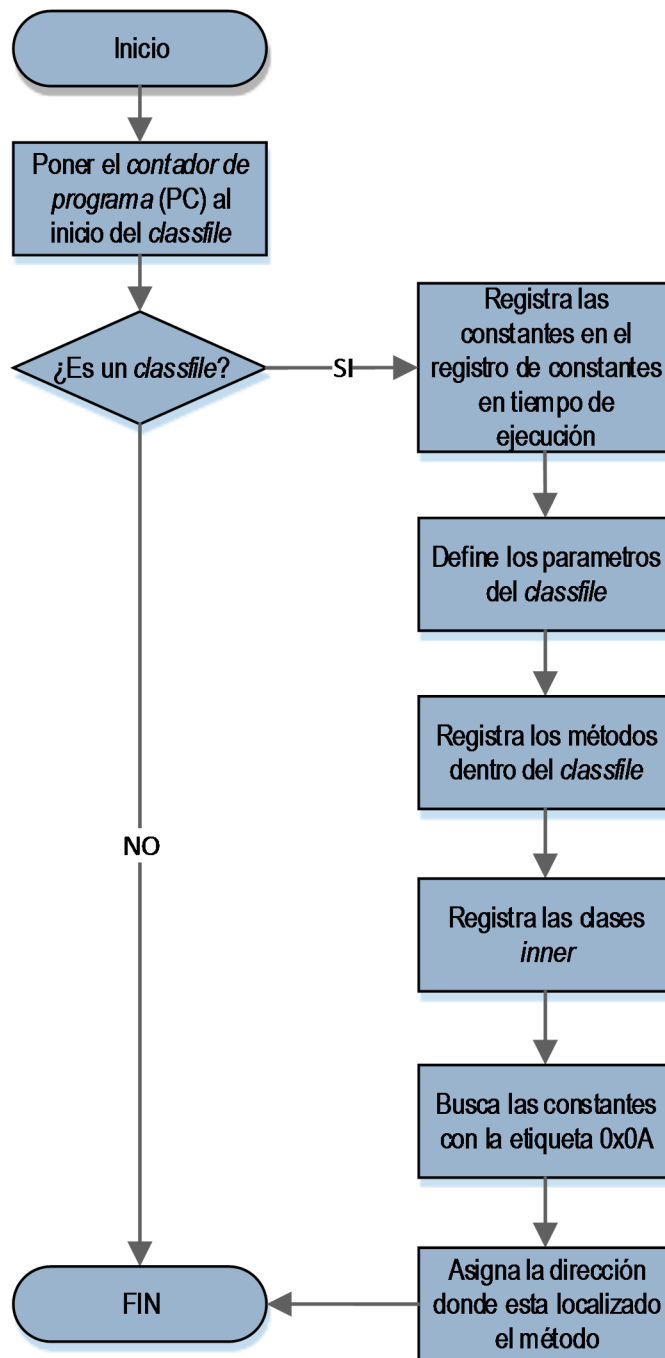


Figura 8.10 Módulo de carga de *classfile* optimizando la referencia a los métodos.

8.2.6 Manejo de Excepciones.

Las excepciones son objetos que son generados automáticamente cuando se produce un acontecimiento circunstancial que impide el normal funcionamiento del programa como pudiese ser la división por cero, no encontrar algún archivo o utilizar un apuntador nulo en lugar de una referencia a un objeto. El objeto generado “*la excepción*” contiene la información sobre el acontecimiento ocurrido y transmite esta información al método desde el que se ha generado la excepción.

La ocurrencia de estas situaciones excepcionales provocará la terminación no controlada del programa o aplicación.

Para el caso particular del lenguaje Java estas pueden originarse de dos modos:

- El programa hace algo ilegal.
- El programa explícitamente genera una excepción.

Actualmente, como todas las excepciones son eventos runtime, sería mejor llamarlas excepciones irreversibles. Esto contrasta con las excepciones que generamos explícitamente, que suelen ser mucho menos severas y en la mayoría de los casos podemos recuperarnos de ellas.

En otras palabras, las excepciones es la forma en que Java indica que la llamada al método a entrado en una condición anormal y cuando eso ocurre el método arroja un objeto especial para el manejo de esta. Para su implementación, es requerida una tabla de referencia donde se enlistan todas las posibles excepciones y el correspondiente objeto para su manejo. No fue implementado por la sobrecarga de procesamiento requerida. En lugar de esto, a fin de soportar el desarrollo rápido de aplicaciones y la reutilización de código, MiniVM es acompañada por un Framework de Desarrollo de aplicaciones en Java, en el cual se tienen varias *clases* y sus “*comportamientos*” apoyados por patrones de desarrollo de software.

8.3 Framework de aplicación.

El lenguaje de programación Java es de 4ta. generación, orientado a objetos, seguro, independiente de plataforma, con mayor facilidad de aprendizaje, además de ser muy popular para la programación de aplicaciones en la Web, una evidencia de como se adapta muy bien en el Internet, uno ambiente híbrido y donde las características de la plataforma final son muy variantes, un escenario muy recurrente en los ambientes de las WSN. Por esta razón, no es de sorprenderse en considerar a Java cuando se trata del ámbito de desarrollo de aplicaciones. Esta tendencia es tan visible que compañías como Intel se han asociado a universidades de U.S.A. para el desarrollo de esta idea [130] inclusive la compañía Oracle, propietaria de Java, promueve su propia versión de plataforma para las WSN [99]. Además, como lenguaje, tiene API nativas que soportan el multi-thread y la colección de basura.

Con respecto al mercado de las plataformas para sistemas embebidos, existen estándares de Java [131] incluídas la J Consortium's Real-Time Core Specification, Personal Java (pJava), Embedded Java, Java 2 Micro Edition (J2ME) ant The Real-Time Specification for Java.

En el caso del Framework de aplicación, apoya el desarrollo para el nodo-SR proveyendo un conjunto de *Java Auxiliary Programming Interface* (API), la cuál es una biblioteca independiente de plataforma que se provee para MiniVM y el nodo-SR, entre otras cosas, para permitir a los programadores ejecutar funciones del sistema y el reutilización de código. Las aplicaciones programadas requieren de la inclusión de las clases de este Framework para su ejecución correcta.

8.3.1 La *clase* LLED.

Uno de los elementos más simples existente en el nodo-SR es el LED (*Light Emisor Diode*). Este esta dentro del empaquetado de clases *io* con el nombre de *clase* **LLED**. Dentro de dicha *clase* (ver Figura 8.11) se tiene un arreglo de *bytes* de 3 elementos, los cuales hacen referencia a los 3 LEDs existentes en el nodo-SR (indicado por sus indices). Esta clase es del

tipo **final** e implementa un estado lógico para el manejo del estado del LED (encendido o apagado).

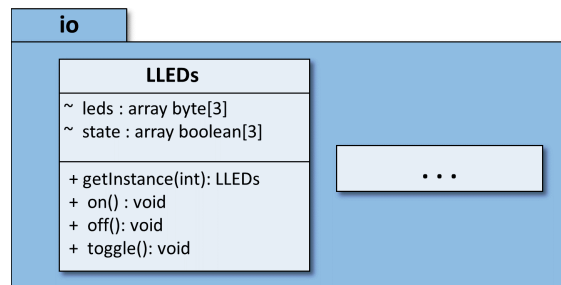


Figura 8.11 Clase **LLED**.

