Desarrollo y evaluación de la fragmentación IPv6 en redes 6LoWPAN

Máster en Software Libre
Trabajo Final - UOC

Antonio J. Cepero López cpro@uoc.edu
Director: Xavi Vilajosana Guillén xvilajosana@uoc.edu

Resumen: El proyecto OpenWSN es una implementación de código abierto de la pila de protocolos estándares del Internet de las Cosas, como 6LoWPAN, RPL y COAP, utilizada por industrias y centros de investigación de todo el mundo, para su uso con dispositivos inalámbricos. El objetivo de este proyecto es proporcionar una implementación de la fragmentación 6LowPAN, basada en el estándar RFC 4944, sobre la plataforma OpenWSN para evaluar el impacto de dicha fragmentación en el rendimiento de las redes de área personal sobre dispositivos inalámbricos de bajo consumo.

I - Introducción

El Internet de las cosas nos acerca a un mundo en el que los objetos cotidianos están interconectados, posibilitando la identificación y el control de esos objetos, incluso, por otros objetos. De este modo, si todo objeto, como libros, lámparas, medicinas, etc., puede ser identificado, en teoría, sabríamos la localización de dicho objeto y podríamos interactuar con el mismo. Dicha interacción se realizará a través de pequeños dispositivos integrados en el objeto, para lo que resulta muy interesante que los mismos estén conectados a Internet, permitiendo una comunicación directa.

No obstante, la red Internet, nacida como una pequeña red académica en los años 70, ha tenido un gran desarrollo incrementando sus funcionalidades y servicios, haciendo que los dispositivos tengan cada vez mayores requisitos de capacidad de procesamiento y energía. Esto hace que sólo sea viable el uso de IP en los dispositivos más potentes, lo que provoca la aparición de tecnologías de red inalámbrica empotradas propietarias que fragmentan el mercado.

En el año 2003 aparece la especificación IEEE 802.15.4 [1][2] que proporciona un estándar para dispositivos inalámbricos de área personal de bajo consumo (WPAN – Wireless Personal Area Network). Poco más tarde la ZigBee Alliance [3] aprueba una especificación (diciembre de 2004) para un conjunto de protocolos de alto nivel para el control ad-hoc de redes de estos dispositivos. Sin embargo, tanto esta tecnología como las propietarias tienen problemas para integrarse en Internet, por lo que el IETF propone un grupo de trabajo para el desarrollo de 6LoWPAN (IPv6

over Low power Wireless Personal Area Networks), que conduce al RFC 4919 [4] y RFC 4944 [5].

El estándar IEEE 802.15.4, aunque basa su definición de niveles en el estándar OSI y prevé la interacción con el resto, define la capa física y la subcapa de control de acceso al medio (MAC – Medium Access Control) sobre dispositivos WPAN, utilizadas para intercambiar información a corta distancia, donde la infraestructura es muy simple o inexistente, característica que permite soluciones pequeñas en tamaño, energéticamente eficientes y económicas.

El nivel físico (PHY - Physical) proporciona el servicio de transmisión propiamente dicho y la interfaz con la subcapa MAC. Contiene una entidad de gestión de nivel físico que proporciona acceso a todos los servicios de gestión y mantiene una base de datos de los objetos gestionados. Es decir, define las características físicas y funciones del enlace inalámbrico, permitiendo controlar el trasnceptor, junto con el control de consumo y la señal. Define un canal (tres en la revisión de 2006) de comunicación en la banda de 868MHz para Europa, diez canales (30 en la revisión de 2006) en la banda de 915MHz en Norte América y 16 canales, disponibles a nivel mundial, en la banda de 2450MHz.

La subcapa MAC permite la transmisión y recepción de tramas a través del canal físico y proporciona servicios para la aplicación de mecanismos de seguridad adecuados en las transmisiones. Entre sus características se encuentra el acceso al canal, la asociación y desasociación (de nodos) y la gestión de *beacons*, tramas de sincronización que envía periódicamente el coordinador de la red al resto de nodos para tener la red sincronizada y evitar colisiones.

Existen dos tipos de nodos, según su funcionalidad: de funcionalidad completa (FFD – Full Function Device) y los de funcionalidad reducida (RFD – Reduced Funtion Device). Los RFD son dispositivos muy sencillos, como interruptores o sensores pasivos, que no necesitan grandes recursos y tienen necesidades de comunicación limitadas; sólo se comunican con un único FFD. Los FFD, sin embargo, pueden tener tres modos de operación distintos: normal -como cualquier otro dispositivo-, coordinador -puede encaminar mensajes- o coordinador de la red de área personal (PAN) -si es responsable de toda la red y no sólo de su entorno-. Los dispositivos tienen direcciones únicas de 64b, aunque pueden utilizar también direcciones cortas de 16b, asignadas cuando se asocian con el coordinador.

La red puede configurarse según dos topologías, en función de los requerimientos de la aplicación: en estrella o punto a punto. La topología en estrella utiliza a un único nodo como controlador central, responsable de iniciar, liberar e intercambiar los mensajes dentro de la red. Como requerimiento, debe estar siempre activo por lo que tendrá un consumo energético superior al resto de dispositivos. Las redes punto a punto disponen también de un coordinador PAN pero todos los nodos pueden intercambiar información con el resto, si están en alcance, permitiendo patrones arbitrarios de conexionado como redes en malla. Este tipo de redes pueden ser ad-hoc autoorganizativas.

La pila del protocolo 6LoWPAN es idéntica a la pila del protocolo IP pero sólo soporta IPv6, incorporando una capa de adaptación para optimizar el uso de IPv6 sobre IEEE 802.15.4. Además, dado el tipo de red de bajo consumo, por razones de eficiencia, el protocolo más utilizado en la capa de transporte es UDP, que puede comprimirse utilizando el formato de la capa de adaptación. El

protocolo ICMPv6 se utiliza para la transmisión de mensajes de control.

Una red LoWPAN se compone de nodos, que pueden jugar el rol de dispositivos host o de encaminadores, pudiendo incluir encaminadores de frontera. Se identifican mediante la dirección del host en la red, compartiendo todos ellos el prefijo de red de 64b, distribuido por los encaminadores de frontera y los encaminadores, a través de la red. Para facilitar la operación, los nodos se registran en un encaminador de frontera, como parte de la operativa del descubrimiento de vecinos (ND – Neighbor Discovery) de IPv6. Su arquitectura queda definida por tres tipos de redes: sencilllas, extendidas y ad-hoc. Una red ad-hoc es aquella que no está conectada a Internet. La red sencilla se conecta a otra red IP a través de un encaminador de frontera LoWPAN (LoWPAN Edge Router). Una red extendida interconecta varias redes sencillas a través de sus encaminadores de frontera. Los encaminadores de frontera juegan un papel fundamental realizando la interconexión de las LoWPAN con otras redes, incluyendo conectividad IPv4. La misma red puede incorporar más de un encaminador de frontera si éstos están conectados al mismo enlace. Los nodos pueden pertenecer a más de una red simultáneamente (multihoming) existiendo tolerancia a fallos entre encaminadores de frontera y pudiendo modificarse la topología por causas como las condiciones del canal, aunque no exista desplazamiento físico.

El direccionamiento IP de 6LoWPAN, como ya se ha comentado, es el de IPv6 de 128b, formado por los 64b de la dirección de red compartida por todos los nodos y los 64b de la dirección de la capa de enlace del dispositivo, eliminando la necesidad de utilizar un protocolo de resolución de dirección (ARP). Adicionalmente, puede soportar direcciones cortas, de 8 ó 16 bits. La característica más relevante se encuentra en la capa de adaptación, que permite la compresión [6] del encabezado IP y de otros protocolos, como UDP, junto con la fragmentación y el direccionamiento en malla.

La codificación de cabecera de compresión (LoWPAN_IPHC – IP Header Compression) utiliza 13b, que puede extenderse con un tercer octeto. Se compone de un valor de gestión que indica el tipo de cabecera (5b), seguido de un valor de compresión de cabecera IPv6 que identifica qué campos se comprimen, seguido de los valores IPv6 sin comprimir. Si el paquete contiene alguna otra cabecera, por ejemplo UDP, también pueden ser comprimidas utilizando compresión de la siguiente cabecera (LoWPAN_NHC – Next Header Compression). En el mejor de los casos, puede comprimir la cabecera IP a dos octetos, para comunicaciones locales, que junto a 4 octetos para la cabecera UDP, permite reducir 48 octetos a sólo 6.

