Capítulo 3

TinyOS y nesC

El objetivo principal de este capítulo es sentar las bases para la programación en el lenguaje nesC. Se trata de una extensión del lenguaje C, con componentes que se enlazan para crear las aplicaciones. El sistema operativo que ejecuta estos programas es TinyOS (en su versión actual 2.1.1).

A la hora de instalar y arrancar el entorno de desarrollo de TinyOS, pueden seguirse las instrucciones de la web oficial, concretamente en:

```
http://docs.tinyos.net/index.php/Getting_started
```

Si se instala en un entorno Windows, será necesario instalar el emulador Cygwin, que proporciona un comportamiento similar a un sistema Unix. Se trata de una interfaz basada en línea de comandos.

Este capítulo podría usarse como guía o tutorial para principiantes en el mundo de la redes de sensores inalámbricos. La bibliografía al respecto en idioma español es escasa, lo que hace que cualquier documento de este tipo sea de gran utilidad.

Se comienza con los conceptos y bloques básicos que componen el lenguaje de programación (componentes e interfaces). A continuación se dan algunas reglas

para la nomenclatura de los ficheros nesC, así como de un ejemplo sencillo de aplicación. También se trata la gestión y funcionamiento de las tareas (añadiendo prioridades). La última parte se centra en la comunicación entre nodos (radiofrecuencia) y con un sistema de gestión de datos (puerto serie). Para profundizar en los conocimientos de programación en TinyOS, se recomienda la lectura de (Levis, 2006) y (Levis, y otros, 2009).

Asimismo, como elementos de ayuda a la hora de programar, es recomendable la inscripción en listas de correo como (Berkeley. University of California) o foros relacionados.

3.1 Elementos básicos

3.1.1 Componente

Un componente es un bloque de código que aporta una funcionalidad determinada. Existen componentes intrínsecos al sistema TinyOS denominados primitivos, pero el programador puede crear a partir de éstos nuevos componentes, que realicen tareas más elaboradas (llamados complejos). La forma de acceder a las funciones de un componente es a través de las interfaces.

Una aplicación o programa de nesC no es más que un conjunto de componentes. La funcionalidad que tiene cada componente se especifica en un fichero de código fuente, el cual toma por comodidad el mismo nombre que su componente. Además, a la hora de nombrarlos, hay que tener en cuenta que no pueden repetirse los nombres, pues el compilador de nesC accede a los componentes necesarios a través de su nombre de archivo.

Se distinguen dos tipos de componentes según su comportamiento: **módulos** *(modules)* y **configuraciones** *(configurations)*.

Todos los componentes (tanto módulos como configuraciones) se dividen en dos bloques muy diferentes: la **signatura** *(signature)* y la **implementa-**

ción *(implementation)*. Precisamente, será en la implementación donde se distinga a módulos de configuraciones.

3.1.2 Interfaz

A través de ella se acceden a determinadas operaciones que puede realizar un componente. También es posible recibir determinados eventos para actuar en consecuencia.

3.1.3 Cableado

Una aplicación nesC se basa en la unión de varios componentes. Se denomina cableado (en inglés *wiring*) a la operación de relacionar las interfaces de los distintos componentes para formar una aplicación de nivel superior.

3.2 Componente. Signatura

El bloque signatura (siempre va al principio del código) indica qué interfaces proporciona y/o usa el componente en cuestión. Esto se especifica mediante las palabras reservadas `provides` y `uses`. Por ejemplo

```
module SenseC {
  uses interface Boot;
  uses interface Leds;
  uses interface Timer<TMilli>;
  uses interface Read<uint16_t>;
}
```

Indica que se trata de un componente de tipo módulo (`module`) llamado `SenseC`. Todo lo que hay detrás de `uses` son interfaces que usa, en el ejemplo: `Boot`, `Leds`, `Timer` y `Read`. En este otro caso

```
configuration LedsC {
  provides interface Leds;
}
```

El componente es ahora de tipo configuración (`configuration`) y se llama `LedsC`. Respecto a las interfaces, sólo proporciona una (`Leds`).

Nada impide que un componente pueda usar y proporcionar interfaces simultáneamente:

```
configuration MainC {
  provides interface Boot;
  uses interface Init as SoftwareInit;
}
```

Se trata de un componente de configuración de nombre `MainC`. Proporciona la interfaz `Boot` y por otro lado usa la interfaz `Init`.

Por último, puede ocurrir que un componente ni use ni provea interfaces. En este caso la signatura estará vacía:

```
configuration BlinkAppC {
}
```

Cuando una signatura trabaje con un número elevado de interfaces, pueden agruparse en interfaces usadas y/o proporcionadas. Esto sólo se hace por legibilidad en el código (no aporta mejoras en el rendimiento). El ejemplo anterior `SenceC` quedaría:

```
module SenseC {
  uses {
    interface Boot;
    interface Leds;
    interface Timer<TMilli>;
    interface Read<uint16_t>;
  }
}
```

3.3 Interfaces

Al igual que ocurría con los componentes, las interfaces también tienen una correspondencia unívoca con el nombre de fichero en el que están definidas, de forma que no puede haber dos interfaces diferentes con el mismo nombre.

Sin embargo, a la hora de la programación, cambian su estructura de código:

```
interface Leds {
  ...
}
```

En el interior se declaran las funciones que la componen. En nesC existen dos tipos de funciones para interfaces: los **comandos** *(commands)* y los **eventos** *(events)*.

Cuando se enlazan dos componentes a través de una interfaz, uno será el que proporcione la interfaz, mientras que el otro será el que la use. El que usa dicha interfaz, puede ejecutar funciones mediante llamadas o comandos. La forma que tiene el otro extremo de responder es a través de los eventos.

Algunos ejemplos de definición de interfaces:

```
interface Leds {
  async command void led0On();
  async command void led0Off();
  async command void led0Toggle();
  ...
}
```

El código anterior define la interfaz denominada `Leds`. Contiene tres comandos o llamadas. El archivo deberá implementar el funcionamiento de dichos comandos.

Este otro caso:

```
interface Boot {
  event void booted();
}
```

La interfaz `Boot` sólo contiene un evento. En este caso, el programador debe implementar algún comportamiento para cuando se reciba el mencionado evento.

De esto se deduce que el componente que proporcione una interfaz, debe implementar todos los comandos que dicha interfaz tenga. Por otro lado, el componente que use una interfaz, debe programar en su código respuestas a los eventos que pueda tener ésta.

La Figura 3.1 muestra de forma esquemática el funcionamiento de las interfaces.

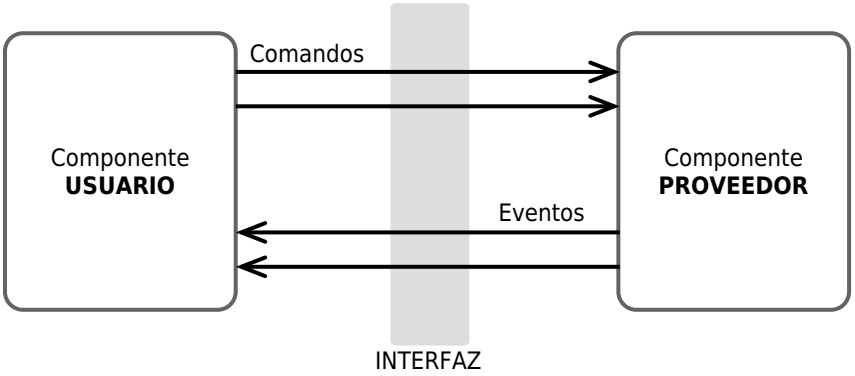

Figura 3.1 Esquema de funcionamiento de una interfaz

El componente usuario utiliza comandos o funciones que le aporta el proveedor. Éste puede avisar al usuario de cualquier suceso a través de los eventos.

Cuando una interfaz tiene tanto comandos como eventos, se denomina **interfaz bidireccional**. Por ejemplo:

```
interface Notify<val_t> {
  command error_t enable();
  command error_t disable();
  event void notify( val_t val );
}
```

Se proporcionan dos comandos denominados `enable` y `disable`; y a su vez se puede recibir un evento de notificación (`notify`).

3.3.1 Interfaz genérica[19]

Como su nombre indica, hay que especificar uno (o más) parámetros para que esté completamente definida. Un ejemplo puede ser la interfaz `Queue` (implementa una cola tipo FIFO, primer elemento en entrar, primero en salir) que está programada como

```
interface Queue<typedef t> {
  command bool empty();
  command uint8_t size();
  command t head();
  ...
}
```

[19] *También denominada paramétrica.*

Mediante el parámetro `t` se indica con qué elementos va a trabajar la cola. Por ejemplo, con `<uint32_t>` o `<bool>` la cola sería de enteros de 32 bit sin signo o de valores booleanos, respectivamente.