Por otra parte, debido al número limitado de recursos físicos existentes en el nodo-SR hace necesario limitar, desde la perspectiva de software, la creación de objetos en la aplicación que operan con ellos. Para lograr esto se implementa el patrón *singleton*. De esta forma cada vez que un objeto del tipo LLED es creado; este es referenciado a un campo estático de la clase, como única instancia del objeto. Así se garantiza la correctitud de la operación entre el hardware y el software en ejecución. Sin importar cuantos objetos se creen estos estarán siempre operando sobre el mismo recurso de hardware. No es crítica la reserva de este recurso puesto que solo es una señalización luminosa.

8.3.2 La *clase* LSwitch.

Otro de los recursos simples existente dentro del nodo-SR es el *Switch*. Este elemento de hardware es usado por el usuario como un dispositivo de entrada y es probado su estado todo el tiempo. Para su manejo se tiene la *clase* **LSwitch**, dentro también del empaquetado de clases *io*. La *clase* **LSwitch** (ver Figura 8.12) es descrita en software por un campo estático (*state*) el cuál es accesible por el método *isPressed* y otro campo estático (*switch*) como única referencia (patrón *singleton*).

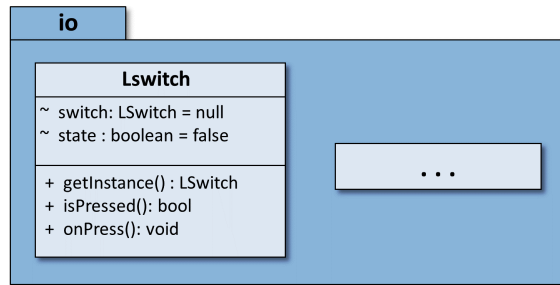


Figura 8.12 Clase **Lswitch**.

El método *onPress* debe de ser sobrescrito en una *clase final* que hereda de la *clase Lswitch*.

8.3.3 Las *clases* **LHumiditySnr**, **LTemperatureSnr** y **LLightSnr**.

Los sensores de Luz, temperatura y humedad en el nodo-SR son inmutables (de manera muy similar a las clases estáticas en Java). Por esta razón las clases que se refieren al manejo de estos recursos son estáticos y empaquetados en el grupo de clases *sensors* (ver Figura 8.13 bajo el nombre de *LHumiditySnr* para el sensor de humedad, *LTemperatureSnr* para el sensor de temperatura y *LLightSnr* para el sensor de Luz).

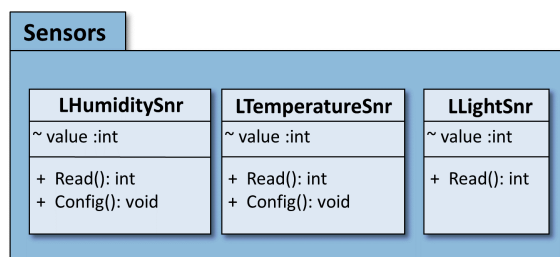


Figura 8.13 El paquete **Sensors**.

Estas clases solamente tienen los métodos funcionales de leer (*read*) y configurar (*config*) las cuales son las únicas requeridas para su operación.

8.3.4 La clase **LTimer** y **LTimerTask**.

La programación de eventos síncronos (aquellas que están basadas en temporizadores) implica el uso de “escuchas” (*listeners*, patrón *observador*). En este patrón se tiene una entidad (*observer*) llevando el registro de todas las entidades interesadas en el evento que esta *observando*, cuando el evento ocurre, esta entidad se encarga de notificar a todos los “interesados” de dicho suceso. De esta forma es como están implementadas las *clases* **LTimer** y **LTimerTask** para el manejo de los temporizadores.

La creación y uso de los temporizadores en el nodo-SR involucra el uso de manejadores de interrupción del “timer” del microcontrolador. Como su “escucha” es implementada (ver Figura fig:timers) dentro del empaquetado *Util* la *clase* **LTimer** quien posee una colección del tipo de *clase* **LTimerTask** como “observadores” ha ser notificados por el tiempo transcurrido. La cantidad de “observadores” es limitada.

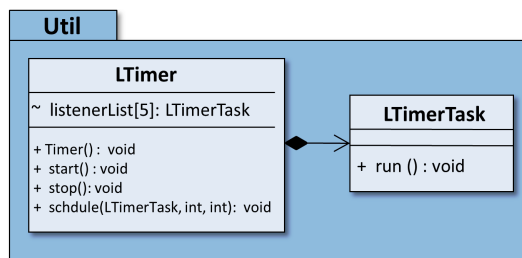


Figura 8.14 Las *clases* **LTimer** y **LTimerTask** para el manejo de Temporizadores.

La *clase* **LTimerTask** debe de ser heredada a una clase final donde se implementa el método *run*.

8.3.5 La *clase* **LPinout**.

El nodo-SR cuenta con puertos de expansión digitales para el manejo de dispositivos de expansión. En el manejo de este se hace uso de la *clase* estática **LPinout** (ver Figura 8.15) que tiene descritos como campos cada uno de los “pines” disponibles (*pinoutA1*, *pinoutA2*, *pinoutA3*, *pinoutB1*, *pinoutB2*, *pinoutC1*, *pinoutC2* y *pinoutC3*), los métodos *configPinoutX*

asigna la dirección del puerto, los métodos *readX* permite leer todos los *pinouts* y el método *writeX* escribir en ellos. Si se desea acceder a uno en particular se puede leer su estado en particular.

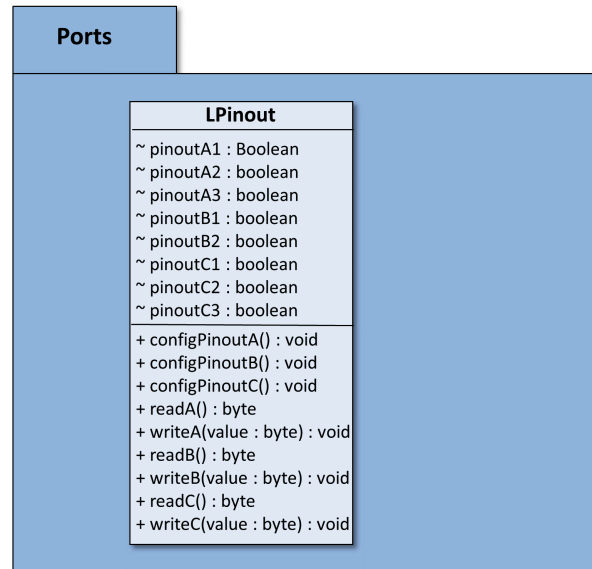


Figura 8.15 La *clase* **LPinout**.

8.3.6 La *clase* **LADC**.

Los mismos “pinouts” que pueden ser E/S digital pueden funcionar de entrada como señales analógicas convertidas a digitales (ADC). Para su manejo se tiene la *clase* **LADC** (ver Figura 8.16) la cual tiene definida cada una de las entradas de manera numérica al pin que corresponde *ADCx*, para la configuración se hace uso del método *configADC* al que se le indica el número del pin que se configurara; de forma muy similar para leer el valor dado por el ADC se hace uso del método *readADC* dando como parametro el número del pin que se desea leer.

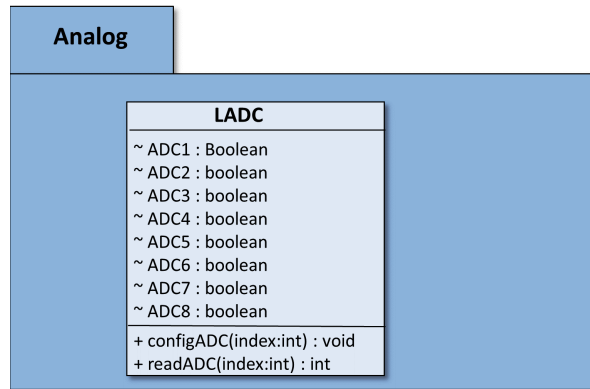


Figura 8.16 La *clase* LADC.