La mayoría de las aplicaciones de los dispositivos que utilizan 6LoWPAN implican sistemas y redes autónomos, sin intervención humana. Esto implica una autoconfiguración por parte de los dispositivos pero también por parte de la red, implementando mecanismos de descubrimiento de vecinos. El descubrimiento de vecinos (Neighbor Discovery) es una característica de IPv6 para gestionar el arranque y el mantenimiento de nodos en enlaces IPv6. Sin embargo, este método no es el idóneo para una red 6LoWPAN por lo que se ha propuesto un método optimizado [7] para el descubrimiento de vecinos en redes 6LoWPAN que describe la autoconfiguración de la red y la operativa de los diferentes nodos.

Una aplicación común de las redes 6LoWPAN es la construcción de redes en malla que reducen el coste de infraestructura mientras extienden el área de cobertura. Para ello es preciso que los distintos nodos sean capaces de retransmitir la información entre los mismos; puede implementarse

de tres modos diferentes: mallado en la capa de enlace, en 6LoWPAN y a través de encaminamiento IP. Los dos primeros, denominados *mesh-under* son transparentes al protocolo IP, mientras que el tercero, denominado *route-over*, se basa en encaminamiento IP. Diferentes estudios [8][9] llegan a la conclusión que el encaminamiento IP es más robusto para la entrega de mensajes de fuente a destino, sin que ello suponga un mayor coste energético que si se utiliza la técnica de mallado. Para realizar el encaminamiento IP se utiliza el protocolo de encaminamiento para redes de bajo consumo y con pérdidas (Routing Protocol for Low-Power and Lossy Networks – RPL) [10], que permite almacenar la mayor parte, si no todo, de las rutas en uno o más encaminadores y reenviar los datagramas evitando mantener grandes tablas de encaminamiento en los encaminadores. Este protocolo es un protocolo vector distancia diseñado para aquellos casos en que los nodos pueden almacenar un número pequeño de rutas por defecto.

Hemos de tener en cuenta que, cuando conectamos una red LoWPAN a otra red IP o a Internet, 6LoWPAN posibilita IPv6 en los dispositivos simplificando y comprimiendo la cabecera IPv6, a la vez que se intenta minimizar lo posible el tamaño de los paquetes en la red; pero también que la especificación de IPv6 requiere soportar una MTU de 1280 octetos, algo imposible en redes IEEE 802.15.4. La solución que ofrece 6LoWPAN es un protocolo para la realización del fragmentado y reensamblado de paquetes, aunque ello suponga una penalización del rendimiento.

II - Fragmentación 6LoWPAN

El algoritmo que utiliza 6LoWPAN para la fragmentación es similar al utilizado en IPv4 pero, en lugar de incorporar un bit que indica que más fragmentos siguen al actual, indica el tamaño total del paquete fragmentado. Este tamaño sólo es obligatorio incluirlo en el primer fragmento pero, dado que los fragmentos pueden llegar en desorden, incluirlo en todos los fragmentos facilita la tarea del reensamblado al permitir reservar un buffer para alojar al paquete completo desde el momento en que cualquiera de ellos sea recibido, por lo que la especificación obliga a hacerlo.

La especificación de fragmentación, como sucede con el soporte para reenvío en mallado, se

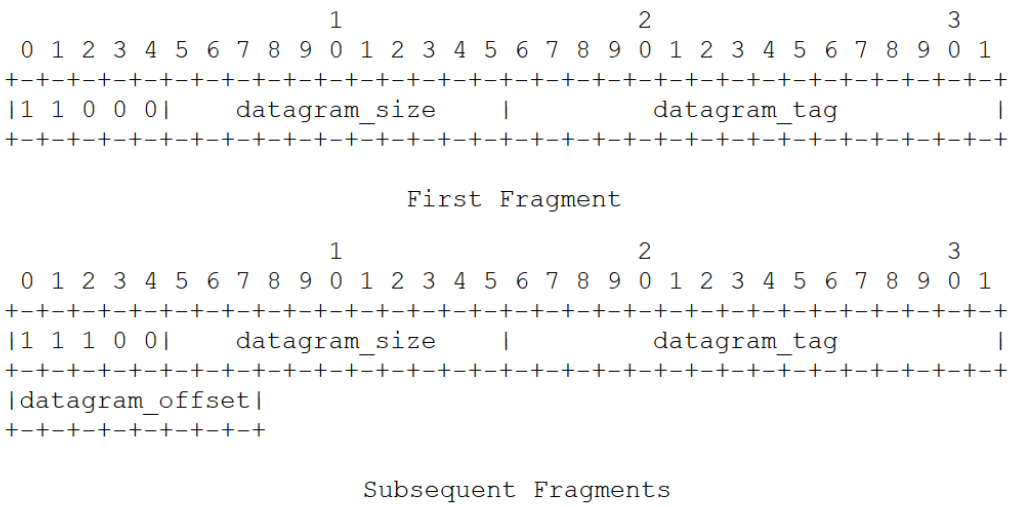

Ilustración 1: Cabeceras de fragmentación según RFC 4944

construye anteponiendo una cabecera 6LoWPAN opcional (LoWPAN_FRAG1 o LoWPAN_FRAGN) a la cabecera de compresión IP (LoWPAN_IPHC). Como en el caso de la cabecera de compresión IP, se define un valor de gestión de 5b pero éste es seguido por un valor de

tamaño del datagrama (*datagram_size*) de 11b; un valor de 11b permite hacer el reensamblado de mensajes de hasta 2047 octetos, suficiente para dar soporte a los 1280 octetos de la MTU de IPv6). A continuación, un campo etiqueta (*datagram_tag*) de 16b identifica el paquete al que pertenece el fragmento, ya que debe tener el mismo valor para todos los fragmentos del paquete; no tiene un valor inicial definido hasta la primera vez que se fragmenta un paquete, pero debe incrementarse para cada paquete que sea fragmentado, pasando a cero cuando alcance el valor 65536. Por último, se incorpora un campo de 8b de desplazamiento (*datagram_offset*) que nos indica la posición del fragmento en el paquete reensamblado, en unidades de 8 octetos, dando soporte a paquetes de hasta 2047 octetos; este campo no se especifica para el primer fragmento ya que el valor implícito del desplazamiento es cero.

El estándar proporciona también otras pautas a seguir con paquetes fragmentados. En primer lugar, indica que sólo es posible fragmentar paquetes cuya cabecera comprimida pueda incluirse en el primer fragmento o paquetes sin comprimir. Si se recibe un fragmento que se superpone en algún modo con otro fragmento del mismo paquete, debe desecharse la información del paquete que se había recibido hasta ahora y empezar de nuevo con el fragmento recién recibido. Si se disocia un enlace durante la transmisión de un paquete fragmentado, los nodos emisor y receptor deben descartar, respectivamente, los fragmentos pendientes de envio y la información del paquete que se había recibido. Análogamente, el nodo receptor debe programar un temporizador de 60 segundos cuando se recibe por primera vez un fragmento cualquiera de un paquete, desechándose la información de dicho paquete si no ha sido reesamblado por completo cuando el temporizador expira.

A continuación, detallo un algoritmo básico para la fragmentación y reensamblado de un paquete. En primer lugar necesito conocer el tamaño que ocupa la cabecera (*header_size*) 6LowPAN, incluyendo todos los valores LoWPAN_IPHC y LoWPAN_NHC, y el *payload* (*data_size*) del paquete. A continuación, debo consultar cuál será el tamaño del *payload* de la trama para este paquete y si contiene otros encabezados 6LoWPAN (redireccionamiento por malla y/o difusión). Al tamaño del *payload* de la trama le restaré el de todas las cabeceras 6LoWPAN y el *payload* del paquete; si el tamaño obtenido es mayor o igual a cero, no necesito comprimir. Si el valor que obtengo es negativo, sumaré al mismo el tamaño del *payload* del paquete y restaré 4, valor correspondiente a los octetos que ocupa la cabecera de fragmentación del primer fragmento (LoWPAN_FRAG1). Si el resultado es mayor que cero, puedo proceder a la fragmentación del paquete con cabeceras comprimidas; en caso contrario, procederé a la fragmentación del paquete IPv6, sin comprimir.

