Universidad Politécnica de Cartagena

Escuela Técnica Superior de Ingeniería de Telecomunicación

Proyecto de fin de carrera:

Uso del protocolo CoAP para la implementación de una aplicación domótica con redes de sensores inalámbricas

Autor:
Jorge Castro Heredia.
Director de proyecto:
Fernando Losilla López.

Septiembre 2014

Autor	Jorge Castro Heredia
E-mail del Autor	**Israel1023@gmail.com**
Director	Fernando Losilla López
E-mail del Director	Fernando.losilla@upct.es
Codirector(es)	
Título del PFC	**Uso del protocolo CoAP para la implementación de una aplicación domótica con redes de sensores inalámbricas.**
Descriptores	

Resumen

El creciente uso del protocolo IPv6 está facilitando la adopción en un futuro del "Internet de las cosas", en el que objetos cotidianos son accesibles desde cualquier parte de Internet. Reciente se está trabajando en el desarrollo del protocolo CoAP *(Constrained Application Protocol)* para las comunicaciones máquina a máquina entre dispositivos limitados en recursos y direccionables mediante IP.

En el presente proyecto se pretende desarrollar una aplicación ejecutable en los nodos de una red de sensores inalámbrica (Wireless Sensor Network, WSN) que haga uso tanto de IPv6 como del protocolo CoAP. Al ser todos los nodos direccionables será posible acceder a ellos desde el exterior de la WSN donde se encuentran. Además se aprovechará este direccionamiento global de los nodos para que puedan comunicarse con nodos de otras WSN. Finalmente, en el proyecto también se usará el protocolo CoAP no sólo para las comunicaciones entre nodos, sino también para recibir información que permita configurar dinámicamente el funcionamiento de los nodos.

Titulación	I.T.T. Especialidad Telemática
Intensificación	
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha de Presentación	**Septiembre 2014**

INDICE

INTRODUCCION 3
Objetivos 3
1.- REDES DE SENSORES 4
1.1.- NODO SENSOR 4
1.2.- CARACTERISTICAS DE UNA WSN 5
1.3.- DESAFIOS DE UNA WSN 5
1.4.- HARDWARE DE UN SENSOR 5
1.5.- SISTEMAS OPERATIVOS PARA MOTES 6
1.6.- PROTOCOLOS DE ENRUTAMIENTO PARA WSN 6
1.7.- ESTADARES IEEE 6
1.7.1.- CARACTERISTICAS DEL ESTANDAR 802.15.4 7
1.8.- 6LoWPAN 8
1.8.1.- CARACTERISTICAS 6LoWPAN 8
1.8.2.- MODELO DE PILA 6LOWPAN 8
2.- INTRODUCCION A WEB SERVICES 9
2.1.- SOAP Web SERVICES 9
2.1.1.- CARACTERISTICAS SOAP 10
2.2.- RESTfull 10
2.2.1.- CARACTERISTICAS REST 11
2.3.- REST VS SOAP 11
3.- CoAP : PROTOCOLO DE APLICACION RESTRINGIDA 12
3.1.- CARACTERISTICAS CoAP 12
3.2.- MODELO DE MENSAJERÍA 13
3.3.- FORMATO DE UN MENSAJE CoAP 14
3.3.1.- OPCIONES CoAP 14
3.4.- COORDINACION PETICION / RESPUESTA 16
3.5.- CACHING 17
4.- CORE LINK FORMAT 17
4.1.- DESCUBRIMIENTO DE RECURSOS (well-known core) 18
4.2.- OBSERVACION DE RECURSOS 19
4.3.- CARACTERISTICAS DE UNA NOTIFICACIÓN 20
4.4.- POLLING 20
4.5.- LONG POLLING 20
5.- TINYOS OPERATING SYSTEMS 21
5.1.- CARACTERISTICAS DE TINYOS 21
5.2.- PROPIEDADES DE MEMORIA Y LLAMADAS SPLIT-PHASE 22
5.3.- LENGUAJE DE PROGRAMACIÓN NesC 23
5.3.1 INTERFACES 24
5.3.2.- MODULOS 24
5.3.3.- CONFIGURACIONES 25
5.4.- MODELO CONCURRENTE 25
5.5.- COMANDOS BÁSICOS PARA NESC 26
6.- IMPLEMENTACION DE CoAP EN TINYOS 27
6.1.- HARDWARE 27
6.2.- SOFTWARE 27
6.3.- PETICIONES / RESPUESTAS CoAP 28
7.- DESARROLLO DEL PROYECTO 30
7.1.- DECRIPCION Y FUNCIONAMIENTO DE COMPONENTES 32
7.2.- CONCLUSIONES Y LINEAS FUTURAS: 36

UNIVERSIDAD POLITECNICA DE CARTAGENA
Fechos Allend Mar

INTRODUCCION:

Las redes se sensores están teniendo un éxito en nuestra sociedad por ser básicamente un conjunto de dispositivos inteligentes que gestionan tareas como la recogida de información, temperatura, humedad y luminosidad, cuyas aplicaciones están presentes en monitorización de una casa, seguridad y salud.

Este proyecto utiliza sensores de bajo consumo (*TelosB*) con los que se pretende simular la gestión de una vivienda o edificio, saber qué habitaciones están encendidas, seguimiento de temperatura y luminosidad de cada habitación o piso en general y configuración de la misma por medio de ***umbrales*** *(acciones programadas tras medir ciertos valores de las lecturas de sensores)*. La comunicación de los nodos será a través de internet adaptando las órdenes del usuario al protocolo *CoAP* que es utilizado en estos dispositivos de baja potencia y de recursos limitados.

Imaginemos una aplicación en que una persona que esté en su casa pueda saber la temperatura en días calurosos o fríos como otro tipo de cosas, humedad, cantidad de luz que recibe una habitación o voltaje de las lámparas por medio de órdenes enviadas a sensores (*previamente instalados en casa*), ellos recogen datos y al gestionarlos se realizan acciones tales como encender la calefacción, lámparas o modificar la temperatura de una habitación, todo esto desde su PC. La comunicación de los sensores (*que actúan como nodos dentro de una red*) se programa para que puedan reaccionar ante determinados *eventos* para que así generen notificaciones a otros nodos o PCs realizando de esta manera acciones provocadas por las lecturas de sensores.

Objetivos:

- Aprendizaje del sistema operativo *TinyOS* para WSN.
- Implementación de una aplicación en los nodos sensores que permita la detección de *eventos* en dichos sensores como sobrepasar niveles de umbrales en lecturas recogidas, comunicaciones mediante el protocolo *CoAP o* reprogramación del comportamiento de los sensores.
- Aprendizaje del lenguaje de programación *nesC* para la programación de ***recursos*** *(funcionalidades que nos permiten recoger lecturas de los sensores)* como también el tipo de *programación concurrente* utilizada para el ahorro de energía.

1.- REDES DE SENSORES:

Una red de sensores es un conjunto de elementos autónomos *(nodos sensores)* interconectados de manera inalámbrica distribuidos alrededor de un objeto que colaboran con el objetivo de realizar una tarea en común como monitorización, capacidad de cómputo y almacenamiento de datos.

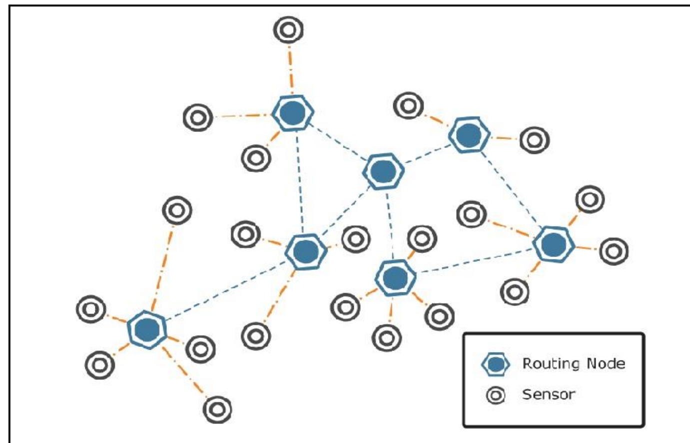


FIGURA 1: Ejemplo de una red de sensores.

1.1.- NODO SENSOR

Los *nodos sensores* son los que realizan las mediciones de luz, temperatura, humedad, etc. Para estas tareas se utiliza un componente *RADIO* que sirve para *enviar / recibir* mensajes y una CPU para su procesamiento.

Para afrontar la limitación energética y prologar el tiempo de vida de los nodos, la CPU se queda en *estado **"sleep"** (estado en que el dispositivo se mantiene inactivo para ahorrar de energía)* mientras no tenga nada que hacer, esto convierte a la *comunicación* como el primer consumidor de energía, a nivel software se aplican técnicas de *programación concurrente* que propician el ahorro de la batería.

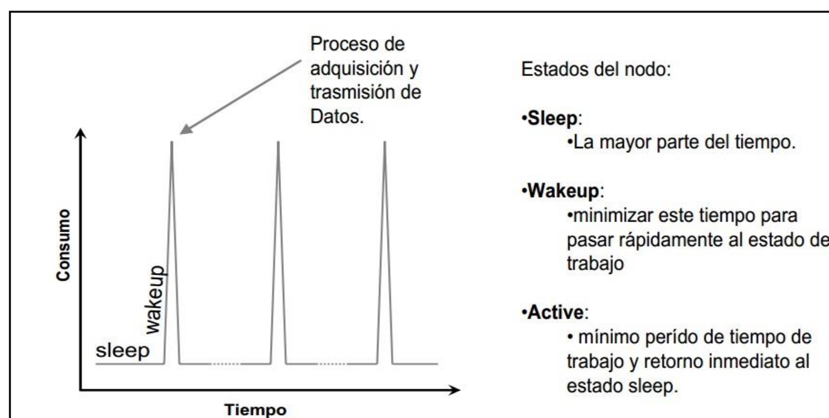


FIGURA 2: Gráfica del consumo de un sensor.

1.2.- CARACTERISTICAS DE UNA WSN

- Un nodo dispone de pocos recursos y es propenso a fallos pero logrando una *cooperación y coordinación* de todos los nodos se puede lograr que la red funcione correctamente.
- Bajos costos de instalación y mantenimiento.
- Facilidad en el reemplazo.
- Pueden tener protocolos que ofrecen un nivel de *"tasas de fallos"* muy bajos.
- Disponibilidad amplia y absoluta en sistema *micro-electrónico-mecánico*.

1.3.- DESAFIOS DE UNA WSN

- *Conservación energética:* debido a la dificultad de remplazar las baterías de los nodos, el ahorro energía es vital en este tipo de redes.

- *Comunicaciones de baja calidad:* al ser dispositivos de recursos limitados el desafío es que funcionen correctamente aún con comunicaciones de baja calidad.

- *Procesamiento en dispositivos limitados:* los recursos disponibles son aún más críticos por lo que tienen que funcionar bien consiguiendo una calidad del servicio lo más alta posible.

1.4.- HARDWARE DE UN SENSOR

- *Unidad de proceso:* formada por microcontroladores, su función es gestionar todas las actividades del nodo entre las que destacan la *captura y procesado de datos*, *comunicación con otros nodos* y *la gestión eficiente de energía disponible*.

- *Memoria:* se complementa con la unidad de procesamiento en el almacenamiento de datos de sensores como la información relevante para tareas de comunicación.

- *Módulo de comunicaciones inalámbricas:* Es la interfaz a través de la cual el nodo interacciona y se comunica con los nodos vecinos, es de corto alcance por las restricciones de los dispositivos.

- Sistema de alimentación son baterías autónomas aunque también se contempla el uso de placas solares en los dispositivos.

1.4.1.- Motes comerciales.

- Crossbow: entre varios productos podemos encontrar: *MICA, MICA2 TELOS, TELOSB IRIS E IMOTE2.*

- Sentilla: motes *TmoteSky* y *TmoteInvent.*

1.5.- SISTEMAS OPERATIVOS PARA MOTES

TINYOS: desarrollado por la universidad de Berkeley (California). TinyOS es un *"sistema operativo"* diseñado para *sistemas embebidos inalámbricos de baja potencia*. Fundamentalmente, se trata de un trabajo de programador y una colección de controladores para microcontroladores y otros circuitos integrados de uso común en plataformas integradas inalámbricas.

Utiliza *nesC*, lenguaje de programación derivado de C, donde cada componente usa *eventos* y *comandos* que permiten el paso de un estado a otro.

La cola que utilizan es FIFO.

