Metodología de diseño para dispositivos empotrados de escasos recursos conectados en red

David Villa, Cleto Martín, Félix Jesús Villanueva,
Francisco Moya, María José Santofimia, Juan Carlos López

Escuela Superior de Informática

Universidad de Castilla-La Mancha

{david.villa, cleto.martin, felix.villanueva, francisco.moya, mariajose.santofimia, juancarlos.lopez}@uclm.es

Resumen

El desarrollo de sistemas empotrados sigue siendo complejo y tedioso, sobre todo para el personal con poco entrenamiento. La adaptación a las herramientas y sistema de trabajo habitual en este entorno suele ser lenta debido a que puede implicar importantes diferencias respecto al desarrollo de software convencional para PC.

El problema se agrava cuando el sistema empotrado debe cooperar estrechamente con sistemas convencionales de la infraestructura IT de la organización. Dicha cooperación requiere necesariamente comunicaciones a través de una red de datos.

Este trabajo propone una metodología y un conjunto integrado de herramientas para el desarrollo de sistemas empotrados con graves restricciones —en cuanto a prestaciones de cómputo— que se comunican entre sí y con el sistema de información convencional por medio de una red de datos.

1. Introducción

El diseño e implementación de sistemas empotrados implica con frecuencia largos periodos de depuración y pruebas. La dificultad intrínseca reside en analizar el comportamiento interno de un dispositivo de cómputo independiente y con acceso limitado. En concreto, nuestro campo de aplicación son dispositivos con importantes limitaciones en cuanto a capacidad de cómputo y memoria: microprocesadores de baja gama y microcontroladores de 16 u 8 bits. En adelante cuando se hable de *sistema o dispositivo empotrado* nos estaremos refiriendo a éstos y no a sistemas como DSP's, PC's enracables u otros dispositivos mucho más potentes que habitualmente también reciben el adjetivo *empotrado*.

Ciertamente hoy en día existe un amplio abanico de posibilidades que facilitan enormemente el trabajo del programador. Podemos clasificarlas en:

- Simuladores, con capacidad para reproducir hasta los más pequeños detalles de la plataforma (incluso periféricos).

- Depuradores empotrados, que permiten ejecutar el programa paso a paso, conocidos como ICD (*In-Circuit Debugger*) o ICE (*In-Circuit Emulator*).

A pesar de todo ello, se requiere formación específica para el uso provechoso de estas herramientas, lo que implica una curva de aprendizaje larga y que el personal no es productivo durante un período no despreciable.

El motivo más importante parece ser el bajo nivel al que se requiere trabajar a pesar de que el objetivo sea de alto nivel. Estos dispositivos se programan en la gran mayoría de los casos en lenguajes como C o ensamblador (ver figura 1). Aunque existen sistemas que permiten utilizar lenguajes como Java, C#, Visual Basic o incluso utilizando metáforas gráficas suelen estar restringidos a entornos y plataformas muy específicas y no es posible generalizar su uso.

El reto es crear herramientas que permitan realizar una especificación de comportamiento de alto nivel, que pueda ser aplicable a cualquier plataforma empotrada, pero que permita aislar al programador de las peculiaridades arquitecturales propias de cada dispositivo.

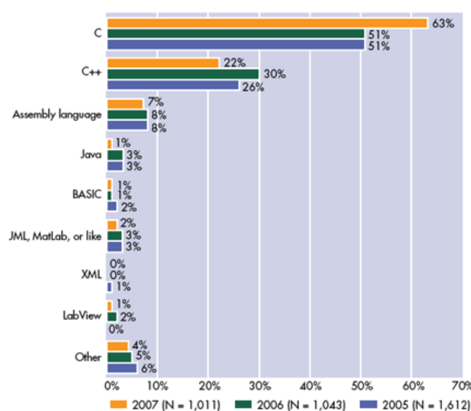

Figura 1: Lenguajes de programación en sistemas empotrados (fuente: http://www.embedded.com)

Además, consideramos dispositivos que pueden comunicarse entre sí y con el sistema de información convencional por medio de redes de datos de tecnología arbitraria. Si estos dispositivos tienen capacidades de sensorización y/o actuación estamos hablando de un superconjunto de las denominadas SAN (*Sensor Actuator Networks*) aunque ése es solo un caso paradigmático ya que existen otras aplicaciones posibles: control industrial, domótica, inmótica, sistemas integrados para tele-control, telemedida, etc.