El parámetro no tiene porqué indicar un tipo de variable. Por ejemplo, en la interfaz `Timer`

```
interface Timer<typedef precision_tag> {
  ...
}
```

`precision_tag` indica la granularidad del temporizador (resolución de milisegundos, microsegundos...).

Si se deben especificar varios tipos, se hacen separados por comas:

```
interface Alarm<typedef precision_tag, typedef size_type> {
  ...
}
```

Como es lógico, si se proporciona una interfaz de un tipo determinado para que un componente la use, es necesario que ambos extremos coincidan en ese tipo, es decir, no es posible usar una interfaz `Timer` con precisión de milisegundos que esté programada para microsegundos.

Este tipo de interfaces es muy útil en cuanto a reutilización de código: si no existieran habría que crear interfaces `Queue` (en el ejemplo) para cada tipo de variable que está definida en nesC, o bien usar *cast* para convertir de unos tipos a otros.

3.3.2 Interfaces. Instancias

Puede ocurrir que una interfaz se quiera utilizar varias veces. A este sistema se le llama **crear instancias** de una interfaz. Como el nombre de las interfaces será el mismo, es necesario distinguirlas de alguna forma. Esto se consigue mediante la palabra reservada `as` (en inglés *como*).

```
module BlinkC
{
  uses {
    interface Timer<TMilli> as Timer0;
    interface Timer<TMilli> as Timer1;
```

```
    interface Timer<TMilli> as Timer2;
    interface Leds;
    interface Boot;
  }
}
```

En el módulo `BlinkC` se quieren utilizar tres temporizadores. La forma de usarlos es a través de la interfaz `Timer` (con precisión de milisegundos). Para distinguirlas unas de otras, se escribe la palabra `as` seguida del *alias* de la interfaz. Así, cuando sea necesario referirse a una interfaz, se usará el nuevo nombre: `Timer0`, `Timer1` o `Timer2`.

Realmente esta técnica de renombrar las interfaces se usa siempre. Es decir, cuando se escribe

```
uses interface Boot;
```

es exactamente igual a escribir

```
uses interface Boot as Boot;
```

Si bien a la hora de programar se prefiere la primera forma (siempre que sea posible).

3.3.3 Interfaz parametrizada

Es importante no confundir esta interfaz con una genérica ni con la creación de instancias. Se trata de una interfaz simple, que en determinados casos, es necesaria distinguirla (porque se puede usar varias veces).

Por ejemplo, el componente `CollectionC` posee una interfaz de recepción de datos:

```
interface Receive[collection_id_t id];
```

El parámetro `id` permite diferenciar qué tipos de mensajes se están usando, para que se puedan tratar de forma más adecuada. Este concepto de parametrización también puede aplicarse directamente a un componente. Por ejemplo:

```
CollectionSenderC(collection_id_t id)
```

El significado del valor `id` es el mismo que anteriormente.

3.4 Componentes. Implementación.

Recuérdese que existen dos tipos de componentes: módulos y configuración. Por otro lado, todos ellos están compuestos de dos bloques: signatura e implementación. En el punto 3.2 se ha estudiado la signatura. Ahora se tratará el segundo bloque: la implementación. Se distinguirá la implementación de configuraciones y de módulos.

3.4.1 Implementación de configuraciones

Como ya se sabe, las configuraciones permiten realizar el *wiring*, cableado o unión de unos componentes con otros (a través de interfaces) para dar lugar al programa final.

La primera parte de una configuración es la signatura, donde se especificaban las interfaces a usar y/o a proporcionar.

En la segunda parte, implementación, es donde se realiza el conexionado propiamente dicho. Mediante la palabra reservada `components` se listan todos los componentes que van a usarse en la aplicación (sin importar el orden de éstos). Puede usarse las veces que se desee, siempre que se haga antes de la conexión. Por ejemplo:

```
components MainC, BlinkC, LedsC;
```

COMPONENTES GENÉRICOS

Los componentes estudiados hasta ahora son únicos (en inglés *singleton*). Por ejemplo, dos configuraciones enlazan al componente `LedsC`, estarán ejecutando el mismo código y accediendo a las mismas variables que contiene.

Los componentes genéricos pueden tener creadas varias **instancias** (algo parecido a lo que ocurría con las interfaces). La idea que está detrás es realizar una abstracción de alto nivel de recursos *hardware* que tenga el mote. Un ejemplo es el componente `TimerMilliC`, que proporciona un temporizador configurable a nivel de usuario. Es posible hacer uso de varios temporizadores virtuales (a nivel

de *software*) gracias a estos componentes genéricos. Aunque el mote cuente con dos temporizadores *hardware*, se pueden utilizar muchos más a alto nivel mediante los componentes genéricos.

Para indicarlos, se utilizan las palabras reservadas `new` (para crear la instancia) y `as` (para renombrarlos, pues tienen el mismo nombre). Un ejemplo completo de implementación:

```
implementation
{
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as TimerA;
  components new TimerMilliC() as TimerB;
  components new TimerMilliC() as TimerC;
}
```

La primera línea `components` declara el uso de tres componentes (únicos). Las tres siguientes crean sendas instancias de `TimerMilliC`, renombrando a cada una para poder diferenciar los componentes. Se tienen así tres temporizadores independientes.

* * *

Con todo esto, el siguiente paso es la conexión o cableado. Se tienen tres operadores fundamentales: `->`, `<-` e `=`. El operador `=` se verá más adelante.

Las flechas `->` `<-` indican qué componente es el usuario de una interfaz y cuál el que la proporciona: la flecha siempre irá desde el usuario hasta el proveedor. El formato genérico es por tanto

```
CompUsuario.InterfazUsuario -> CompProveedor.InterfazProveedor;
```

Por ejemplo

```
BlinkC.Boot -> MainC.Boot;
```

El componente `BlinkC` hace uso de la interfaz llamada `Boot`, pero el que proporciona la interfaz es el componente `MainC`.

El sentido de la flecha puede ser el contrario, pero no se recomienda por comprensión del código. Esta línea es similar a la anterior:

```
MainC.Boot <- BlinkC.Boot;
```

La sintaxis del cableado puede simplificarse en muchos casos. Si el nombre de la interfaz que usa un componente es el mismo que el nombre de la que proporciona el otro componente, puede omitirse:

```
BlinkC -> MainC.Boot;
```

El compilador entiende que `BlinkC` usará una interfaz de nombre `Boot`.

También se puede especificar así

```
BlinkC.Leds -> LedsC;
```

De igual forma, el componente `LedsC` se supone proporcionará una interfaz denominada `Leds`.

El operador = tiene un uso algo distinto. Las flechas <- y -> se usan para realizar un cableado directo, uniendo interfaces proporcionadas por un componente y usadas por otro. Sin embargo, un componente de configuración también puede proporcionar/usar interfaces, pero no puede programar ninguna funcionalidad en su código. La forma de hacerlo es renombrando interfaces de otros componentes, lo que se consigue con el operador =. Es decir, el operador = conecta interfaces de la signatura con interfaces de componentes declarados mediante `components`. Por ejemplo:

```
configuration LedsC {
  provides interface Leds;
}
implementation {
  Components LedsP, PlatformLedsC;
  Leds = LedsP.Leds;
...
}
```

El componente de configuración `LedsC` proporciona la interfaz `Leds`, si bien esa interfaz es realmente proporcionada por el componente `LedsP`. El operador = permite que se exporte una interfaz que la proporcionan a su vez otro componente.

3.4.2 Implementación de módulos

El componente de tipo módulo incorpora la funcionalidad del programa propiamente dicha.

En la primera parte (signatura) se especificaban las interfaces usadas y/o proporcionadas.

Es en la implementación donde se hará uso de las interfaces, concretamente de los comandos y los eventos. Se distinguen dos casos, según si el módulo utiliza o proporciona la interfaz.

INTERFACES USADAS

Cuando se usa una interfaz, el programa puede hacer llamadas a las funciones que dicha interfaz proporcione. Esto se consigue usando la palabra reservada `call`. De forma general

```
call Interfaz.NombreComando();
```

Por otro lado, el componente proveedor de la interfaz puede avisar mediante eventos. Para detectar estos eventos se usa la palabra `event`:

```
event Interfaz.NombreEvento() {
  ...
}
```

Todo lo que se escriba entre las llaves se ejecutará cuando tenga lugar el evento[20].

Ejemplo de implementación de un módulo:

```
implementation {
  event void Boot.booted() {
    call Timer0.startPeriodic( 256 );
    call Timer1.startPeriodic( 512 );
    call Timer2.startPeriodic( 1024 );
  }

  event void Timer0.fired() {
```

[20] *Puede ocurrir que no se requiera hacer nada cuando suceda el evento. Basta con no escribir entre las llaves. Sin embargo, siempre hay que escribir la línea de event y las llaves (aunque no se haga nada).*

```
    call Leds.led0Toggle();
  }
  ...
}
```

El primer evento a tratar se llama `booted`, y lo aporta la interfaz `Boot`. Cuando tiene lugar, se ejecutan tres llamadas: comando `startPeriodic` de las interfaces `Timer0`, `Timer1` y `Timer2`.

