Desarrollo de Aplicaciones para Redes de Sensores Inalámbricas utilizando TinyOS y nesC

Jorge A. Atempa*, Aram Tafoya*, Arnoldo Díaz-Ramírez*, Juan F. Ibañez* y Claudia Martínez*
{jatempa, aramtd, adiaz}@itmexicali.edu.mx, {pacois20, claudia.itm}@gmail.com

***Resumen*—En los últimos años las redes inalámbricas MANET (*Mobile Ad hoc Networks*) han recibido gran atención por parte de la comunidad científica. Estas redes se caracterizan por no requerir de infraestructura fija (por ejemplo, un punto de acceso). Un caso especial de las redes MANET son las redes de sensores inalámbricas o WSN (*Wireless Sensor Networks*), que están formadas por nodos capaces de monitorizar el medio ambiente y de comunicarse vía inalámbrica. Para poder aprovechar la funcionalidad de las redes de sensores inalámbricas, sobre todo tomando en consideración los recursos limitados de los nodos, es necesario contar con una plataforma de software que optimice los recursos y que facilite el desarrollo de aplicaciones. Dos de los componentes mas importantes son el sistema operativo y el lenguaje de programación. En este artículo se presentan las principales características del sistema operativo TinyOS y del lenguaje de programación nesC, que han sido diseñados especialmente para redes de sensores inalámbricas. Se presenta también una aplicación utilizando estas herramientas.**

I. Introducción

Las redes MANET están compuestas por un conjunto de nodos móviles inalámbricos, capaces de auto-organizarse dinámicamente utilizando diversas topologías de red [5]. Una de las características mas interesantes de este tipo de redes consiste en que no requieren de una infraestructura fija, por lo que cada nodo funciona como encaminador al retransmitir paquetes hasta el nodo destino. La Fig. 1 muestra un ejemplo de una red MANET formada por diversos dispositivos móviles, que pueden comunicarse entre sí sin necesidad de un nodo coordinador o punto de acceso. Con redes de este tipo se puede compartir información en lugares en donde no existe infraestructura fija o en los que ha ocurrido algún desastre y se ha destruido la infraestructura existente, permitiendo el rápido establecimiento de una red de comunicación.

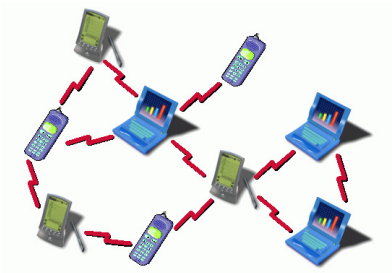

Figura 1: Ejemplo de red Ad hoc.

*Instituto Tecnológico de Mexicali
Departamento de Sistemas y Computación
Av. Tecnológico s/n, Col. Elías Calles
Mexicali, B.C. CP: 21376
Tel. (686) 580-4980

Estas redes deben adaptarse dinámicamente ante los cambios continuos en la topología de la red debido a la movilidad de los nodos, así como a la desaparición y aparición de nuevos nodos. Por esta razón se han propuesto protocolos de encaminamiento diferentes a los utilizados en las redes con infraestructura y que presuponen que la topología de la red cambia poco. A continuación se presenta la clasificación de los protocolos de encaminamiento para redes *Ad hoc* [3].

1. *Pro-activos.* Periódicamente envían información de encaminamiento para que en cualquier momento cualquier nodo pueda comunicarse con cualquier otro de la red.
2. *Reactivos o Bajo demanda.* Solo cuando es necesario se crean las rutas de comunicación.
3. *Jerárquico.* Su función en la retransmisión depende del nivel en el que se encuentre el nodo.
4. *Planos.* Todos los nodos están al mismo nivel, tienen las mismas funciones y responsabilidades.
5. *Geográficos.* Se tiene en cuenta la posición geográfica exacta de cada nodo para realizar los encaminamientos.
6. *Encaminamiento en el origen.* Cada paquete de datos lleva incorporado el camino que debe seguir en la red hasta el destino.
7. *Encaminamiento salto a salto.* Cada nodo decide cuál será el nodo siguiente.

Debido a que los nodos que forman la red inalámbrica generalmente utilizan energía de baterías, se han propuesto protocolos que consideren el consumo de energía buscando incrementar la vida útil de los nodos [9].

Un caso especial de las redes MANET son las redes de sensores inalámbricas o WSN (*Wireless Sensor Networks*), que están formadas por nodos capaces de monitorizar el medio ambiente y de comunicarse vía inalámbrica. Las WSN han mostrado un gran avance recientemente debido a que los sensores cada vez son mas pequeños, económicos e inteligentes. El diseño de una WSN depende significativamente del objetivo de la aplicación, el entorno donde se utilizarán y los costos que esto implica.