2. Aplicaciones distribuidas

La programación distribuida clásica entiende la ejecución de porciones de código en computadores remotos como una generalización de la programación convencional. En la programación distribuida orientada a objetos, estas porciones de código se corresponden con métodos de objeto, en el contexto de la POO (*Programación Orientada a Objetos*). Eso permite hacer diseños más cohesivos, flexibles y con mucho menos acoplamiento que otras alternativas como RPC (*Remote Procedure Call*). Esto es posible gracias a los llamados middlewares de comunicaciones. Algunos ejemplos de este tipo de plataforma pueden ser CORBA, Java RMI, .Net Remoting, ZeroC Ice, etc.

En estas plataformas el diseño de la aplicación está fuertemente condicionado por la definición de las «interfaces remotas» que han de proporcionar los servidores (objetos) y usar los clientes. Las interfaces[1] son el contrato entre los distintos componentes los sistema; aquellos que proporcionan servicios y los que los consumen. Además se define una segunda interfaz menos evidente: el protocolo de aplicación propio del middleware.

Ambos interfaces (a nivel de programación y de protocolo) permiten que el acoplamiento entre las entidades mencionadas sea mínimo. Pueden utilizarse arquitecturas, sistemas operativos y lenguajes distintos siempre que se respeten las interfaces y el protocolo establecido. Llevando esta situación al extremo es posible desarrollar dispositivos capaces de asumir alguno de los roles (cliente/objeto) implementando la funcionalidad imprescindible para asumir esos dos condicionantes.

La gran ventaja práctica que conlleva este enfoque es que en un mismo sistema distribuido puede haber, a la vez, implementaciones muy dispares en todos los aspectos mencionados (lenguaje, SO, arquitectura, etc.) pero también dispositivos empotrados. Todos estos elementos pueden trabajar conjuntamente de forma completamente transparente.

La importante reducción de acoplamiento y la posibilidad de interoperabilidad con los middlewares de comunicaciones de propósito general son suficientemente interesantes como para justificar esta investigación. Sin embargo, hemos llevado aún más lejos las implicaciones del paradigma de invocación a método remoto. Consideramos el middleware de comunicaciones OO también como marco de referencia en otros aspectos:

- Desde el punto de vista conceptual, hemos llevado las abstracciones habituales en los sistemas distribuidos orientados a objetos a la especificación de la funcionalidad de los dispositivos, explicitando esos conceptos incluso en el propio fichero fuente del programa. Este hecho queda plasmado en la creación de un nuevo lenguaje de programación denominado *IcePick*.

- Desde el punto de vista arquitectural, proporcionamos un entorno de ejecución para objetos distribuidos portables. Esto es posible gracias

[1] Las interfaces remotas se escriben en un lenguaje específico dependiente del middleware: IDL, Slice, XDR, etc.

al concepto de *picoObjeto*. En lo concerniente al sistema IT, los picoObjetos se comportan como objetos distribuidos convencionales que hubieran sido desarrollados de la forma habitual (utilizando las herramientas proporcionadas por el fabricante del middleware).

La especificación del picoObjeto incluye toda la información necesaria para el reconocimiento y generación de mensajes para protocolos de aplicación arbitrarios, incluyendo los procedimientos de (de)serialización. Dicha especificación se proporciona como un programa FSM: *bytecode* que incluye todos los detalles de una máquina de estados finitos (de ahí su nombre). Para ejecutar dicho código se requiere una implementación de la máquina virtual FSM. Lo interesante es que este *bytecode*, y por tanto, la máquina virtual que lo ejecuta es extremadamente simple. Una implementación funcional completa de la V-ISA (*Virtual Instruction Set Architecture*) de FSM en lenguaje ANSI C requiere 683 SLOC. Por tanto, adaptar cualquier dispositivo de cómputo para que pueda ejecutar picoObjetos es una tarea relativamente simple. Este sistema permite implementar un objeto distribuido no trivial incluso en microcontroladores de 8 bits con 128 bytes de RAM y 4K palabras de memoria de programa. En [10] se aborda en profundidad el concepto de picoObjeto y su funcionamiento detallado.

3. Trabajo previo

A continuación se describen brevemente los diferentes modelos y enfoques que se utilizan actualmente en la programación de sistemas empotrados en red, así como algunas de las soluciones comerciales más utilizadas.