Existe otro evento: `fired`, perteneciente a la interfaz `Timer0`. Cuando se reciba, se ejecuta el comando `led0Toggle` de la interfaz `Leds`.

INTERFACES PROPORCIONADAS

También puede ocurrir que el módulo proporcione ciertas interfaces. Ahora la situación es la inversa. Si la interfaz proporciona algún comando, la forma de programarlo será usando `command`:

```
command Interfaz.NombreComando() {
  ...
}
```

Todo lo que vaya entre las llaves se ejecutará cuando se llame a dicho evento (desde otro componente).

Si la interfaz utiliza eventos, la forma de activarlo y así avisar al otro extremo es mediante la palabra reservada `signal` y la siguiente línea de código:

```
signal Interfaz.NombreEvento();
```

* * *

La Figura 3.2 aclara los conceptos de interfaz usada y proporcionada.

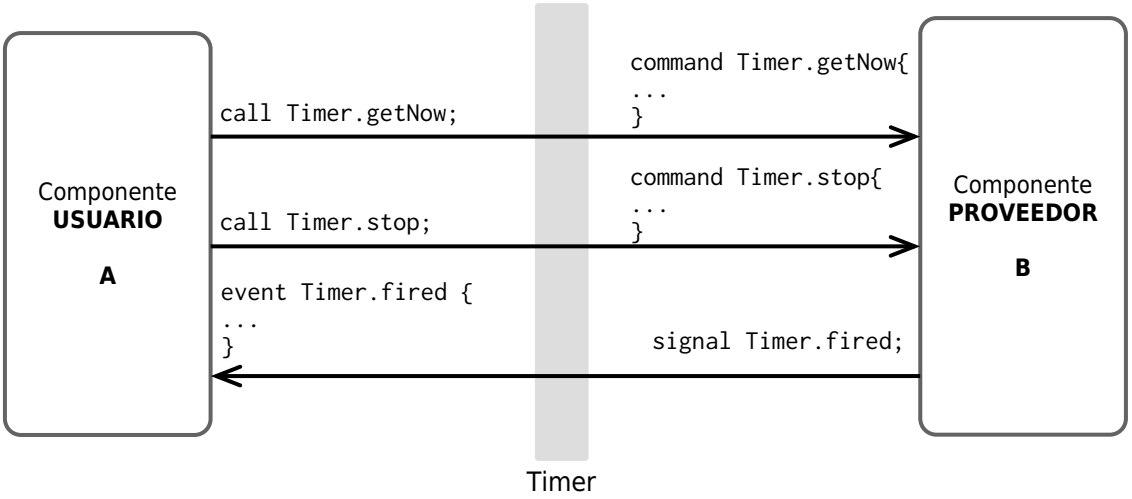

Figura 3.2 Comandos y eventos de una interfaz

El sentido de las flechas indica en qué componente se genera el comando o evento. En la figura, el componente A usa la interfaz "Timer" proporcionada por B. Por ejemplo, cuando el componente A realiza la llamada al comando "getNow", el B (proveedor) ejecuta el código asociado a dicho comando. Por su parte el componente B puede enviar el evento "fired" hacia el A (usuario). Cuando esto suceda, se ejecutará el código perteneciente al evento.

3.5 Archivos nesC

En este punto se tratan los nombres que tienen los archivos de nesC y sus extensiones. Algunas normas de uso:

- Todos los ficheros con código nesC deberán tener extensión `.nc`.
- Los nombres de directorios deben ir en minúsculas.
- Todos los componentes privados o primitivos (del sistema operativo) terminarán en `P`.
- Asimismo, los componentes públicos o complejos deberán terminar en `C`.
- Si se trata de un componente público de tipo configuración, se suele acabar en `AppC`, indicando así que se trata de la definición de una aplicación (aunque no es obligatorio).
- Nombres de interfaces y componentes pueden mezclar minúsculas y mayúsculas, siempre que comiencen por mayúscula. Evitar que terminen en `P` o en `C`.
- En nesC también es posible usar ficheros de cabecera (*header*) igual que en lenguaje C. La extensión será `.h`.

- El fichero tipo *makefile* (necesario para realizar la compilación y generar el ejecutable) no tiene extensión alguna.

En la Figura 3.3 aparecen algunos ejemplos de nombres de fichero.

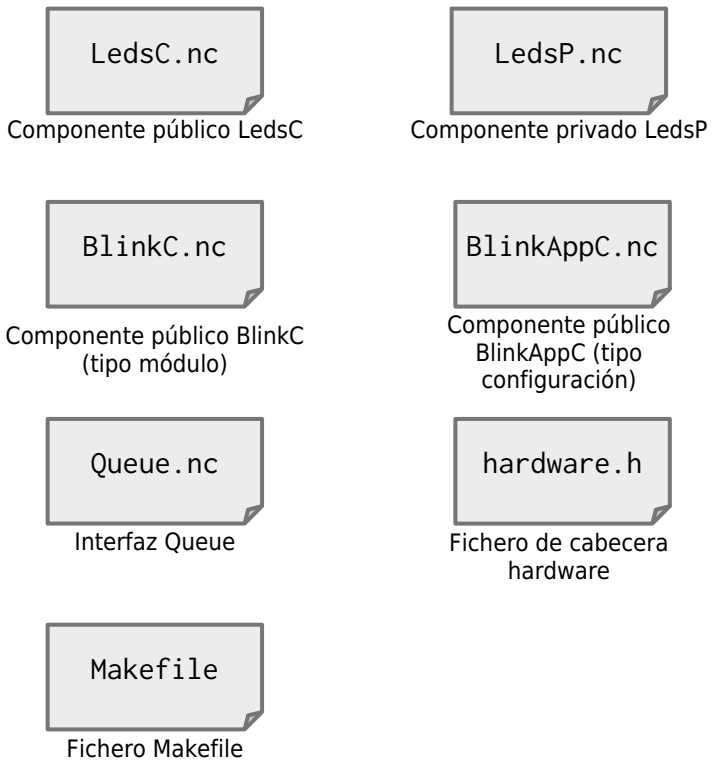

Figura 3.3 Ejemplos de nombres de fichero en TinyOS

Nomenclatura y extensiones de los archivos relacionados con el lenguaje nesC.

3.6 Ejemplo de código

El siguiente programa nesC implementa un contador binario en el mote, a través de sus tres LED.

Para conseguirlo, se hace parpadear un LED a una frecuencia determinada, por ejemplo 1 Hz el bit más significativo, el segundo a 2 Hz (el doble) y el menos significativo a 4 Hz. El efecto final es un contador binario de tres bits.

La aplicación contiene tres ficheros:

- `BlinkAppC.nc`: componente de configuración de la aplicación.

- `BlinkC.nc`: componente de tipo módulo de la aplicación.
- `Makefile`: archivo para generar el ejecutable.

Código fuente del componente `BlinkAppC.nc`

```
1  configuration BlinkAppC {
2  }
3
4  implementation {
5    components MainC, BlinkC, LedsC;
6    components new TimerMilliC() as TimerA;
7    components new TimerMilliC() as TimerB;
8    components new TimerMilliC() as TimerC;
9
10   BlinkC -> MainC.Boot;
11   BlinkC.Timer0 -> TimerA;
12   BlinkC.Timer1 -> TimerB;
13   BlinkC.Timer2 -> TimerC;
14   BlinkC.Leds -> LedsC;
15 }
```

La línea 1 indica que se trata de un componente de configuración, llamado `BlinkAppC`. Lo que está entre llaves es la signatura: en este caso no existe i.e. este componente ni usa ni proporciona alguna interfaz.

A partir de la línea 4 se encuentra el bloque de implementación. Para el caso de una configuración, se tienen que especificar los componentes que se van a usar en el programa: `MainC`, `BlinkC`, `LedsC`. También se van a utilizar tres temporizadores. Como el módulo que los implementa es de tipo genérico, hay que crear tres instancias del mismo, nombrándolas de forma distinta: `TimerA`, `TimerB` y `TimerC`.

Una vez hecho esto, se procede al cableado o *wiring*. Por ejemplo, la línea 10 indica que la interfaz `Boot` del módulo `BlinkC` usa la interfaz del mismo nombre que la proporciona `MainC`. Para los temporizadores se procede igual: la interfaz llamada `Timer0` del módulo `BlinkC` utiliza la interfaz del mismo nombre del componente `TimerA`[21].