Otros lenguajes dedicados a sensores son *PalOS* SOS

1.6.- PROTOCOLOS DE ENRUTAMIENTO PARA WSN

Los nodos no tienen un conocimiento de la topología de la red, deben descubrirla. La idea básica es que cuando un nuevo nodo aparece en la red, anuncia su presencia y escucha los anuncios *broadcast* de sus vecinos. Posteriormente este nodo se informa con los nuevos nodos que están a su alcance y de la manera de cómo enrutarse a ellos, a su vez puede anunciar al resto de nodos que pueden ser accedidos desde él. Transcurrido un tiempo, cada nodo sabrá qué nodos tiene alrededor sabiendo una o más formas de cómo alcanzarlos.

Los algoritmos de enrutamiento en redes de sensores inalámbricas tienen que cumplir las siguientes normas [11]:

- Mantener una tabla de enrutamiento razonablemente pequeña.
- Elegir la mejor ruta para un destino dado *(ya sea la más rápida, confiable, de mejor capacidad o la ruta de menos coste)*.
- Mantener la tabla regularmente actualizada para la caída de nodos, cambio de posiciones y apariciones.
- Requerir una pequeña cantidad de mensajes y tiempo para converger.

1.7.- ESTADARES IEEE

En el entorno de comunicaciones inalámbricas podemos encontrar:

- **Bluetooth (IEEE802.15.1):** radio de comunicación ***720 Kbps*** (*1 Mbps de capacidad bruta*), radio de cobertura entre 10 y 100 metros con un consumo de corriente de 40 mA.

- **Wimax (IEEE802.16):** radio de comunicación de ***70 Mbps*** con un radio de cobertura de hasta 49 Km a frecuencias de 2,5 y 3,5 GHz.

- **Wifi (IEEE802.11)** transmisiones de ***11Mbps*** (802.11b) hasta ***54Mbps*** (802.11g) y opera en las bandas de radio de 2,4 -2,5 GHz.

- **IEEE802.15.4:** permite transmisiones de datos entre ***20 a 250 Kbps*** en radios de cobertura de entre 10 y 75 metros soporta bandas de radio entre 24000-2483,5 MHz como también compatible con las bandas de 868-868.8 MHz en Europa

1.7.1.- CARACTERISTICAS DEL ESTANDAR 802.15.4

- *Flexibilidad* en la red debido a la facilidad de integración en la red mostrada por sus dispositivos ya que cada nodo puede iniciar su participación en la red y el intercambio de información se realiza sin demasiado esfuerzo.

- *Bajo coste:* debido al uso de sus componentes de coste reducido.

- *Bajo consumo de energía:* como depende de baterías, la potencia de transmisión es muy corta.

El estándar define dos tipos de nodo en la red:

Dispositivo de funcionalidad completa (full-function device, FFD): actúa como *coordinador* en la red de área personal (PAN) o como *nodo normal*, permite establecer un intercambio de información con otros dispositivos, encaminar mensajes, en algunos casos actúa como coordinador de la red en caso de que sea el único responsable.

Dispositivos de funcionalidad reducida (reduced-function device, RFD): son dispositivos muy sencillos con recursos y necesidades de comunicación muy limitadas. Por ello, sólo pueden comunicarse con *FFDs* y nunca pueden ser coordinadores.

Las redes de nodos pueden construirse como redes "***punto a punto***" o "***estrella***" compuestas por grupos de dispositivos separados por distancias suficientemente reducidas, cada dispositivo posee un identificador único de 64 bits aunque si se están en ciertas condiciones de entorno pueden utilizarse identificadores cortos de 16 bits.

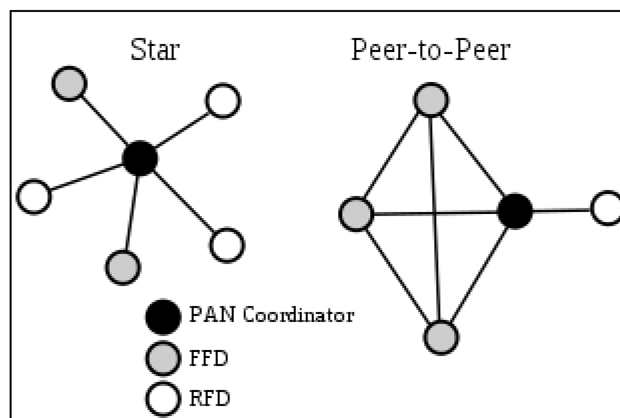


FIGURA 3: Tipos de redes, estrella y punto a punto.

1.8.- 6LoWPAN

6loWPAN (IPv6 over Low power Wireless Personal Area Networks*)* es un estándar que permite que los dispositivos simplifiquen su cabecera de forma muy compacta *(ideal para el protocolo CoAP),* introduciendo de esa manera tecnologías basadas en IPs de nodos inalámbricos de baja potencia para resolver problemas de *interoperabilidad* (*aspecto muy importante a la hora de elegir un sistema operativo*).

Los sensores pueden ser conectados a IPs de redes externas permitiendo la reutilización de tecnologías existentes (ipv4).

1.8.1.- CARACTERISTICAS 6LoWPAN

- Se adapta perfectamente a dispositivos de bajo consumo permitiendo el modo "*sleep*" cuando no estén funcionando.

- Compatibilidad con la multidifusión, permite ***multicast*** *(envío de información en múltiples redes a múltiples destinos simultáneamente).*

- Los protocolos de enrutamiento IP permiten una amplia cobertura mediante ***topología mallada*** *(red en que cada nodo está conectado a todos los nodos)* y tecnologías RADIO integradas.

- El ancho de banda es 1280 Bytes.

- Las direcciones IPv6 son de 128 bits de longitud y consisten en dos partes [2], una de 64 bits que corresponde a un prefijo y otra de 64 bits que corresponden al identificador de interfaz que casi siempre se genera de la dirección MAC de la interfaz que ha sido asignada la dirección obtenida del servidor DHCPv6 *(Protocolo de Configuración Dinámica de Hosts para IPv6)* o por asignación manual.

1.8.2.- MODELO DE PILA 6LOWPAN

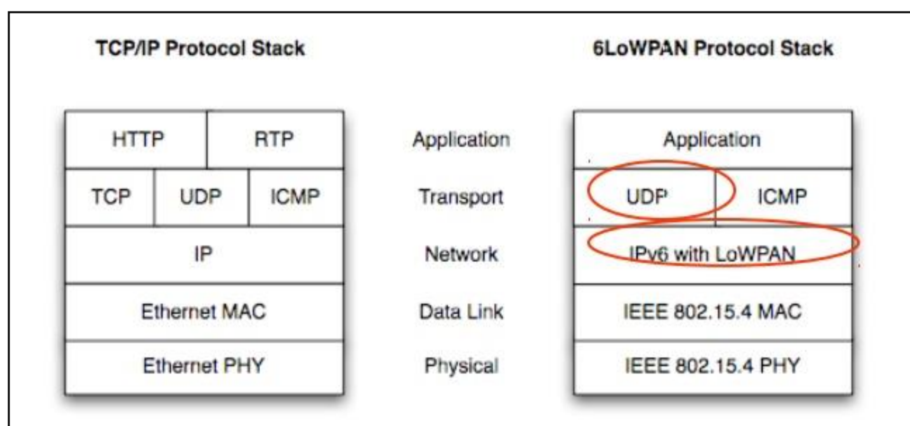


FIGURA 4: Modelo de capas 6loWPAN [4].

La pila de 6loWPAN se basa en el modelo de capas OSI, como podemos observar en la capa de transporte se utiliza el protocolo UDP. El reenvío y enrutamiento de paquetes tiene lugar en la capa 2 (*Data Link*) utilizando la técnica ***"mesh under".***

2.- INTRODUCCION A WEB SERVICES

Un *Web Service* es un servicio ofrecido por una aplicación que expone su lógica a clientes a cualquier plataforma, es decir conectan programas que interactúan con los usuarios mediante una interfaz accesible por medio de la red utilizando protocolos estándares de Internet. En ese proyecto se consideran dos tipos de *Web Service* como los más importantes a destacar, *SOAP* y Web *REST*.

2.1.- SOAP WEB SERVICE

Abreviación de *Simple Object Access Protoco*l, es un protocolo de intercambio de información basado en *XML (lenguaje de marcas extensible utilizado para almacenar datos de forma legible)* diseñado para Internet, se usa para codificar información de los requerimientos de los *Web Services* y responder a los mensajes antes de enviarlos a la red.

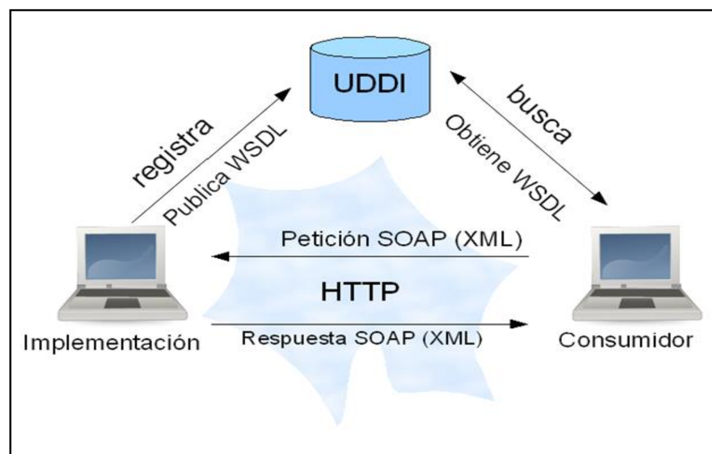


FIGURA 5: Representación del Servicio Web SOAP.

SOAP utiliza **WSDL** [5] (*Lenguaje de descripción Web Service)* que es un formato XML que describe los servicios de red como un conjunto de *puntos finales* que operan los mensajes de *petición / respuesta* y *UDDI* ***(Integración universal de descripción y descubrimiento***) que siendo una plataforma independiente, es una extensión del lenguaje XML que almacena y localiza aplicaciones *Web Service*.

2.1.1.- CARACTERISTICAS SOAP

- Sencillo de implementar, probar y usar ya que atraviesa "Firewalls" y routers por lo cual parece que es una comunicación HTTP.

- Permite el intercambio de documentos o llamada a procedimientos remotos (RPC).
- Extensibilidad: seguridad y WS-routing son extensiones aplicadas en el desarrollo.
- Neutralidad: puede ser utilizado por cualquier protocolo de transporte como HTTP, SMTP, TCP o JMS.
- Permite la interoperabilidad entre múltiples entornos, al no importar la plataforma, los mensajes SOAP son independientes de los sistemas operativos y pueden ser transportados por los protocolos que funcionan en la Internet.

- Aprovecha los estándares existentes en la industria, SOAP aprovecha XML para la codificación de los mensajes, en lugar de utilizar su propio sistema de tipo que ya están definidas en la especificación esquema de XML.

2.2.- RESTful WEB SERVICE

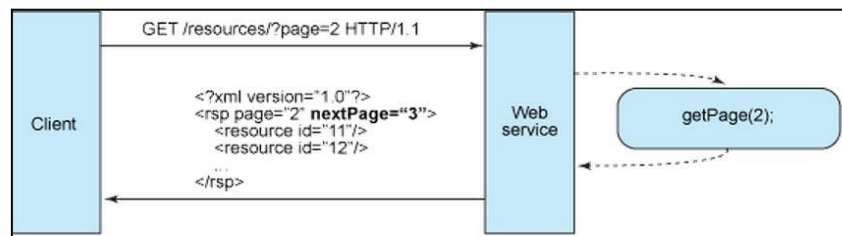


FIGURA 6: Ejemplo de modelo RESTFull.

REST (*Representational State Transfer)*, se basan en HTTP para intercambiar información y no necesita encapsulado extra para ello. Es más ligero, menos engorroso pero al mismo tiempo tiene más limitaciones. En vez de hacer peticiones encapsuladas en un "*sobre SOAP*" para solicitar un servicio para lo que es necesario el *wsdl*, en *REST* las peticiones se hacen mediante el protocolo HTTP con métodos *GET, POST...* sin necesidad de encapsularlo.

REST utiliza *URIs* [7], que son espacios unificados que ofrecen una densa red de enlaces que permiten que la web sea tan utilizada, identificando recursos, los cuales se representan en la misma web como servicios.

2.2.1.- CARACTERISTICAS REST:

- Identificación de recursos a través de URIs.

- Interfaz uniforme que tiene un conjunto fijo de operaciones para interactuar con los recursos, estas operaciones son: GET, POST, PUT y DELETE.

HTTP	CRUD	Descripción
POST	CREATE	Crear un nuevo recurso
GET	RETRIEVE	Obtener la representación de un recurso
PUT	UPDATE	Actualizar un recurso
DELETE	DELETE	Eliminar un recurso

FIGURA 7: Métodos RESTFull.