Algoritmo de fragmentación

- Calculo el tamaño del primer fragmento restando al tamaño del *payload* de la trama el del tamaño de las cabeceras 6LoWPAN (*frame_size*), 4 y dividiendo todo por 8; redondeo al mayor entero inferior de dicho valor y lo multiplico por 8; de esta manera he obtenido (*datagram_frag1_size*) el mayor valor múltiplo de 8 que es inferior al tamaño máximo que puedo almacenar en el fragmento.
- Selecciono *datagram_frag1_size* octetos del paquete y los precedo de los cuatro octetos de la cabecera *LoWPAN_FRAG1*, para la que habré seleccionado un identificador

(*datagram_tag*) nuevo; a continuación procederé a añadir, en caso necesario, otras cabeceras 6LowPAN y enviarlo a la capa de enlace.

- Para el cálculo del tamaño del resto de fragmentos, calcularé, como antes, el mayor entero inferior a la división por 8 de la resta *frame_size – 5*, que multiplicaré por 8 (*datagram_fragn_size*).
- Fijo un valor *datagram_position* a *datagram_frag1_size*, que indica el punto del paquete a partir del cual debo construir el siguiente fragmento.
- Mientras *frame_size – datagram_position > datagram_fragn_size – 5*, repito el proceso de
  - enviar *datagram_fragn_size* octetos, precedidos de los 5 octetos de la cabecera *LoWPAN_FRAGN* (y otras cabeceras 6LoWPAN que deban añadirse) en la que el valor *datagram_offset* será igual a *datagram_position* dividido por 8
  - actualizo el valor de *datagram_position* sumándole *datagram_fragn_size*.
- Él último fragmento se construye utilizando los octetos restantes del paquete de manera que el valor *datagram_offset* es igual a *datagram_position* dividido por 8.

Algoritmo de reensamblado

- Construyo un registro con los valores *datagram_size*, *datagram_tag* y las direcciones de origen y destino; serán las direcciones IEEE 802.15.4 de emisor y receptor o las de origen y destino final si se ha definido reenvío en malla en la cabecera 6LowPAN.
- Reservo un buffer de tamaño *datagram_size* para almacenar el paquete, creo una lista de fragmentos recibidos y activo un temporizador asociado al mismo, si es el primer fragmento recibido para ese paquete.
- Si es un fragmento del tipo LoWPAN_FRAG1 descarto los cuatro octetos de la cabecera y fijo el valor *datagram_offset* a cero; si es del tipo LoWPAN_FRAGN, fijo el valor *datagram_offset* al indicado por la cabecera y descarto cinco octetos.
- Compruebo que el tamaño de dicho fragmento es múltiplo de 8 o que se corresponde con el último fragmento del paquete; si no se cumple alguna de las dos condiciones se descarta el paquete.
- Si el fragmento ya ha sido recibido con anterioridad, se descarta el fragmento.
- Si el fragmento se solapa con otro recibido con antelación, descarto los contenidos recibidos hasta ahora y considero éste como el primer fragmento recibido, reiniciando el temporizador.
- Añado la información de este fragmento a la lista de recibidos.
- Copio los contenidos del *payload* al buffer que almacena el paquete a partir de la posición *datagram_offset * 8*.
- Si se han recibido todos los fragmentos, el paquete está completo: procesar la cabecera 6LoWPAN descomprimiendo, si es necesario y entregar a la capa de transporte.

III - Integración de la fragmentación en OpenWSN

A. Organización y estructura de OpenWSN

La pila de protocolos OpenWSN [11] se basa en la organización de los mismos como componentes funcionales independientes que utilizan funciones *Send* y *Receive* para enviar los mensajes, respectivamente, hacia abajo y hacia arriba en la pila. Todos los datos de un mensaje se organizan dentro del componente *OpenQueue*, que define registros *OpenQueueEntry_t* conteniendo tanto el mensaje como metadatos; es a través de estos metadatos como los diferentes componentes de la pila se comunican entre ellos, en lugar de utilizar parámetros y evitando así también la copia de datos entre capas. Si no se ha producido ningún error, cuando la función *Send* llega al final de la pila, en lugar de esperar, entrega el mensaje para que éste sea transferido por la red; por ello, los distintos componentes utilizan también una función *SendDone* que indica si el mensaje efectivamente ha sido transmitido o ha habido un error durante la transmisión.

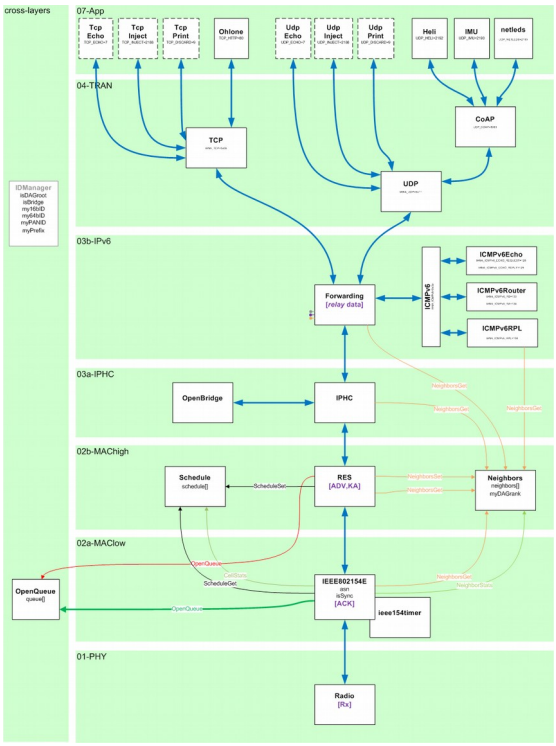

Ilustración 2: Arquitectura OpenWSN

La introducción de la fragmentación debe realizarse en 03a-IPHC, entre los componentes IPHC y RES de la figura anterior, dividiendo los mensajes en otros más pequeños e incluyendo la cabecera de fragmentación correspondiente cuando el mensaje a enviar sea demasiado grande. Del mismo modo, para los mensajes recibidos, los diferentes fragmentos que se reciban deberán ser ensamblados antes de que sean procesados por IPHC. De este modo, para la integración, necesitaríamos bien incorporar directamente el código en IPHC haciendo la comprobación correspondiente y actuando como sea preciso en las mismas funciones de interfaz de IPHC o definir un nuevo componente que se encontrará entre IPHC y RES.

Por su propia naturaleza -nodos que se comportan como sensores o actuadores-, las LoWPAN van a tender a la utilización de mensajes de pequeño tamaño, por lo que dotar a toda la infraestructura del soporte necesario para albergar mensajes de grandes dimensiones en una plataforma que no utiliza memoria dinámica supone una sobrecarga innecesaria. Además, teniendo en cuenta que una gran cantidad de mensajes van a ser reenviados entre los nodos de la red y este reenvío se realiza en la capa tres, se hace imprescindible ensamblar para analizar los mensajes y volver a fragmentar para reenviar. Por otro lado, el componente *OpenBridge*, que posibilita la comunicación entre la LoWPAN y una red externa, envía y recibe los mensajes sin procesar, llevando a cabo dicho procesamiento en una parte de *OpenWSN* denominada *OpenVisualizer* -un programa desarrollado en Python que se ejecuta en un PC-, por lo que también el ensamblado/fragmentación se realizará en *OpenVisualizer.* A propuesta del director de este TFM, Xavi Vilajosana, se va a introducir una solución que se basa en reducir al mínimo dicha sobrecarga, tratando de evitar en todo momento el reensamblado.

Para ello debemos, en primer lugar, analizar el camino que sigue un mensaje en la implementación actual de OpenWSN para determinar qué puntos del código deben ser tenidos en cuenta para la fragmentación. Nos centramos en la capa 3 y en qué partes del código interaccionan con las capas 2 y 4.

Supongamos que queremos enviar un mensaje. Desde la capa de aplicación, éste irá desplazándose en la pila hasta la capa de transporte, que lo encapsulará en el datagrama correspondiente. A continuación, el control del mensaje pasará al componente *forwarding* mediante la función *forwarding_send*; esta función añadirá la cabecera IPv6 y lo enviará a *forwarding_send_internal_RoutingTable* para que determine cuál es el nodo siguiente al que debe enviar el mensaje, según la tabla de encaminamiento. Una vez determinado se cede el control a *iphc_sendFromForwarding* que comprime la cabecera IPv6 en una IPHC y solicita la construcción de la trama a *sixtop_send* el cual invoca a *sixtop_send_internal*, que deja la trama preparada para su envío a través de la radio.