8.3.7 La comunicación.

El nodo-SR tiene dos formas posibles para la comunicación:

- De forma estandarizada por el puerto serie (*USART*). Usada comúnmente para comunicarse con una PC u otro dispositivo para la descarga de datos o control y
- Por el transreceptor inalámbrico. Utilizado para comunicarse inalámbricamente de manera no estándar a través de un canal libre con otro nodo-SR o con la puerta de enlace de la red LiSANDRA (*Gateway*).

La *clase* LUSART.

La *clase* LUSART (ver Figura 8.17) maneja la comunicación serie y aunque permite en teoría crear un objeto que señale a cualquier puerto de serie; sin embargo, solo es posible crearlo al *cero* por ser el único existente en el nodo-SR. Se tiene el método *getInstance* para así tener un control de la cantidad de objetos que pueden ser creados y haga referencia a uno único. Los métodos *getInputStream* y *getOutputStream* regresan un objeto *stream* que permite escribir o leer del puerto serie.

LUSART
~ baudRate : int ~ dataBits : int ~ stopBits : int ~ parity : int ~ os : LOutputStream ~ is : LInputStream ~ uart : LUSART
+ LUSART(comx : int, baud : int) + getInstance(comx : int, baud : int) : LUSART + getInputStream() : InputStream + getOutputStream() : OutputStream

Figura 8.17 La *clase* LUSART.

La *clase* socket.

Los *sockets* son un sistema de comunicación entre procesos de diferentes máquinas de una red. Es decir, es un punto de comunicación por el cuál un proceso puede emitir o recibir una información. Originalmente desarrollados y popularizados por *Berckley Software Distribution*, de la universidad norteamericana de Berkley. Los *sockets* han de ser capaces de utilizar el protocolo de *streams* TCP (*Transfer Control Protocol*) y de de datagramas UDP (*User Datagram Protocol*). Sin embargo, estos son protocolos muy “pesados” para ser adoptados como tal dentro de los ambientes de las WSN, recordemos que las comunicaciones de red son desarrolladas sobre medios inalámbricos con bajo ancho de banda y que generalmente no están estandarizadas. Pero la abstracción que hace con respecto a las comunicaciones es tan completa que resulta idóneo emplearlas aunque a bajo nivel se trabajen otro tipos de protocolos. Para el manejo de la comunicación inalámbrica se hace uso de dos *clases* (ver Figura 8.18), una es para crear un servicio de “escucha” que espera por alguna petición de conexión para iniciar la comunicación (*ServerSocket*) y otro para abrir la conexión a cualquier nodo de la red (*Socket*). Ambos hacen uso de un objeto del tipo *stream* para enviar (*método* *getOutputStream*) o recibir (*método* *getInputStream*) datos en la red.

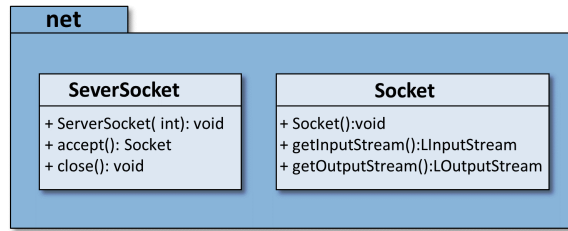


Figura 8.18 Las *clases* **ServerSocket** y **Socket**.

Capítulo 9

Uso de la Framework

El uso de la Framework de Java para Desarrollo de Aplicaciones en WSN comprende la tupla descrita en el capítulo anterior, sin embargo su uso requiere de algunas consideraciones tanto de programas “extra” como de configuración de la computadora personal en la que se efectuará el desarrollo.

9.1 Requisitos.

Las herramientas de software requeridas para la programación del nodo-SR propuesto estan listadas en la siguiente tabla:

Herramienta	Parámetros	Descripción
javac	<i>-g:none -source 1.5</i>	Compilador estándar de Java usado para compilar las aplicaciones. Este debe de ser compilado en la versión 1.5 del <i>classfile</i>
CargaClass	<i>-COMx -v -r</i>	Usado para descargar la aplicación compilada al nodo-SR a través del puerto <i>COM</i> con la verificación del <i>classfile</i>
mixClass	<i>Inclass Outclass</i>	Empaqueta la <i>classfile</i> de entrada y sus <i>clases</i> referenciadas en una sola clase con el nombre dado en <i>Outclass</i>

Para el uso de ellas no es requerido el uso de un sistema operativo en específico ya que estas herramientas son independientes de plataforma. De las herramientas descritas anteriormente, solo es importante describir la operación de *mixClass* ya que *javac* es el

compilador estándar del lenguaje Java y la operación de *loadClass* solo es la carga de la aplicación en el nodo-SR.

9.1.1 Configuración.

La configuración de la máquina que será usada para el desarrollo de las aplicaciones requiere de los siguientes pasos:

- **Copiar** el folder **JavELiNA Framework** a un directorio Local del sistema en la computadora donde se desarrollará la aplicación.
- **Instalar** el Java Development Kit, la versión no importa mientras permita compilar el *classfile* en versión 1.5.
- **Referenciar** en las **variables de entorno** dentro de las propiedades del sistema. Es decir, en el *classpath* deberá de tener rutas como “...c: \LiSANDRA framework \LiSANDRA \src \Timer \;...”.

9.1.2 La herramienta *mixClass*.

La aplicación compilada de Java, ha ser descargada en el nodo-SR esta limitada por la cantidad de archivos *classfile* que puede administrar *MiniVM*, de hecho solo puede manejar un único archivo. Por esta razón, durante el proceso de desarrollo (ver Figura 9.1), además de compilarse con la Framework de Aplicaciones en Java (la cuál debe de estar referenciada en el *classpath*) usando un compilador estándar de Java, se debe de pre-enlazar (haciendo uso de *mixClass*) en una única imagen de aplicación en Java donde es colocado todos los *classfiles* usados para la ejecución de la aplicación (tanto la aplicación del usuario como las clases del Framework de aplicación).

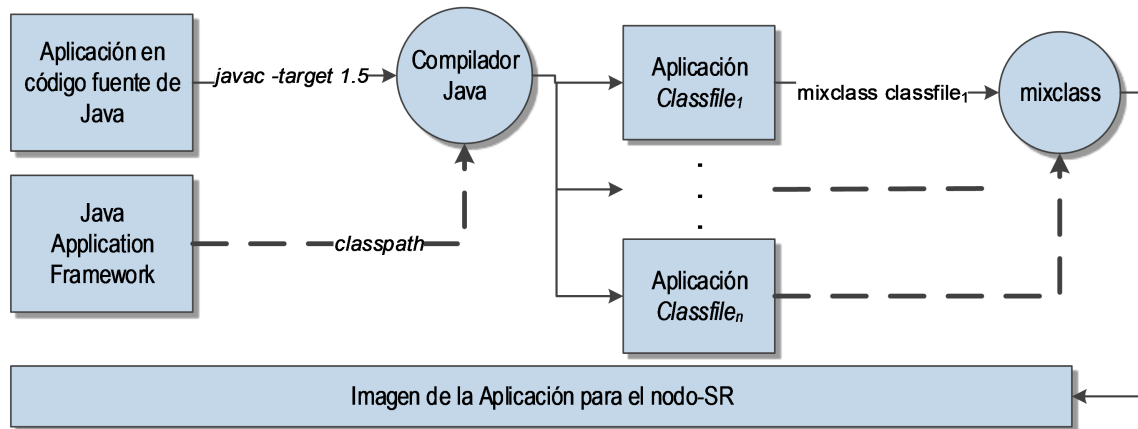


Figura 9.1 Proceso de desarrollo de aplicaciones usando Java y el Framework.