- Los recursos pueden ser desacoplados de su representación mediante mensajes auto-descriptivos HTTP que pueden ser interpretados por los intermediarios y se ejecuten en nombre del usuario.

- *Hipermedia* como estado de la aplicación, el estado actual de una aplicación Web es capturado en uno o más documentos de hipertexto residiendo en el cliente como en el servidor.

- Compatibilidad con los componentes intermedios como *PROXYs* y *GATEWAYs* que permiten esforzar la seguridad y encapsular sistemas no propiamente Web.

2.3.- REST VS SOAP

Muchos factores juegan a la hora de elegir una de estos dos tipos de arquitecturas, SOAP es interesante a la hora de hablar de la seguridad por lo que soporta WS-Security y muchos protocolos relacionados con las siglas “WS”, que añade características de seguridad Enterprise. Aparte de eso proporciona un estándar de integridad y privacidad de datos lo cual REST no tiene y no puede lidiar con la comunicación de fallos.

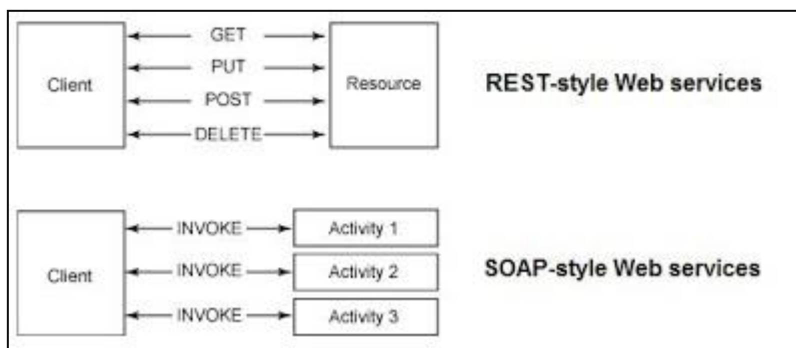


FIGURA 8: Diferencias entre SOAP y REST [8].

Por otro lado, la **interoperabilidad** en SOAP se maneja con mensajes XML personalizados en un único formato de carga útil (*payload*), para la **identificación de recursos** se utiliza un direccionamiento por medio de referencias y para la **descripción del mismo** se ofrece un resumen de las operaciones **WSDL** *(donde las operaciones y mensajes se describen de forma abstracta al ser unidos en puertos WSDL a un formato de un protocolo de red).* Mientras REST se centra en el *nombramiento de recursos mediante URIs*. La interoperabilidad es más fácil porque los mensajes son autónomos lo cual permite una semántica personalizada para utilizar un formato de mensaje más apropiado dependiendo de la aplicación.

Por esas Razones REST es una buena opción para una WSN debido a su baja complejidad.

3.- CoAP PROTOCOLO DE APLICACION RESTRINGIDA

Es un protocolo especializado para el uso de nodos inalámbricos restringidos y limitados de baja potencia que pueden comunicarse de forma interactiva a través de internet, su modelo de interacción *cliente / servidor* es similar al de HTTP con la diferencia que CoAP realiza estas interacciones *(intercambios de mensajes)* de forma asíncrona por medio del protocolo de transporte UDP.

Utiliza *REST (por los métodos GET, POST, PUT, etc..)*, bajo coste operativo, aplicaciones M2M (***máquina a máquina***), se adapta al formato de *nodos-sensores* cuyos controladores son de 8 bits con ROM y RAM limitadas, también permite la utilización de redes limitadas 6loWPAN, que fragmentan los paquetes IPv6 en pequeñas tramas de "capa 2".

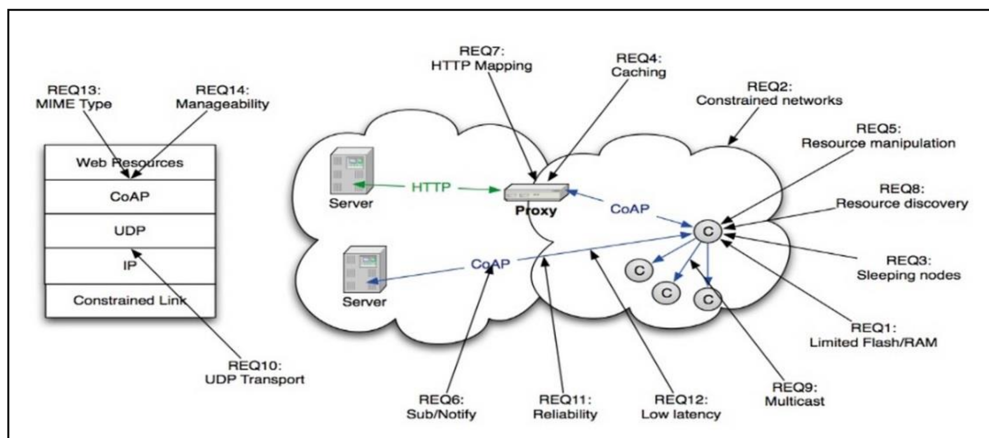


FIGURA 9: Capa y diseño CoAP.

3.1.- CARACTERISTICAS CoAP

- Reducción de sobrecargas TCP por medio de una vinculación UDP en opciones de fiabilidad *unicast* y apoyo *multicast* de peticiones.
- El diseño del protocolo traduce fácilmente a HTTP la integración simplificada con la web.
- Intercambio de mensajes asíncronos.

- Baja sobrecarga de cabeceras para reducir la complejidad al analizar el mensaje.
- Apoyo de URI y contenido (*Content-Type*).
- Búsqueda de recursos y mecanismos de suscripción opcional.
- Caching simple basado en *MAX-AGE*.

Cuando el cliente realiza una petición, mediante una opción que indica el método a utilizar para solicitar un recurso (*identificado por una URI*), el servidor envía una respuesta con un código de respuesta que puede incluir una representación de dicho recurso, esta comunicación es asíncrona a través de transporte UDP. Esto se realiza utilizando una capa de mensajes que soporta una fiabilidad opcional, los mensajes pueden ser: *Confirmable* (CON), *No-Confirmable* (NON), *asentimientos o reconocimiento* (ACK) y *Reset* (RST) , estos mensajes se encuentran en la cabecera *CoAP*.

3.2.- MODELO DE MENSAJERÍA

El modelo de mensajería CoAP se basa en el intercambio de mensajes a través de UDP entre *puntos finales "Host"*, usa un encabezado binario de 4 bytes que puede ser seguido de opciones binarias y una carga útil (*payload*), este tipo de formato utilizado en los mensajes de *solicitud / respuesta*. Para evitar la duplicación de mensajes y la fiabilidad cada mensaje tiene un ID de 16 bits de tamaño y por cada segundo pueden haber 250 mensajes entre dos puntos.

La confirmación de cada mensaje [9] es comprobada por un *tiempo* hasta que el destinatario envía un mensaje de *asentimiento* (ACK) con el mismo ID y en caso de que el destinatario no sea capaz de enviar un ACK, responderá con un mensaje *Reset* (RST).

Si se realiza una comunicación que no requiera confirmación "*No-Confirmable* (NON)", no se envía un ACK por parte del destinatario pero se envía un *RST* si es recibido.

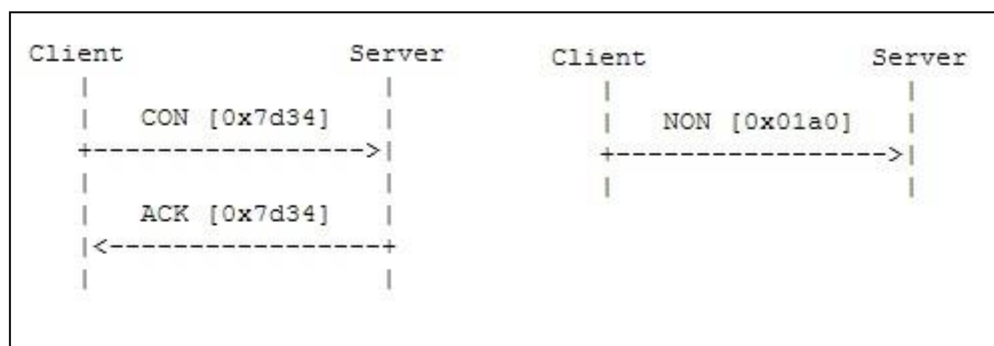


FIGURA 10: Envío de mensaje Confiable / No-Confirmable.

3.3.- FORMATO DE UN MENSAJE CoAP

Los mensajes CoAP están divididos en tres partes:

- **Cabecera CoAP**: contiene información básica que permite el reconocimiento de un la versión CoAP, tipo, código y el ID del mensaje.

  La cabecera está dividida en cuatro partes:

  Versión (Ver): indica el número de versión de CoAP.

  Tipo (T): si el mensaje es tipo *CON, NON, ACK o RST*.

  Contador de opciones (OC): número de opciones después de la cabecera, si este campo es rellenado con un 0 es que porque no hay opciones y se pasa directamente al campo de carga útil (*payload*).

  Código (): indica si el mensaje es una petición (con valores de 1 a 31) indicando en el *Resquest Method*, si es una respuesta (valores de 64 a 191) indicando el Response *Code* o si está vacío (valor 0).

- **Opciones CoAP:** provee los parámetros para rellenar peticiones.

- **Carga útil CoAP:** contiene el cuerpo del mensaje.

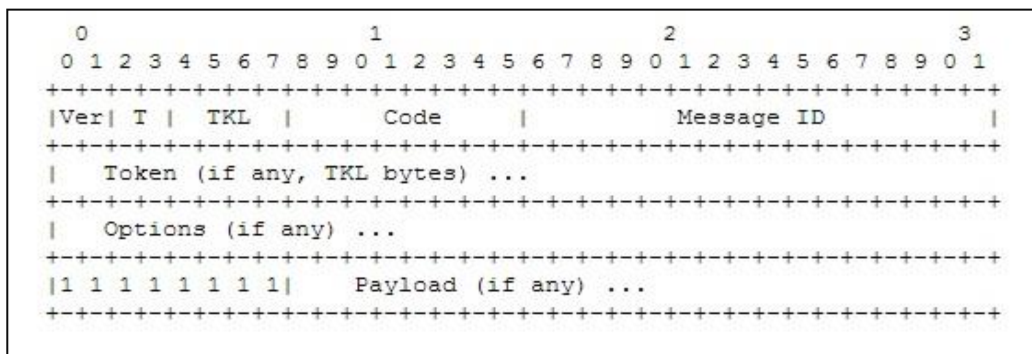


FIGURA 11: Formato de trama de un mensaje CoAP.

3.3.1.- OPCIONES CoAP

CoAP define una serie de opciones que pueden ser incluidas en un mensaje, especificadas por el "*número de opciones*" definido, su longitud y valor. Estas instancias *(opciones de mensajes)* son calculadas por la suma de los *números de opciones* del mensaje anterior, que deben aparecer en el orden definido más un delta de codificación.

.

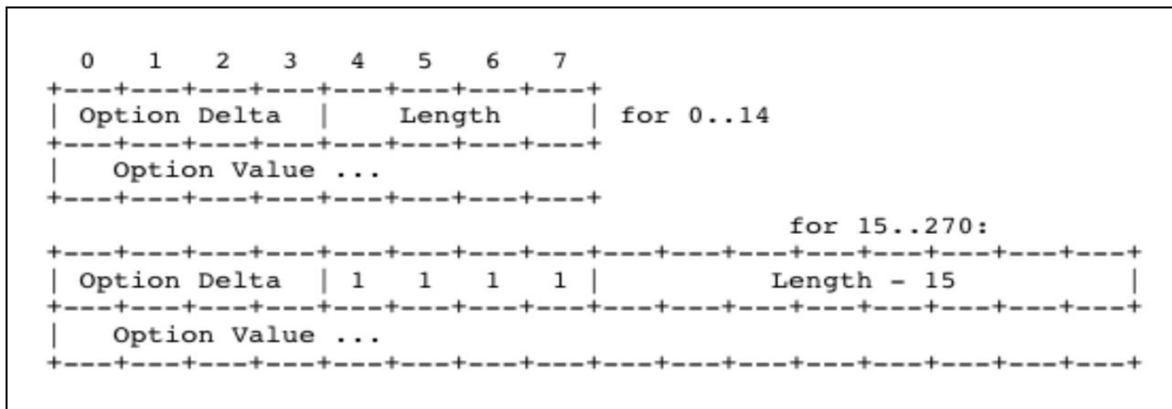


FIGURA 12: Trama opción CoAP.

El formato de las opciones es el siguiente:

Opción Delta: entero son signo de 4 bits que indica la diferencia entre el número de opción actual del anterior.

Longitud (*length*): entero sin signo de 4 bits que como su nombre indica la longitud del valor de la opción, si este campo se establece a 15, se añade un entero sin signo de 8 bits que permite longitudes que van desde 15 hasta 270 bytes (como indica la figura).