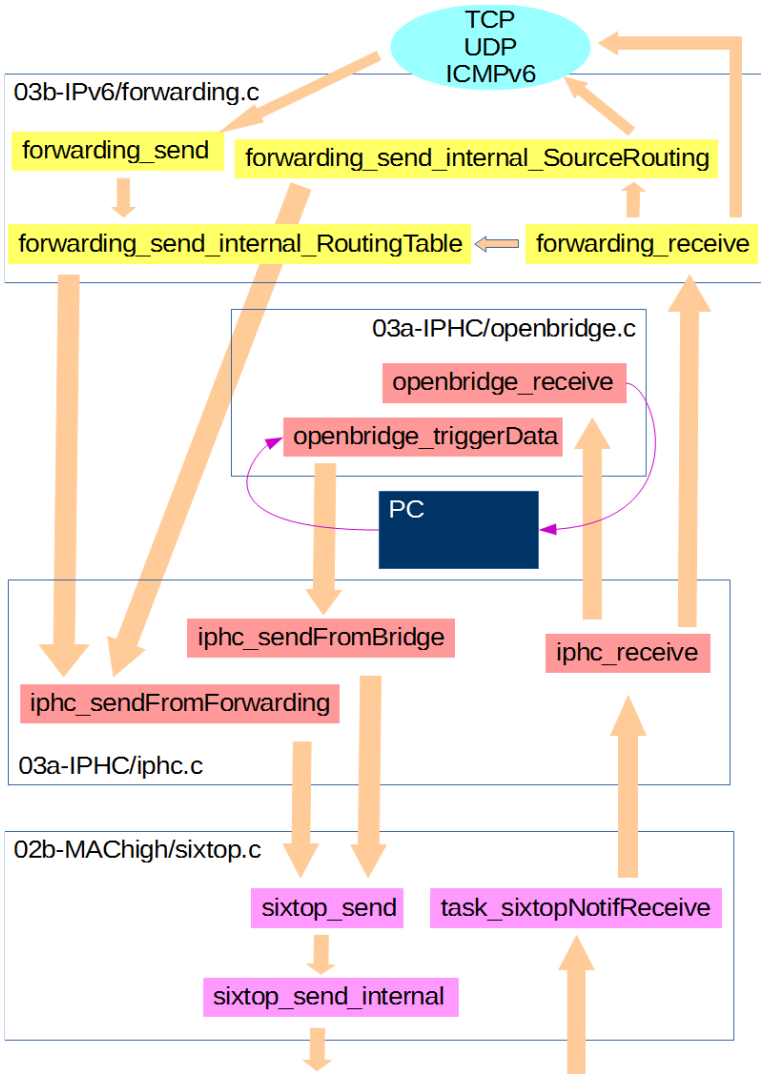

Ilustración 3: Camino seguido por un paquete en OpenWSN

Cuando un mensaje es recibido se elimina la información relativa a la capa MAC y *task_sixtopNotifReceive* cede el control del mismo a *iphc_receive* que, según el nodo sea un encaminador de frontera o no, lo reenviará a *openbridge_receive* o descomprimirá la IPHC para comunicárselo a *forwarding_receive*. Ésta puede determinar que el mensaje es para la mota actual, entregándolo a la capa de transporte o que debe ser reenviado, según incorpore *sourceRouting* o no.

Un caso particular es el del encaminador de frontera ya que los mensajes son procesados por *OpenVisualizer*. La configuración típica es una mota conectada al PC a través del puerto USB de manera que la comunicación entre la mota y el PC se realiza por el puerto serie. La implementación de la subcapa LoWPAN y las capas superiores se realiza dentro de *OpenVisualizer*, aunque sigue una mecánica similar. Como se puede ver en la figura, cuando se envia un mensaje, se dispara la función *openbridge_triggerData* que recibe el mensaje a través del puerto serie y lo reenvia a *sixtop_send* vía *iphc_sendFromBridge*, aunque esta

función no realiza actividad alguna sobre el mensaje. Cuando el mensaje es recibido, análogamente, es tratado por *openbridge_receive* para enviarlo a *OpenVisualizer* a través del puerto USB.

B. Integración de la fragmentación

Para los mensajes, enviados o recibidos por la mota, habría que disponer de buffers de al menos 1280B, en función de la cantidad de memoria que tenga el dispositivo. En estos momentos, la estructura *OpenQueueEntry_t* contiene metadatos y un buffer de 127B que almacena el mensaje a través de la pila, manteniendo un puntero *payload* al inicio del mismo para ir gestionando dónde empiezan los datos o cabecera actuales. Para este caso, la idea sería hacer transparente la dirección de buffer visible a la capa superior, apuntando al buffer del paquete de 127B (como ahora) o a uno de 1280B, según sea éste y tratando de igual forma el tamaño del mismo. Presuponemos que el número de mensajes que van a necesitar un buffer de grandes dimensiones va a ser pequeño por lo que se propone crear una estructura simple que almacene unos pocos bloques de gran tamaño y gestionarlo para que los metadatos trabajen con el interno, de 127B o uno externo, de 1280B.

Para enviar mensajes, el proceso general consistiría en, una vez generada la cabecera comprimida en *iphc_sendFromForwarding*, comprobar si el paquete debe y puede ser fragmentado: si no necesita ser fragmentado se envía a *sixtop_send*; en caso contrario, se redirecciona a una función encargada de anteponer la cabecera de fragmentación que, en realidad, generaría los paquetes necesarios para enviar los diferentes fragmentos a través de *sixtop_send*. En el caso de *iphc_sendFromBridge* no resulta necesario dado que los mensajes ya estarían fragmentados por *OpenVisualizer.*

En nuestra implementación, sin embargo, no vamos a seguir esta técnica porque al utilizar *OpenWSN* la estructura *OpenQueueEntry_t*, es un elemento de este tipo el que debemos pasar a la capa 2. Por ello, mantendremos el registro correspondiente al mensaje original y solicitaremos al sistema tantos registros como fragmentos debamos enviar. Si hiciéramos esto, no obstante, podría darse el caso en el que el sistema no dispusiera de registros suficientes por lo que aprovecharemos la circunstancia de que, cada vez que enviemos un mensaje, cuando éste haya sido enviado, se activará una notificación *sendDone* para indicarnos el éxito o fracaso en la transmisión. Capturando esta notificación podemos reutilizar el registro para el envío de otro fragmento. Así pues, dividimos la transmisión de fragmentos en dos procesos diferenciados: el primero genera la información necesaria para fragmentar (*fragment_preprendHeader*) y el segundo será el que construya efectivamente esos fragmentos (*fragment_tryToSend*) para enviarlos a la capa de enlace.

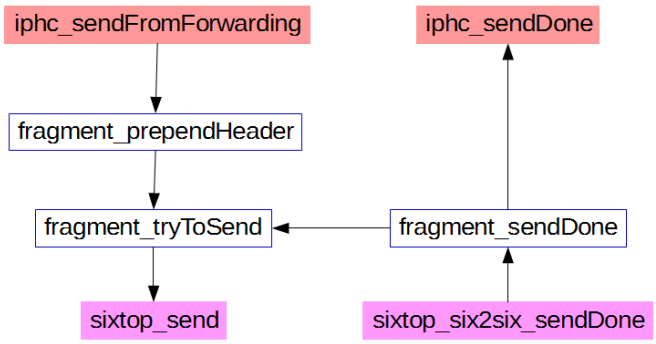

Ilustración 4: Camino para el envío de fragmentos en mi implementación

Cuando el fragmento es enviado, recibimos la notificación correspondiente. Si se ha producido un error, lo reenviaremos a la capa superior vía *iph_sendDone*. En caso contrario, si necesitamos seguir enviando fragmentos, lo haremos nuevamente a través de *fragment_tryToSend*; si no, notificaremos el éxito de la transmisión.

En la recepción, el proceso general consistiría en que, si lo que estábamos recibiendo es un mensaje fragmentado, reservaríamos -si correspondía con el primer fragmento que recibimos del mensaje al que pertenece- o utilizaríamos el bloque de memoria grande en el que ensamblaríamos el mensaje, copiaríamos el contenido del fragmento y liberaríamos el bloque. Una vez que todo estuviera ensamblado, lo pasaríamos a IPHC para que siguiera el curso normal de cualquier mensaje. En primer lugar, si el nodo que recibe el mensaje es un encaminador de frontera, *OpenBridge* espera recibir un mensaje que será procesado en *OpenVisualizer*, por lo que no resulta necesario el ensamblado pero además, nuestra implementación se ha propuesto minimizar en lo posible el ensamblado.