De este proceso, es necesario describir la herramienta de pre-enlazado llamada **mixClass**. Esta es necesaria porque cuando el código fuente de la aplicación hecha en Java es compilada por el comando *javac*, este separa cada objeto contenido dentro del código fuente en un archivo *classfile*, esto se traduce que para *n* número de *clases inner* o *nested* va a resultar en el mismo número *n* de *classfile* también. Cada *objeto* es referenciado en el *classfile main* por una tabla de atributos donde son descritos sus nombres y su correspondiente *classfile*. Esta acción es estándar en los compiladores de Java.

Cada *objeto* es descrito dentro del *formato estándar del classfile*. La herramienta *mixclass* une todos los *objetos* (con sus *classfile*) en un único archivo (véase Figura 9.2), nombradas dentro de una única *tabla de constantes*, sus referencias a métodos y todos los campos de las clases referenciadas por los nombres de sus propietarios. Obviamente se le agrega el atributo *InnerClasses* en la tabla de constantes.

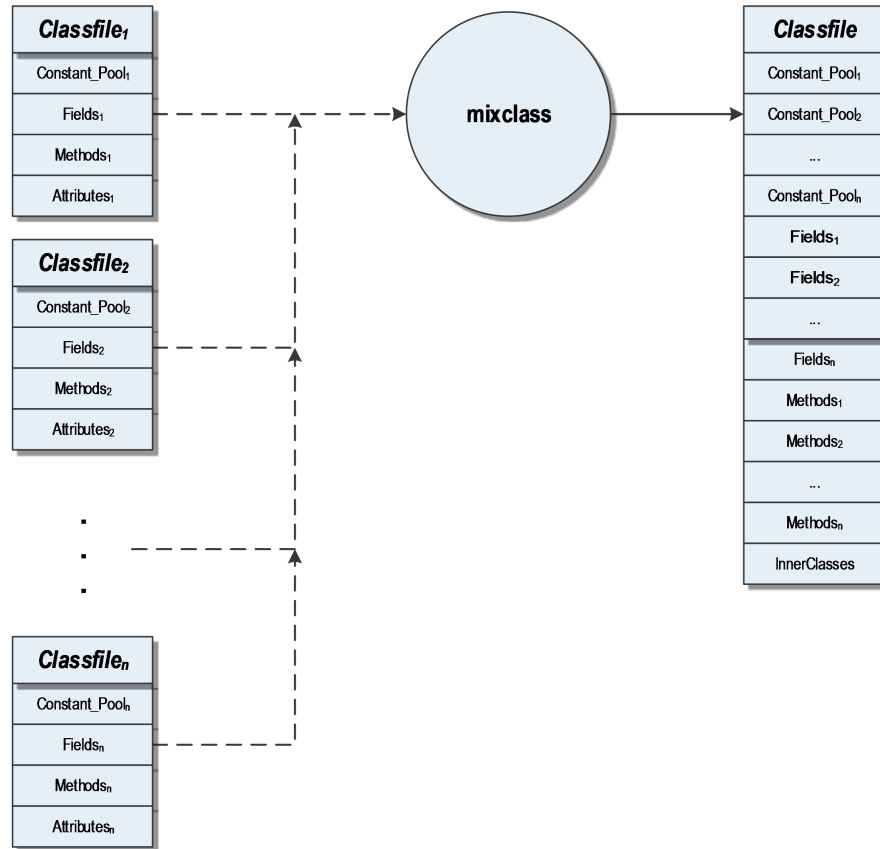


Figura 9.2 Proceso de serialización del *classfile*.

También durante el proceso de la generación de la imagen de aplicación, la tabla de constante es optimizada, eliminando la redundancia dentro de esta (generalmente son del tipo *string*) y actualizando la referencia a donde se encuentra. Como resultado, reduce el tamaño del archivo *classfile*, el cuál será variable en función de la cantidad de constantes repetidas en la tabla de constantes. Otra razón para usar un único archivo *classfile* en la imagen de la aplicación es por el tiempo de retardo dado por MiniVM al momento de inicializar el *classfile*.

9.2 Código de Ejemplos.

Para el uso del Framework de Java para el Desarrollo de Aplicaciones en las WSN requiere de que este instalada MiniVM en el nodo-SR. Para codificar la aplicación, como se comentó en la sección anterior, debe estar referenciada a las clases de la Framework de Aplicación para las WSN. Esto permitirá que el nodo-SR sea programado haciendo uso de las clases de alto nivel del Framework sumado a las clases creadas por el programador para la aplicación final. Los ejemplos de código de las siguientes subsecciones mostrarán como usar el Framework para hacer aplicaciones.

9.2.1 Uso de los LEDs.

Esta es una sencilla aplicación (ver Figura 9.3) donde se muestra como se puede acessar y operar los recursos de LED dentro del nodo-SR, para esto se declara un objeto del tipo *LedRed* (línea 5) al que se le asigna el objeto haciendo la petición del recurso con el método *LiSANDRA.Resource* (línea 6). Finalmente dentro de un ciclo *while* infinito se alterna el estado del LED haciendo uso del método *toggle* (línea 8).

```
1:import LiSANDRA.resources.LEDs;
2:
3:public class SimpleLEDRed{
4:    public static void main(String args[]){
5:        LedRed _red =null;
6:        _red = (LedRed)LiSANDRA.Resource(RED);
7:        while(true){
8:            _red.toggle();
9:            for(int i=0;i<65536;i++);
10:        }
11:    }
12:}
```

Figura 9.3 Aplicacion SimpleLEDRed.java

9.2.2 Uso de Lswitch.

La clase Lswitch es una clase abstracta que en la implementación final se sobrescribe el método *onPress()* (línea 15). La Figura 9.4, muestra la parte implementada de la *clase Lswitch*. De esta implementación se aclara que el método *readSwitch* es nativo, el cual regresa el valor de *true* y el método *getInstance* es la implementación singleton del *Lswitch*.

```
1:public abstract class Lswitch{
2:    static public Lswitch switch;
3:    static public boolean state = false;
4:
5:    public final boolean isPressed(){
6:        return readSwitch();
7:    }
8:    public Button getInstance()
9:    {
10:        if(switch == null){
11:            switch = new Lswitch();
12:        }
13:        return switch;
14:    }
15:    public abstract void onPress();
16:}
```

Figura 9.4 Implementación de la *clase Lswitch*.

9.2.3 Haciendo uso de la comunicación UART.

Una aplicación muy sencilla que ejemplifica como hacer uso de la comunicacion UART en el nodo-SR esta dada por la Figura 9.5 en ella el nodo-SR espera por un *byte* del puerto UART de la PC y regresa este por el mismo en forma de eco. Para llevar a cabo esto se extrae la instancia del puerto UART indicando los *baud* a los que se solicita el puerto (línea 7), una vez obtenido el objeto se extraen los *streams* de *entrada* y *salida* (líneas 8 y 9). Por último, espera por un byte recibido para escribir ese dato de vuelta por el puerto (línea 12).

```

1:import java.io.InputStream;
2:import java.io.OutputStream;
3:import java.io.IOException;
4:
5:public class EchoUart{
6:    public static void main(String args[]){
7:        UART _serialPort = UART.getInstance(9600);
8:        InputStream is = _serialPort.getInputStream();
9:        OutputStream os = _serialPort.getOutputStream();
10:        while(true){
11:            try{
12:                os.write(is.read());
13:            }catch(IOException e){}
14:        }
15:    }
16:}

```

Figura 9.5 EchoUart para ejemplificar el uso de la comunicación UART.