Las opciones *CoAP* son **pares (opciones opcionales)** y **impares *(opciones críticas)*,** estas opciones no son obligatorias y la diferencia entre ellas se basa en cómo se trata la situación que se da cuando se recibe una opción no reconocida.

Opcionales: deben ignorar mensajes que no tienen opciones reconocidas.

Críticas: ocurre si un mensaje de petición *confirmable* (CON) es devuelto por un código de respuesta 402. Otro caso sería si un mensaje de respuesta *Confirmable* y un *No -Confirmable* son ignorados.

Hay que tener en cuenta que una opción CoAP siempre será opcional por lo que la utilización de estos dos opciones son para permitir implementaciones que detienen alguna opción de procesamiento que esté incompleta o no implementada (*esto lo podemos ver en algunas partes del código CoAPBlipP.nc que hay opciones que aún están por implementar*).

Además de estas dos características de opciones, también existen otros tipos de opciones:

- ***Token:*** sirve para asociar peticiones con sus respuestas asociadas.
- ***Uri-Host:*** se especifica para solicitar el host de Internet del recurso.
- ***Uri-Port:*** número de puerto del recurso.
- ***Uri-Path:*** segmento de la ruta absoluta al recurso.
- ***Uri-Query:*** cadena de consulta.

- ***Proxy-Uri:*** se utiliza para hacer una petición a un proxy.

- ***Content-Type:*** se indica el formato de representación de la carga útil (*payload*) del mensaje dado como un valor numérico.

- ***Max-Age*****:** tiempo de respuesta en caché que el cliente tiene para esperar la respuesta del servidor.

- ***E-Tag*****:** una respuesta proporciona el valor actual de la *entidad-tag* para la representación del recurso de destino. Una *entidad-Tag* es como un identificador local de recursos para diferenciar las representaciones de recursos que varían con el tiempo.

- ***Location-Path and Location-Query***: indica la ubicación de un recurso, creado por un *POST*, como la ruta URI absoluta.

- ***If-Match:*** se utilizar para hacer una solicitud condicional de una actual existente o de un valor *E-tag* para una o más representaciones del recurso destino.

- ***If-None-Match****:* útil para solicitudes de creación de recursos, tales como *peticiones PUT* y como un medio de protección que evita sobrescribir *IDs* de clientes cuando varios de estos están ejecutándose en paralelo en el mismo recurso.

3.4.- COORDINACION PETICION / RESPUESTA

CoAP se basa en el intercambio de mensajes asíncronos entre dos nodos, un nodo actuando como cliente envía una o más peticiones sobre uno o más recursos alojados en un determinado servidor que atenderá la petición. El servidor responderá a la petición indicando si la petición recibida es exitosa o no.

En esta coordinación *petición / respuesta* utiliza una técnica **"*Piggy-backing*"** que consiste en que el servidor responde inmediatamente a una petición recibida de tipo *Confirmable* y envía un mensaje de tipo *ACK*. Este tipo de respuestas se dan independientemente de si la respuesta indica éxito o fallo al tratar la petición, esta coordinación sigue los siguientes pasos

Las peticiones se envían mediante mensajes *CON* o *NON* y consiste en que la petición ejecuta un método *(GET, PUT, POST y DELETE)* sobre un recurso, el cual viene identificado por una ruta contenida en el campo *URI-Path* del paquete CoAP.

Otro método utilizado para poder determinar que un mensaje recibido es la respuesta a un mensaje concreto y no a otro es *"el uso de Token"*. Un *Token* es un valor pensado para ser gestionado de forma local, para que el cliente pueda diferenciar de forma concurrente las peticiones que tienen en curso. El valor que toma el *Token* debe ser actualizado para que todas las peticiones que un cliente tenga pendientes de recibir respuestas tenga un *valor Token* distinto.

3.5.- CACHING

CoAP provee un método para que los nodos puedan almacenar en su memoria caché respuestas obtenidas a partir de una petición y que podrán ser utilizadas posteriormente bajo ciertas condiciones.

Lo que se pretende con esto es optimizar el uso de la red para evitar el envío de paquetes redundantes. Una respuesta almacenada puede ser *re-usada* para responder una petición de igual característica que la petición original que causó esa respuesta sin necesidad de enviar otra petición al servidor.

Caching reduce el tiempo de respuesta y el consumo ancho de banda de red. Las condiciones en que un nodo pueda re-utilizar una respuesta almacenado son:

- El método de la petición y de el que se utiliza para obtener la respuesta almacenada debe coincidir.

- Las opciones contenidas en la petición recibida deben de ser las mismas que las que estaban presentes en la original, excepto *Max-Age o E-Tag*.

- La respuesta es almacenada debe estar validada o aún no haber caducado.

Hay dos formas de decidir si una caché puede utilizarse para satisfacer una petición:

• ***Modelo "freshness":*** se entiende por "fresh" una respuesta almacenada en caché que puede ser usada por un nodo para responder a una petición recién llegada. Este atributo se determina en la opción CoAP "Max-Age" que indica la *edad máxima* en segundos a partir de la cual una respuesta deberá dejar de considerarse *fresh* y pasará a estar obsoleta. Esta opción es añadida por el servidor original que contiene el recurso y determina cuando expirará el periodo de validez de la respuesta.

• ***Modelo de Validación:*** se utiliza para renovar en el servidor la validez de una respuesta de un nodo CoAP almacenada en caché que ha dejado de ser válida porque expiró su periodo de validez. El nuevo periodo de validez de la respuesta indicada con la opción E-Tag vendrá indicado mediante la inclusión de la opción Max-Age.

4.- CORE LINK FORMAT

Es una serialización [10] particular tipo "*links*" basada en enlaces HTTP, describe las relaciones entre recursos y páginas web a través de las aplicaciones para M2M, estas relaciones permiten que un cliente de una máquina busque y ejecute un recurso del servidor. *Core Link* fue creado para cumplir con las siguientes necesidades:

Descubrimiento de recursos: En una comunicación M2M los nodos no son intervenidos por personas y es importante que sean capaces de descubrir que recursos almacena cada uno para que puedan dirigir sus peticiones a uno u otro nodo en función del recurso que quieren buscar. Esto se aplica más en casas automatizadas donde la detección de recursos se realiza de forma *"unicast"* (*envío de mensajes entre un emisor y un receptor*) o "*multicast*" (*envío a muchos receptores*).

Para realizar el descubrimiento se establece una petición tipo "GET" dirigida sobre el recurso "**/.well-known/core**" que es respondida por el servidor con un mensaje cuya carga útil *(Payload)* contiene una descripción del recurso siguiendo el formato *CORE LINK.*

Este formato contiene ciertos ***atributos*** que describen el recurso y de esta manera el cliente sabrá que recursos tiene a su disposición, ya que en el *Payload* vendrá incluida la URI del recurso.

Colecciones de recursos: los servidores pueden hacer uso de colecciones de recursos, como por ejemplo una lista de recursos o un grupo de sensores que miden la temperatura. Core link es capaz de entrar a la lista de recursos del servidor y navegar por ellos por medio de una interfaz de descripción de recursos.

Directorio del recurso: es un motor de búsqueda limitada de recursos para M2M donde los servidores almacenan enlaces a recursos almacenados en otros servidores.

CORE LINK FORMAT crea la relación entre *recursos y servidores* por medio de un enlace que transmite la URI *(dirección del recurso)* que describe un recurso alojado en el servidor. Los *atributos* que tiene el fin de describir la información útil del acceso al enlace destino de una relación son:

Atributo recurso tipo "rt": es un sustantivo que especifica el recurso.

Atributo interfaz "if": proporciona un nombre o la URI que indica una definición de interfaz en concreto para interactuar con el recurso destino.

Atributo estimación de tamaño "sz": indica el tamaño máximo de la representación de recursos devueltos por medio de un *GET* al URI destino.

4.1.- DESCUBRIMIENTO DE RECURSOS (well-known core)

Como ya hemos visto, el descubrimiento de recursos en *CORE LINK* se realiza mediante la URI del recurso que si es reconocido devuelve una *lista de enlaces* sobre los *recursos alojados* en el servidor. Los recursos empiezan con un componente de ruta "*/.well-known/*" y una vez pasado el reconocimiento, el servidor aporta por el puerto predeterminado una lista de enlaces para tareas en concreto, *CORE* para esto cuenta con las siguientes interacciones:

- *GET* al puerto por defecto para que el servidor devuelva un conjunto de enlaces disponibles en *CORE-FORMAT*.
- Filtrado: se realiza desde cualquiera de los atributos del *CORE-LINK* utilizando una cadena de consulta, sin embargo un servidor no está obligado a realizar el filtrado. *Ejemplo: GET /.well-known/core?rt=temperature-c*, esta línea de comando pide un recurso tipo *TemperatureC*.
- Directorio de recursos mediantes servidores capaces que describen estos recursos con distintos criterios de valoración permitiendo solicitudes POST de "*./well-known/core*".

```
REQ: GET /.well-known/core
RES: 200 OK
</sensors>;ct=40;rt="index";rt="Sensor Index", </sensors/temp>;rt="TemperatureC";if="sensor",
</sensors/light>;ct=41;rt="LightLux";if="sensor",
<http://www.example.com/sensors/t123>;anchor="/sensors/temp"
;rel="describedby",
</t>;anchor="/sensors/temp";rel="alternate"
```

FIGURA 13: Ejemplo de una petición GET por consola.

En este ejemplo particular, el servidor presenta un directorio llamado sensores que contiene dos recursos: *temperatura y luz* disponibles a través de la URL / *sensores* / *temp* y / sensores / luz respectivamente.

4.2.- OBSERVACION DE RECURSOS

El estado de un servidor *CoAP* puede cambiar con el tiempo ya sea por **Polling** o long-polling generando una sobrecarga significativa, para evitar esta sobrecarga se utiliza un patrón de diseño "*observer*" que implica mantener una lista de partes interesadas de un objeto, notificando automáticamente cuando se produce un cambio de condición, evento o estado.

Este mecanismo de observación consta de tres fases:

Establecimiento:
Un cliente registra un recurso mediante la realización de un GET que incluye la opción "*observe*", al ser recibida esa solicitud en el servidor se establece una relación *(en caso de que esa petición sea exitosa)* entre el *cliente* y el *recurso destino*. Las notificaciones son hechas por el *Token* por parte del cliente que incluye la opción *observe* y permite saber si la relación se ha establecido correctamente. En caso de que el servidor no pueda establecer la relación, éste debe ignorar la esta opción y procesar la solicitud como usual.

Mantenimiento:
El cliente es capaz de refrescar o actualizar la observación en cualquier momento, este proceso se realiza por medio de la repetición de la petición GET con la opción "observar", por otro lado el servido al recibir una solicitud como esta, sólo reemplaza la solicitud existente si la relación cumple con que la *URI* y *origen* de las dos peticiones están enlazadas y los Id del mensaje o *Token* no deben de ser tomados en cuenta en este proceso.

Fin de la observación:
La observación termina cuando se produce las siguientes condiciones:

El servidor envía una respuesta de notificación con un código de error.
El cliente rechaza una notificación *CON* con un mensaje *RST*.
Un *time-out* de retransmisión de una notificación a los clientes *CON*.

4.3.- CARACTERISTICAS DE UNA NOTIFICACIÓN

Cada respuesta de notificación incluye la opción "*observar*" y el "*echo-Token*" especificado por la solicitud del cliente.

- Debe de tener un contenido válido.
- El formato de representación tiene que ser el mismo que la solicitud *GET* de lo contrario el servidor la devuelve un *"internal server error"*.
- Pueden enviarse con mensajes *CON* o *NON*.
- Si el cliente no reconoce el *token* en un mensaje *CON* será rechazado con un mensaje *RST*.
- Si la notificación llega antes que la respuesta a la solicitud, el cliente puede tomar esa notificación como una respuesta inicial en lugar de la respuesta real.

4.4.- POLLING

Polling o "sondeo" es una técnica que cuando se recibe una petición y no hay ninguna actualización o información sobre el recurso, se devuelve al cliente una respuesta vacía y este esperará un tiempo hasta realizar otra *petición poll*.