Modelos de programación

Desde el punto de vista del programador de los nodos empotrados de la red, existen dos paradigmas para modelar su comportamiento y, por tanto, definir la aplicación:

- *Programación dirigida por eventos*: el comportamiento de los nodos de la red puede verse como una máquina que reacciona ante eventos programables (internos o externos).
- *Máquinas de estado*: el comportamiento se define en términos de estados y transiciones entre ellos.

Sin embargo, la arquitectura y estructura de las redes de dispositivos empotrados también influye en la filosofía de programación de los nodos. A continuación se muestran algunos enfoques arquitecturales actuales:

- *Base de datos*: este enfoque de desarrollo muestra la red de dispositivos empotrados como una base de datos virtual que puede ser manipulada utilizando sentencias muy similares a SQL para consultar y cambiar el estado de los nodos de la red.

  Un buen ejemplo de este modelo de desarrollo es TINYDB[2]. Las aplicaciones para TINYDB se ejecutan sobre TINYOS[3] y pueden ser construidas en Java y nesC.
- *Macro-programación*: el modelo de macro-programación consiste en dos fases bien diferenciadas. En primer lugar, se define todo el sistema de red: los nodos que lo forman así como su comportamiento en un lenguaje de alto nivel. Una vez descrito, se realiza un proceso de despliegue en el que se carga en cada nodo el software correspondiente.

  Ejemplos de plataformas que utilizan este tipo de paradigma son Maté [8, 7], Magnet [2] y SensorWare [5, 4].
- *Clusters*: la red de dispositivos se divide en conjuntos (*clusters*) donde uno de los integrantes toma el rol de líder. Éste se encarga de la obtención y filtrado de datos y realiza tareas de rutado.

  SINA [11] y DSWARE [9] son plataformas basadas en la arquitectura *cluster*, ofreciendo un modelo de programación de base de datos.
- *Máquinas virtuales*: en este enfoque se instala en cada uno de los nodos una máquina virtual que interpreta un *bytecode* determinado. La máquina virtual abstrae al programador del acceso al hardware y hace que el código pueda ser portado a implementaciones para otras plataformas de la misma máquina virtual.

[2]http://telegraph.cs.berkeley.edu/tinydb/
[3]http://docs.tinyos.net

En Maté se utiliza una máquina virtual para la ejecución de las aplicaciones. Esta máquina proporciona un repertorio de instrucciones (ISA) con las que el programador puede construir las aplicaciones. Sin embargo, estas instrucciones son muy cercanas a las de la plataforma física subyacente. Por ejemplo, existe una instrucción para encender un LED, un recurso hardware que depende de la plataforma.

SensorWare, WSP [3] y VDM [1] también utilizan máquinas virtuales en los nodos empotrados de la red.

Muchas de las plataformas anteriores utilizan varios enfoques arquitecturales. Sin embargo, todas ellas suponen la red como un sistema aislado de la red principal. No ofrecen una visión integradora en la que nodos de tecnología tan dispar como un PC y un dispositivo empotrado puedan interactuar de igual forma que lo harían dos PC. Por ello, se hace necesario el uso de dispositivos y/o programas adaptadores entre la red principal y la red de dispositivos empotrados.

Plataformas comerciales

Waspmote es un producto creado por la empresa Libelium orientado a las redes de sensores. En estos dispositivos se pueden montar placas con sensores y actuadores y, junto con el producto, se proporcionan entornos de programación. El lenguaje de programación es C++ simplificado, restringiéndolo a la API[4] proporcionada por Libelium y a una sintaxis concreta.

Sin embargo, los entornos y librerías proporcionadas con Waspmote son exclusivamente para esta plataforma. De hecho, de forma parecida al problema de Maté, los componentes del API proporcionado dependen exclusivamente del hardware concreto.

Otros dispositivos reseñables son la gama Sun SPOT[5] y Lego Mindstorms que tienen implementaciones de la máquina virtual de Java (JVM), por lo que son programables en este lenguaje. Más concretamente, la máquina virtual de Sun SPOT (conocida como Squawk) es una versión J2ME compatible con la especificación CLDC 1.1[6] de Java. Por otro lado, LEJOS[7] es un ejemplo de JVM para dispositivos Lego Mindstorms.