9.2.4 Uso de Threads, Sensores y Sockets.

Un ejemplo más complejo de las aplicaciones que se pueden crear, como el hacer uso de *threads*, *sensores* y *sockets* de comunicación esta mostrado en la Figura 9.6. En este, se asegura de que las instancias a los recursos requeridos sean *singletón*, ya que solo existe un recurso de este tipo en el nodo-SR. Por otra parte, la creación de objetos y especialmente de *sockets* implementan el uso del patrón *observador* teniendo un número limitado de conexiones que pueden ser creadas (debido al tamaño de la memoria RAM), pero que trabaja de forma suficiente para conexiones de punto a punto. En el código se muestra como se configuran los parámetros del nodo SR a través de la aplicación (de la línea 8 a la línea 11). Se utilizan clases *inner* que heredan de la clase *Thread* (líneas 18 y 34) dentro de ellas se hace petición del recurso de sensor de humedad y temperatura (líneas 21 y 37). Ambas clases en su constructor (líneas 22 y 38) crean un *socket* de *radio* para solo enviar la información (es decir no reciben ninguna conexión) del *socket* creado extrae el *stream* de salida (línea 24 y 40). En el método *main* de la aplicación se crean dos objetos del tipo de las clases creadas y por último se dan inicio llamando a sus métodos *start* (de la línea 13 a la 16).

```

1:import LiSANDRA.net.Socket;
2:import LiSANDRA.io.OutputStream;
3:import LiSANDRA.Sensors.*;
4:import LiSANDRA.lang.Thread;
5:public class SimpleThreadApp{
6:    final int FIVE_SECONDS = 5000;
7:    final String COMM_MEDIA = "radio";
8:    final int N1_addr = 138;
9:    final int N1_port = 25;
10:    final int N2_addr = 100;
11:    final int N2_port = 12;
12:    public static main( String arg[] ){
13:        HumidityTx neighborHum = new HumidityTx();
14:        TemperatureTx neighborTem = new TemperatureTx();
15:        neighborHum.start();
16:        neighborTem.start();
17:    }
18:    private class HumidityTx extends Thread {
19:        Socket sock_n1;
20:        OutputStream outRF_n1;
21:        LHumiditySnr humidity = (LHumiditySnr) LiSANDRA.Resources(LHumiditySnr);
22:        public HumidityTx ( int node_addr, int port ){
23:            sock_n1 = new Socket( COMM_MEDIA, node_addr, port );
24:            outRF_n1 = (OutputStream) sock_n1.getOutputStream();
25:        }
26:        void run(){
27:            while(true){
28:                outRF_n1.write( "H:" + humidity.read() );
29:                Thread.sleep( FIVE_SECONDS );
30:            }
31:        }
32:    }
33:
34:    private class TemperatureTx extends Thread {
35:        Socket sock_n2;
36:        OutputStream outRF_n2;
37:        LTemperatureSnr temperature = (LTemperatureSnr) LiSANDRA.Resources(LTemperatureSnr);
38:        public TemperatureTx ( int node_addr, int port ){
39:            sock_n2 = new Socket( COMM_MEDIA, node_addr, port );
40:            outRF_n2 = (OutputStream) sock_n2.getOutputStream();
41:        }
42:        void run(){
43:            while(true){
44:                outRF_n2.write("T:" + temperature.read());
45:                Thread.sleep( FIVE_SECONDS );
46:            }
47:        }
48:    }
49:}

```

Figura 9.6 Aplicación que muestra el uso de *sockets*, *threads* y *sensors*.

Capítulo 10

Conclusiones.

El desarrollo de software embebido para las WSN es complicado, generalmente este hace uso del lenguaje de programación C implementado dentro de sus herramientas de desarrollo. Cuando esto no es así, se implementan paradigmas de programación muy distintos de los comúnmente usados en la programación de sistemas clásicos.

Una buena aproximación para simplificar el desarrollo de software en este tipo de sistemas está en la adopción de la tecnología de Java. Para esto, es requerido de una middleware del tipo máquina virtual compatible (MiniVM) con el *bytecode* de Java como *entorno de ejecución* para agilizar el desarrollo de aplicaciones, además de proveer seguridad al momento de la ejecución y por último la abstracción de alto nivel de la plataforma. En consecuencia, esto hace la curva de aprendizaje pequeña, de tal forma que inclusive profesionales fuera del área de la computación pueda desarrollar su propias aplicaciones en este tipo de redes.

Obviamente al estar de intermediaria, la MiniVM, debe de implementar un núcleo de sistema. Este es de tipo de tiempo real y permite la programación multi-hilo propias de Java, además administra los recursos compartidos y tiene un planificador basado en la prioridad dada por la especificación de Real-Time for Java Expert Group.

Por otra parte, para la programación en el lenguaje Java en las WSN, es tan solo requerido un subconjunto de la especificación del lenguaje, ya que si bien esta tiene elementos relevantes para las redes híbridas, los nodos dentro de las WSN están limitados de recursos y fuente de poder. A fin de lograr esto se hace uso de una biblioteca independiente de plataforma como un Framework de Aplicación para WSN. Este grupo de *clases* estan basadas en la edición estándar de Java 2. Esto ayuda al programador a mantenerse enfocado en la aplicación, dándole el manejador interno de hardware que la middleware tiene desde la perspectiva de

la aplicación.

Para la comunicación entre los nodos WSN el uso de los *BSD sockets* en la programación facilita el intercambio de datos durante la programación. Independientemente de que no esta implementada como tal a bajo nivel, este modo permite al programador ver la red como un grupo de conexiones que se abren o se cierran de acuerdo a las necesidades de intercambio de datos.

Por último, tener una abstracción completa del hardware en el nodo-SR, hace posible extender la Framework de Aplicación con otros grupos de clases que describan ámbitos de aplicaciones.

10.0.5 Trabajo Futuro.

Para facilitar el uso del Framework desarrollado debe de ser adoptado el uso de un ambiente de desarrollo popular como Eclipse o NetBeans, obviamente con la Framework desarrollada referenciada en la ruta (*classpath*) o en la dependencia de bibliotecas en la que se este codificando la aplicación (asistente de código dentro de NetBeans o asistente de contenido en Eclipse) pero al compilarse este genera un archiv del tipo *jar*. En este archivo además de tener todos los archivos *classfile* utilizados de forma comprimida, la cuál deba de ser descomprimida para su ejecución. Realizar esta tarea requiere una gran cantidad de recursos de computación y esta es la razón porque *mixclass* es usada. Por esta razón, se requiere de un “plug-in” en estos ambientes, el cual use *mixclass* para crear el archivo final de la aplicación.

Adicionalmente, para la depuración (con la aplicación ejecutandose dentro del nodo-SR) es necesario el uso de un observador la cual tiene que comunicar constantemente el estado de la plataforma, con la ejecución de la aplicación a la PC, este debe de ser implementado en la middleware, con posibilidades para recibir el comando *trace* para el rastreo del programa en ejecución. De igual forma se requiere de un “plug-in” en el ambiente de desarrollo para comunicar y desplegar este estado al programador.

Para poder asegurar la extensión de la Framework de Aplicación, se debe de experimentar

el desarrollo de aplicaciones en los distintos ámbitos a fin de cumplir con los requisitos genéricos de las WSN en cada uno de ellos. Así mismo, la abstracción del nodo basado en Java, permite la creación de Frameworks de desarrollo que favorecen la visión de las WSNs como una red de servicios. Algunas de estas abstracciones son descritas en las especificaciones de JINI [134] y JXTA [135]. Sin embargo, deben de ser estudiadas a mayor detalle para su adecuada adopción dentro de las WSN.