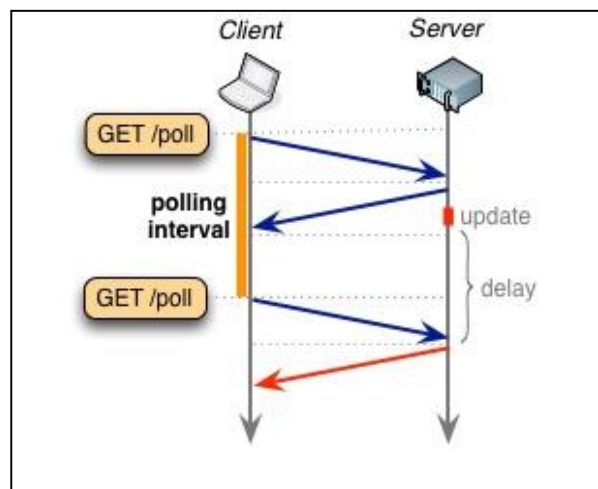
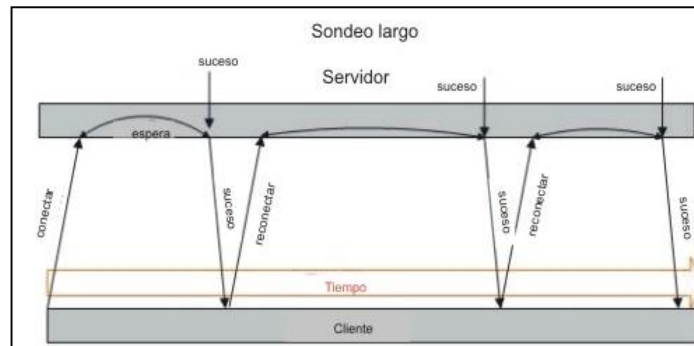


FIGURA 14: Polling en una conexión.

4.5.- LONG-POLLING

Long-polling es una variación de la técnica tradicional *polling* en que el servidor puede esperarse a enviar la respuesta a una petición si no hay información nueva o disponible.

Cuando un servidor recibe una petición *long-polling* y no ha ocurrido ningún evento ni ha expirado un determinado tiempo *timeout* a partir del cual se debería enviar nueva información al cliente, el servidor simplemente espera a enviar una respuesta. El cliente una vez que recibe su respuesta inmediatamente envía una nueva petición *long-poll* para que cuando un nuevo evento ocurra pueda volver a ser informado.



FIGUR 15: Long Polling.

5.- TINYOS OPERATING SYSTEMS

TinyOS es un sistema operativo *BSD* con licencia de código abierto diseñado para dispositivos inalámbricos de bajo consumo de energía, como los utilizados en redes de sensores, computación ubicua, redes de área personal, edificios inteligentes y contadores inteligentes.

5.1.- CARACTERISTICAS DE TINYOS

- ***Recursos limitados:*** Los Motes (sensores *TelosB* utilizados en el proyecto) tienen pocos recursos físicos debido a su tamaño, bajo costo y bajo consumo de energía, los motes están compuestos por procesador de 1 MIPS y decenas de Kilobits de almacenamiento.

- Las **tareas *(task)***, que es una forma de llamada a procedimiento diferido (DPC) que permite aplazar un cálculo y operación hasta un momento posterior, se ejecutan de forma síncrona.

- **Concurrencia:** en TinyOS se da la concurrencia a través de *tasks* (tareas) y eventos que se ejecutan de forma atómica, los eventos pueden interrumpir otros eventos o tareas para cumplir los requerimientos de tiempo real.

- **Flexibilidad:** TinyOS se adapta a la variación de hardware de aplicaciones.

- **Bajo consumo:** dada la amplia gama de aplicaciones para redes de sensores, TinyOS proporciona una gran flexibilidad en las estrategias de gestión de energía y ciclo de trabajo.

- Basado en el modelo de componentes, utiliza **NesC** como lenguaje de programación.

- **Modelo de ejecución** concurrente que permite la creación de tareas y administrar las preferencias de ejecución. Es capaz de utilizar muchos componentes a la vez, mientras estos requieran poca RAM.

- Cada llamada *In/ Out* es ***split-phase*** que consiste en devolver la respuesta de una solicitud inmediatamente en vez de esperar a que se termine de ejecutar un bloque de código o procesos posteriores.

- Utiliza **interfaces de programación** y bibliotecas de componentes para simplificar el escribir código de una nueva aplicación, como por ejemplo la ***interfaz read*** que se utiliza en cualquier componente para realizar una lectura de un sensor.

TinyOS es una plataforma estándar diseñada para la programación de sensores, en este caso mi proyecto utiliza sensores *TelosB* que están programados bajo esta plataforma y cuyas aplicaciones, escritas en *nesC*.

5.2.- PROPIEDADES DE MEMORIA Y LLAMADAS SPLIT-PHASE

La memoria en los motes *TelosB* es un recurso muy valioso porque está limitado y porque administra el consumo de energía cuando el mote está en modo “sleep”. Los *módulos* que son las aplicaciones en *nesC*, asignan las variables privadas al componente. En la RAM no se guardan variables constantes, por lo que se utilizan “*enumeraciones*” que se definen como directivas en enumeraciones tipo de lenguaje C.

Split-Phase: 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 donde la llamada a un comando detiene la ejecución de cualquier otro código hasta que no finalice la operación no se continúa la ejecución.

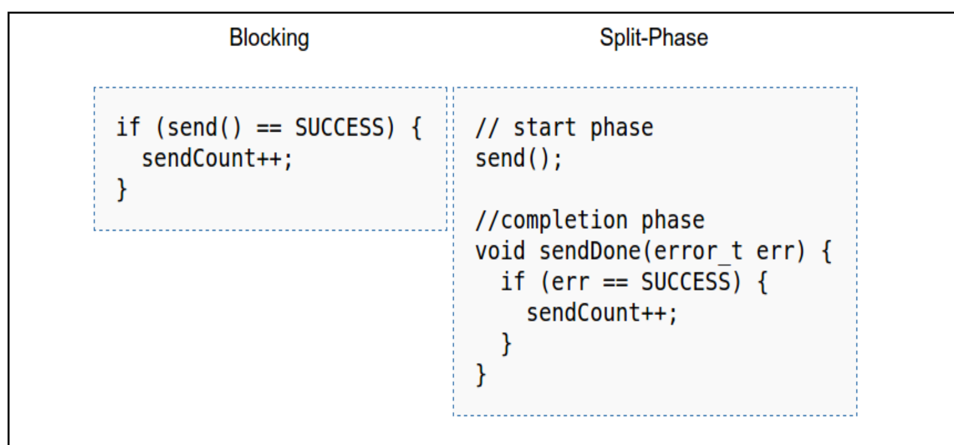


FIGURA 16: Split-Phase.

Sin embargo, los motes tienen una limitación considerable en cuanto a memoria RAM. Cada hilo (*proceso*), cuando se mantiene suspenso debe reservar un cierto espacio de memoria RAM para almacenar las variables que está usando, lo cual es un desperdicio de espacio, cuanto más hilos se mantengan en ejecución, más gasto de memoria habrá.

La solución que da TinyOS es el uso de interfaces Split-Phase, de forma que cuando se llame una función, ésta comienza y luego un evento avisará que esa función ha finalizado. Esto hace que los programas de TinyOS sólo requieran usar una única pila de memoria RAM (*lista ordenada o estructura de datos muy simple*).

Aunque estas interfaces son algo más complejas a la hora de la programación pero aportan ciertas ventajas:

Ahorro de memoria: las operaciones no tienen que almacenar variables de estado mientras se ejecutan (reduciendo el tamaño de la pila).

El programa no pierde capacidad de respuesta: al no existir ninguna función que mientras se ejecute bloquee el sistema, éste puede reaccionar ante cualquier otro evento casi de forma inmediata.

5.3.- LENGUAJE DE PROGRAMACIÓN NesC

NesC es un dialecto del lenguaje C diseñado para desarrollar aplicaciones para TinyOS, es un lenguaje basado en componentes y dirigido a eventos, implementa tres modelos: *a eventos*, *concurrente* y a *componentes* previstos por TinyOS. Los componentes conectan sus interfaces entre sí *(wiring)* en el archivo *"configuración"* para dar lugar a las aplicaciones.

Gracias a estas conexiones, una misma aplicación puede correr en distintas arquitecturas hardware sin apenas tocar código, simplemente modificado el *wiring*, esto significa la reutilización de módulos. NesC las siguientes características:

- *Separación de la construcción y la composición:* los programas se crean a partir de componentes, los cuales se tienen que unir para formar programas enteros. Los componentes cuentan con dos archivos, uno de ellos especifica el código, que es donde se llevará a cabo programa especificando el comportamiento de una o más interfaces, a esto no referimos como *"módulo"*. El otro archivo se llama *configuración* que como se ha dicho antes, se realiza el conexionado de las interfaces que algunos módulos usan y que son provistas por otros módulos.

- *Programación por interfaces, módulos y configuraciones.*

- *Especificación del comportamiento del componente mediante una serie de interfaces*: las interfaces son *provistas* o bien *usadas* por los componentes. Las que son *provistas* representan la funcionalidad que un componente ofrece al usuario y las que son *usadas* representan las herramientas necesarias para poder desarrollar las interfaces que el módulo provee.

- *Las interfaces son bidireccionales:* especifican una serie de funcionalidades que son implementadas por el proveedor de la interfaz (*comandos*) y otro número de funciones que tienen que ser especificadas por el componente que usa esa interfaz (*eventos*).

- *Los componentes están enlazados estáticamente a través de sus interfaces.*

5.3.1 INTERFACES

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

```
interface interfazPRB {
  command void comando1();
  command int comando2(int c);
}
```

FIGURA 20: Modelo de interfaz en NesC.

Cada interfaz se declara con la palabra "interface". Las interfaces pueden ser:

Interfaz genérica: se especifican uno o más parámetros para que esté completamente definida, se utilizan para la reutilización de código.

interface Timer<typedef precision_tag>{…}

Interfaz instancia: puede ocurrir que se quiera utilizar varias veces y para esto se crean instancias de una interfaz, se distinguen con nombres personales mediante la palabra "as".

interface Timer<TMilli> as Timer2

Interfaz parametrizada: se trata de una interfaz simple, que en determinados casos es necesaria distinguirla (*porque se pueden usar varias veces*).

interface Receive[collection_id_t id]

5.3.2.- MODULOS

Implementan bloques funcionales generalmente de un nivel de abstracción más bajo que el bloque que lo referencia. Pueden proveer varias interfaces y por lo tanto puede implementar distintas funcionalidades.

```
module modulo2C {
  provides {
    interface interfaz2;
    /*
      Única interfaz proporcionada por el módulo
    */
  }
}

implementation
{
  command void interfaz2.i2c1() {
    int v1 = 5;
    int v2;
    v2 = v1;
  }

  command uint16_t interfaz2.i2c2(int c) {
    return (uint16_t)2*c;
  }
```

Figura 21: Formato módulo en NesC.

Observamos que el módulo se divide en dos partes, la primera se declara todas las interfaces como ya dijimos y en la segunda parte los comandos y el funcionamiento de cada uno de ellos, en caso de que un comando tenga un evento relacionado se implementará en la parte final del programa.

5.3.3.- CONFIGURACIONES

Permiten la creación de la aplicación, un archivo de configuración consiste básicamente en módulos o componentes que actuarán en conjunto unidos por interfaces, a esto se le denomina *wiring* o conexionado.

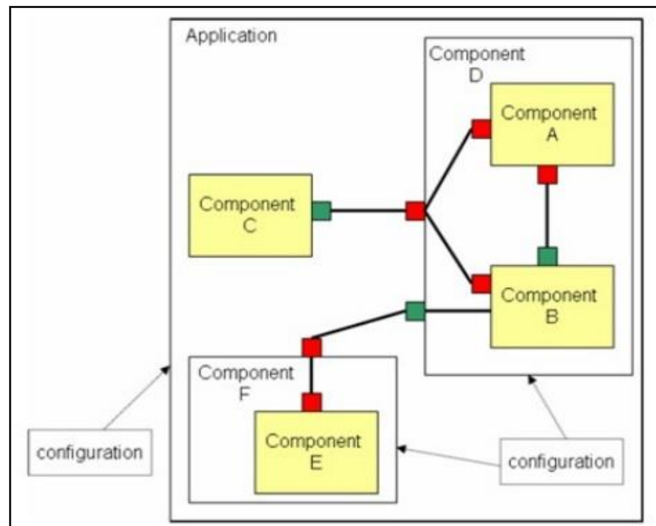


Figura 22: Ejemplo de configuración.

La razón porque un componente se diferencia entre módulos y configuraciones es para favorecer la modularidad de una aplicación que permite al desarrollador ensamblar velozmente las aplicaciones.

5.4.- MODELO CONCURRENTE

El lenguaje nesC proporciona un modelo de ejecución que consiste en tareas y en interrupciones. El código se reparte en:

Código síncrono: que puede ser enviado en ejecución solo desde una tarea.
Código asíncrono: que puede ser enviado en ejecución después una interrupción hardware, como parte de la elaboración de un evento notificado.