Para poder aprovechar la funcionalidad de las redes de sensores inalámbricas, sobre todo tomando en consideración los recursos limitados de los nodos, es necesario contar con una plataforma de software que optimice los recursos y que facilite el desarrollo de aplicaciones. Dos de los componentes mas importantes son el sistema operativo y el lenguaje de programación, ya que interactúan de manera directa con el hardware y proporcionan una interfaz a los programas de

los usuarios. En este artículo se presentan las principales características del sistema operativo TinyOS y del lenguaje de programación nesC, que han sido diseñados especialmente para redes de sensores inalámbricas. Además, se presenta un ejemplo de una aplicación que utiliza estas herramientas.

El resto del documento esta organizado de la siguiente manera. La sección II presenta las características más importantes de las redes de sensores inalámbricas. En la sección III se estudia brevemente el sistema operativo TinyOS, mientras que en la sección IV se explican aspectos relevantes del lenguaje nesC. En la sección V se explica brevemente el proceso de instalación de TinyOS. La sección VI muestra un ejemplo sencillo de una aplicación para redes de sensores que utiliza TinyOS y nesC. Finalmente, en la sección VIII se presentan las conclusiones y trabajo futuro.

II. Redes de Sensores Inalámbricas

Una red de sensores inalámbrica es un sistema distribuido formado por nodos, también llamados *motes*, que interactúan con el entorno físico. Un *mote* es un pequeño dispositivo compuesto de un microprocesador con memoria, uno o varios sensores, una radio de baja potencia, una fuente de energía y un actuador. Cada nodo puede contar con diversos tipos de sensores, tales como térmicos, biológicos, químicos, ópticos o mecánicos, por mencionar algunos, y que miden las propiedades del ambiente. La información recabada por los sensores es generalmente procesada por los nodos. Sin embargo, debido a las limitaciones en memoria y capacidad de procesamiento, los nodos cuentan con un radio para transmitir de manera inalámbrica únicamente la información requerida y parcialmente procesada. Esta información es enviada hacia una estación base, como por ejemplo, una computadora portátil o una tableta electrónica. La estación base recibe la información de los nodos, la procesa si es necesario y lleva a cabo las acciones necesarias, como por ejemplo, informar a los usuarios si algún evento de interés ha ocurrido. Generalmente la fuente primaria de energía de los nodos son baterías, y en algunos casos del medio ambiente se obtiene una segunda fuente de energía (por ejemplo, utilizando paneles solares). Por otra parte, los actuadores pueden ser incorporados a los nodos dependiendo del tipo de sensores y de la aplicación.

Existen una gran cantidad de escenarios en los que pueden aplicarse las redes de sensores, entre los que se encuentran aplicaciones militares de rastreo y vigilancia, pronóstico de desastres naturales, monitorización de condiciones médicas en pacientes, exploración de ambientes peligrosos, detección de sismos, entre otros. Yick *et al.* en [16] clasifican a las aplicaciones para redes de sensores en dos categorías: *monitorización* y *rastreo* (*tracking*).

La Fig. 2 muestra un ejemplo de una aplicación WSN de monitorización. En el ejemplo, la aplicación monitoriza el entorno físico de un viñedo, midiendo condiciones de humedad, temperatura, aparición de plagas, entre otras.

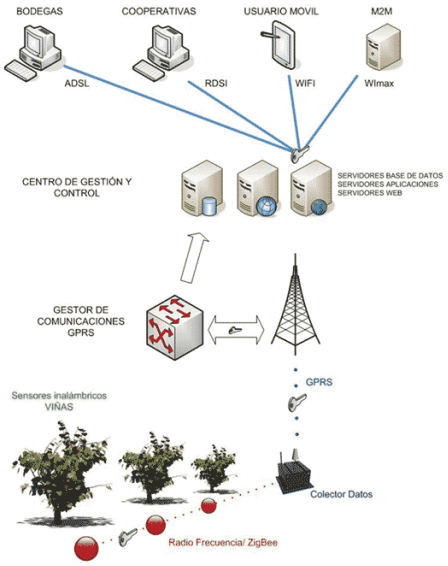

Figura 2: Ejemplo de WSN.

Para desarrollar aplicaciones que optimicen el uso de los recursos de las WSN es necesario contar con una plataforma de software. Entre los componentes de la plataforma de software se encuentra el sistema operativo y el lenguaje de programación (compiladores, entornos de desarrollo, bibliotecas, etc). Un sistema operativo es un componente de software que administra los recursos de hardware de un sistema informático y proporciona una interfaz a los programas de aplicación de los usuarios [14]. El diseño de un sistema operativo para redes de sensores es diferente al de un sistema operativo tradicional debido a las características específicas de las WSN, tales como recursos limitados, gran dinamismo y a la dificultad de acceso a los entornos en los que se distribuyen los nodos. A la fecha se han propuesto algunos sistemas operativos para redes de sensores [12]. A continuación se presentan algunos de ellos:

- *Bertha.* Una plataforma de software diseñada e implementada para modelar, evaluar e implementar una red de sensores distribuida de muchos nodos idénticos.
- *Nut/OS.* Es un pequeño sistema operativo para aplicaciones en tiempo real, que trabaja con CPUs de 8 bits.
- *Contiki.* Es un Sistema Operativo de libre distribución para usar en un limitado tipo de computadoras, desde los 8 bits a sistemas embebidos en micro-controladores, incluidos *motes* de redes inalámbricas.
- *TinyOS.* Primer sistema operativo desarrollado específicamente para redes de sensores.
- *MANTIS.* Sistema Operativo escrito en C de código abierto basado en POSIX.
- *LiteOS.* Sistema operativo desarrollado en principio para calculadoras, pero que ha sido también utilizado para redes de sensores.

En [6], Farooq *et al.* presentan el estado del arte de los sistemas operativos para redes de sensores, con la finalidad de apoyar a diseñadores y programadores de aplicaciones para WSN en la elección del sistema operativo más adecuado al hardware y tipo de aplicación específica. Es importante considerar cuatro requerimientos de las aplicaciones para redes de sensores al seleccionar el sistema operativo y lenguaje de programación. Estos requerimientos son [15]:

- Uso eficiente de la energía
- Escalabilidad
- Resistencia a fallos
- Colaboración

El uso eficiente de la energía de los nodos es un requerimiento de todas las aplicaciones para redes de sensores. Debido a que la comunicación inalámbrica es la actividad que más consume energía, es importante reducir en la medida de lo posible el envío innecesario de información. Por esta razón, el modelo de programación utilizado debe ayudar a los programadores en el fácil desarrollo de aplicaciones con bajo consumo de energía.

La escalabilidad es un requerimiento importante debido a que muchas aplicaciones utilizan cientos de nodos. Uno de los factores que más influye en la escalabilidad es el limitado ancho de banda del estándar IEEE 802.15.4 y que es utilizado por muchos sistemas operativos para WSN. Debido a esta limitante, la principal preocupación relacionada con la escalabilidad en redes de sensores consiste en reducir la comunicación para utilizar eficientemente el limitado ancho de banda disponible. El modelo de programación utilizado debe permitir el desarrollo de aplicaciones escalables y mecanismos en tiempo de ejecución para lograr un uso eficiente del ancho de banda.

La resistencia a los fallos es también importante en aplicaciones que se ejecutan por un largo periodo de tiempo. Estas aplicaciones deben mantenerse funcionando a pesar de la existencia de un canal de comunicación poco confiable, fallos en los nodos o fallos inesperados. El desarrollo de aplicaciones resistentes a los fallos y que sean capaces de adaptarse a nuevas condiciones de operación requieren del soporte del modelo de programación.

Finalmente, la colaboración es también un requerimiento importante, debido a que es posible utilizar de manera eficiente los recursos existentes (por ejemplo, el ancho de banda) si los nodos colaboran entre si al procesar y enviar la información. Sin embargo, el desarrollo de algoritmos colaborativos no es sencillo y se benefician en gran medida si pueden apoyarse en un buen modelo de programación.

Para cumplir con los requerimientos de las aplicaciones para redes de sensores es importante seleccionar, además del sistema operativo, un modelo de programación adecuado. A la fecha se han propuesto diversos modelos de programación para WSNs. Sugihara y Gupta en [15] clasifican a los modelos de programación en dos categorías: modelos de programación de *bajo nivel* y modelos de programación de *alto nivel*. Los primeros están centrados en la plataforma, se enfocan en la abstracción del hardware y permiten un flexible control de los nodos. Un ejemplo de este modelo es el lenguaje nesC [7] utilizado en conjunción con el sistema operativo TinyOS [8], que se han constituido en el estándar *de facto* en el desarrollo de aplicaciones para WSNs.

Los modelos de programación de alto nivel están centrados en la aplicación y tienen como objetivo facilitar el desarrollo de las aplicaciones, en contraste con el modelo anterior que busca optimizar el desempeño del sistema. Ejemplos de macrolenguajes de esta categoría son TinyDB [11] y Regiment [13].

III. TinyOS

TinyOS [8] es un sistema operativo diseñado específicamente para sensores inalámbricos con bajo consumo de energía. Se distribuye como código abierto (*open source*) con licencia Berkeley Software Distribution (BSD) [1]. A diferencia de otros sistemas operativos, el diseño de TinyOS se enfoca en operaciones de *ultra* bajo consumo de energía. Su arquitectura está basada en componentes, permite una rápida implementación de aplicaciones al mismo tiempo que minimiza el tamaño del código requerido, algo que es de suma importancia debido a las severas limitaciones de memoria que existen en en los nodos. La biblioteca de componentes de TinyOS incluye protocolos de red, servicios de distribución, manejadores de sensores y herramientas para la adquisición de datos, entre otros. TinyOS fue originalmente desarrollado como un proyecto de investigación de la Universidad de California en Berkeley, y a partir de entonces se ha conformado una gran comunidad internacional de desarrolladores y usuarios.