Además de permitir el desarrollo de aplicaciones centradas en el nodo-WSN la plataforma puede ser adaptada a su uso en la programación para el Internet de las cosas (IoT) [136]. Obviamente, en aquellos nodos dentro del Internet de las Cosas que requieran de programas para el procesamiento y transmisión de la información generada dentro de la red.

Otra parte importante en el desarrollo futuro es la implementación de la **serialización** de objetos, parte de la naturaleza de Java, donde objeto completo (en el caso de nuestra implementación representaría una aplicación) puede ser representado por una secuencia de *bytes* donde se incluyen los datos del objeto (como información) y el tipo de datos almacenados en él. Este puede ser almacenado como un archivo y puede ser **deserializado** para recuperar de manera completa el objeto en el estado en que fue guardado. Esto serviría enormemente para la migración de *threads*, donde la serialización del *thread* permite que este pueda ser transmitido a través de la WSN a un nodo en específico dando soporte para diseminación de código [137] simplificando el despliegue y programación de las WSN.

Finalmente, haciendo uso de este Framework es posible crear un Framework que soporte la programación orientada a agentes en las WSN [138]. Para esto se requiere del diseño de un nuevo grupo de clases las cuales describirían los principios de la programación orientada a agentes, como proposito, autonomía y reactividad [139],[140] y [141].

Referencias

- [1] J. Hill and D. Culler, “MICA: A Wireless Platform for Deeply Embedded Networks.” *IEEE Micro* 22(6): 12-24, 2002
- [2] TelosB Mote Platform. CrossBow product documentation, 2009.
- [3] Tmote Sky. Ultra low power IEEE 802.15.4 compliant Wireless Sensor module. Moteiv Corporation, 2006.
- [4] Tmote Invent. User’s Manual. Moteiv Corporation, 2006.
- [5] Sentilla At a glance. Infraestructure Intelligence. Sentilla Corporation, 2013.
- [6] TinyNode. User’s Manual. Shockfish, 2010.
- [7] Raymond S. Wagner, “Standard-Based Wireless Sensor Networking Protocols for Spaceflight Applications,” *WiSens4Space* 2009, Conference Paper.
- [8] 2.4 GHZ IEEE 802.15.4/ZIGBEE RF TRANSCEIVER. Texas Instrument product documentation, 2007.
- [9] MSP430x1xx Family User’s Guide. Texas Instruments product documentation, 2004.
- [10] ATmega 128(L) Preliminary Complete. ATmel product documentation, 2004.
- [11] Ying Liang, “An Energy-Efficient Routing Protocol Based Dynamic Cooperative MIMO for Wireless Sensor Network,” *ICINIS* 2010
- [12] Waltenegus Dargie and Christian Poellabauer, *Fundamentals of Wireless Sensor Networks: Theory and Practice*, John Wiley, Ltd. 2010.
- [13] G. Boriello and R. Want, “Embedded Computation Meets the World Wide Web,” *Communications of the ACM*, 43(5):59-66, 2000.

- [14] I.F. Akyildiz, W. Su, Y. Sankarasuramaniam, and E. Cayirci, "A survey on Sensor Networks," *IEEE Communications Magazine*, Vol. 40, no. 8, 2002, pp. 102-114.
- [15] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless Sensor Networks: a survey," *Computer Networks Journal*, Vol. 38, No. 4, 2002, pp. 393-422.
- [16] G. Asada, M. Dong, T.S. Lin, F. Newberg, G. Pottie and W. J. Kaiser "Wireless Integrated Network Sensors: Low power Systems on a chip," *In Proceedings of 1998 European Solid State Circuits Conference*, The Hague, Netherlands, 1998.
- [17] G.J. Pottie and W.J. Kaiser. "Embedding The Internet: Wireless Integrated Network Sensors. *Communications of the ACM* 43(5):51-58, 2000
- [18] Intel StringARM SA-1100 Microprocessor Brief Data Sheet. Intel product documentation, August 2000.
- [19] MicroChip PIC Family User's Guide. Arizona Microchip product documentaion, 2006.
- [20] The PROFIBUS Technology and Application - System Description. PROFIBUS Users, 2013.
- [21] LonWorks Fundamentas -A Guide to a Basic Understanding of the LonTalk Protocol. North Beach COnsulting, LLC, 2003.
- [22] CAN Specification 2.0 CAN in Automation, Am Weichselgarten 26, 2010
- [23] TR1001 868.35 Mhz Hybrid Transceiver. RF Monolithics, Inc. 2012.
- [24] CC1000 Single Chip Very Low Power RF Transceiver. Texas Instruments SWRS048A Data Sheet.
- [25] TDA525x Series ASK/FSK Transceiver Family. Product Brief Infineon.
- [26] V. Raghunathan, C. Schurgers, S. Park, and M.B. Srivastava, "Energy-Aware Wireless Sensor Networks," *IEEE Signal Processing Magazine*, Vol. 19: 40-50, 2002.

- [27] K. Chakrabarty, S. S. Iyengar, H. Qi and E. Cho. "Coding Theory Framework for Target Location in Distributed Sensor Networks," *Proceedings of the International Symposium on Information Technology: Coding and Computing* pages 130-134, Las Vegas, NV, 2001.
- [28] K. Chakrabarty, S. S. Iyengar, H. Qi and E. Cho. "Grid Coverage for Surveillance and Target Location in Distributed Sensor Networks," *IEEE Transactions on Computers*, 51(12):1448-1453, 2002
- [29] H. Kopetz, *Real-Time Systems Design Principles for Distributed Embedded Applications*, Kluwer Academic Publisher, Dordrecht, The Netherlands, 1997.
- [30] A. Chandrakasan, S. Sheng and R. Brodersen, "Low-Power CMOS Digital Design," *IEEE Journal of Solid-State Circuits*, 27(4): 473-484, 1992.
- [31] J. M. Khan, R. H. Katz and K. S. J. Pister, "Emerging Challenges: Mobile Networking for Smart Dust," *Journal of Communications and Networks*, 2(3): 188-196, 2000.
- [32] J. M. Rabaey, M. J. Ammer, J. L. da Silva, D. Patel and S Roudy, "PicoRadio Supports Ad Hoc Ultra-Low Power Wireless Networking," *IEEE Computer* 33(7): 42-48, 2000.
- [33] S. Roundy, D. Steingart, L. Frechette, P. Wright and J. Rabaey, "Power Sources for Wireless Sensor Networks," *Proceedings of 1st European Workshop on Wireless Sensor Networks (EWSN)* pages 1-17, LNCS Springer volume 2920 January 2004.
- [34] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister, "System Architecture Directions for Networked Sensors. " *Communication of the ACM* 47(6): 41-46, 2000.
- [35] Karl, H and Willig, A, *Protocols and Architectures for Wireless Sensor Networks*, Editorial John Wiley & Sons Ltd. ISBN 13978-0-470-09510-2.