Escenarios de acceso concurrente:

- se produce una actualización al estado compartido donde sólo puede ser ejecutado desde código asíncrono.
- se produce una actualización al estado compartido que puede ser ejecutado tanto desde código síncrono como desde código asíncrono.

5.5.- COMANDOS BÁSICOS PARA NESC

Primeramente tenemos que cambiar los permisos al mote para poder instalar programas, tenemos que tener los permisos de *súper usuario* o en su defecto cambiárselos mediante:

$ sudo chmod 666,/dev/ttyUSBX

Si no sabemos el puerto del mote, escribimos en consola la orden "*motelist*" que nos mostrará por pantalla los motes que tenemos conectados.

Para compilar del código nos dirigimos al directorio del programa y escribimos:

$ make telosb

Al usar la orden "make" compilaremos el programa donde si hay un error, el compilador nos avisará en qué línea se encuentra ese error.

Para instalar el programa en el mote escribiremos:

$ make telosb make install, /dev/ttyUSBX

Indicaremos al compilador en qué mote queremos instalar el programa mediante "/dev/ttyUSBX" conectado al pc vía USB. Si queremos reinstalar el código lo único que tenemos que hacer es cambiar la orden *install* por *reinstall*.

Existe una forma de compilar-instalar donde se puede asignar un número que identifique el nodo (mote) que estamos programando mediante la orden:

$ make telosb make install bsl, /dev/ttyUSBX

Este identificador se le conoce como *dirección AM*, se debe evitar en lo posible asignar el identificador 0 al mote porque ese valor suele estar reservado para la estación base y podría dar problemas a la hora de ejecutar alguna aplicación.

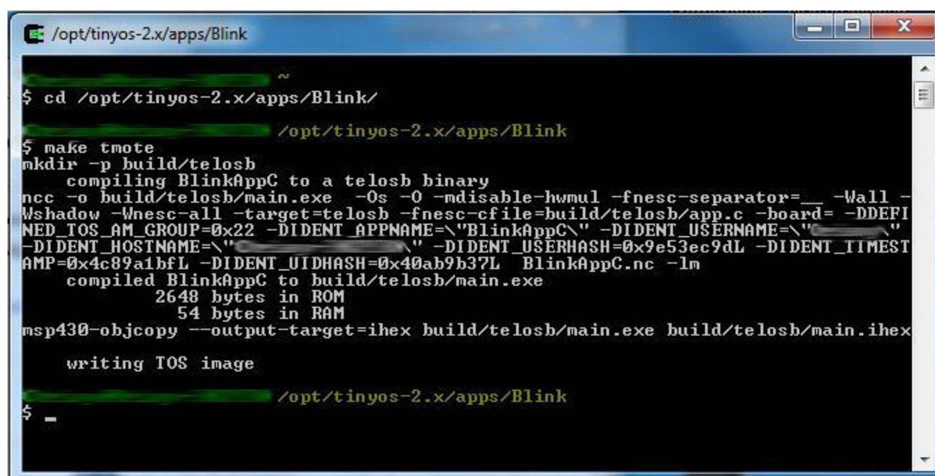


Figura 23: Compilación de una aplicación en consola.

6.- IMPLEMENTACION DE CoAP EN TINYOS

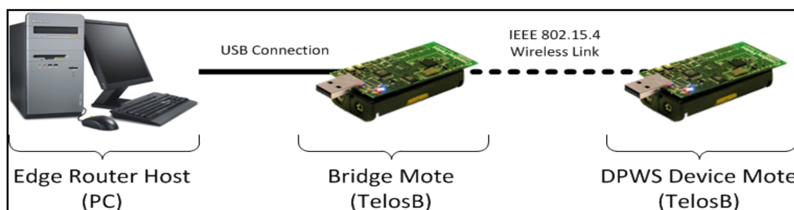
6.1.- HARDWARE

El hardware que utilizaremos en este proyecto son motes *TELOSB* que son dispositivos inalámbricos de bajo consumo y pueden ser operados sin vigilancia durante mucho tiempo. Al hablar de las limitaciones de recursos, estos motes son limitados en cuanto a su comunicación como también de almacenamiento.

TelosB fue publicado y desarrollado por la comunidad de investigación de la Universidad de Berkeley que agrupa todos los elementos esenciales para los estudios de laboratorio incluyendo la capacidad de programación USB, una radio IEEE 802.15.4, MCU de bajo consumo de memoria extendida y un conjunto de sensores para recoger datos como temperatura, humedad y luminosidad.

Algunas características de este dispositivo son :

- EEE 802.15.4 / ZigBee transceptor de RF compatible.
- Banda ISM: 2,4 a 2,4935 GHz.
- 250 Kbps tasa.
- Integrado con antena a bordo.
- 8 MHz microcontrolador MSP430 con 10 KB de RAM.
- Bajo consumo de corriente.
- 1MB flash externo para registro de datos.
- Programación y recopilación de datos a través de USB.



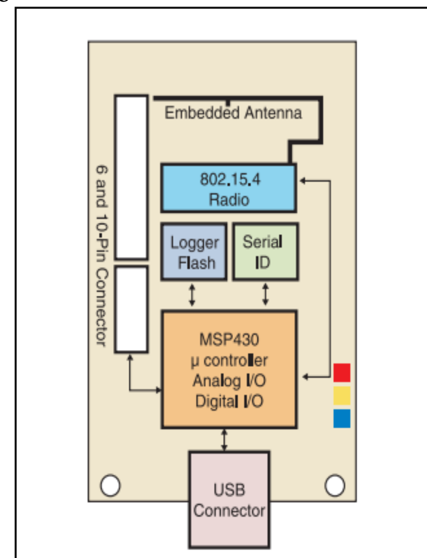


Figura 24 y 25 Mote Crosbow Telosb.

6.2.- SOFTWARE

En este proyecto se utiliza la versión 03 de *CoAP* que se basa en librerías *CoAP* que tienen como requisitos un diseño de *utilización de menos RAM (en motes TelosB la RAM es de 10 KBytes)* y un diseño de programación lo más *simple posible*, al utilizar menos RAM se consume menos energía, lo cual aumenta la vida de la batería.

Las librerías CoAP son:

- **CoapPdu:** asigna nuevas estructuras tipo *coap_pdu_t* (*estructuras que llevan el mensaje CoAP*) y permiten consultar o establecer valores en las opciones CoAP.

- **CoapList:** asigna nuevas estructuras tipo *coap_list_t* para manejar listas enlazadas en que se manejan un conjunto de nodos (*añadiendo o quitando nodos*).

- **CoapOption:** asigna estructuras tipo *coap_option_t.*

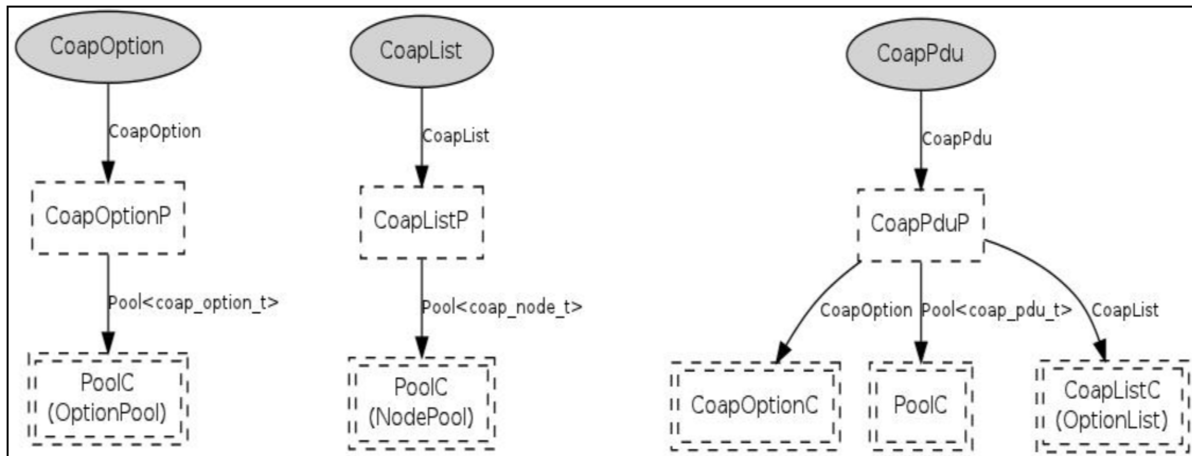


Figura 26: Librerías CoAP.

Observamos que en cada librería CoAP hay un componente "*Pool"*, pues éste componente se utiliza para especificar dónde conseguir un espacio en una región de la memoria RAM.

6.3.- PETICIONES / RESPUESTAS CoAP

Las *peticiones / respuestas* se basan en el uso de dos interfaces, *CoapClient* y *CoapServer* , que proporcionan comandos y eventos para unir un servidor UDP con su puerto correspondiente así como el añadir recursos al servidor .

Interfaz CoapClient: provee comandos y eventos que notificarán cuando se reciba una respuesta CoAP del servidor.

Interfaz CoapServer: provee comandos para inicializar el servidor y realizar el "bind" por medio del puerto especificado.

Al hablar de recursos en el servidor CoAP, se utiliza una interfaz llamada ***CoapResource*** que inicializa un nuevo componente del servidor CoAP pasándole una lista de estructuras tipo ***coap_resource_t***.

```
Typedef struct {
uint8_t key;
uint8_t uri[MAX_URI_LEN];
uint8_t len;
uint8_t rt[10];
uint8_t iff[10];
uint8_t sz;
uint8_t ct;
uint8_t is_subscription;
} coap_resource_t
```

Cuando llega una petición a un recurso que pertenece al servidor, un módulo de tipo *Coap_ReadResource* la atiende. Será este módulo el encargado de mantener las listas de clientes que están observando al recurso. La lista de clientes se maneja mediante una estructura de lista enlazada que guarda todos los clientes que han manifestado su interés por recibir notificaciones de un recurso.

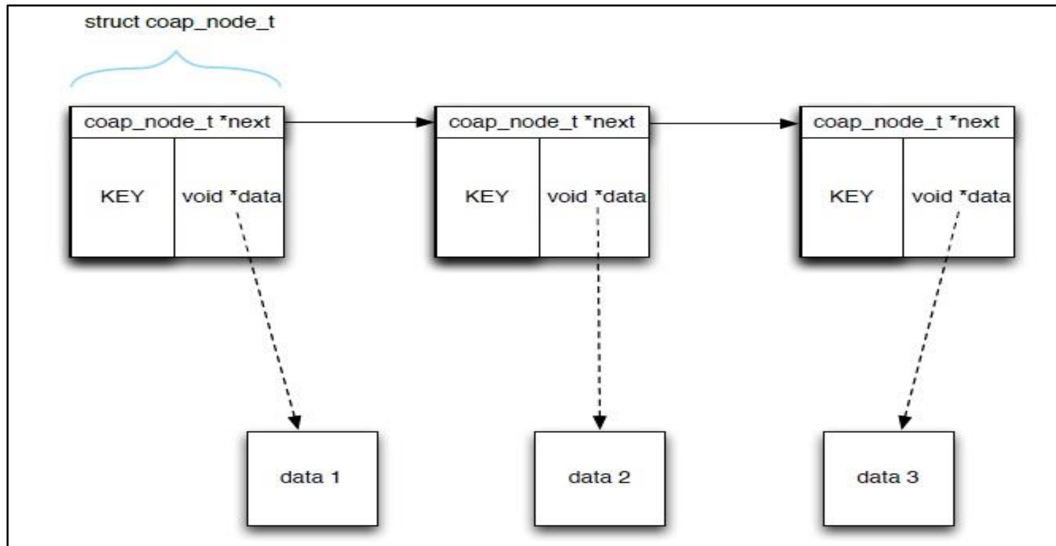


Figura27: Lista enlazada de clientes.

La lista está formada por una estructura tipo *coap_node_t,* dónde los nodos (clientes) van formando sucesivamente una lista enlazada. Los campos de esta estructura son:

- ✓ **Puntero next:** apunta al siguiente nodo de la lista.
- ✓ **Key:** identifica y diferencia a cada nodo de la lista.
- ✓ **Puntero data:** apunta al contenido de cada nodo de la lista (*información*).

En el servidor podemos encontrar dos *componentes* y *tareas* para el *tratamiento de colas* que se pueden formar. Estos componentes son:

IncomingQueue: para conexiones entrantes.
ProcessQueue: otro para las conexiones de procesamiento.

Al llegar un paquete se realiza una tarea de procesado ***"processIncoming"***, tras su finalización el servidor toma el primer nodo de la cola y revisa el tipo de mensaje que lleva (*CON* o *NON)*. En ese momento se ejecuta una nueva tarea ***"processResource"*** que se encarga de poner en marcha el método que se solicita (GET, POST, PUT y DELETE) del recurso.