Para ambas plataformas existen numerosas herramientas de desarrollo que facilitan la construcción de aplicaciones proporcionadas por el fabricante o por terceras partes. Por ejemplo, existen plugins para el entorno Eclipse o Netbeans para construir aplicaciones J2ME, que incluyen emuladores, depuradores y otras herramientas similares. Lego Mindstorms, por su parte, proporciona utilidades para programar con metáforas gráficas.

Sin embargo, y pese a las facilidades de portabilidad que ofrece Java, el número de plataformas destino está acotado a unos cuantos dispositivos concretos. El tamaño de una implementación mínima de una JVM y sus requisitos hardware hacen inviable la carga de estas aplicaciones en dispositivos de baja gama como microcontroladores. Además, ni Squawk ni LEJOS proporcionan mecanismos para la programación orientada a objetos distribuida, por lo que los dispositivos no son fácilmente integrables en un sistema distribuido ya implantado.

4. Descripción de escenarios

Además de lo anterior, se asume que cualquier dispositivo empotrado susceptible de ser conectado a una red puede trabajar de forma autónoma; aunque ello no evita que pueda cooperar con otros. A este dispositivo autónomo lo denominaremos «nodo» a partir de este momento.

En la gran mayoría del trabajo previo no existe una forma de especificar cómo es la estructura del nodo, con qué otros nodos tiene relación o cómo se establecen dichas relaciones.

Las aplicaciones distribuidas que utilizan un middleware de comunicaciones, como CORBA o Web Services, utilizan una declaración de interfaces y posiblemente una lista de los objetos o servicios que residen en el nodo. Pero además de esta información, nosotros proponemos añadir detalles que permitan especificar otra información: qué estímulos internos o externos puede responder, sobre qué otros nodos tendrán consecuencias, de qué forma, qué debe ocurrir cuando el nodo arranca, etc.

Ese tipo de información es útil en dos sentidos:

[4] http://www.libelium.com/development/waspmote
[5] http://uk.sun.com/specials/sunspot/spec.jsp
[6] http://java.sun.com/products/cldc/
[7] http://lejos.sourceforge.net

- Ofrece al programador una visión funcional del nodo. Le permite (entre otras cosas) conocer con detalle cuáles son las relaciones entre nodos, cómo y cuándo (bajo qué circunstancias) ocurrirá. Es decir, proporciona un enfoque holístico del sistema distribuido, o al menos de una parte.

- Proporciona a nuestro entorno de desarrollo una especificación completa a partir de la cual se puede afrontar la generación de código final, creación de baterías de pruebas útiles tanto en la fase de desarrollo con en el entorno en producción, validación de requisitos o incluso generación de documentación.

IcePick

Para materializar toda esta información hemos creado un lenguaje de programación completamente especializado: IcePick. Tal como se indicaba anteriormente, el lenguaje está diseñado de modo que el programador utiliza los conceptos básicos de los middlewares de sistemas distribuidos como elementos de construcción del programa. Veamos una breve descripción de dichos conceptos:

node El nodo es un dispositivo físico autónomo con capacidad de cómputo y comunicaciones, al menos. Aunque nuestro interés está en dispositivos empotrados de muy baja gama, esta definición tan general no limita su aplicabilidad.

endpoint Es un punto lógico de conexión a la red de comunicaciones. En redes TCP/IP implica una dirección IP y un puerto. En otro tipo de redes puede ser una dirección MAC u otro dato que permita localizar un nodo unívocamente en su dominio. Nótese que es factible que un único nodo disponga de varios *endpoints*.

object Corresponde casi exactamente con el concepto equivalente de la POO. La principal diferencia es que el objeto distribuido no representa forzosamente una instancia. Varios objetos pueden estar respaldados por una única implementación (llamada «sirviente»). Todo objeto debería tener un identificador globalmente único.

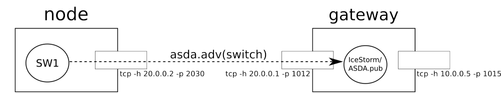

Figura 2: Esquema de la especificación descrita en el algoritmo 1

adapter El adaptador es el encargado de ofrecer los objetos a la red. Cada adaptador engloba un conjunto de objetos que podrán ser invocados desde clientes remotos a través de uno o más *endpoints*.

Un programa IcePick describe un escenario distribuido en el que existen *objetos* vinculados a *adaptadores*, que a su vez residen en *nodos*. El fichero fuente incluye también referencias a los ficheros que definen las interfaces remotas (clausula `uses`). Para cada objeto debe indicarse su tipo, es decir, alguna de las interfaces definidas en dichos ficheros. En los prototipos actuales usamos ZeroC Ice [6] como middleware de comunicaciones.