- [36] Dulman, S. Havinga, P., "Architectures for Wireless Sensor Networks." *Proceedings of the IEEE ISSNIP 2005* pp. 31-38.
- [37] NeTS-NOSS: Creating and Architecture for Wireless Sensor Networks, (2007). Disponible en <http://snap.cs.berkeley.edu/documents/architecture.pdf>
- [38] F. Zhao and L. Guibas, *Wireless Sensor Networks: An Information Processing Approach*, Morgan Kaufman Publishers, San Francisco, CA, 2004.
- [39] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J Anderson, "Wireless sensor networks for habitat monitoring," *Proceedings of 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, Sept. 2002 pp 88-97.
- [40] R. Jagari, A. Encarnacao, A. Zahoory, F. Dabiri, H. Noshadi and M. Sarrafzadeh, "Wireless sensor networks for health monitoring," *Proceedings of 2nd Annual International Conference on Mobile and Ubiquitous systems: Networking and Services (MobiQuitous'05)*, July 2005, pp 479-481.
- [41] D. Trossen and D. PAvel, "Sensor networks wearable computing and healthcare Applications," *IEEE Pervasive Computing* vol. 6, no.2, App-June 2007, pp58-61.
- [42] C. Herring and S. Kaplan, "Component-based software systems for smart environments," *IEEE Personal Communications*, Vol. 7, no. 5, Oct 2000, pp. 60-61.
- [43] L. Aguilar, G. Licea, G. and J. A. García-Macias, "An Experimental Wireless Network applied to engineering courses," *Computer Applications in Engineering Education*, Wiley-InterScience, May 2009.
- [44] L. Aguilar, *LiSANDRA: An Experimental Wireless Sensor Network*, PhD thesis, Universidad Autónoma de Baja California, 2009.

- [45] J. J. Evans, "Undergraduate Research Experiences with Wireless Sensor Networks," 37th ASEE/IEEE Frontiers in Education Conference, IEEE CS Press, Milwaukee, Wisconsin, USA, 2007, pp. 7-12.
- [46] J. Frolik and T. Keller, "Wireless Sensor Networks: An Interdisciplinary Topic for Freshman Design," *American Society for Eng. Education Annual Conference & Exp.*, ASEE, Portland, Oregon, USA, 2005, pp. 1-7
- [47] M. Ilyas and I. Mahgoub, *Handbook of Sensor Networks: Compact Wireless and Wired Sensing Systems*, CRC Press, Florida 2004
- [48] L. Aguilar, G. Licea, "LEARN: Light-weight Embedded Academic Real-Time Nucleus," *Internal technical report*, Universidad Autónoma de Baja California, 2000.
- [49] A.K. Dwivedi, O. P. Vyas, *Wireless Sensor Networks: At a Glance*.
- [50] Blumenthal J, Handy M, Golatowski F, et al., "Wireless Sensor Networks -New challenges in software engineering," *In proceedings of the 2003 IEEE Conference on Emerging Technologies and Factory Automation (ETFA'03)*, pp. 551-556.
- [51] R. E. Schantz, D. C. Schmidt, *Middleware for Distributed Systems: Envolving the Common Structure for Network Centric Application*, Wiley & Sons, New York, 2001.
- [52] Miao-Miao Wang, Jian-Nong Cao, Jing Li and Sajal K. Das, "Middleware for Wireless Sensor Networks: A Survey," *Journal of Computer Science and Technology*, May 2008.
- [53] K. Sohraby, D. Minoli, T. Znati, *Wireless Sensor Networks Technology. Protocols and Applications*, Wiley 2007.
- [54] P. Levis and David Culler, *Maté: A Tiny Virtual Machine for Sensor Networks*, University of California Berkley, California.
- [55] <http://labs.oracle.com/projects/squawk/squawk-rjvm.html>, Marzo 2011.

- [56] <http://research.microsoft.com/apps/video/default.aspx?id=104765>, Marzo 2011
- [57] Johannes Gehrke, Samuel Madden, "Query Processing in Sensor Networks," *Pervasive Computing*, 2004.
- [58] S. Li, S. H. Son, J. Stankovic, "Event Detection Services Using Data Services Middleware in Distributed Sensor Networks," *Proceedings of the Workshop on Information Processing in Sensor Networks (IPSN'03)*, Palo Alto, CA, 2003.
- [59] Srisathapornphat, C. Jaikaeo y C. Chien-Chung Shen, "Sensor Information Networking Architecture," *International Workshops on Parallel Processing*, 2000.
- [60] F.C. Delicato, P. Pires, J. Ferreira, L. Rust, Service Oriented Middleware for Wireless Sensor Networks.
- [61] E. Souto, G. Guimaraes, G. Vasconcelos, M. Vieira, et al., "Mires: a publish/subscribe middleware for sensor networks," *Pers Ubiquit Comput.* 2006.
- [62] A. Ribeiro, F. Silva, L. Freitas, et al., "SensorBus: a middleware model for wireless sensor networks," *LANC'05 Proceedings of 3rd. International IFIP/ACM Latin American conference on Networking*, USA 2005.
- [63] Jeffret M. Bradshaw, KaoS: An Open Agent Architecture Supporting Reuse, Interoperability, and Extensibility. (<http://ksi.cpsc.ucalgary.ca/KAW/KAW96/bradshaw/KAW.html>)
- [64] <http://agent.omg.org/> , Noviembre 2012.
- [65] <http://www.fipa.org/> , Marzo 2010.
- [66] D. Barnes, S. Sankaranarayanan, S. Ganesan, "Performance Analysis of Client/Server Agent Based Communication in Wireless Sensor Networks For Health Applications." *IEEE Journal*, 2009.

- [67] M. CHen, T. Kwon, Y. Yuang, "Mobile Agent Based Wireless Sensor Networks," *Journal of Computers*, April 2006.
- [68] <http://www.tinyos.net> , Diciembre 2012.
- [69] Amol B. Bakshi, Viktor K. Prasanna, "Architecture-Independent Programming for Wireless Sensor Networks." Wiley 2009.
- [70] L. Mottola, G. P. Picco, "Programming Wireless Sensor Networks: Fundamental Concepts and State of Art." *ACM Journal*, Vol. V, No. N, Month 20YY.
- [71] <http://www.contiki-os.org/> , Diciembre 2012.
- [72] <http://particle.teco.edu> , Diciembre 2012.
- [73] J. Lifton, D. Seetharam, M. Seltzer, J. Paradiso, Bertha: The OS for Pushpin Computers.
- [74] <http://www.ethernut.de/en/software/index.html> , Diciembre 2012.
- [75] M. O. Papazoglou, "Service-Oriented Computing: COnccepts, characteristics and directions." WISE.
- [76] R. Flenner, M. Abbott, T. Boubez, et. al., Java P2P Unleashed. With JXTA, Web Services, XML, Jini, JavaSpaces and J2EE. Editorial Sams.
- [77] I. Amundson, M. Kushwaha, X. Koutsoukos, OASIS: A Service-Oriented Middleware for Pervasice Ambient-Aware Sensor Networks.
- [78] E. Aviles-Lopez, J. GarciaMacias, TinySOA: a service-oriented architecture for wireless sensor networks.
- [79] Chenyang Lu, Servilla: Service Oriented Computing in Wireless Sensor Networks.
- [80] F. C. Delicato, et. al., "Reflective middleware for wireless sensor networks.", *SAC'05*, Santa Fe, New México, USA, 2005.

- [81] F. C. Delicato, et. al., “A felxible middleware system for wireless sensor networks.” *IFIP, LNCS 2672*, pp. 474-492, 2003.
- [82] <http://www.eclipse.org> ,Abril 2011.
- [83] <http://www.microsoft.com/visualstudio/en-us> ,Abril 2011.
- [84] <http://www.embarcadero.com> ,Junio 2011.
- [85] J. D. Monhammad Al Saad, Jochen Schiller, “ScatterEditor: An Eclipse based Tool for Programming. Testing and Managing Wireless Sensor Networks,” *SENSORCOMM*, Valencia, España, 2007.
- [86] <http://www.cygwin.com> ,Agosto 2011.
- [87] <http://www.tinydt.sourceforge.net> ,Diciembre 2010.
- [88] <http://tinysize.net> ,Enero 2010.
- [89] <http://selab.csuohio.edu/tosdev/> ,Agosto 2011.
- [90] N. Burri, R. Schuler, R. Wattenhofer, “YETI: A TinyOS Plug-in for Eclipse.”
- [91] E. Cheong, E. A. Lee, Y. Zhao, “Viptos: A Graphical Development and Simulation Environment for TinyOS-based Wireless Sensor Networks.”, *Electrical Engineering and Computer Sciences University of California at Berkeley*, 2006.
- [92] <http://www.isis.vanderbilt.edu/projects/nest/gratis/index.html> ,Junio 2010.
- [93] <http://www.atmel.com/dyn/products/> ,Junio 2010.
- [94] <http://www.iar.com/en/Products/IAR-Embedded-Workbench/TI-MSP430/> ,Junio 2010.
- [95] <http://sourceforge.net/apps/mediawiki/mspgcc/> , Junio 2010.