A continuación se muestra el código del módulo `BlinkC.nc`:

[21] *Nótese como ahora se trabaja con el alias del componente "TimerA" y no con el nombre genérico "TimerMilliC".*

```
1  #include "Timer.h"
2  module BlinkC {
3    uses {
4      interface Timer<TMilli> as Timer0;
5      interface Timer<TMilli> as Timer1;
6      interface Timer<TMilli> as Timer2;
7      interface Leds;
8      interface Boot;
9    }
10 }
11
12 implementation {
13   event void Boot.booted() {
14     call Timer0.startPeriodic( 256 );
15     call Timer1.startPeriodic( 512 );
16     call Timer2.startPeriodic( 1024 );
17   }
18   event void Timer0.fired() {
19     call Leds.led0Toggle();
20   }
21   event void Timer1.fired() {
22     call Leds.led1Toggle();
23   }
24   event void Timer2.fired() {
25     call Leds.led2Toggle();
26   }
27 }
```

La primera línea comienza con una directiva del compilador: `#include`. Indica un fichero de cabecera con definiciones.

La línea 2 comienza el primer bloque (signatura), indicando que se trata de un componente de módulo llamado `BlinkC`. Aquí se definen las interfaces que se usarán (en este caso no se proporciona ninguna) en el módulo.

La interfaz `Timer` es genérica, y hay que especificar algún parámetro para definirla. Haciendo `Timer<TMilli>` se indica que la precisión que se requiere es de milisegundos. Además, como van a hacer falta tres interfaces, se han de crear instancias de ellas. Esto se consigue con la palabra `as` seguida del alias: `Timer0`, `Timer1` y `Timer2`.

También se incluyen `Leds` y `Boot`. Los nombres de estas cinco interfaces son los que se usan en el cableado visto anteriormente (`BlinkAppC.nc`, líneas 10 a 14).

Así finaliza la parte de la signatura. El resto es la implementación. La implementación de un módulo ha de contener respuesta a los diferentes eventos.

El primero (línea 13) es el evento `booted` de la interfaz `Boot`. Tiene lugar cuando el mote ha arrancado y está preparado para funcionar. Cuando se produce, se realizan tres llamadas a los temporizadores a través de sus interfaces (líneas 14-16). El comando `startPeriodic` inicia el temporizador correspondiente, haciendo que lance un evento según el tiempo especificado como parámetro. Por ejemplo el primero de ellos tiene valor 256 , indicando que el temporizador asociado a la interfaz `Timer0` lanzará un evento cada $250\,\mathrm{ms}$ [22] (de forma periódica). El resto de comandos son análogos, para los dos temporizadores restantes. Con esto se consigue que cuando se inicie el dispositivo, se pongan en marcha los tres temporizadores (cada uno controla a un LED).

Los eventos restantes son para cuando venza cada uno de los temporizadores. En la línea 18 se recibe el evento `fired` de la interfaz `Timer0`, es decir, cada $250\,\mathrm{ms}$ se ejecutará lo que se especifique en dicho evento. La llamada al comando `led0Toggle` de la interfaz `Leds` provoca que el LED0 del mote conmute de estado (encendido/apagado) cada vez que se llame.

Con los otros dos temporizadores se procede igual, en este caso para el LED1 y el LED2.

Para terminar, el fichero `Makefile` contiene las órdenes de compilación:

```
1 COMPONENT=BlinkAppC
2 include $(MAKERULES)
```

La primera línea indica al compilador con qué componente de tipo configuración tiene que trabajar para generar la aplicación. La segunda indica las reglas de compilación (varían según el modelo concreto de mote).

[22] *Los milisegundos que se especifican como parámetro son binarios, lo que significa que 1 segundo tiene 1024 milisegundos binarios (y no 1000).*

3.7 Representación gráfica de componentes

Existe una forma de representación gráfica para comprender mejor la estructura de una aplicación de TinyOS. Los elementos básicos se muestran en la Figura 3.4.

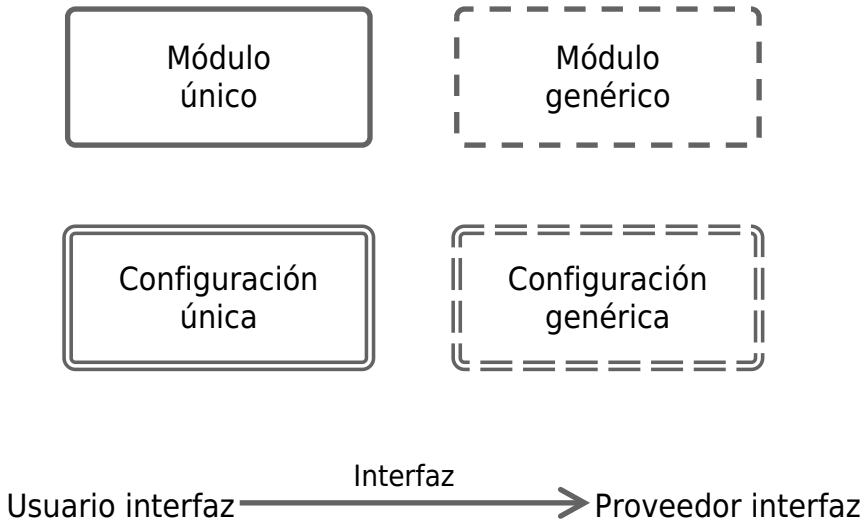

Figura 3.4 Elementos básicos para diagramas nesC

En la Figura 3.5 se ilustra cómo quedaría el esquema aplicado al ejemplo anterior `Blink`.

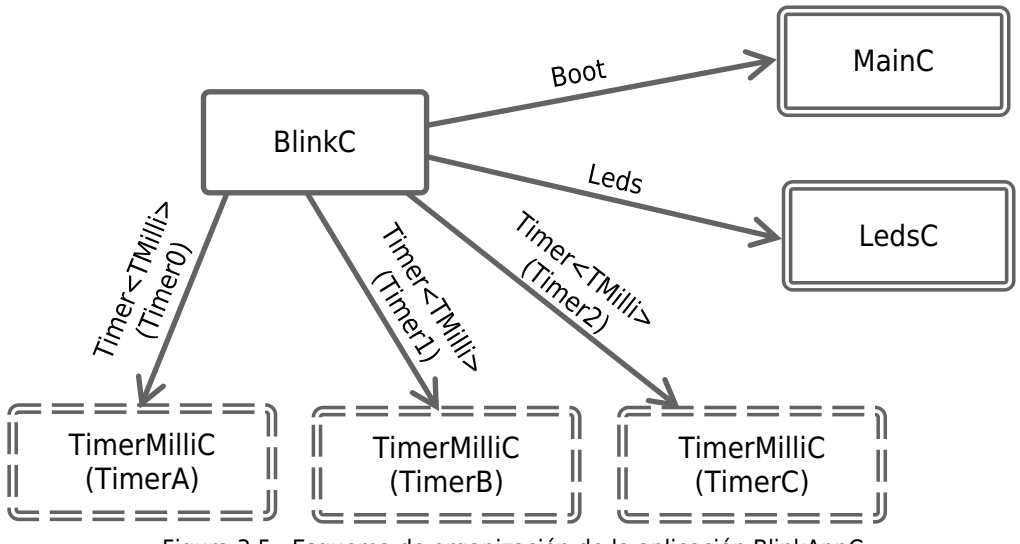

Figura 3.5 Esquema de organización de la aplicación BlinkAppC

Todas las relaciones que se muestran se especifican en el fichero "BlinkAppC.nc" (no aparece en el diagrama). El módulo "BlinkC" utiliza interfaces proporcionadas por "MainC", "LedsC" y el genérico "TimerMilliC", del cual se crean tres instancias. El nombre entre paréntesis representa el alias de una instancia.

3.8 Tareas

La mayoría del código de TinyOS visto es de tipo síncrono. Es decir, todo el programa se ejecuta en un solo contexto y no hay ningún tipo de prioridad de una función frente a otra: mientras se está ejecutando el código correspondiente (síncrono), no se podrá ejecutar otro hasta que el primero no finalice su ejecución en la CPU.

Esto permite al programador de tareas de TinyOS poder minimizar el consumo de memoria y mantener el código lo más simple posible. Sin embargo, hay ocasiones en las que este sistema no es ventajoso. Por ejemplo, un código que realice algún cálculo complejo puede tardar demasiado tiempo en ejecutarse; el mote puede estar perdiendo paquetes vía radio porque no puede atenderlos. El objetivo es "fraccionar" de alguna manera el código para el cálculo de forma que se vaya realizando poco a poco, evitando saturar a la CPU con un único proceso.

Este es el cometido de las **tareas**. La tarea permite ejecutar código en segundo plano (o *background*), de forma que se le indica al microprocesador la necesidad de ejecutarlo, y es éste el que decide cuándo hacerlo.

Para declarar una tarea en el lenguaje nesC, se hace

```
task void NombreTarea() {
  ...
}
```

Obsérvese que no recibe ni devuelve parámetro alguno. Esto permite un programador de tareas más simple. Una vez hecho esto, es posible lanzarla (o postearla, *post*):

```
post NombreTarea();
```

Un componente puede lanzar la tarea dentro de un comando, de un evento, o incluso de una tarea.