El listado 1 muestra un ejemplo sencillo en el que se declaran dos objetos. Una representación gráfica del mismo escenario se muestra en la figura 2.

```
uses "DOBS/DUO.ice";
uses "DOBS/ASDF.ice";

object DUO.IBool.W switch;
object ASD.Listener asda;

local adapter node {
   endpoint = "tcp -h 20.0.0.2 -p 2030";
   objects = {"SW1": switch};
};

remote adapter gateway {
   endpoint = ["tcp -h 20.0.0.1 -p 1012",
               "tcp -h 10.0.0.5 -p 1015"];
   objects = {"IceStorm/ASDA.pub":asda};
};
```

Algoritmo 1: Ejemplo de escenario distribuido

El objeto `switch` se añade a un adaptador local (desde el punto de vista del nodo) y es accesible a través del `endpoint` especificado. Por otro lado, se define un nodo remoto (`gateway`) en el que se encuentra registrado un objeto (`asda`) que recibe anunciamientos de los dispositivos. Se trata de un canal de eventos que implementa el protocolo ASDF [12].

IcePick permite especificar el comportamiento de los objetos utilizando un esquema dirigido por eventos. Las estructuras que permiten especificar el comportamiento son los *triggers*, es decir, bloques de invocaciones secuenciales que se ejecutarán cuando se produzca el evento asociado. Se pueden efectuar invocaciones tanto a objetos de nodos externos (remotas) como a internos (locales al nodo).

Por defecto, IcePick ofrece cuatro tipos de *triggers*:

- `boot`: es útil para definir invocaciones que deben ejecutarse en la inicialización del nodo.

- `timer(n)`: las invocaciones asociadas se ejecutarán cada `n` ticks. En el ejemplo de la figura 2, el objeto `SW1` se anuncia periódicamente a un canal de anunciamientos. Una posible implementación podría ser la siguiente:

  ```
  timer(60) { asda.adv(switch); }
  ```

  El identificador `switch` se utiliza como argumento para enviar la referencia remota al objeto, de forma que otros objetos remotos que reciban el anunciamiento puedan contactar con él.

- `event E do`: permite definir comportamiento proactivo, es decir, se ejecutarán las invocaciones del bloque asociado cuando ocurra el evento interno `E`.

- `when INVOCACIÓN do`: las invocaciones asociadas se ejecutarán cuando se realice la invocación especificada. De esta forma, es posible definir comportamiento reactivo a eventos externos.

5. Flujo de desarrollo

Aparte de las ventajas prácticas ya comentadas, este nuevo enfoque para especificación e implementación de nodos de red empotrados ofrece interesantes ventajas metodológicas que veremos a continuación.

En el desarrollo de aplicaciones distribuidas con middlewares orientados a objetos, la definición de las interfaces remotas es determinante. El diseñador (rol 'a' de la figura 3) debe decidir qué objetos habrán de alojarse en cada nodo. La elección es crucial ya que determina aspectos muy importantes: la invocación remota es de media dos órdenes de magnitud más cara (sobre todo en tiempo) que la local aparte de implicar semánticas de fallo parcial, que no ocurren en una invocación convencional. Además se debe tener presente que los mismos objetos que prestan servicios a clientes remotos también pueden hacerlo localmente, y en algunos sistemas, existe además la posibilidad de migración de objetos incluso con la aplicación en ejecución.

En las aplicaciones convencionales (utilizando PC's o sistemas sin limitaciones relevantes en cuanto a recursos) y una vez están definidas las interfaces remotas, la implementación de los clientes y objetos implica aspectos muy similares. De hecho, es muy frecuente que los programas ejerzan ambos papeles al tiempo, aunque en la mayoría de los casos hay un sesgo predominante. Eso significa que el personal encargado de la programación de clientes y servidores tiene básicamente la misma formación, requisitos y aplica las mismas consideraciones.

Sin embargo, cuando hay sistemas empotrados involucrados en la aplicación (normalmente como parte de un sistema de sensorización) suele implicar una excepción en el proceso de desarrollo. Las abundantes propuestas en middlewares para redes de sensores y dispositivos similares asumen casi por definición que dichos dispositivos deben considerarse aparte. Esto lleva a situaciones en las que el sistema de información de la organización utiliza herramientas, abstracciones y protocolos distintos que las redes de dispositivos. Mediante el uso de pasarelas se representan los dispositivos empotrados por medio de delegados de modo que aparecen en la red troncal como si se tratara de objetos convencionales.