Al acabar el procesamiento de solicitud se genera el ***evento "Done"*** que devuelve una respuesta que puede ser un mensaje *CON*. Se reinicia el temporizador "TIMEOUT" que controla el tiempo de transmisión de una respuesta, en caso de que el servidor no es capaz de procesar un mensaje *CON* se envía un *RST*, de lo contrario un *ACK*. Sino la respuesta se envía directamente al Cliente CoAP.

7.- DESARROLLO DEL PROYECTO

Figura 28: Ejemplo del proyecto.

Propuesta del proyecto:

La idea principal de este proyecto es enviar una petición (GET o PUT) por medio de HTTP que posteriormente se convertirá en una orden *CoAP*, que se cargará en los motes TelosB donde se ejecutará la aplicación CoAP para recoger lecturas de sensores. Posteriormente se realizarán ciertas acciones *"en función de las lecturas de los sensores"* como enviar notificaciones a otros PCs, encender la calefacción o encender una o varias lámparas de una casa.

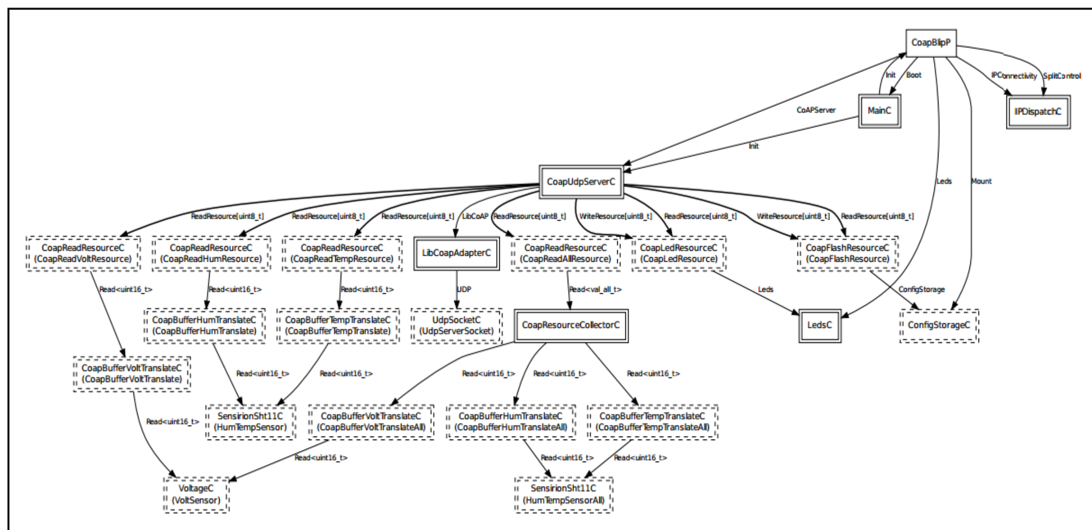


Figura 39: Modelo CoapBlip.

Basándome en el programa *CoapBlip* , que realiza una lectura de datos, temperatura, humedad , luz por medio de las interfaces a componentes correspondientes , mi proyecto utiliza un componente ***CoapUmbralResourceC*** en comparación con *CoapBlip* que utiliza el componente *CoapBufferTranslate* para el cálculo o adaptaciones (*conversiones de valores recogidos a unidades de medida ºC, V y Lux*) de las lecturas de los sensores, mi componente notifica una petición del cliente al servidor por una medio de una interfaz ***PfcInter*** que proporciona el paquete recibido al componente *CoapBlip.*

Una vez en CoapBlip se ejecuta la ***función "comprobación ()"*** que se encarga de copiar los valores del mensaje recibido a la estructura de datos tipo "*destino_t*".

struct destino_t{
unsigned char **sensor**; → tipo de sensor a utilizar (*letras "t", "h", "v" y "l"*).
unsigned char **umbral**; → *umbral que determina el comportamiento una lectura del sensor.*
unsigned char **accion**; → *determina que Leds se encienden al ejecutar el programa.*
unsigned char **direccion_dest[17]**; → *contiene la dirección IPv6 destino.*
uint16_t **lecturas**;} → *aquí se almacena el valor de una lectura del sensor*.

Todas las conversiones de valores recogidos por parte de los sensores se realizan en el mismo programa *CoapBlip*, respetando la concurrencia para el bajo consumo de energía.

El esquema de mi programa consta de tres partes:
Primera parte: el temporizador que comprobará cada segundo los umbrales para determinar el comportamiento de una lectura.

Segunda parte: la recepción de una petición en *CoapUmbralResource* que realizará una notificación y envía el mensaje a CoapBlip para rellenar la estructura *destino_t.*

Tercera parte: son las lecturas por los sensores *SensirionC(), Voltage() y Hamamatsu().*

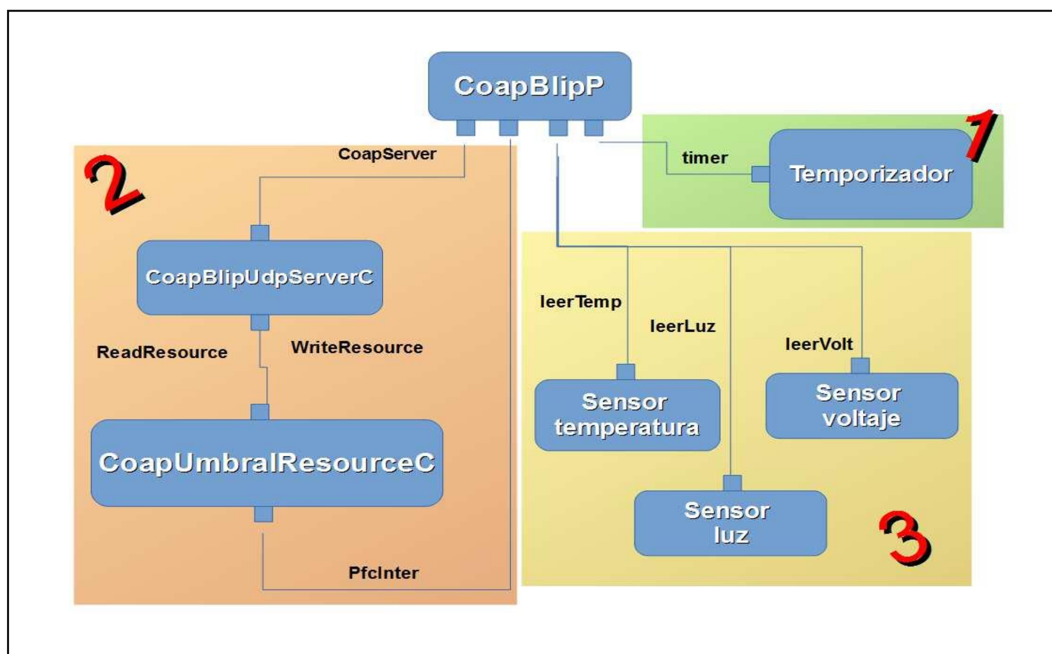


Figura 30 proyecto de fin de carrera.

7.1.- DESCRIPCION Y FUNCIONAMIENTO DE COMPONENTES

Empezaremos con el componente ***CoapUmbralResourse***, donde se destacan dos comandos, lectura y escritura pero el que interesa es el método escritura:

command int WriteResource.put(uint8_t *val, size_t buflen, coap_tid_t id):
Este comando se activa cuando el cliente envía un paquete que es apuntado mediante la variable "val" y su tamaño es *buflen*, luego se realiza la notificación al componente *CoapBliP* por medio de la interfaz ***Pfcinter***, esta interfaz proporciona un comando ***"int aviso"*** que se encarga de enviar el paquete y su tamaño.

El contenido del paquete son parámetros que se ejecutarán en el servidor para la lectura de los sensores.

```
36
37   command int WriteResource.put(uint8_t *val, size_t buflen, coap_tid_t id) {
38     //if (*val < 8) {
39       if (lock == FALSE) {
40         lock = TRUE;
41         temp_id = id;
42         //call Leds.set(*val);
43         call PfcInter.aviso(val,20);
44         post setLedDone();
45         return COAP_SPLITPHASE;
46       } else {
47         return COAP_RESPONSE_500; //503
48       }
49   //}
50      //else {
51       //return COAP_RESPONSE_500;
52     //}
53   }
54 }
```

Figura 31: Código del comando WriteResource.put.

Componente CoapBlip:
Es el componente principal donde está todo el núcleo del programa, se encarga del arranque del programa y de la comprobación del mensaje recibido por el usuario. Aquí se definirá una serie de interfaces que serán utilizadas para conectar los diferentes componentes como el temporizador principal, sensores y el programa cliente que viene por defecto.

La secuencia de arranque empieza en el componente MainC, con comandos que están dentro de otras interfaces como:

> ***Interfaz Init():*** proporciona una interfaz síncrona secuencial que permite la inicialización, ningún componente puede empezar a funcionar hasta que haya terminado la inicialización.
>
> ***Interfaz PfcInter:*** es una interfaz que conecta el componente *UmbralResourse* con *CoapBlipP* para la notificación y envío del mensaje del cliente. Consta de un comando que devuelve un entero, *"command **int aviso**(int *paquete, size_t len)"* que tiene como parámetro un puntero al mensaje y su tamaño, este mensaje será copiado posteriormente a la estructura "*destino_t*".

Dentro de este comando, implementamos una función "***conmprobación(int*mensaje)***" que se encarga de trocear el mensaje en partes y con ellas rellenar la estructura *destino_t*.

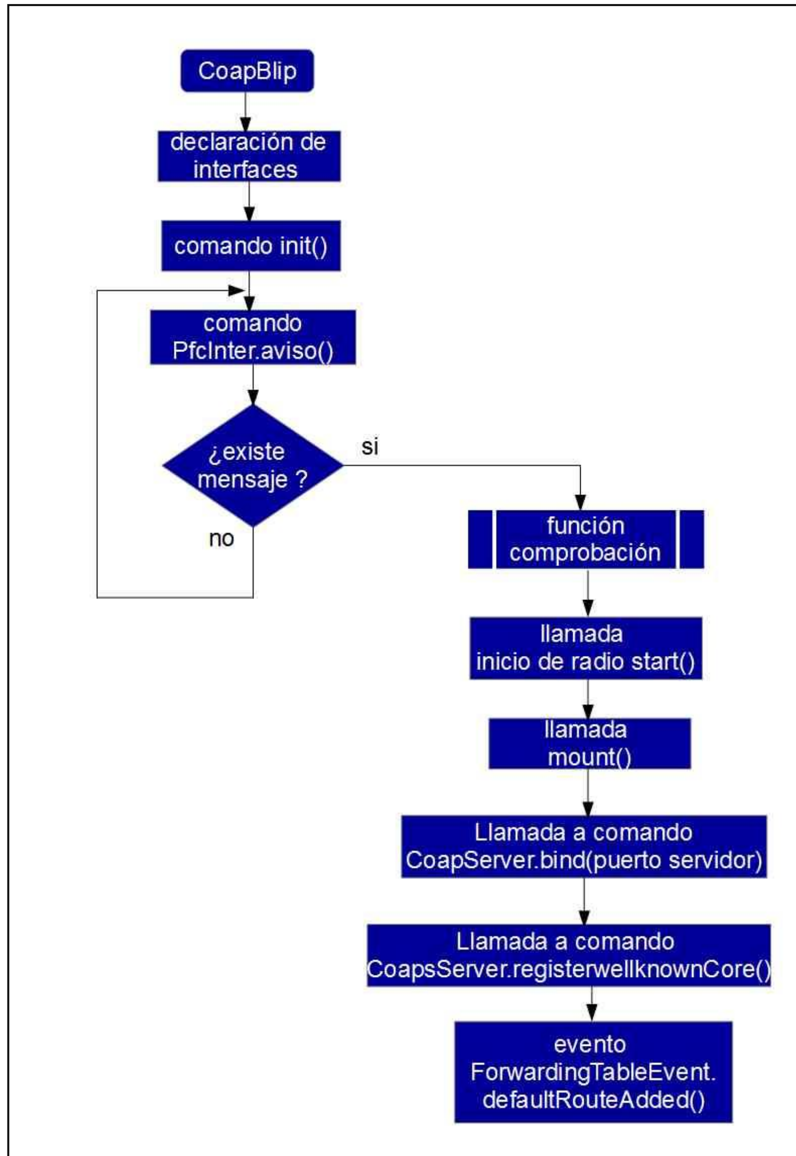


Figura 32: Diagrama de flujo del programa CoapBlip en el proyecto.