La operación `post` almacena la tarea en una cola de tareas, que se procesa según la regla FIFO de forma que hasta que no se finaliza la ejecución de una tarea no se pasa a la siguiente (si la hubiere). Es decir, las tareas no se pisan entre ellas, su eje-

cución es secuencial. Lo único que puede pausar una tarea es una interrupción *hardware*.

3.8.1 Llamadas split-phase[23] y tareas

Una operación *split-phase* es típica de componentes periféricos *hardware*. Cuando se realiza una llamada, comienza a ejecutarse la orden, pero el programa continúa realizando otras operaciones. Mediante un evento, la función puede avisar de que se ha completado la ejecución.

En el lado opuesto se encontraría una operación bloqueante *(blocking)*. La llamada a un comando detiene la ejecución de cualquier otro código; hasta que no finalice la operación no se continúa la ejecución.

Sea el siguiente ejemplo nesC:

```
estado = ESPERA;
op1();
call Timer.startOneShot( 512 );

event void Timer.fired() {
  op2();
  estado = EJECUCION;
}
```

La llamada al temporizador `Timer.startOneShot()` es de tipo *split-phase*. Se lanza un temporizador de $500\,\mathrm{ms}$, pero la CPU puede estar realizando cualquier otro proceso. La forma que tiene de avisar (de que ha transcurrido el tiempo programado) es enviando el evento `Timer.fired()`.

Véase ahora este otro ejemplo:

```
estado = ESPERA;
op1();
sleep( 512 );
op2();
estado = EJECUCION;
```

[23] *Se puede traducir como llamada en dos fases, o de fases partidas.*

El comportamiento es el mismo que anteriormente (espacio de $500\,\mathrm{ms}$ entre una operación y la siguiente). La diferencia estriba en que ahora no se permite la ejecución de otro código mientras transcurre el tiempo: la CPU se bloquea durante los $500\,\mathrm{ms}$. Se ha hecho una llamada bloqueante.

Otro ejemplo claro de estos modos de comportamiento puede observarse en el convertidor analógico-digital muy propio de los microcontroladores.

Podría plantearse el uso de hilos *(threads)*, en los cuáles, cada operación que requiera un cierto tiempo se ejecuta en un hilo diferente. Por ejemplo, cuando se llama al CAD para efectuar una conversión, el propio sistema operativo realiza la llamada, y mantiene al hilo en memoria, a la espera de que finalice la conversión. Mientras tanto, continúa con la ejecución de otro hilo. Cuando el CAD termina, el SO recibe una interrupción, recupera el hilo de memoria y termina con la operación.

Sin embargo, los sistemas de motes tienen una limitación considerable en cuanto a memoria RAM. Cada hilo, cuando se mantiene suspenso debe reservar un cierto espacio en RAM para almacenar las variables que está usando, y se estaría desperdiciando dicho espacio. Lógicamente, cuantos más hilos se mantengan en ejecución, más gasto de memoria habrá.

La solución que da TinyOS por tanto es el uso de las *split-interfaces*, de forma que cuando se llama a la función, ésta comienza, y es un evento el que avisa que ha finalizado, lo que hace que los programas de TinyOS sólo requieran usar una única pila de memoria RAM.

Aunque estas interfaces son algo más complejas a la hora de la programación aportan ciertas ventajas: ahorro de memoria, pues las operaciones no tienen que almacenar variables de estado mientras se ejecutan (reduciendo el tamaño de la pila), y el programa no pierde capacidad de respuesta: al no existir ninguna función que mientras se ejecute bloquee al sistema, éste puede reaccionar ante cualquier otro evento casi de forma inmediata.

Las tareas son el modo que tiene TinyOS de emular el comportamiento de dispositivos periféricos pero desde el punto de vista *software* y no *hardware*. La tarea se postea y es el propio sistema el encargado de ejecutarla en segundo plano.

3.8.2 Gestión de tareas con prioridad

Las operaciones o cálculos que se realizan en el mote se llevan a cabo mediante tareas. Sin embargo, el sistema de gestión de tareas por defecto de TinyOS puede no ser suficiente: son necesarias **prioridades**.

Por ejemplo, algoritmos con cierta complejidad pueden llegar a consumir demasiados recursos de CPU. Sería conveniente fijar una prioridad baja con el fin de que se puedan seguir realizando otras operaciones de forma más continuada y sin bloqueos.

Para ello se han creado un conjunto de componentes para dotar de prioridad al programador de tareas de TinyOS, bajo el nombre de The TinyOS-2.x Priority Level Scheduler.

En el componente de configuración (en la parte de `implementation`) hay que incluir:

```
components PrioritySchedulerC;
```

Este componente implementa una cola de tareas pero con diferentes prioridades (hasta cinco posibles grados) optimizando además el consumo de memoria, facilitando así la labor del programador.

A continuación se definen las tareas a ejecutar, y dependiendo de su prioridad se usa el componente adecuado. Por ejemplo:

```
components new VeryLowTaskC() as TareaA;
components new LowTaskC() as TareaB;
components new HighTaskC() as TareaC;
components new VeryHighTaskC() as TareaD;
```

Según el código, la `TareaA` está definida como de muy baja prioridad, le sigue la `TareaB`. `TareaC` es de alta prioridad; y `TareaD` puede bloquear a todas las demás (máxima prioridad).

Como se observa, se trata de componentes genéricos, con lo que se pueden crear instancias de todos ellos, es decir, varias tareas con la misma prioridad (para diferenciarlas se usa as seguida del nombre específico). Falta un quinto tipo de prioridad, denominado *BASIC*, y que se correspondería con una tarea por defecto de TinyOS.

Cada componente proporciona una interfaz llamada `TaskBasic`. En caso de que existan varias tareas, habrá que distinguir estas interfaces usando la palabra reservada as seguida del *alias*. Por ejemplo (en la signatura de un módulo):

```
interface TaskBasic as Conversion;
```

Para añadir a la cola una tarea y esperar su posterior ejecución, se usará el comando `postTask` de dicha interfaz:

```
call Conversion.postTask();
```

Por último, faltaría definir la tarea propiamente dicha. Esto se hace en el evento `runTask`. Este evento tiene lugar cuando se ha posteado la tarea correspondiente y el programador decide que es el momento de ejecutarla. Siguiendo con el ejemplo anterior:

```
event void Conversion.runTask() {
  ...
}
```

3.9 Mensajes en TinyOS 2.x

Existen diferencias entre la estructura de mensajes en la versión actual de TinyOS y la anterior. Antes, el mensaje se abstraía en una estructura denominada `TOS_Msg`; ahora, la estructura está definida en `message_t`, y es de tipo opaco, por lo que se aconseja no acceder directamente a leer/escribir de este *buffer*, sino usando las interfaces creadas a tal efecto.

La estructura del mensaje está definida en un fichero de cabecera de nesC:

```
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;
```

Como se observa, la estructura `message_t` está compuesta de cuatro elementos internos:

- `header` (o cabecera). Se trata de un vector, cuyo tamaño viene dado a su vez por otra estructura, denominada `message_header_t`. Es en esta última donde se almacenan las cabeceras según la plataforma que se esté usando. De esta forma, el *buffer* `message_t` se hace más genérico. Por ejemplo, para el chip de radio CC2420, la cabecera ocupa 11 B, mientras que si es una cabecera de protocolo serie, el tamaño es de 5 B. En caso de que haya que efectuar un relleno (*padding*) lo hará el propio sistema operativo y nunca el usuario.
- `data`, o campo de datos. El tamaño viene dado por la constante `TOSH_DATA_LENGTH`, que es posible modificar en tiempo de compilación (por ejemplo, a través de la línea de comandos). Por defecto, es de 28 B. Si la capa de mensajes recibiera un mensaje de mayor tamaño que el especificado en la variable, lo debe descartar automáticamente.
- `footer`, final del mensaje. Permite ajustar el tamaño del paquete al especificado para las MTU[24] del sistema.
- `metadata` (metadatos). Datos que aportan información adicional. Por ejemplo, la estructura `message_metadata_t` para el chip CC2420 incluye información sobre potencia de transmisión, RSSI, códigos CRC, asentimientos, marcas de tiempo...

Como se ha comprobado, no se debe acceder a estos campos directamente, pues es el SO (TinyOS) el que gestiona las asignaciones de memoria y las posiciones que ocupa cada campo.

Añadir que esta estructura se utiliza en cualquier tipo de mensaje, ya sea enviado a través de la radio o por un puerto de comunicaciones serie.

[24] *Maximum Transfer Unit. Expresa el tamaño máximo de una unidad de datos que puede transmitirse en un protocolo específico.*

3.10 Comunicación del nodo

A continuación se explican las interfaces básicas para la comunicación del sensor inalámbrico.