Para facilitar el desarrollo de aplicaciones, TinyOS proporciona un importante conjunto de servicios, componentes y abstracciones, tales como de comunicación y almacenamiento, así como para medir (*sense*) condiciones del ambiente. Además, se ejecuta en una gran cantidad de plataformas genéricas, siendo sencillo adaptarlo para que soporte nuevas plataformas.

TinyOS proporciona tres elementos de alto nivel con la finalidad de simplificar el desarrollo de sistemas y aplicaciones [10]:

- un modelo de componentes, que define cómo escribir pequeñas y reutilizables piezas de código, así como la manera de integrarlas para construir abstracciones de mayor tamaño;
- un modelo de ejecución concurrente, que define la manera en que los componentes pueden entrelazar sus ejecuciones, y la manera en que el código interactúa con interrupciones y sin interrupciones;
- interfaces de programación de aplicaciones (APIs), servicios, componentes de bibliotecas, así como una estructura general de componentes que simplifica la escritura de nuevas aplicaciones y servicios.

Las aplicaciones y sistemas en TinyOS, así como el sistema operativo, están escritos en lenguaje nesC.

IV. nesC

nesC [7] es una extensión del lenguaje de programación C y ha sido diseñado para ajustarse a los conceptos estructurales y modelo de ejecución de TinyOS. Tiene como objetivos el desarrollo de aplicaciones que reduzcan el tamaño del código y de la utilización de memoria RAM, permitir la optimización en el uso de los recursos disponibles, así como ayudar en la prevención de errores de bajo nivel, tales como las condiciones de carrera.

Los conceptos básicos de nesC se explican a continuación [4]:

- Separación de construcción y composición: los programas son construidos a partir de componentes, los cuales

son enlazados ("*wired*") para formar programas. Los componentes son definidos en dos ámbitos, uno para la especificación (contiene los nombres de las instancias de las interfaces) y una para la implementación. Los componentes tienen concurrencia interna en forma de tareas (*tasks*). Los hilos de control pueden pasar en un componente a través de las interfaces. Estos hilos tienen sus raíces en una tarea o en una interrupción por hardware.

- Especificación del comportamiento de los componentes en términos del conjunto de interfaces. Las interfaces pueden ser proporcionadas o utilizadas por los componentes. Las interfaces buscan representar la funcionalidad que proporciona el componente al usuario. Las interfaces utilizadas representan la funcionalidad que el componente requiere para desempeñar su trabajo.
- Las interfaces son bidireccionales: especifican un conjunto de funciones que serán implementadas por el proveedor de la interfaz (comandos) y un conjunto de funciones que serán implementadas por el usuario de la interfaz (eventos). Esto permite que una interfaz represente una compleja interacción entre componentes (por ejemplo, el registro del interés en algún evento, seguido de una notificación cuando el evento ocurre). Esto es crítico debido a que todos los comandos de larga duración en TinyOS, como por ejemplo el envío de un paquete, son operaciones *sin bloqueo*; su terminación es notificada a través de un evento (envío realizado). Al especificar interfaces, un componente no puede invocar el comando `send` al menos que proporcione una implementación del evento `sendDone`.
- Los componentes están ligados estáticamente entre sí a través de sus interfaces. Esto incrementa la eficiencia en tiempo de ejecución, promueve un diseño robusto y permite un mejor análisis estático de los programas.
- nesC ha sido diseñado bajo la premisa de que el código será generado por compiladores de programas completos. Esto también debe permitir una mejor generación de código y análisis del mismo.

V. Instalación de TinyOS

En esta sección se explica el proceso de instalación de TinyOS en Ubuntu Lucid [2]. Para aprovechar el sistema de instalación de paquetes de Ubuntu, es recomendable agregar algún repositorio que contenga TinyOS. Por ejemplo, en el archivo de repositorios `/etc/apt/sources.list` se agrega la siguiente línea:

```
deb http://tinyos.stanford.edu/tinyos/dists/ubuntu
    lucid main
```

Una vez hecho esto es necesario actualizar el listado de paquetes disponibles en los repositorios. Esto puede hacerse ejecutando desde una terminal lo siguiente:

```
$ sudo apt-get update
```

Para conocer cuál es la más reciente versión de TinyOS existente en el repositorio, puede escribirse:

```
$ apt-cache search tinyos
```

Suponiendo que el paquete con la versión más reciente sea `tinyos-2.1.1`, para instalarlo del repositorio se ejecuta en una terminal el siguiente comando:

```
sudo apt-get install tinyos-2.1.1
```