El uso de picoObjetos permite eliminar este tipo de pasarelas, pero a pesar de ello, seguía existiendo un salto muy importante entre la implementación de los objetos distribuidos dentro y fuera de la red de sensores. Gracias a IcePick esta diferencia es mucho menor. Aunque la implementación de un nodo convencional y uno empotrado sigue difiriendo, ahora se emplean los mismos conceptos (introducidos en la sección 4) de modo que no se requiere personal con una formación y experiencia tan radicalmente distinta como antes.

Un programador con experiencia en middlewa-

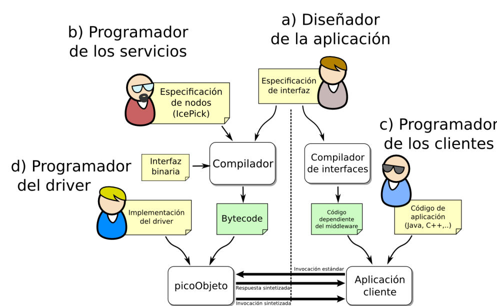

Figura 3: Propuesta de los distintos roles en el desarrollo de sistemas empotrados en red

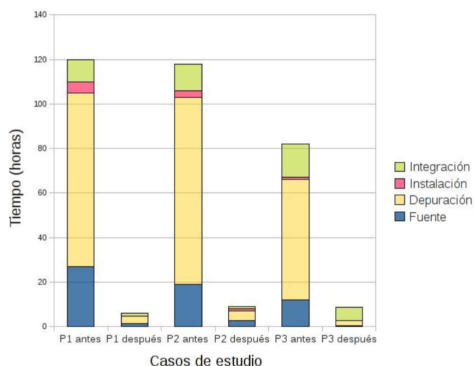

Figura 4: Comparativa de tiempo invertido en cada fase de desarrollo de diferentes sujetos de prueba.

res de comunicaciones orientados a objetos (como CORBA o Java RMI) puede dedicarse a la implementación de servicios sobre nodos empotrados sin apenas formación específica adicional. Es decir, en la figura 3, los roles de 'b' y 'c' son similares.

La figura 4 muestra una comparativa de tiempos consumidos en las diferentes fases de desarrollo de una aplicación estándar por tres sujetos de prueba, cada uno de ellos con formación previa diferente. La tarea que se les propone es la programación de un nodo que realiza una invocación a un objeto remoto. Además el nodo aloja un objeto que puede recibir invocaciones a un único método que no tiene ni argumentos ni valor de retorno.

Sin el framework, los sujetos deben aprender el formato de los mensajes IceP[8], el lenguaje FSM (que es de bajo nivel) y el manejo de las herramientas específicas para compilar a bytecode, depurar y programar el dispositivo final.

Utilizando el framework, la duración de las fases se reduce y equilibra respecto a la situación anterior. Ello supone una ventaja ya que hace posible que la planificación de proyectos sea más sencilla y con menor contención entre tareas.

Acceso al hardware

Los nodos empotrados no tendrían utilidad si no integraran algún tipo de periférico (normalmente sensor o actuador) ya que la capacidad de cómputo es, y debe ser, mínima.

El acceso a los periféricos requiere inevitablemente de software específico (controladores) ya que la arquitectura del nodo y las interfaces de las que disponga (I2C, UART, SPI, etc.) son determinantes en la implementación. Estos *drivers* proporcionan o consumen los datos relevantes para la aplicación, es decir, son los *sirvientes* aplicando la terminología de los middlewares orientados a objetos.

Tanto IcePick como FSM son lenguajes portables. A diferencia de otras iniciativas, hemos evitado el acoplamiento que conllevan las instrucciones de alto nivel relacionadas con hardware concreto. A cambio, se debe disponer de los controladores correspondientes para cada plataforma. Consideramos que el esfuerzo para desarrollar los controladores necesarios no es relevante en comparación con el esfuerzo total para desarrollar la aplicación; se pueden re-aprovechar los existentes y además puede ser desarrollado en paralelo. En este caso se requiere personal especializado (rol 'd' de la figura 3).