Interfaz RadioControl.Start(): se activa el componente *RADIO* para el envío de mensajes.

Interfaz CoapServer.bind(): esta interfaz asigna el puerto al servidor, esta predefinido por el puerto 61616.

RegisterWellKnownCore: aquí es donde las *opciones coap* rellenan la estructura ***resgisterResourse***, cuyos campos especifican la URI, tamaño, tipo de operación y si es solo se puede hacer una lectura o escritura de recurso.

Finalmente tenemos un evento dentro del programa CoapBlipP ***"ForwardingTableEvent.defaultRouteAdded()"*** que consta de una estructura sockaddr_in6 donde uno de sus campos indica la dirección *destino (esa dirección nos proporciona el mensaje recibido anteriormente)*.Lo que hacemos es copiar la dirección destino de la estructura ***destino_t*** al campo dirección de la *estructura sockaddr*, que en este caso es

sa6.sin6_addr.

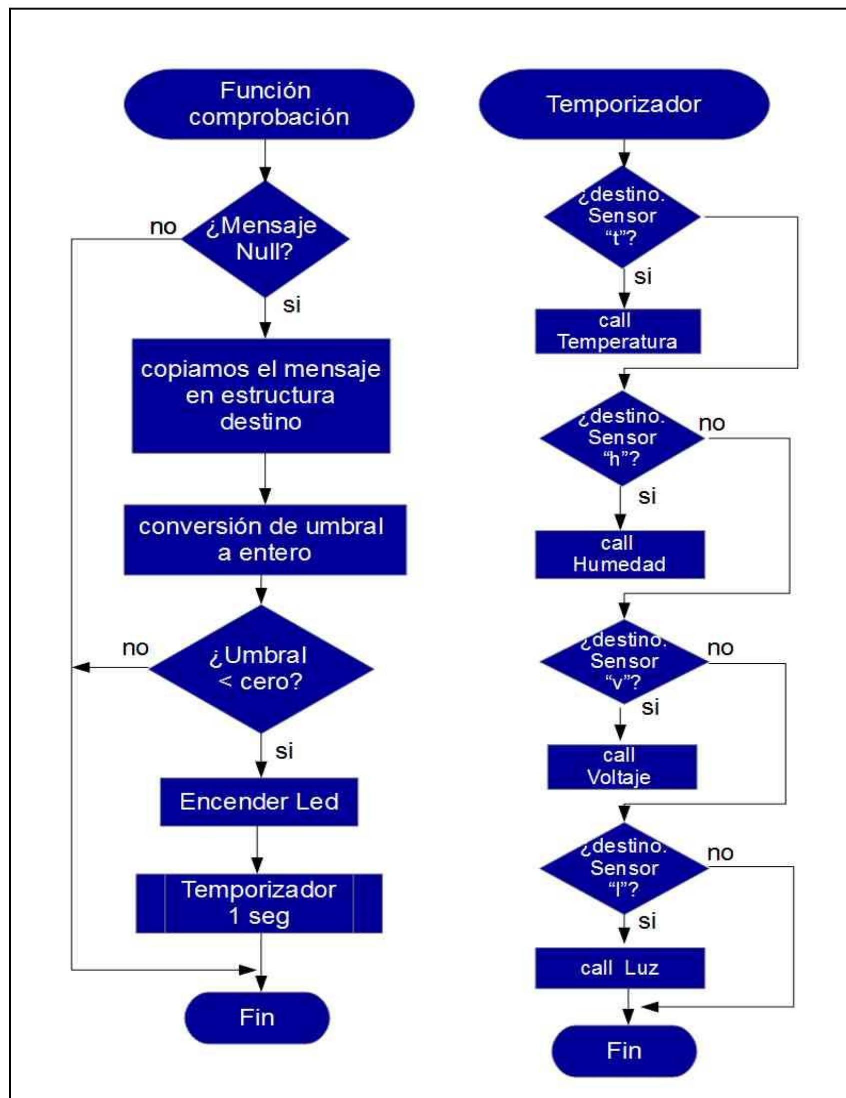


Figura 33: Función comprobación y temporizador general .

Cuando expire el temporizador del programa, se generara el evento *Temporizador.fired()*, se procederá a saber qué tipo de sensor utilizaremos mediante el campo "*destino.sensor"* que llevar una letra que identifica el tipo de sensor.

Cada comprobación tiene una llamada a un comando de una *interfaz relacionada* con un sensor en concreto. Una vez que se active esa llamada, el evento correspondiente se encarga de las conversiones y de guardar el valor del sensor en "*destino.lectura*".

Al utilizar nesC como lenguaje de programación, sabemos que la programación se basa en la creación de componentes que tienen módulos, estos van conectados entre sí en un archivo de *configuración*. La configuración de CoapBlipP entre los *componentes / interfaces* (wiring) se muestra en la siguiente figura:

```
118 #ifdef COAP_RESOURCE_UMBRAL
119         components new CoapUmbralResourceC(KEY_UMB) as CoapUmbralResource;
120         CoapUmbralResource.Leds -> LedsC;
121         CoapUmbralResource.PfcInter -> CoapBlipP;
122         CoapUdpServerC.ReadResource[KEY_UMB] -> CoapUmbralResource.ReadResource;
123         CoapUdpServerC.WriteResource[KEY_UMB] -> CoapUmbralResource.WriteResource;
124         CoapBlipP.leerTemp -> HumTempSensorAll.Temperature;
125         CoapBlipP.leerHum -> HumTempSensorAll.Humidity;
126         CoapBlipP.leerVolt -> VoltSensor.Read;
127         CoapBlipP.leerLuz -> SensorLuz.Read;
128
```

Figura 34: Conexionado de interfaces y componentes.

Observamos claramente las conexiones de las interfaces que utilizamos para comunicar los componentes de los sensores con las interfaces de *CoapBlip* para que la aplicación funcione correctamente.

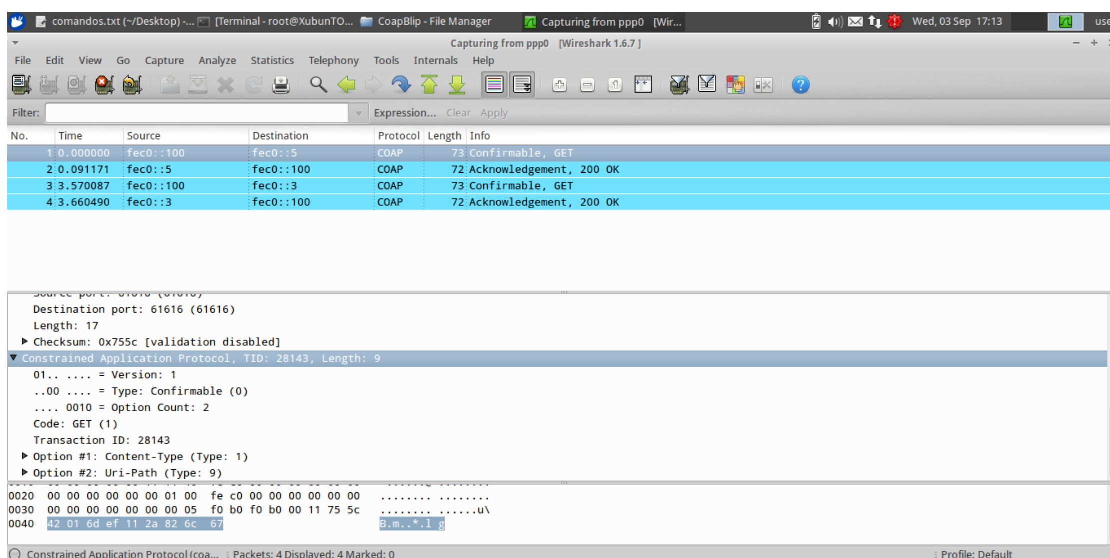


Figura 35: Captura Wireshark del recurso Luz.

En esta captura de Wireshark de una petición GET al *Sensor Luz*, observamos el tipo de protocolo (CoAP), opciones CoAP, payload del paquete, las direcciones de origen y destino y el tipo de mensaje que estamos enviando (*de tipo CONFIRMABLE con su ACK*).

7.2.- CONCLUSIONES Y LINEAS FUTURAS:

Trabajar con esta tecnología es muy interesante, sabemos que está un en crecimiento y que todo lo que se desarrolla está prácticamente dentro del marco de la investigación, los motes TelosB no son sensores comerciales que se pueden comprar para instalarlos en una casa o edificio y empezar a trabajar con ellos. Sin embargo nos sirven para simular resultados que nos pueden ayudar, como en mi caso, a llevarlos al desarrollo de aplicaciones domóticas.

En este proyecto se ha desarrollado una aplicación con redes de sensores que permite la programación del *comportamiento* de los mismos mediante un servicio web a través del protocolo *CoAP*. Gracias a esta programación del *comportamiento* de los nodos se puede definir valores de las lecturas de los sensores (*umbrales*) que al ser sobrepasados desencadenan otras acciones en el mismo nodo o en otros.

En una aplicación domótica esto puede servir, por poner un ejemplo, para abrir o cerrar persianas en función de la temperatura o de la luz en una habitación o bien detectar la presencia de personas y encender las luces antes de que entre a una habitación.

- El consumo de energía es importante, sobre todo por aprovechar las baterías al máximo gracias al modo “sleep” que ahorra energía a más de que son cómodas de cambiar. Otra forma de ahorro de batería es la programación concurrente.

- A la hora de cargar el fichero en los nodos, se ha utilizado un servicio web REST, con las operaciones GET y PUT. Para la tarea de traducción del contenido de los ficheros a CoAP, se ha usado un proxy que se encarga de transformar las peticiones y respuestas entre HTTP y CoAP.

- Me pareció interesante la adaptación de las órdenes que envía el usuario por medio de HTTP que finalmente se transforma en CoAP utilizando protocoles como UDP como medio de transporte.

Como líneas futuras se pueden mencionar las siguientes:

- Se puede trabajar con otro tipo de aplicaciones no sólo para ordenadores sino también para teléfonos “Android” permitiendo una mejor comodidad sin la necesidad de utilizar un pc para el envío de órdenes.

- Ampliar los recursos de la lista que nos ofrece ya los motes TelosB como por ejemplo alguno que gestione el ruido de una habitación.

- Notificaciones y gráfica de seguimiento de todas las lecturas para tener un control semanal de la temperatura y luz de una casa que se puede aplicar a investigar los costos que se genera cunado se active la calefacción o aire acondicionado, en caso de la temperatura y los vatios consumidos en caso de la luz.

- El enrutamiento de los motes ha sido probado con IPs privadas pero como línea futura se puede ampliar al uso de IPs públicas y que se puedan acceder de forma remota para ejecutar la aplicación, claro está que para ello se tomarán en cuenta protocolos de seguridad para que su uso remoto sea fiable.

Me ha gustado mucho trabajar con estos motes, es cierto que los resultados no son tan tangibles al ser un programa informático que gestiona órdenes de usuarios, pero al ser una tecnología que está en continuo crecimiento he aprendido mucho como telemático a desenvolverme en un lenguaje de programación que desconocía y ver las diversas aplicaciones y capacidades que puede tener un dispositivo de recursos limitados.

Bibliografía:

[1] C. Bormann Z. Shelby. 6LoWPAN: The Wireless Embedded Internet. 1th ed.; John Wiley & Sons Ltd: Chichester, UK, 2009.

[2] S. Deering R. Hinden. IP Version 6 Addressing Architecture. Available online at: http://tools.ietf.org/html/rfc4291. February 2006. [Last access: 30 September 2011].

[3] http://www.sase.com.ar/2013/files/2013/09/SASE2013-6LOWPAN-A-Diedrichs.pdf

[4] The Wireless Embedded Internet book

[5] W3C. Web Services Description Language (WSDL) 1.1. Web Services Description Language (WSDL) 1.1. Available online at: http://www.w3.org/TR/wsdl. [Last access: 30 September 2011].

[6] R.T Fielding. Architectural Styles and the Design of Network-based Software Architectures. PhD Tesis, University of California. Irvine, 2000.

[7] http://users.dsic.upv.es/~rnavarro/NewWeb/docs/RestVsWebServices.pdf

[8] http://information-technology-forum.blogspot.com.es/2009/06/technical-interoperability-of- disparate.html

[9] https://tools.ietf.org/html/draft-ietf-core-coap-18#page-9

[10] https://tools.ietf.org/html/draft-ietf-core-link-format-14

[11] Wireless Sensor Networks. F. L. LEWIS. Smart Environments: Technologies, Protocols, and Applications Ed. D.J. Cook and S.K. Das, John Wiley, New York, 2004.
Web: http://arri.uta.edu/acs/networks/WirelessSensorNetChap04.pdf