- `Packet`. Acceso básico a la estructura de mensajes `message_t`. Para borrar el contenido de un paquete existe el comando `clear`; `maxPayloadLength` devuelve el tamaño máximo de datos (carga útil) que soporta el protocolo. Por su parte, `getPayload` retorna un puntero al área de datos de un paquete. Esto facilita la labor del programador a la hora de leer/escribir información en un mensaje.
- `Send`. Como su nombre indica, permite enviar un mensaje desde un nodo. Mediante el comando `getPayload` se obtiene un puntero hacia el área de carga útil del mensaje (esto evita tener que trabajar directamente sobre el mensaje, junto con las cabeceras y otros campos de control). También se tiene `maxPayloadLength`, que indica el tamaño máximo para los datos.
  Para enviar el mensaje, se usa el comando `send`. Si devuelve un valor satisfactorio, indica que posteriormente se recibirá el evento `sendDone`. Esto significa que el mensaje está en cola listo para ser enviado (si existiera algún problema a la hora de transmitirlo, el evento retornaría el código de error asociado). El comando `cancel` puede abortar el envío de un mensaje (siempre que éste no haya sido transmitido).
- `Receive`. Recepción de paquetes. Únicamente tiene un evento: `receive`. Cuando se recibe, se tiene un puntero hacia el paquete recibido. También se tiene información acerca del tamaño de la carga útil del mismo.
- `PacketAcknowledgements`. Permite activar o desactivar el uso de asentimientos para cada mensaje transmitido. Por ejemplo, mediante la orden `requestAck` se utilizan asentimientos síncronos para el paquete; con `noAck` se indica lo contrario.

Es muy común que una aplicación TinyOS tenga varios componentes que usen la transmisión de mensajes simultáneamente. Para estos casos, se tienen otro tipo de interfaces, denominadas *Active Message* (AM). Cada tipo de mensaje utilizará un identificador o tipo AM. Es similar al uso del número de puerto en el protocolo

UDP: permite multiplexar distintos servicios sobre un mismo soporte de comunicación.

Este protocolo es de tipo *single-hop*, es decir, el mensaje se genera en un nodo y el nodo destino lo consume: no hay nodos intermedios en la comunicación.

- `AMPacket`. Es similar a la interfaz `Packet`: se pueden usar todos los comandos ya vistos. Sin embargo, ahora es posible obtener la dirección de un nodo (mediante `address`), hacia qué nodo va dirigido un paquete (comando `destination`) o discernir si un paquete recibido es realmente para el nodo, con el comando `isForMe`...
- `AMSend`. Es prácticamente igual a la interfaz `Send`. La única diferencia radica en el comando `send`, que en este caso hay que especificar una dirección (AM) de destino para el paquete a enviar.
- `ActiveMessageAddress`. Permite leer y/o realizar modificaciones en la dirección de un nodo. Esta interfaz no está creada para el uso general: el cambio en el identificador de red puede provocar fallos en ésta, por lo que se aconseja que no se use a menos que sea estrictamente necesario.

3.11 Comunicación radio. Protocolos de red

Usando las interfaces para la comunicación vistas en el punto anterior, existen varios componentes que implementan su funcionalidad:

- `AMSenderC`. Proporciona las interfaces para el envío de mensajes: `Packet`, `AMPacket`, `AMSend` y `PacketAcknowledgements`. Este componente está parametrizado: cuando se declare su uso hay que especificar un valor de identificación único (recuérdese que pueden existir varios componentes que envían mensajes).
- `AMReceiverC`. Funcionalidad complementaria al anterior: interfaces para recibir mensajes. `Packet`, `AMPacket` y `Receive`. Este componente también está parametrizado (indicación con un identificador).

- `AMSnooperC`[25]. Este componente se usa para observar la cabecera de los paquetes que se reciben, y comprobar si el nodo debe o no debe recibirlo. Interfaces que proporciona: `Packet`, `AMPacket` y `Receive`. Componente parametrizado.
- `ActiveMessageAddressC`. Permite leer y/o modificar la dirección de un nodo. Incluye la interfaz `ActiveMessageAddress`, cuyo uso estará restringido a casos muy particulares.
- `ActiveMessageC`. Es uno de los componentes más generales, pues incluye la gran mayoría de las interfaces vistas para la comunicación de nodos. El hecho de que existan muchos modelos de motes haría que este componente cambiara según la plataforma en la que se usara. Para ello, se crean componentes de bajo nivel que son usados por `ActiveMessageC`, de forma que para el programador sea algo transparente. Por ejemplo, en el caso del modelo Tmote Sky, el chip de radio es el CC2420, y el componente auxiliar se denomina `CC2420ActiveMessageC`.

A continuación se detallan algunos protocolos de red (radio) muy útiles para el trabajo que se desarrolla.

3.11.1 Colección

Este protocolo tiene como fin reunir todos los datos que se generen en una red de sensores y llevarlos hacia un nodo principal, también conocido como nodo base o raíz[26].

El uso más común es establecer un nodo como raíz, y los demás nodos que conforman la red han de enviar sus datos hasta él. La estructura de red que queda se denomina árbol. En la Figura 3.6 se muestra un ejemplo con una configuración, así como una posible ruta de datos.

[25] *Snoop se traduce como fisgonear, curiosear…*

[26] *Root node.*

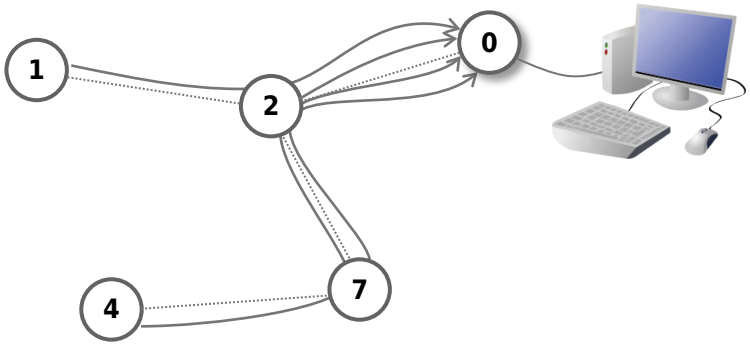

Figura 3.6 Red de colección de datos

Dentro de cada nodo aparece el identificador único para la red: el nodo 0 representa al nodo raíz (conectado a un sistema para la recepción de datos). En línea discontinua se muestra el enlace elegido por cada nodo (por ejemplo, el nodo 4 decide que la mejor ruta para llegar al raíz es a través de su vecino 7). La línea continua representa el flujo de información (p.ej. los datos del nodo 1 pasan a través del número 2 antes de llegar a la base).

En la actual versión del sistema operativo TinyOS, se cuenta con dos tipos de protocolos de colección: CTP[27] y MultihopLQI.

COLLECTION TREE PROTOCOL (CTP)

Entre sus características destaca que no depende de la plataforma (tanto microcontrolador como radio) en la cual se implemente. Además, posee un rendimiento aceptable. Un punto negativo es su excesivo consumo de memoria de código.

El algoritmo implementado usa el parámetro denominado ETX[28]. Por ejemplo, un nodo raíz tiene por definición $ETX = 0$; un nodo cualquiera tendrá un valor igual al que tenga el nodo inmediatamente por encima de él (denominado nodo padre) más el valor ETX del enlace entre ellos. Es decir, dado un conjunto válido de rutas para llegar a un determinado destino, el protocolo deberá tomar aquella con el valor ETX mínimo. En particular, este valor se almacena en un entero sin signo de $16\,\mathrm{b}$, resultando un rango $\left[0, 65535\right]$.

En la Figura 3.7 se muestra un esquema con los distintos elementos (se traducen a componentes de nesC) que implementan este protocolo.

---

[27] *Collection Tree Protocol.*

[28] *Expected Transmissions. De alguna forma representa el coste que tiene un determinado camino hacia un nodo.*

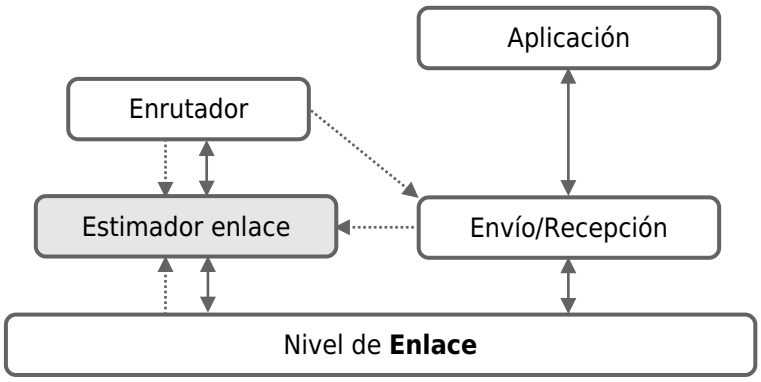

Figura 3.7 Diagrama de bloques protocolo CTP

Las flechas continuas indican transferencia de mensajes de datos, mientras que las discontinuas muestran transferencia de información de control del protocolo.