- [96] C. Cota, “Un Middleware para las Redes De Sensores Inalámbricas”, Master Thesis, Universidad Autónoma de Baja California, 2008.
- [97] Tim Lindholm, Frank Yellin, “The Java Virtual Machine Specification”, Sun Microsystems, 1999.
- [98] Donald Alpert, “A Pascal P-Code Interpreter for the Stanford Emmy”, Technical Note No. 164, Departments of Electrical Engineering and Computer Science, Stanford, California, 1979.
- [99] <http://www.sunspotworld.com/> , Diciembre 2012.
- [100] Michael Sipser, “Introduction to the Theory of Computation”, Thompson, 2006.
- [101] James Gosling, Bill Joy, “The Java Language Specification. Third Edition”, Addison-Wesley, 1996.
- [102] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, Jan Vitek. “High-level Programming of Embedded Hard Real-Time Devices”, *EuroSys’10* April 13-16, 2010, Paris, France. ACM
- [103] Auerbach, J.S., Bacon, D.F., Et. Al. “Design and implementation of a comprehensive conference on Embedded software” (EMSOFT) 2007, pp. 349-258.
- [104] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers, Craig Chambers. “Extensibility, Safety and Performance in the SPIN Operating System”, *Proceedings of the 15th Symposium on Operating Systems Principles*, pp 267-284, Copper Mountain, Colorado, 1995
- [105] HAVi, <http://www.havi.org>, Noviembre 2012
- [106] John K. Ousterhout, “Scripting: Higher-Level Programming for the 21st Century”, 1998, IEEE.

- [107] Nik Shaylor, Et.Al., “A Java Virtual Machine Architecture for Very Small Devices”, *ICTES’03 June 11-13*, 2003, San Diego, California, USA.
- [108] Thomas Fuhrmann and Till Harbaum, “A Platform for Lab Exercises in Sensor Networks”, Universidad Karlsruhe.
- [109] Memo Lindwer, “Java in Embedded Systems”, *Xootic Magazine*, May 2001.
- [110] D. Verkest, Et. Al., “CoWare-A Design Environment for Heterogeneous Hardware/Software Systems”, *Design Automation for Embedded Systems*, pp 357-286, October 1996.
- [111] S. Wuytack, Et. Al, “Memory Management for Embedded Network Applications”, *IEEE Transaction On Computer-Aided Design of Integrated Circuits and Systems*, May 1999.
- [112] Mohamed Shalan, Vicent J. Mooney, “A Dynamic Memory Management Unit for Embedded Real-Time System-on-a-Chip”.
- [113] P. R. Wilson, Et. Al., “Dynamic Storage Allocation: A Survey and Critical Review”, *Proceedings 1995 Int’l workshop on Memory Management*, Scotland. September 1995.
- [114] Andre Ventevogel, Jeroen Voeten, “Platform-independent Design for Embedded Real-time Systems”.
- [115] L. Lavagno, S. Dey, and R. Gupta, “Specification, modeling and design tools for system-on-chip”. In *Design Automation Conference, 2002. Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design*. Proceedings, pages 21-23. IEEE Computer Society Press, 2002.
- [116] A. Sintotski, D. K. Hammer, O. van Rossmale and J. Hooman. “Formal platform-independent design of real-time systems”. In *Proceedings 13th Euromicro Conference on Real-Time Systems*, pages 163-170. IEEE Computer Society Press, 2001

- [117] The Free Software Foundation, “The GNU Compiler for the Java Programming Language”, <http://www.gnu.org/software/gcc/java>
- [118] Per Bothner, “Compiling Java for Embedded Systems”. *IEEE Compcon*, Spring 1997.
- [119] Lars Raeder Clausen, Et. Al., “Java Bytecode Compression for Low-Embedded Systems” *ACM Transactions on Programming Languages and Systems*, Vol. 22, No. 3, May 2000, Pages 471-489.
- [120] A. Krall, R. Grafl, “CACAP - A 64 bit JavaVM Just-in-Time Compiler”, *Institut fuer Computersprachen* Technische Universitaet Wien, Vienna, 1998.
- [121] Sun Microelectronics, “JavaSoft: The Source for Java Technology”, <http://www.javasoft.com>
- [122] Holger Karl, Andreas Willig, *Protocols and Architectures for Wireless Sensor Networks* John Wiley, 2005
- [123] BTnodes A distributed Environment for Prototyping Ad Hoc Networks, October 2004. Website <http://www.btnode.ethz.ch>
- [124] Bruce P. Douglass, *Real-Time UML: Developing Efficient Objects for Embedded Systems*, Pearson Education, 1999.
- [125] Bruce P. Douglass, *Real Time UML: Advances in the UML for Real-Time Systems*, Addison-Wesley Professional, 2004.
- [126] A. Pantelopoulos and N. Bourbakis, “A survey on wearable biosensor systems for health monitoring,” *Conf Proc IEEE Eng Med Biol Soc*, vol. 2008, pp. 4887-90, 2008.
- [127] R. Madden, M. J. Franklin, J. M. Hellerstein, W. Hong, “TinyDB: An Acquisitional Query Processing System for Sensor Networks,” *ACM Transactions on Database Systems*.

- [128] Cota C., Aguilar L., Licea G., “A Java-compatible Virtual Machine as an Embedded Middleware for Wireless Sensor Networks”, CERMA 2010, Cuernavaca, México.
- [129] E. Douglas Jense, “The Real-Time Specification for Java”, Sun Microsystems, 2000.
- [130] Intel Wireless promise, Accessed Sept. 30 2010.
http://blogs.intel.com/intellabs/2008/10/02/rattner_the_promise_of_wireles/ pdf
- [131] Greg Bollela, Et. Al., The Real-Time for Java Expert Group, Addison Wesley, 2002.
- [132] Teemu Laukkarinen, Jukka Suhonen, Et. Al., A Survey of Wireless Sensor Network Abstraction for Application Development, International of Distributed Sensor Networks, Hindawi Publishing Corporation, 2012.
- [133] K. Sohraby, D. Minolu, T. Znati, Wireless Sensor Networks Technology. Protocols and Applications, Wiley 2007.
- [134] Waldo Jim, “The JINI specification”, Second Edition, Sun Microsystems 2001.
- [135] Brookshier Daniel, Et. Al. “JXTA: Java P2P Programming”, SAMS 2002.
- [136] CASAGRAS EU project final report, <http://www.rfidglobal.eu/userfiles/documents/Finalreport.pdf> 2009.
- [137] Qiang Wang, Et. Al., Reprogramming Wireless Sensor Networks: Challenges and Approaches, IEEE Network, June 2006.
- [138] Lesser V, Et. Al., Distributed Sensor Networks: A Multiagent Perspective, Klumer Academic, New York, 2003.
- [139] Yoav Shoham, Agent-oriented programming, Elsevier, 1993.
- [140] Gerhard Weiss, Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence, MIT, 2001.
- [141] Mark d’Inverno, Et. Al., Understanding Agent Systems, Springer, 2001.