La interfaz binaria entre el código FSM y el controlador se especifica mediante un lenguaje muy simple denominado SIS (Servant Interface Specification) que es reconocido por el compilador de IcePick.

6. Herramientas y prototipos

Hemos desarrollado un compilador que materializa todo el proceso descrito. A partir de especificaciones Slice[9], un fichero en lenguaje IcePick y una

[8] IceP es el protocolo de red de ZeroC Ice.
[9] Slice es el lenguaje de descripción de interfaces de ZeroC Ice.

descripción de interfaces binarias (lenguaje SIS) es capaz de crear una máquina de estados completa, en bytecode FSM, para el nodo descrito. También hemos creado herramientas para la asistencia al proceso de programación, instalación y depuración de la máquina virtual de picoObjetos en PC, microcontroladores de MicroChip y motas soportadas por TinyOS. Puede encontrar información detallada sobre el uso de estas herramientas, instrucciones de instalación y ejemplo de uso en la página web del proyecto IcePick: http://arco.esi.uclm.es/icepick.

7. Conclusiones

Hemos presentado un conjunto de herramientas que, a partir de una especificación de alto nivel, permiten implementar dispositivos empotrados en red. Al basarse dicha especificación en los mismos conceptos que los middlewares de comunicaciones orientados a objetos, un programador experimentado en herramientas habituales en la programación de aplicaciones distribuidas podrá ser productivo en poco tiempo sin necesidad de formación específica.

Eso proporciona una metodología de desarrollo con importantes diferencias en todas las fases del desarrollo: especificación de los requisitos, diseño de la aplicación distribuida, implementación de los nodos, simulación, pruebas, etc.

Referencias

[1] Balani, R., Han, C. C. Rengaswamy, R. K., and Tsigkogiannis, I. and Srivastava, M., *Multi-level software reconfiguration for sensor networks*,EMSOFT'06: 6th ACM & IEEE International conference on Embedded software, 112–121, 2006.

[2] Barr, R., Bicket, J. C., Dantas, D. S., Du, B., Danny, T. W., Bing, K., Emin, Z., Sirer, G., *On the need for system-level support for ad hoc and sensor networks*, Operating System Review, vol. 36, pp. 1-5, 2002.

[3] Bosman, R., Lukkien, J., Verhoeven, R., *An integral approach to programming sensor networks*, Consumer Communications and Networking Conference, IEEE, http://www.win.tue.nl/san/wsp, 2009.

[4] Boulis, A., Han, C. C., Shea, R., Srivastava, M. B., *Sensorware: Programming sensor networks beyond code update and querying*, Pervasive Mob. Comput., vol. 3, no. 4, pp. 386-412, 2007.

[5] Boulis, A., Han, C. C., Srivastava, M. B., *Design and implementation of a framework for efficient and programmable sensor networks*, MobiSys'03. New York, NY, USA: ACM, pp. 187-200, 2003.

[6] Henning, M., Spruiell, M., *Distributed Programming with Ice* (Revision 3.4), ZeroC Inc., 2010.

[7] Levis, P., Culler, D., *Mate: A tiny virtual machine for sensor networks*, International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA, Oct. 2002.

[8] Levis, P., Gay, D., Culler, D., *Active sensor networks*, in NSDI'05: Actas de 2nd conference on Symposium on Networked Systems Design & Implementation. Berkeley, CA, USA: USENIX Association, 2005, pp. 343-356.

[9] Li, S., Lin, Y., Son, S. H., Stankovic, J. A., Wei, Y., *Event detection services using data service middleware in distributed sensor networks*, Information Processing in Sensor Networks. p. 557, 2003

[10] Moya, F., Villa, D., Villanueva, F.J., Rincón, F., Barba, J., López, J.C., *Embedding Standard Distributed Object-Oriented Middlewares in Wireless Sensor Networks*, Journal on Wireless Communications and Mobile Computing (WCMC), 1 Mar 2009.

[11] Srisathapornphat, C., Jaikaeo, C., Shen, C. C., *Sensor information networking architecture and applications*, IEEE Personal Communications, vol. 8, pp. 52-59, 2001.

[12] Villa, D., Villanueva, F. J., Moya, F., Rincón, F., Barba, J., López, J. C., *ASDF: an object oriented service discovery framework for wireless sensor networks*, International Journal of Pervasive Computing and Communications (IJPCC), 2008.