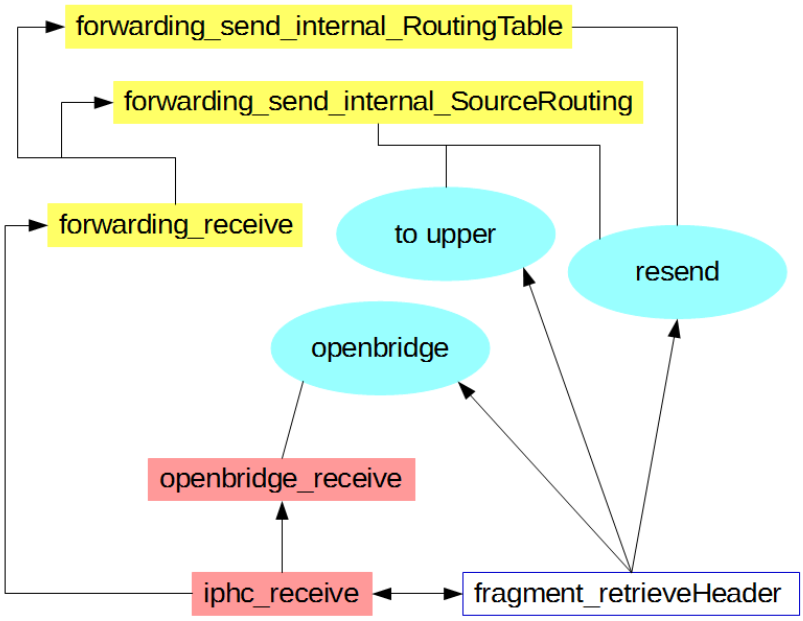

Ilustración 5: Camino para la rececpción de mensajes en mi implementación

Vamos a utilizar que la especificación obliga a que toda la IPHC debe encontrarse en el primer fragmento para evitar el ensamblado; de esta manera, toda la información sobre qué hacer con el mensaje se encuentra en el primer fragmento y en consecuencia, sólo deberemos promocionar a través de la capa de red dicho primer fragmento que, en el peor de los casos, podría ser el último recibido pero que, en general, no será así. A partir de ahí, haremos un seguimiento del mismo para que, en el momento del código en que se toma la decisión sobre qué hacer, en lugar de hacerlo, realizaremos esa operación para todos los fragmentos de dicho mensaje, salvo que deba ir a la capa superior, en cuyo caso lo ensamblaremos. Esta operación se puede realizar estableciendo alguna función como las representadas en la figura con fondo azul, que se deberán ejecutar tanto para los fragmentos recibidos antes que el inicial como para los que lleguen posteriormente, para el mismo mensaje.

IV - Implementación

En la implementación he creado un nuevo módulo que he denominado COMPONENT_FRAGMENT [12], siguiendo el estilo de *OpenWSN*, para el que he creado dos ficheros fragment.h [13] y fragment.c [14], intentado restringir al máximo todo lo desarrollado a dichos ficheros, siempre que ha sido posible.

A. Soporte para mensajes de gran tamaño

Para proporcionar soporte he definido una estructura muy sencilla que sólo contiene un buffer del tamaño deseado que, aunque es posible redefinir, yo he ajustado al mínimo marcado para IPv6 [15], y un centinela que indique si está en uso o no [16]. A continuación, para asociar un buffer de gran tamaño al registro *OpenQueueEntry_t*, he definido una función openqueue_toBigPacket [17]. La

función se encarga de comprobar si existe algún buffer disponible, reservándolo si es el caso y copiar los contenidos del buffer de 127B al reservado.

La idea principal de esta función es que la asignación de memoria sea transparente a la aplicación. Dado que la memoria asignada no es dinámica, el método utilizado por *OpenWSN* para añadir datos o cabeceras consiste en utilizar un función *packetfunctions_reserveHeaderSize* [18] que yo he adaptado para que, primero, permita reservar un tamaño mayor que el soportado por la implementación original y segundo, que en caso necesario invoque a la función de conversión a paquete de gran tamaño.

Incorpora también un parámetro *start* que, aunque en este punto puede parecer innecesario, es utilizado durante el ensamblado para definir a partir de dónde debe copiar los datos del buffer actual al que acabamos de asignar.

B. Fragmentación: introducción

Tanto si estoy enviando un mensaje como si lo estoy recibiendo, necesito algún tipo de entidad que me permita gestionar la operación que realizo. En el primer caso, tendré que saber qué fragmentos he enviado y cuáles faltar por enviar; en el segundo, necesito saber qué fragmentos he recibido y cuáles he ensamblado o reenviado. Análogamente, tendré que almacenar la información que indentifica al mensaje de manera unívoca, que según la RFC 4944 es la composición de la etiqueta y el tamaño del mensaje, con las direcciones de origen y destino en la capa de enlace.

Para la gestión de toda esta información, en línea con lo que hace el elemento *OpenQueue*, he definido un registo FragmentQueueEntry_t [19] que contiene los elementos básicos requeridos por la especificación como el tamaño del mensaje, la etiqueta, la dirección de origen, la dirección de destino y una referencia a un temporizador que permitirá determinar la recepción de un mensaje en el tiempo requerido. Incorpora también una referencia al mensaje que estamos enviando o recibiendo y una lista que va a almacenar la información [20] -tamaño y offset- de los fragmentos que vamos recibiendo para controlar que no haya duplicados u otros que se superponen a alguno que ya hubiéramos recibido.

C. Envío de un mensaje: fragmentación

El algoritmo básico de la RFC 4944 consiste en generar las tramas correspondientes a los fragmentos copiando en las mismas los contenidos correspondientes desde el mensaje original. El principal problema reside en que podemos agotar los recursos disponibles (elementos OpenQueueEntry_t) antes de terminar la transmisión, lo que generaría un error y el mensaje no podría marcarse como enviado.

La solución implementada consiste en aprovechar la lista [20] que contiene la información de los fragmentos recibidos para construir un lista con la información de los fragmentos a enviar, pero sin enviarlos. Dicha lista se construye en fragment_prependHeader [21]. Esta función solicita información sobre qué tamaño va a tener la cabecera de la capa de enlace para comprobar si el mensaje cabrá completo en una única trama; si cabe se envía a *sixtop_send* y si no, comienza el proceso de fragmentación.

No profundizo aquí en el algoritmo seguido en la implementación porque es básicamente el

explicado en la sección II, a excepción de que no se copia ningún dato ni se transmite ninguna trama, sino que tan sólo se genera la información correspondiente a los fragmentos resultantes del mensaje y se almacena en la lista. Por último, esta función invoca a aquella que intentará la transmisión de los fragmentos y que explico en el apartado E.

D. Recepción de un mensaje

Desde *task_sixtopNotifReceive* el mensaje llega a fragment_retrieveHeader [22] que comprueba si el paquete recién recibido se corresponde con un fragmento 6LoWPAN y en caso de no ser así lo cede a *iphc_receive* para que éste lo procese en la forma habitual.

El algoritmo es similar al explicado en la sección II, aunque cambia un poco el orden en que se realizan las tareas. En primer lugar, solicitamos un registro para contener el mensaje al que corresponde este fragmento; eso se hace en la función fragment_searchBuffer [23]. Esta función localiza si ya existe un registro para dicho mensaje comprobando su tamaño, etiqueta y direcciones de origen y destino; en caso de no encontrarlo, solicita uno libre y asigna al mismo los valores de identificación del mensaje.

A continuación, inicia el proceso de comprobación que asegura que no hay errores y el mensaje puede procesarse: es múltiplo de 8 o es el último fragmento del mensaje, no está duplicado y no se superpone a ningún otro fragmento que ya hubiéramos recibido. En el último caso, si se da, se elimina toda la información recibida hasta ahora, a excepción de los datos que identifican al mensaje y se trata a este fragmento como si fuera el primero que recibimos del mensaje.

Elimino la cabecera de fragmentación y almaceno el fragmento para su tratamiento posterior. Si es el primer fragmento que recibo del mensaje solicito la activación de un temporizador de sesenta segundos pero si, por el contrario, es el último que debo recibir, lo desactivo. Para finalizar, si este fragmento es el primero -FRAG1-, lo cedo a *iphc_receive* para que determine cuál es el destino de este mensaje. En caso contrario, si no es el primero y si ya sé lo que debe suceder con el mensaje, procedo a procesar el fragmento.