Una vez instalado TinyOS puede ser necesario hacer algunos ajustes en la configuración del sistema. Por ejemplo, es recomendable definir en las variables de entorno del sistema, el directorio en el que se encuentran los archivos importantes de TinyOS, tales como componentes, interfaces, bibliotecas, entre otros. Una manera de hacerlo consiste en agregar la siguiente línea en los archivos `/.bashrc` o `/.profile` que se encuentran en el directorio personal (*home directory*):

```
source /opt/tinyos-2.1.1/tinyos.sh
```

Para verificar si el entorno esta configurado correctamente, debe ejecutarse:

```
$ tos-check-env
```

En algunos casos, el comando anterior puede generar un aviso de advertencia (*warning*) al no encontrar el archivo `tinyos.jar` en la ruta de clases (*classpath*). Para corregirlo, es necesario cambiar el valor de la variable CLASSPATH que se encuentra en el archivo `/opt/tinyos-2.x/tinyos.sh`. En nuestro caso, la ruta del archivo es *`/opt/tinyos-2.1.1/support/sdk/java/tinyos.jar`*, por lo que la variable de entorno fue modificada de la siguiente manera:

```
export CLASSPATH=/opt/tinyos-2.1.1/support/sdk/java/
    tinyos.jar:.
```

En caso de que se utilice Ubuntu para 64 bits es probable que la versión de Python incluida en la distribución sea diferente a la esperada. Por tal motivo, es necesario modificar el archivo `/opt/tinyos-2.1.1/support/make/sim.extra` en la línea `PYTHON_VERSION` para indicar la versión correcta. En nuestro caso, esta línea se modificó de la siguiente manera:

```
PYTHON_VERSION=2.6
```

Estas modificaciones son necesarias para poder ejecutar sin problemas el simulador de TinyOS TOSSIM.

Por otra parte, puede ocurrir que la versión utilizada de TinyOS no incluya el soporte para la base programadora utilizada, o que la información de ésta sea incorrecta en el sistema. Esto puede corregirse en ocasiones de la siguiente manera. Suponiendo que la base programadora a utilizar sea *mib520*, será necesario editar el archivo `/usr/bin/motelist` y corregir la sección en la que describen los parámetros de la base programadora. Por ejemplo, en nuestro caso, en el que se utilizó la base programadora *mib520*, fue necesario modificar la siguiente sección de texto:

```
#Scan /sys/bus/usb/drivers/usb for FTDI devices
  my @ftdidevs = grep { ($_->{UsbVendor}||"") eq "
      0403" && ($_->{UsbProduct}||"") eq "6001" }
```

por:

Figura 3: Ejemplo de una aplicación para WSN

```
#Scan /sys/bus/usb/drivers/usb for FTDI devices
  my @ftdidevs = grep { ($_->{UsbVendor}||"") eq "
      0403" && ($_->{UsbProduct}||"") eq "6010" }
```

Para verificar que nuestro dispositivo es reconocido por TinyOS, es necesario escribir sobre una terminal:

```
$ motelist
```

Si después de estos cambios la base programadora es soportada por la versión del sistema operativo, el comando anterior mostrará el nombre y modelo de nuestro dispositivo.

VI. Ejemplo

En esta sección se presenta un ejemplo con la finalidad de mostrar la manera en que pueden crearse aplicaciones para WSNs utilizando TinyOS y nesC. En la aplicación se utilizó TinyOS versión 2.1.1 y el compilador de nesC versión 1.2.4. El *mote* esta compuesto por una plataforma de hardware *IRIS* con una placa sensora modelo *MTS420/400CC*. La estación base esta compuesta por una plataforma de hardware *IRIS* con un procesador *ATmega1281*, en una base programadora para la adquisición de datos modelo mib520. La estación base está conectada a una computadora portátil *Compaq Presario* con procesador *AMD Athlon II P320* con dos núcleos, y con sistema operativo *Ubuntu 10.04 Lucid Lynx* de 32 bits. Se utilizó el kit de desarrollo para redes de sensores *Memsic* modelo *WSN-START2110CA*.

En el ejemplo se explica la manera en que un nodo puede medir el valor de la temperatura del sensor correspondiente, y enviar un mensaje a la estación base que contenga el valor leído, como se observa en la Fig. 3. El valor de la temperatura es leído y enviado a la estación base de manera periódica. El *mote* utilizado cuenta con dos sensores de temperatura. En la aplicación ejemplo se utilizó el sensor denominado *Sensirion SHT11*.

TinyOS proporciona un número de interfaces para la abstracción de servicios de comunicación y componentes que proveen estas interfaces. Todas las interfaces y componentes usan en común un *buffer* de mensaje, llamado `message_t`, que implementa una estructura nesC (similar a las estructuras en C). `message_t` es un tipo de dato abstracto; los miembros pueden leer y escribir en él usando funciones de acceso a recursos como semáforos (*mutex*).