- **Estimador de enlace**. Este bloque monitoriza la calidad del enlace que posee con sus sensores vecinos. Como se ve en el esquema, está desacoplado del bloque de establecimiento de rutas (enrutamiento). La información que se transfieren deberá ser un indicador de la calidad del enlace. Para conseguir este seguimiento, se envían periódicamente mensajes de control o balizas (*beacons*), si bien la orden para enviarlos proviene del enrutador. Es importante destacar que se encarga de monitorizar el enlace hasta sus nodos vecinos (lo que se denomina *single-hop*).
- **Enrutador** *(router)*. Utiliza una métrica basada en el parámetro ETX. El periodo para enviar una baliza (a través del estimador de enlaces) es configurable, desde los $64\,\mathrm{ms}$ hasta varios minutos. El criterio para seleccionar otra ruta diferente es que sea mejor en al menos $1.5{\cdot}ETX$ a la ruta actual.
- **Bloque envío/recepción**. Interactúa con la capa de enlace enviando y recibiendo los mensajes. Debe evitar la duplicación de mensajes, controlar los temporizadores para las retransmisiones, así como detectar posibles bucles. También posee una cola para los mensajes, decidiendo así cuando enviar cada uno. Existe una comunicación con el estimador de enlaces, por ejemplo para avisar si un enlace ha perdido calidad (por ejemplo, demasiadas retransmisiones).

MULTIHOP LQI

Se trata de una variante del protocolo CTP. Está adaptada para que funcione con los chips de radio modelo CC2420. Este dispositivo aporta información sobre la capa física de datos, denominado LQI[29]. Este valor se calcula a partir de la variable RSSI, si bien se calcula también una correlación de los símbolos recibidos para evitar valores falsos de la calidad.

La diferencia con el protocolo CTP es que no existe un módulo de estimación de enlaces propiamente dicho: éste ya está implementado en el propio chip de radio.

Como ventajas respecto del anterior, es que el consumo de memoria de código es menor (pues necesita menos componentes). Por el contrario, la calidad de estimación puede no ser tan buena y depende de la plataforma (solo aquéllas que incluyan el chip CC2420 podrán usarlo).

3.11.2 Diseminación

Este protocolo tiene la funcionalidad inversa al anterior. En este caso se trata de enviar un determinado mensaje (desde el nodo raíz) y que lo reciban el resto de nodos de la red.

En la Figura 3.8 se ilustra el esquema de una red usando la diseminación de datos.

[29] *Link Quality Indication.*

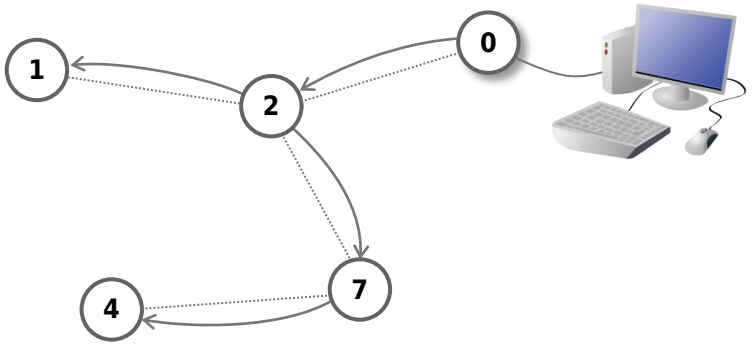

Figura 3.8 Red de diseminación de datos

Dentro de cada nodo aparece el identificador único para la red: el nodo 0 representa al nodo base o raíz (conectado a un equipo). La línea discontinua muestra qué enlaces tiene cada nodo con sus vecinos (por ejemplo, el nodo 2 para enviar un mensaje al número 4 lo hace a través de su vecino 7). Por su parte, la línea continua ilustra el camino que sigue el mensaje generado en la base (p.ej. el nodo 2 recibe el mensaje y lo reenvía hacia 1 y 7).

En cuanto a la implementación en TinyOS, se tienen hasta tres versiones de este protocolo, establecidas en dos grupos: Drip por un lado, y DIP y DHV por otro.

DRIP

Cada elemento de datos para diseminar se trata como una entidad individual, por lo que ofrece un grado de control bastante fino, por ejemplo, para especificar en qué momento preciso se deberá enviar un paquete de datos.

Como se ha comentado, cada paquete de datos se envía y disemina de forma independiente.

En cuanto al dispositivo radio, es el programador quién debe activarlo o desactivarlo según corresponda.

Este protocolo debería usarse en casos en los que se tienen pocos tipos de datos y no se sabe a priori qué tipos de datos usa cada nodo. Esta flexibilidad implica una cantidad mayor de mensajes de control por toda la red.

DIP Y DHV

Todos los elementos de datos o paquetes se tratan en conjunto, como un grupo. Los parámetros de control y configuración se aplicarán a todos los tipos de mensajes por igual.

Se tiene un temporizador configurable para todos los tipos de datos. En este caso, todos los nodos participantes en la red deben prepararse para trabajar con el mismo tipo de mensajes.

Respecto a la radio, DHV la activa de forma automática cuando sea necesario, mientras que DIP no lo hace.

Este grupo debería ser usado cuando, efectivamente todos los nodos trabajen con los mismos mensajes y se necesite una alta eficiencia en la red. En la mayoría de los casos, DHV transmite menos mensajes y la red converge hasta dos veces más rápido que con DIP.

3.11.3 Deluge T2

Corresponde a un protocolo para diseminar de forma segura grandes cantidades de datos, como por ejemplo, archivos binarios de aplicación. De hecho, cuando funciona en conjunto a la aplicación denominada *bootloader*, permite reprogramar toda una red de sensores.

El protocolo está concebido originalmente para la versión 1 de TinyOS (llamado Deluge 2.0). Aunque la mayor parte del código se puede reutilizar en TinyOS 2.x, el comportamiento no es exactamente el mismo.

Hay que añadir que a fecha de elaboración de este documento, está en fase experimental, por lo que su código no está totalmente depurado y podrían observarse errores de cualquier tipo. Además, solo funciona en algunas plataformas de sensores inalámbricos, como Tmote Sky, MicaZ, Iris o mulle.

Uno de los usos más interesantes de este protocolo es programar una red completa de nodos de forma automática, e inalámbrica (Figura 3.9). Esto repercutiría en un gran ahorro de tiempo y recursos, pues no es necesario programar (vía USB) cada mote individualmente.

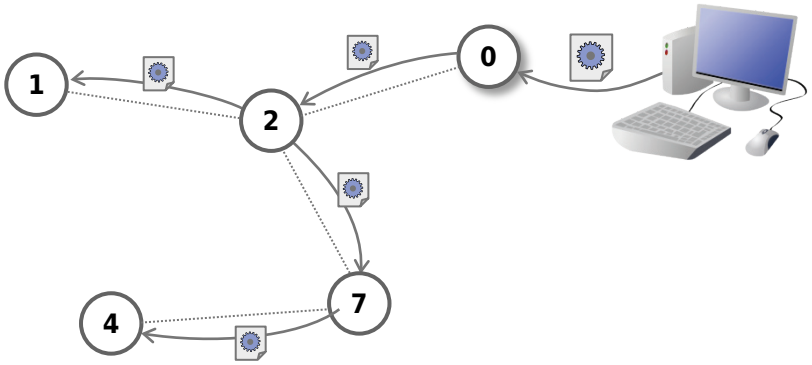

Figura 3.9 Red de protocolo Deluge T2

Para este protocolo, se distinguen dos tipos de nodos: el nodo estación-base (en la figura, 0) y el resto de nodos (que van a recibir el nuevo programa). Realmente se trata de una red de diseminación de datos, vista anteriormente. La imagen del programa (en formato binario) se envía desde un equipo hacia el nodo base. Después, basta con enviarle el comando para que comience la diseminación (en este caso de la imagen de la aplicación) hacia el resto de nodos de la red.

Antes de comenzar a usar Deluge T2, hay que comprobar que se tenga instalado el intérprete de lenguaje Python y la herramienta PySerial, para poder comunicar con el puerto serie (USB).

Los códigos de Deluge T2 no se instalan por defecto con TinyOS. Se pueden descargar desde el repositorio y compilar la aplicación.

Para empezar, hay que compilar la aplicación `tinyos-2.x/tos/lib/tosboot` para la plataforma concreta. Esto permite arrancar el mote de una forma determinada. Por ejemplo (Tmote Sky)

```
make telosb
```

Un primer ejemplo para comprender el funcionamiento de Deluge T2 se incluye en el directorio `tinyos-2.x/apps/tests/Deluge/Blink`. Básicamente, la aplicación hace parpadear el LED2 o el LED0 dependiendo de si se especifica o no (respectivamente) la bandera `BLINK_REVERSE` a la hora de compilar. Lógicamente, incluye el componente `DelugeC`.