D1. Mecanismo genérico

Para determinar que tipo de procesamiento deben seguir los fragmentos de un mensaje he definido una serie de acciones [24] que son asignadas al campo ***action*** del registro [19]. De este modo, cuando se determina la operación que debe realizarse con el mensaje que está siendo analizado por IPHC, se comprueba si es un mensaje fragmentado para, en lugar de hacerlo, asignar dicha acción.

La primera posibilidad que analiza IPHC es que el nodo sea un encaminador de frontera, en cuyo caso cede el control a *OpenBridge* [25]. Si se determina que el mensaje es para la mota, éste debe ensamblarse para cederlo a ICMP o a la capa de transporte [26][27]; cuando se realiza esta operación la cabecera IPv6 ya ha sido analizada y eliminada del primer fragmento. Por último, puede ocurrir que el mensaje deba ser reenviado, por lo que sigue el mecanismo habitual de reconstrucción de la cabecera IPv6 y su compresión en IPHC, tomando el control del mismo justo antes de que fuera a ser entregado nuevamente al componente de fragmentación [28].

D2. Openbridge

Este es el caso más simple y se implementa en la función fragment_openbridge [29]. Su cometido

es el de recuperar la cabecera de fragmentación que había sido eliminada en la recepción del fragmento cederlo sin modificación a *OpenBridge* para que éste lo transfiera a *OpenVisualizer* a través del puerto serie.

D3. Ensamblado

La función fragment_assemble [30] es la encargada de este cometido. Ésta va a ser invocada cada vez que recibamos un fragmento de este tipo por lo que, para realizarlo, definimos un autómata que tendrá asociados una serie de estados [31] que van a ser aquellos en los que se pueda encontrar un fragmento en cada momento; para el caso que nos ocupa, estos pueden ser NONE -sin asignar-, RECEIVED -recibido y no procesado-, PROCESSED -contenidos copiados al mensaje- y FINISHED -liberada la estructura *OpenQueueEntry_t* que contenía el fragmento-.

La primera vez que es invocada esta función se hace para el fragmento inicial -FRAG1-, que ya no contiene la cabecera IPv6 por lo que existe un desplazamiento con respecto al tamaño actual de este fragmento y el tamaño total del mensaje, cosa que debe ser tenida en cuenta al realizar el reensamblado. Así pues, la primera operación consiste en calcular dicho desplazamiento y almacenarlo para que se tenido en cuenta con el resto de fragmentos.

De cara a minimizar el uso de bloques de gran tamaño, al existir un gran número de mensajes que deben ser fragmentados para su transmisión pero cuya información de capas 3 y superior tiene cabida en el buffer de 127B, se hace una comprobación sobre el tamaño del mensaje para reutilizar el actual, en caso de ser posible o reservar un buffer de gran tamaño. En cualquiera de los casos, este registro va a ser utilizado para pasar la información a la siguiente capa por lo que, aun cuando no ha sido liberado, se indica que su estado es FINISHED para que no sea procesado de nuevo.

Para el resto de fragmentos, cuando su estado es RECEIVED se procede a copiar la información en el buffer del mensaje y su estado cambia a PROCESSED. A continuación, se liberan los recursos y su estado cambia a FINISHED; en la práctica, el estado PROCESSED es un mero convencionalismo que indica si los recursos de ese fragmento ya han sido liberados o no. El estado FINISHED es utilizado para determinar si todos los fragmentos del mensaje han sido procesados y liberados, en cuyo caso se procede a liberar los recursos utilizado durante el ensamblado y comunicar la recepción del mensaje a la capa superior.

D4. Reenvío

De esta operación se encarga la función fragment_forward [32] que tiene un cometido especial puesto que, inicialmente, el tratamiento que reciben los fragmentos es el mismo que se proporciona a los mensajes que se deben reensamblar pero, una vez que se encuentran en el estado PROCESSED, en lugar de lugar de ser liberados sus recursos, son reutilizados para enviarlos de nuevo, como se explica en el apartado D.

En esta función estamos simulando el comportamiento que tendría un mensaje que es ensamblado, su cabecera IPv6 es eliminada creando una nueva y finalmente es fragmentado para enviarlo al nodo siguiente. La principal misión de esta función es generar una nueva cabecera de fragmentación para cada paquete recibido que debe ser reenviado. Para ello, genera una nueva etiqueta y un nuevo tamaño teniendo en cuenta el desplazamiento que se haya producido en la cabecera del mensaje. La función implementada se basa en que la diferencia entre la cabecera de llegada y la de salida será de

ocho octetos, debida a la eliminación del nodo actual en la cabecera de encaminamiento IP, por lo que el resto de condiciones -el tamaño es múltiplo de ocho e IPHC está contenida en el primer fragmento- no afectan a otros fragmentos.

Una vez procesado, dado que va a ser reenviado, su estado pasa a RESERVED y se invoca a la función encargada de la transmisión, como indicaba en D.

E. Transmisión de fragmentos

La transmisión es el proceso más complejo al deber tener en cuenta muchos y diferentes aspectos. La responsable principal es la función fragment_tryToSend [33] que junto con sus funciones auxiliares utiliza los siguientes estados del autómata: ASSIGNED -la función fragment_PrependHeader [21] ha creado la lista con la información necesaria para generar los fragmentos-, RESERVING -se están intentando asignar recursos a un fragmento para que pueda ser enviado-, RESERVED -los recursos necesarios para enviar el fragmento han sido asignados-, SENDING -el fragmento a sido entregado para que pueda ser enviado a través de la radio- y FINISHED -el fragmento puede haber sido enviado o no pero sus recursos ya han sido liberados-.

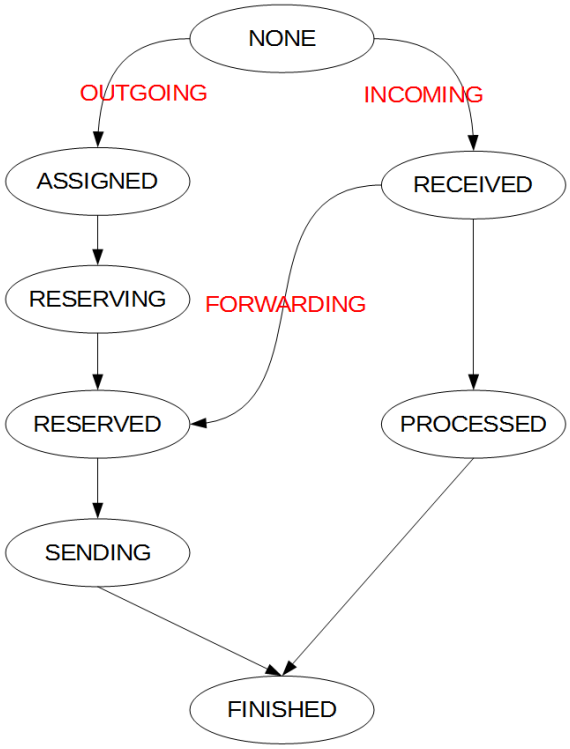

Ilustración 6: Diagrama de estados de los fragmentos en la implementación

Cuando el estado de un fragmento es ASSIGNED, la función fragment_reservePkt [34] se encarga de reservar los recursos necesarios para almacenar los contenidos del fragmento, copiándolos desde el mensaje y generando su cabecera. Si tiene éxito, llegará al estado RESERVED permitiendo que fragment_tryToSend [33] continúe. Una vez un fragmento se encuentra en el estado RESERVED, porque se está enviando o reenviando, invoca a la funcion *sixtop_send* para transmitirlo. Si tiene éxito, modifica su estado a SENDING y finaliza; en caso de que no tenga éxito finaliza el envío a través de fragment_finishPkt [35], que explico más adelante. Una circunstancia de esta implementación, que espera a que el mensaje haya sido enviado por la red para enviar el siguiente mensaje, es que podría llegar a suceder que no haya sido posible reservar los recursos necesarios para un fragmento por lo que no ha podido ser enviado y nos encontraríamos en la situación en la que no hemos terminado de enviar el mensaje pero esta función no volverá a ser invocada. Se realiza una comprobación al final del código para finalizar el envío si esta situación se produjera.

El siguiente paso para el fragmento será su transmisión por la red y la notificación del éxito o fracaso de la misma a través de fragment_sendDone [36]. Dado que la notificación es sobre un registro *OpenQueueEntry_t*, localizamos el mensaje al que pertenece y cedemos el control a fragment_finishPkt [35].