La definición de la estructura `message_t` para la versión utilizada de TinyOS se muestra a continuación.

```
typedef nx_struct message_t {
  nx_uint8_t header[sizeof(message_header_t)];
  nx_uint8_t data[TOSH_DATA_LENGTH];
  nx_uint8_t footer[sizeof(message_footer_t)];
  nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```

Esta estructura esta definida en el archivo `tos/types/message.h`

VI-A. *Interfaces básicas de comunicaciones*

Para crear una aplicación en TinyOS, deben primero definirse las interfaces a utilizar. TinyOS incluye un buen número de interfaces y componentes que utilizan `message_t` como estructura de datos. El programador puede elegir las que requerirá en su aplicación, o crear nuevas. Las interfaces se encuentran en la carpeta `tos/interfaces`. Algunas de ellas son:

- `Packet` Provee el acceso básico a la estructura `message_t`.
- `Send` Interfaz básica de envío de mensajes. Proporciona comandos para el envío y la cancelación de mensajes pendientes. Incluye un evento para indicar si un mensaje fue enviado exitosamente, o si no pudo enviarse.
- `Receive` Es la interfaz básica de recepción de mensajes. Proporciona un evento para la recepción de mensajes, además de comandos para conocer la longitud del mensaje, entre otros.
- `PacketTimeStamping` Proporciona información de marcas de tiempo en la transmisión y recepción.

VI-B. *Interfaces de mensaje activo (Active Message)*

Es muy común tener múltiples servicios que utilizan el mismo radio de comunicación. Para evitar condiciones de carrera u otro tipo de problemas relacionados con el acceso a los recursos, como por ejemplo el radio, es necesario utilizar mecanismos de sincronización. Por tal razón, TinyOS provee la capa de Mensaje Activo o *Active Message* (AM) para coordinar los múltiples accesos al radio. El término *AM type* se refiere al uso de multiplexación. Algunos servicios que soporta AM son:

- `AMPacket` Es similar a la interfaz `Packet`; provee el acceso básico a `message_t`.
- `AMSend` Similar a la interfaz `Send`; interfaz básica de envío para AM.

En el ejemplo se utiliza `AMSend`.

VI-C. *Componentes*

Los componentes implementan las interfaces básicas de comunicación y de mensajes activos (AM). Al igual que las interfaces, el programador debe elegir los componentes que utilizará en su aplicación. Los componentes de TinyOS se encuentran en el directorio `/tos/system`. Algunos de ellos se muestran en la Tabla 1. Puede observarse que el nombre del componente siempre contiene la letra C al final.

En el ejemplo se utilizo el componente `AMSenderC`.

Componente	Interfaz que provee
AMReceiverC	Receive, Packet, AMPacket
AMSenderC	AMSend, Packet, AMPacket PacketAcknowledgements, Acks
AMSnooperC	Receive, Packet, AMPacket
AMSnoopingReceiverC	Receive, Packet, AMpacket

Tabla 1
Componentes Active Message(AM)

VI-D. Definiendo la estructura del mensaje

Es común que los nodos de una WSN envíen un mensaje a la estación base al detectar algún evento importante. Por esta razón es necesario definir la estructura del o los mensajes a enviar. En el ejemplo se utilizó la siguiente estructura:

```
enum{
  AM_DATATORADIOMSG=1,
  TIMER_PERIOD_MILLI = 256 };
typedef nx_struct datatoradiomsg{
  nx_uint16_t NodeID;
  nx_uint16_t Temp_data;
}datatoradiomsg_t;
```

Puede observarse que se definen unas constantes de tipo enumerado (*enum*), llamadas `AM_DATARADIOMSG` y `TIMER_PERIOD_MILLI`. Posteriormente se define la estructura llamada `datatoradiomsg` que contiene dos tipos de datos enteros sin signo de 16 bits; estos datos son `NodeID` y `Temp_data`, que contendrán el número de identificación del nodo emisor y el valor de la temperatura, respectivamente. La definición de la estructura de los mensajes se almacena habitualmente en un archivo de cabecera con extensión *h*.

VI-E. Definición de módulo para envío

A continuación deben definirse los módulos que serán utilizados por la aplicación. En el ejemplo se utiliza un módulo al que se nombró `DataToRadioC`. Este módulo utiliza las interfaces para el envío de AM.