Basta con ejecutar un *script* incluido en el mismo directorio:

```
./burn bsl,<PuertoCOM-1> serial@COM<PuertoCOM>:115200 telosb
```

La variable `PuertoCOM` indica el número de puerto asignado al mote conectado al PC (se supone un sistema operativo Windows). Por ejemplo, si el puerto COM es el número 15, quedaría

```
./burn bsl,14 serial@COM15:115200 telosb
```

Lo primero que se hace es compilar `Blink` con una única bandera: `DELUGE_BASESTATION`. Esto indica que se requiere una compilación para Deluge. Una vez compilada, se instala en el mote y el usuario debería observar el LED0 (rojo) parpadear.

El siguiente paso (tras pulsar la tecla `ENTER`) es compilar la misma aplicación pero añadiendo el modificador `BLINK_REVERSE`: se compila una aplicación distinta. En vez de instalarse de la forma común, se copia la imagen del programa en una zona de la memoria flash externa, de forma que no sobrescribe al programa anterior. Durante este proceso, se muestra información de dicha imagen (tamaño, bloques de memoria) y progreso de copiado en la memoria. Al finalizar, aparece información acerca del nombre del programa, cuándo fue compilado, identificadores...

Si se vuelve a pulsar `ENTER`, se ejecuta un comando para reiniciar el mote, y hacer que el nuevo programa sea el último en grabarse en memoria. En este caso debería parpadear el LED2 (azul).

COMANDOS TOS-DELUGE

La utilidad `tos-deluge` permite realizar operaciones sobre la memoria externa del mote. La forma general de usarlo (a través de Cygwin) es:

```
tos-deluge <fuente_datos> <comando> numero_imagen <opciones>
```

En `fuente_datos` se especifica cómo se accede al sensor: por puerto serie (escribiendo `serial@<puerto>:<velocidad>`) o una plataforma de programación Ethernet Crossbow MIB600 (`network@<equipo>:<puerto>`).

En comando se especifica qué operación realizar:

- `-p` (`--ping`). Muestra el estado de una imagen almacenada en flash.
- `-i` (`--inject`). Guarda la imagen de una aplicación TinyOS compilada. En el campo `opciones` puede especificarse la ruta de fichero.
- `-e` (`--erase`). Borra una imagen de la memoria.
- `-b` (`--boot`). Fuerza el reinicio en el mote.
- `-r` (`--reprogram`). Reprograma un mote.

- `-d` (`--disseminate`). Disemina una imagen guardada en la memoria a toda la red inalámbrica.
- `-dr` (`--disseminate-and-reprogram`). Diseminación y posterior reprogramación.
- `-s` (`--stop`). Detiene la diseminación de la imagen.
- `-ls` (`--local-stop`). Detiene la diseminación pero solo en el mote local.

En `image_number` se indica con un entero sobre qué imagen (en memoria) se está realizando una operación (borrado, programado...).

* * *

En este punto, ya se puede pasar a reprogramar toda una red de sensores usando `DelugeC`. Se tienen dos tipos de motes:

- *Basestation*. Es el nodo base. Puede recibir las imágenes de las aplicaciones compiladas a través del puerto serie para después diseminarlas al resto de nodos. Para indicar que un mote debe funcionar como base, hay que especificar la bandera `DELUGE_BASESTATION`.
- *Client motes*. El resto de los nodos de la red. Éstos reciben el código compilado desde el nodo base, quedando a la espera de ejecutarlo. No es necesario especificar ninguna bandera, aunque si se necesita comprobar el estado del mote (`--ping`) hay que añadir `DELUGE_LIGHT_BASESTATION`.

A la hora de programar, sólo hay que realizar algunos cambios en el código para añadir la funcionalidad de Deluge T2. En el componente de configuración (según la nomenclatura de archivos comentada en 3.5 terminaría en `*AppC.nc`) hay que incluir el componente `DelugeC`, de la siguiente forma:

```
components DelugeC;
```

Asimismo, en la parte de cableado o *wiring*, hay que enlazarlo con la interfaz `Leds` (ya que utiliza éstos para indicar la operación que está realizando sobre la memoria del mote):

```
DelugeC.Leds -> LedsC;
```

Otro fichero a modificar será `Makefile`, con la siguiente línea

```
BOOTLOADER=tosboot
```

Con lo que se indica que el programa de arranque es ahora `tosboot` (ha tenido que ser compilado previamente).

Además, en este tipo de aplicaciones es necesario incluir un fichero XML[30], definiendo las zonas de la memoria flash externa del mote (ST M25P80) en las cuales se almacena el código. Un ejemplo de configuración es

```
<volume_table>
<volume name="GOLDENIMAGE" size="65536" base="983040" />
<volume name="DELUGE1" size="65536"/>
<volume name="DELUGE2" size="65536"/>
<volume name="DELUGE3" size="65536"/>
</volume_table>
```

Como se observa, hay que especificar el nombre del volumen, el tamaño de dicho volumen (en Byte) y como valor opcional, a partir de qué posición de memoria se almacena. En el caso concreto de Tmote Sky, el tamaño del volumen habrá de ser múltiplo de 64 kB . Más información sobre el chip de memoria en 4.9 y en (STMicroelectronics, 2002).

El archivo (para este modelo de sensor) se denomina `volumes-stm25p.xml`.

A continuación se muestran los pasos para actualizar una aplicación en todos los nodos de la red de forma automática.

Primeramente se carga la imagen del programa `Blink` en cada uno de los nodos, indicando que se trata de nodos cliente:

```
CFLAGS+=-DDELUGE_LIGHT_BASESTATION make telosb install,<ID_nodo>
  bsl,<Puerto_COM-1>
```

En el parámetro `ID_nodo` se fija un identificador único para la red (por ejemplo 1, 2, 3...).

Para el nodo base, se instala la misma imagen como sigue

[30] *EXtensible Markup Language, metalenguaje extensible usado como estándar para el intercambio de información estructurada.*

```
CFLAGS+=-DDELUGE_BASESTATION make telosb install,<ID_nodo>
  bsl,<Puerto_COM-1>
```

El `ID_nodo` deberá ser distinto a los anteriores.

En este momento, todos los nodos de la red deberían estar ejecutando la misma versión de la aplicación `Blink` (parpadeo del LED0).

Ahora (solamente) se compila `Blink` pero es el LED2 el que parpadeará:

```
CFLAGS+=-DBLINK_REVERSE\ -DDELUGE_BASESTATION make telosb
```

Y con ayuda de `tos-deluge` se graba en el bloque 1 de memoria del nodo base:

```
tos-deluge serial@COM<Puerto_COM>:115200 -i 1
  build/telosb/tos_image.xml
```

Empezará el proceso de volcado de la imagen desde el ordenador hasta el mote (en pantalla se muestra información acerca del proceso). Cuando finalice, el nodo base tendrá (en la imagen número 1) la nueva versión del programa, si bien seguirá ejecutando la versión previa.

Para empezar a diseminar el programa a todos los nodos de la red se usa

```
tos-deluge serial@COM<Puerto_COM>:115200 -dr 1
```

Los LED de los nodos cliente parpadearán indicando que están recibiendo el nuevo programa. Después, se reiniciarán y arrancarán con el nuevo programa.

JUSTIFICACIÓN DE SU NO USO

La razón principal para no usar este protocolo en el trabajo es su consumo de memoria de programa. Por ejemplo, en el código de ejemplo anterior (`Blink`) si no se usa el componente `DelugeC` la aplicación compilada ocupa unos $2520\,\mathrm{B}$ en memoria ROM. Al añadirle la funcionalidad Deluge T2 pasa a ocupar $32834\,\mathrm{B}$.

La memoria principal del Tmote Sky es de $48\,\mathrm{kB}$. Teniendo en cuenta que la aplicación desarrollada en el trabajo está próxima a este valor máximo sin utilizar Deluge T2, la inclusión de este protocolo haría que se excedieran los límites de dicha memoria.

3.12 Comunicación serie

Se trata de establecer una conexión entre un nodo y un ordenador o PC. Esto permite la realización de un gran número de tareas adicionales: recibir y procesar todos los datos que se generan en una red, monitorizar el tráfico que se produce en la misma, o enviar mensajes de control hacia los motes desde el PC.

El envío de datos a través del puerto serie del mote es muy similar a la comunicación radio vista en 3.10. Para hacerlo más sencillo, los nombres de los componentes son iguales que para la radio, pero con la palabra `Serial` para distinguirlos. Los más importantes:

- `SerialAMSenderC`. Análogo al componente `AMSenderC`. Interfaces para el envío de mensajes a través de un puerto serie: `Packet`, `AMPacket`, `AMSend` y `PacketAcknowledgements`. Este componente está parametrizado: cuando se declare su uso hay que especificar un valor de identificación único.
- `SerialAMReceiverC`. Funcionalidad complementaria. Interfaces para recibir mensajes serie: `Packet`, `AMPacket` y `Receive`. Este componente también está parametrizado.
- `SerialActiveMessageC`. El componente más general para la transmisión/recepción de mensajes vía serie. Contiene la mayor parte de interfaces vistas para comunicación.