La función de finalización de fragmentos puede ser invocada si un fragmento ha sido transmitido, si se ha producido un error cuando se intentaba transmitir un fragmento o porque se desea dejar de transmitir un mensaje -como por ejemplo, un mensaje que está siendo retransmitido pero no ha sido recibido completo dentro de los sesenta segundos que marca la especificación-. Su operativa básica consiste en liberar los recursos del fragmento por el que ha sido invocada y comprobar las condiciones en que lo ha sido ora para notificar a las capas superiores el éxito o fracaso de la transmisión de un mensaje ora para invocar de nuevo a la función de transmisión [33] si el fragmento ha sido transmitido con éxito y aún quedan por transmitir otros fragmentos del mensaje al que pertenecía.

F. OpenVisualizer

Una parte muy importante de *OpenWSN* es el software que implementa la funcionalidad de encaminador de frontera y que, además, proporciona un simulador de redes 6LoWPAN con el que resulta posible verificar el software que se desea incorporar a la red, antes de hacerlo. *OpenVisualizer* está desarrollado en Python y utiliza la pila de *OpenWSN*, comunicándose con la misma a través del puerto serie. No es mi objetivo dar una visión completa del funcionamiento del mismo por lo que sólo indicaré brevemente la parte que compete a este trabajo para explicar cómo he integrado la fragmentación el mismo.

Dado su carácter, puede trabajar tanto con motas reales como simuladas; para cada una de ellas incorpora un objeto *MoteProbe* que gestiona los mensajes recibidos desde la mota. Éstos son comunicados a otro objeto *Parser* que, trabajando con diferentes procesadores de mensajes especializados, comunican la información recibida al objeto que deba tratar con ella. En nuestro caso, cuando se recibe un mensaje desde la red 6LoWPAN, se genera una señal *fromMote.data* que es capturada por el objeto *OpenLbr* equivalente componente IPHC de la pila *OpenWSN*. En sentido inverso, cuando un mensaje llega desde Internet hacia la red, el objeto *OpenLbr* comprime la cabecera IPv6 y genera una señal *bytesToMesh*, que será capturada por el objeto encargado de transferir el mensaje a *OpenBridge*.

F1. Fragment

La implementación de la fragmentación la he hecho incorporando un nuevo objeto Fragment [37], que captura la señal *fromMote.data* y la trata en el método _assemble_notif [38] que, como sucedía en *OpenWSN*, transfiere el mensaje si no es un fragmento, pero que, en este caso sí, realiza el ensamblado antes de transferirlo a *OpenLbr*. Por su parte *OpenLbr* captura ahora una señal *meshToV6*.

El envío hacia la red desde Internet también se ve modificado ya que *OpenLbr* genera ahora una señal *fragment* que es capturada por Fragment y tratada por _fragment_notif [39]. Como en *OpenWsn*, este método no fragmenta sino que genera una lista con la información de los diferentes fragmentos, generando finalmente una señal *fragsent* que será capturada por _fragsent_notif [40] que ensamblará un fragmento y lo transmitirá generando la señal *bytesToMesh*. En este caso también se espera la notificación de que el fragmento ha sido transmitido antes de enviar el siguiente.

F2. ParserBridge

En *OpenWSN* no se notifica el éxito o fracaso de la transmisión de un mensaje a través de *OpenBridge*. Ésto supone un problema para esta implementación, que intenta minimizar el uso de los recursos, ya que necesita saber cuándo se ha realizado la transmisión de un fragmento para enviar el siguiente o no continuar con los envíos si se ha producido un error. Análogamente, cuando se están recibiendo mensajes fragmentados, *OpenWSN* está utilizando los recursos de la mota para su gestión, por lo que sería redundante volver a hacer dicha gestión en *OpenVisualizer*; resulta más práctico poder comunicar que se ha producido un error para no continuar ensamblando un mensaje.

Para ello, he introducido un nuevo tipo de mensaje serie, el de *OpenBridge*, denominado SERFRAME_MOTE2PC_BRIDGE [41] y la función constructora de estos mensajes [42]. Si el mensaje se genera por la transmisión de un fragmento, la función que lo notifica es fragment_checkOpenBridge [43]; si se genera por un error en la recepción, lo notifica fragment_cancelToBridge [44].

En la parte de *OpenVisualizer*, el procesador generará dos tipos de señales *fromMote.fragsent* y *fromMote.fragabort* que serán capturadas, respectivamente, por _fragsent_notif_mote y _fragabort_notif [45], según pueda enviarse un nuevo fragmento o deba eliminarse un mensaje.

V - Evaluacion

Para evaluar mi implementación he realizado diferentes pruebas, desarrollando un programa que me ha permitido medir el tiempo empleado, en el simulador, para enviar y recibir mensajes entre el nodo encaminador de frontera y otro nodo de la red. Las pruebas se han realizado utilizando como modelo deLoWPAN un grafo acíclico con 3 nodos: el encaminador y dos motas.

Bytes	32	64	128	256	512	1024
Ida	49,57	140,02	140,37	292,57	499,33	992,27
Vuelta	80,25	149,98	152,51	305,28	492,01	950,37
Total	129,82	290,00	292,87	597,85	991,34	1942,64

Tabla 1: Mensajes entre la mota 2 y el DAGroot

Esta prueba permite medir y relacionar el tiempo empleado en la comunicación con un vecino directo y la comunicación con un nodo intermedio que debe reenviar los datos. Los tamaños elegidos para los mensajes están en función del número de fragmentos que utilizan, para tener una muestra lo más representativa posible. Así, los mensajes de 32B no necesitan ser fragmentados; en el caso de 64B y de 128B ambos se necesitan 2 fragmentos para su transmisión, lo que posibilita observar también el impacto del tamaño de los datos sobre el de los fragmentos; y el resto duplican al caso anterior utilizando diferentes cantidades de fragmentos: 4 para 256B, 6 para 512B y 12 para 1024B.

Bytes	32	64	128	256	512	1024
Ida	127,54	343,84	319,37	799,43	1290,55	2695,51
Vuelta	158,24	387,64	358,32	808,82	1338,40	2837,77
Total	285,78	731,47	677,70	1608,25	2628,95	5533,28

Tabla 2: Mensajes entre la mota 3 y el DAGroot

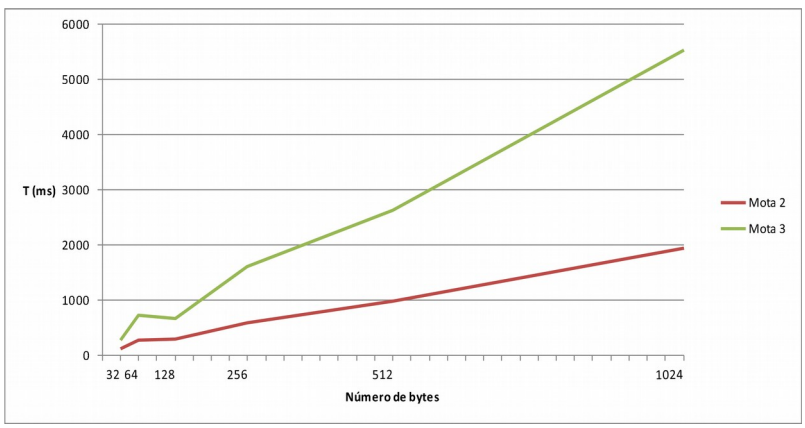

Gráfico 1: Tiempo de transmisión en función del tamaño del mensaje

En las tablas 1 y 2 podemos

apreciar los tiempos empleados, en milisegundos, para transferir diferentes mensajes cuyo *payload* es el mostrado en Bytes. Vemos claramente la sobrecarga impuesta por la fragmentación, dado que hemos de enviar diferentes mensajes con sus respectivas cabeceras. En el gráfico se observa cómo estos tiempos crecen especialmente en función del número de fragmentos que hay que enviar; por ejemplo, entre enviar 64B y 128B apenas hay diferencia debido a que ambos ocupan 2 fragmentos; sin embargo, en el resto de los casos, el mayor número de fragmentos aumenta los tiempos de transmisión de forma mayor que si sólo dependiera del tamaño de los datos.

El tiempo de la transmisión también aumenta en función del número de nodos. En condiciones normales -sin fragmentación-, este tiempo, debería ser el de transmisión a un nodo por el número de nodos de distancia más la latencia impuesta por los nodos intermedios, como sucede en el caso de los mensajes de 32B. Sin embargo, al fragmentar debemos añadir la latencia impuesta por el nodo intermedio para cada uno de los fragmentos por lo que, por ejemplo, en el caso de 1024B (12 fragmentos) el tiempo de transmisión a la mota 3 es más de dos veces y media el que toma para la mota 2 -que es el nodo intermedio-.