```
module DataToRadioC {
  uses interface Boot;
  uses interface Leds;
  uses interface SplitControl;
  uses interface Read<uint16_t> as Temperature;
  uses interface Timer<TMilli> as TimerSense;
  uses interface AMSend;
}
```

En el módulo también deben definirse las variables a utilizar. En el ejemplo, se definieron en la sección de implementación (*implementation*), como se muestra a continuación:

```
implementation {
  uint16_t Temp_data;
  message_t message;
  bool busy = FALSE;
  datatoradiomsg_t* packet;
  ...
}
```

Como se observa, en el ejemplo se declara una variable de tipo entero sin signo de 16 bits llamada `Temp_data`, que contendrá el valor medido de la temperatura. También se define una variable de tipo *booleano* llamada `busy`, que es inicializada con el valor de *falso*, para indicar que es posible enviar un mensaje. Debido a que es necesario enviar mensajes, se definió una variable llamada `message` de tipo `message_t`. Por otra parte, cuando se invoca al evento `Boot.booted` es importante ejecutar el evento `SplitControl`, que se encarga de optimizar el uso de la energía en el envío de mensajes.

```
event void Boot.booted() {
  call SplitControl.start();
  .....
}
```

Además, es también necesario implementar los manejadores de eventos `SplitControl.startDone` y `SplitControl.stopDone` con las siguientes instrucciones

```
event void SplitControl.startDone(error_t err) {
  if(error==SUCCESS){
    call TimerSense.startPeriodic(TIMER_PERIOD_MILLI
        );
  }else{
    call SplitControl.start();
  }
}
event void SplitControl.stopDone(error_t err) {   }
```

Finalmente, es importante mencionar que el evento `Temperature.readDone` recibe del sensor, en el parámetro denominado `val`, el valor medido de la temperatura. El evento almacena el valor recibido en la variable `Temp_data`

```
event void Temperature.readDone(error_t result,
    uint16_t val){
  Temp_data = val;
}
```

Como se explicó anteriormente, el evento de lectura de la temperatura, así como el envío de su valor hacia la estación base, se lleva a cabo de manera periódica. Para implementar este comportamiento periódico es necesario utilizar un evento temporizador. En el ejemplo, el evento temporizador se nombró `TimerSense.fired`.

```
event void TimerSense.fired() {
  call Temperature.read();
  if(!busy){
    packet=(datatoradiomsg_t*)(call AMSend.
        getPayload(&message,sizeof(datatoradiomsg_t)
        ));
  }
  packet->NodeID = TOS_NODE_ID;
  packet->Temp_data =Temp_data;
  busy = TRUE;
  call AMSend.send(AM_BROADCAST_ADDR, &message,
      sizeof(datatoradiomsg_t));
}
```

El evento `TimerSense.fired` inicia invocando al evento `Temperature.readDone` a través de `Temperature.read()`. Posteriormente verifica que el nodo no esté enviando algún otro mensaje para crear el paquete con la estructura `datatoradiomsg_t`, que como se explicó contiene el valor leído de la temperatura. Posteriormente se envía el paquete invocando al evento `AMSend.send`.

Una vez que el envío se ha llevado a cabo, se utiliza el manejador `sendDone`, que verifica si el mensaje enviado es

el deseado. Si lo es, se asigna a la variable `busy` el valor de *false*, para permitir el envío de otro mensaje.

```
event void AMSend.sendDone(message_t* msg, error_t
    error) {
  if (&message == msg) {
    busy = FALSE;
  }
}
```

VI-F. Definición de la configuración para envío

Una vez que se ha definido el módulo para el envío, a continuación debe definirse la configuración con los componentes que utilizarán en la implementación de la aplicación:

```
implementation {
  components MainC;
  components new SensirionSht11C();
  components ActiveMessageC;
  components new AMSenderC(AM_DATATORADIOMSG);
  components DataToRadioC as App;
  components new TimerMilliC() as TimerSense;
  .....
}
```

Después de definir los componentes que se usarán, es necesario enlazar las interfaces que utilizará la aplicación con los componentes que proveen las interfaces:

```
implementation {
  ...
  App.Boot -> MainC;
  App.AMSend -> AMSenderC;
  App.SplitControl -> ActiveMessageC;
  App.TimerSense -> TimerSense;
  App.Temperature -> SensirionSht11C.Temperature;
}
```

Una vez desarrollado el código de la aplicación en los archivos anteriormente descritos, es necesario compilarlos. Una manera conveniente de hacerlo es utilizando el comando `make` y definiendo en un archivo *Makefile* los parámetros de compilación y enlazado. El contenido de este archivo se muestra a continuación:

```
COMPONENT=DataToRadioApp
SENSORBOARD=mts400
include $(MAKERULES)
```

Como puede observarse, el proceso de compilación se simplifica ya que TinyOS define en el archivo *support/make/-Makerules* las reglas necesarias para que el comando `make` puede ejecutarse sin problemas.

Debido a que la plataforma utilizada es iris, la compilación y enlazado se lleva a cabo con el siguiente comando:

```
$ make iris
```

Finalmente, para instalar la aplicación en el dispositivo se escribe en una terminal lo siguiente:

```
$ make iris install,id mib520,/dev/ttyUSB0
```

en donde `id` es el número identificador del dispositivo.

Una vez compilados los archivos descritos en esta sección, y después de instalar en los nodos la aplicación desarrollada, se enviarán periódicamente a la estación base las mediciones de la temperatura recabas en los nodos de la WSN.

AMPacket	Dirección Destino	Dirección Fuente	Tamaño Mensaje	ID del Grupo	ID del Manejador	NodeID	Temp_data
00	FF FF	1C 25	04	00	01	1C 25	19 3A

Figura 4: Formato del Paquete de Datos

VII. Recepción en Estación Base

Para la recepción de mensajes, se utiliza una aplicación llamada `BaseStation` incluida en las herramientas de TinyOS. El paquete de mensajes tiene el formato que se muestra la Fig 4. Para visualizar el contenido de los paquetes, en la estación base se utiliza una herramienta desarrollada en el lenguaje Java. Esta herramienta se llama `Listen`, y para invocarla debe escribirse en una terminal lo siguiente:

```
java net.tinyos.tools.Listen
```

Con esto, la estación base mostrará los mensajes enviados por los nodos.

VIII. Conclusiones y Trabajo Futuro

Las redes inalámbricas ad hoc ó MANETs tienen la capacidad de auto-configurarse utilizando diversas topologías de red, y de comunicarse sin requerir de una infraestructura fija. Las redes de sensores inalámbricas o WSN son un ejemplo de una red MANET. En las WSN, los nodos están equipados con uno o mas sensores para medir las condiciones del ambiente. Debido a las limitaciones de recursos con las que cuentan las redes de sensores, es importante utilizar una buena plataforma de software para el desarrollo de aplicaciones. El sistema operativo es quizás el componente que mas influye en el uso adecuado de los recursos disponibles.

TinyOS es un sistema operativo diseñado para redes de sensores inalámbricas embebidas. La finalidad de TinyOS es minimizar el consumo de energía y espacio en memoria (*storage*). Por otra parte, el lenguaje de programación nesC ha sido diseñado para aprovechar eficientemente la funcionalidad de TinyOS. Con nesC y TinyOS es posible agilizar el desarrollo de aplicaciones para WSN.

En este artículo se presentó una aplicación para redes de sensores que mide los valores de la temperatura del medio ambiente. Los valores obtenidos son enviados por los nodos a la estación base. La aplicación se desarrolló utilizando TinyOS y nesC, y mostró que con estas herramientas es posible desarrollar fácilmente aplicaciones eficientes para WSNs.

Como trabajo futuro, se planea desarrollar una aplicación para detección de incendios e intrusos.

Referencias

[1] Tinyos home page. http://www.tinyos.net/.

[2] Ubuntu home page. http://www.ubuntu.com/.

[3] Mehran Abolhasan, Tadeusz Wysocki, and Eryk Dutkiewicz. A review of routing protocols for mobile ad hoc networks. *Ad Hoc Networks 2*, 2(4):1–22, January 2004.

[4] Eric Brewer, David Culler, David Gay, Phil Levis, Rob von Behren, and Matt Welsh. nesc: A programming language for deeply networked systems, December 2004. (Online http://nescc.sourceforge.net/ Visitadoel 13 de Abril de 2011).

[5] Imrich Chlamtac, Marco Conti, and Jennifer J. N. Liu. Mobile ad hoc networking: imperatives and challenges. *Ad Hoc Networks*, 1(1):13–64, July 2003.

[6] Muhammad Omer Farooq, Sadia Aziz, and Abdul Basit Dogar. State of the art in wireless sensor networks operating systems: A survey. *Lecture Notes in Computer Science*, 6485:616–631, 2010.

[7] David Gay, Matt Welsh, Philip Levis, Eric Brewer, Robert Von Behren, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 1–11, 2003.

[8] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the 9th international conference on Architectural support for programming languages and operating systems*, pages 93–104, 2000.

[9] Christine E. Jones, Krishna M. Sivalingam, Prathima Agrawal, and Jyh Cheng Chen. A survey of energy efficient network protocols for wireless networks. *Wireless Networks*, 7(4):343–358, Sep 2001.

[10] Philip Levis and David Gay. *TinyOS Programming*. Cambridge University Press, 1st edition, April 2009.

[11] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 491–502, San Diego, CA, 2003.

[12] Adi Mallikarjuna, V. Reddy, A.V.U. Phani Kumar, D. Janakiram, and G. Ashok Kumar. Wireless sensor network operating systems: a survey. *International Journal of Sensor Networks*, 5(4):236–255, 2009.

[13] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 489–498, Cambridge, Massachusetts, 2007.

[14] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, 8th edition, July 2008.

[15] Ryo Sugihara and Rajesh K. Gupta. Programming models for sensor networks: A survey. *ACM Transactions on Sensor Networks*, 4(2):1550–4859, April 2008.

[16] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. Wireless sensor network survey. *Computer Networks*, 52(12):2292–2330, August 2008.