VI – Conclusiones

La sobrecarga de tiempo debida a la fragmentación es grande debido a que, además de los datos, debemos enviar las correspondientes cabeceras. No obstante, supone también una gran ventaja ya que permite establecer comunicaciones extremo a extremo siguiendo el estándar, de forma transparente a las aplicaciones -que no deben implementar paginación- y en definitiva, disponer de conectividad IP sin tener que descartar mensajes mayores a 125B. Esta sobrecarga, además, es perfectamente asumible en este tipo de redes de bajo consumo en las que el rendimiento es de 250kbps o menos y no hay demasiado tráfico.

Este trabajo de fin de master, además, me ha servido para desarrollar la implementación de un protocolo de red real, especificado como parte del estándar RFC 4944 y aportarlo a *OpenWSN*, un proyecto *open source* ampliamente utilizado, en el que todavía no estaba disponible.

Bibliografía

[1] IEEE, Institute of Electrical and Electronics Engineers, 2003. IEEE Standard for Information technology - Telecomunications and information exchange between systems - Local and metropolitan area networks - Specific requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low - RateWireless Personal Area Networks (WPANs). IEEE Standard 802.15.4 - 2003, Institute of Electrical and Electronics Engineers, New York.

http://standards.ieee.org/getieee802/download/802.15.4-2003.pdf

[2] IEEE, Institute of Electrical and Electronics Engineers, 2006. IEEE Standard for Information technology - Telecomunications and information exchange between systems - Local and metropolitan area networks - Specific requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low - RateWireless Personal Area Networks (WPANs). IEEE Standard 802.15.4 - 2006, Institute of Electrical and Electronics Engineers, New York.

http://standards.ieee.org/getieee802/download/802.15.4-2006.pdf

[3] Página informativa de la ZigBee Alliance. Recurso Web: http://www.zigbee.org/

[4] IPv6 sobre redes de área personal inalámbricas de bajo consumo (6LoWPANs): descripción general, consideraciones, planteamiento del problea y objetivos. Nandakishore Kushalnagar, Gabriel Montenegro, Christian Peter Pii Schumacher. 2007. IETF RFC 4919.

http://datatracker.ietf.org/doc/rfc4919/?include_text=1

[5] Transmisión de paquetes IPv6 sobre redes IEEE 802.15.4. Gabriel Montenegro, Nandakishore Kushalnagar, Jonathan W. Hui, David E. Culler. 2007. IETF RFC 4944 .

[6] Transmisión de paquetes IPv6 sobre redes IEEE 802.15.4. Jonathan W. Hui, Pascal Thubert. 2011. IETF RFC 6282.

[7] Optimización del descubrimiento de vecinos IPv6 sobre redes de área personal inalámbricas de bajo consumo (6LoWPANs). Zach Shelby, Samita Chakrabarti, Erik Nordmark, Carsten Bormann. 2012. IETF RFC 6775.

[8] Chowdhury A.H., Ikram M., Cha H.-S., Redwan H., Saif Shams S.M., Kim K.-H., Yoo S.-W. Route-Over *vs.* Mesh-Under Routing in 6LoWPAN. Proceedings of IWCMC '09, the 2009 International Conference on Wireless Communications and Mobile Computing: Connecting the World Wirelessly; Leipzig, Germany. Junio 2009.

[9] Forwarding Techniques for IP Fragmented Packets in a Real 6LoWPAN Network. Alessandro Ludovici, Anna Calveras, Jordi Casademont. Enero 2011. Wireless Network Group (WNG), Universitat Politècnica de Catalunya

[10] Un encabezado de ruta IPv6 para rutas del protocolo de encaminamiento para redes de bajo consumo y con pérdidas (RPL). Jonathan W. Hui, JP. Vasseur, David E. Culler, Vishwas Manral. 2012. IRTF RFC 6554

[11] OpenWSN: a Standards-Based Low-Power Wireless Development Environment. Watteyne, T., Vilajosana, X., Kerkez, B., Chraim, F., Weekly, K., Wang, Q., Glaser, S., and K. Pister. Transactions on Emerging Telecommunications Technologies - August 2012

http://onlinelibrary.wiley.com/doi/10.1002/ett.2558/abstract

Referencias al código. Repositorio GitHub.

[12] Declaración del componente: https://github.com/cprouoc/openwsn-fw/blob/develop/inc/opendefs.h#L150

[13] Fichero fragment.h: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/fragment.h

[14] Fichero fragment.c: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/fragment.c

[15] MTU IPv6: https://github.com/cprouoc/openwsn-fw/blob/develop/inc/opendefs.h#L45

[16] Registro para paquetes de gran tamaño: https://github.com/cprouoc/openwsn-

fw/blob/develop/openstack/cross-layers/openqueue.h#L28

[17] Función de conversión a paquete de gran tamaño: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/cross-layers/openqueue.c#L193

[18] Función para reserva de espacio: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/cross-layers/packetfunctions.c#L268

[19] Registro de gestión de fragmentación: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/fragment.h#L101

[20] Registro de información de un fragmento: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/fragment.h#L94

[21] Función de fragmentación: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/fragment.c#L130

[22] Función de recepción de mensajes: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/fragment.c#L226

[23] Función de búsqueda de mensajes fragmentados: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/fragment.c#L764

[24] Tipos de procesamiento de fragmentos: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/fragment.h#L51

[25] Asignación de acción para openbridge: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/iphc.c#L276

[26] Asignación de ensamblado por destinatario de mensaje: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03b-IPv6/forwarding.c#L251

[27] Asignación de ensamblado por último nodo de la lista de encaminamiento: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03b-IPv6/forwarding.c#L517

[28] Asignación de reenvío: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/iphc.c#L206

[29] Función de cesión a openbridge: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/fragment.c#L1138

[30] Función de ensamblado de fragmentos: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/fragment.c#L1021

[31] Estados de un fragmento: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/fragment.h#L27

[32] Función de reenvío de fragmentos: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/fragment.c#L1082

[33] Función de gestión del envío de fragmentos: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/fragment.c#L619

[34] Reserva de recursos para fragmentos: https://github.com/cprouoc/openwsn-

fw/blob/develop/openstack/03a-IPHC/fragment.c#L478

[35] Liberación de recursos de fragmentos salientes: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/fragment.c#L545

[36] Notificación de transmisión: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/fragment.c#L330

[37] Objeto para fragmentación en Python: https://github.com/cprouoc/openwsn-sw/blob/develop/software/openvisualizer/openvisualizer/openLbr/fragment.py

[38] Método para el ensamblado en Python: https://github.com/cprouoc/openwsn-sw/blob/develop/software/openvisualizer/openvisualizer/openLbr/fragment.py#L146

[39] Método para la fragmentación en Python: https://github.com/cprouoc/openwsn-sw/blob/develop/software/openvisualizer/openvisualizer/openLbr/fragment.py#L89

[40] Método para el envío de fragmentos en Python: https://github.com/cprouoc/openwsn-sw/blob/develop/software/openvisualizer/openvisualizer/openLbr/fragment.py#L205

[41] Formato de mensaje serie de OpenBridge: https://github.com/cprouoc/openwsn-fw/blob/develop/drivers/common/openserial.c#L190

[42] Función constructora de mensajes serie: https://github.com/cprouoc/openwsn-fw/blob/develop/drivers/common/openserial.c#L190

[43] Notificación por fragmento trasmitido a través de OpenBridge: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/fragment.c#L395

[44] Notificación de error en recepción a OpenVisualizer: https://github.com/cprouoc/openwsn-fw/blob/develop/openstack/03a-IPHC/fragment.c#L956

[45] Métodos de tratamiento de las notificaciones desde OpenBridge: https://github.com/cprouoc/openwsn-sw/blob/develop/software/openvisualizer/openvisualizer/openLbr/fragment.py#L243

Adicional

Enlace a los tutoriales de OpenWSN para instalación y puesta en marcha:

https://openwsn.atlassian.net/wiki/display/OW/Get+Started

Repositorios que contienen el código desarrollado para este trabajo, derivados de los de OpenWSN:

https://github.com/cprouoc/openwsn-fw.git

https://github.com/cprouoc/openwsn-sw.git