Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Proyecto Final de Máster

Implementación de Protocolos de Transporte en Redes de Sensores

AUTOR: Ernesto J. García Davis

TUTORA: Anna Calveras Augé

FECHA: 23 de Julio de 2009.

AGRADECIMIENTOS

Primero a Dios que me ha permitido llegar hasta aquí, a mi mamá, mi Tía Vilma y mi hermano Gustavo quienes siempre han estado conmigo a pesar de la distancia, a mi esposa Anabelle que siempre ha sido mi apoyo y me ha dado ánimo para seguir adelante.

También agradecer a mi tutora Anna Calveras, por ser mi guía y una valiosa ayuda para llevar a cabo este proyecto.

Un especial agradecimiento a mi compañero de batalla Toni Adame, que me ha brindado su valiosa ayuda y colaboración en el desarrollo del código de este proyecto.

Por último pero no menos importante, a mis amigos de Máster que han compartido conmigo las penas y glorias a lo largo de estos dos años.

CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

La evolución de las tecnologías inalámbricas y el desarrollo de la nanotecnología han permitido la integración del mundo físico con el mundo digital, de manera que prácticamente cualquier objeto que conocemos tiene la capacidad de proporcionar información de los eventos que suceden en su entorno a través del medio inalámbrico, lo que generalmente se conoce como el “Internet de las cosas”.

En este sentido, las redes de sensores inalámbricas (WSN) son las que mayormente han aprovechado esta evolución. Este tipo de redes está formada por nodos sensores los cuales poseen características particulares, como lo son: capacidad limitada de memoria, limitado recursos de energía y procesamiento, y limitado ancho de banda, las cuales las hacen diferentes a las redes inalámbricas tradicionales y por lo tanto traen consigo nuevos retos.

Por otro lado, cada vez más las aplicaciones de este tipo de redes, requieren la interconexión hacia redes externas con el fin de gestionar los datos medidos y controlar el comportamiento de los nodos sensores de forma remota. Por ejemplo, se puede pensar en aquellas aplicaciones donde el despliegue de una red de sensores inalámbrica puede ayudar a proporcionar información valiosa en lugares de difícil acceso o de alto riesgo, como por ejemplo, detección de actividad volcánica, vigilancia de las tropas enemigas entre otras.

Además, este tipo de aplicaciones requieren que la información medida pueda ser entregada de forma fiable al destino, convirtiéndose en un factor crítico a la hora de diseñar y desplegar una WSN.

Esta necesidad de interconexión y la necesidad de proporcionar un transporte fiable de los datos, impone nuevos retos a los protocolos tradicionales. En el caso de protocolo de transporte fiable, como TCP, éste presenta una serie de problemas para trabajar en redes inalámbricas, en términos de rendimiento y eficiencia energética y que por lo tanto deben ser resueltas para que este protocolo sea adecuado para funcionar en las WSN.

Es por esto, que los objetivos principales de este proyecto son: primeramente realizar un estudio de las limitaciones que poseen los protocolos de transportes tradicionales para trabajar en las WSN, y los requisitos que deberían cumplir los protocolos de transporte para ser eficientes en este tipo de redes. Además analizaremos brevemente

las propuestas de protocolos de transporte existentes para WSN, haciendo énfasis en los protocolos de transporte basados en TCP. En este sentido, y como segundo objetivo, realizaremos la implementación del protocolo TSS (TCP Support for Sensor Networks), desarrollado por Adam Dunkels y el equipo de desarrollo del SICS (Sweden Institute Computer Science), el cual proporciona mejora a los mecanismos de fiabilidad en la entrega de mensajes, recuperación de errores y control de congestión de TCP, además de disminuir el consumo de energía de los nodos sensores, factor importante en este tipo de redes.

Esta implementación será realizada sobre un escenario real, utilizando la plataforma de hardware TelosB, y con la cual se pretende validar los objetivos, funcionamiento y ventajas que se han presentado en el estudio realizado por los autores de TSS. Además de presentar los retos que ha conllevado esta implementación en un escenario real, y realizar un análisis de los resultados obtenidos y exponer como líneas futuras de investigación aquellos temas que quedarían pendientes de resolver o mejorar en este protocolo.

El presente trabajo se estructura de la siguiente forma: el capítulo 1 se presenta una breve introducción a las redes inalámbricas de sensores en el que se identifican sus características y limitaciones, la evolución de su arquitectura de protocolos de red, además un análisis de los mecanismos existente para interconectar las WSN a las redes externas y por último una breve descripción de las pilas IP existentes que permiten una conexión directa a estas redes externas. En el capítulo 2 se presenta un estado del arte de los protocolos de transporte para redes WSN, en el que se exponen y se estudian las limitaciones que los protocolos de transporte tradicionales, como TCP y UDP enfrentan en las WSN, y adicionalmente, se realiza un análisis y discusión de los requisitos que deben cumplir los protocolos de transporte para operar en las WSN, y por último, se realiza una breve descripción de los protocolos de transportes existentes en las WSN identificando sus funcionamiento y características más importantes. En el capítulo 3, es donde se realiza un análisis detallado y exhaustivo del funcionamiento del protocolo TSS implementado en este proyecto, y se presentan y se discuten los resultados obtenidos, y los aportes particulares que se tuvieron que realizar para resolver los retos que conllevó la implementación de TSS en las plataformas TelosB. Finalmente en el capítulo 4 se exponen las conclusiones basadas en los resultados obtenidos y se exponen los aspectos que son susceptibles de estudio e investigación futura, producto de los hallazgos encontrados en la implementación de TSS.

1. REDES DE SENSORES INALÁMBRICAS

Los avances tecnológicos en los sistemas micro-electromecánicos (por sus siglas en inglés MEMS) y en las comunicaciones inalámbricas han permitido el desarrollo y despliegue de las redes de sensores inalámbricas (WSN) las cuales superan las capacidades que hasta el momento proporcionaban las redes de sensores tradicionales.

Las WSN están formadas por un conjunto de nodos sensores conectados a través de tecnologías inalámbricas y cuya finalidad es la detección o estimación precisa de las características de eventos obtenidas de la información colectiva proporcionada por los nodos sensores **(ver figura 1.1)**.

La detección o estimación de un evento y su correspondiente transmisión se realiza a través del trabajo colectivo de todos o de cierta parte de los nodos sensores en la red, lo que disminuye la cantidad de energía y costo computacional utilizado.

En este tipo de redes, cada sensor se convierte en un nodo con capacidad de transmitir, a través del medio inalámbrico, los datos obtenidos de su entorno a un punto central donde son procesados y gestionados.

Aunque existen similitudes con otros tipos de redes inalámbricas como las tradicionales redes ad hoc inalámbricas, como por ejemplo: la comunicación se realiza a través de un medio inalámbrico compartido y la ausencia de una infraestructura de red, así también existen claras diferencias entre estas. La principal diferencia podría ser que las redes inalámbricas ad hoc generalmente son construidas con el propósito de extender la comunicación en entornos con limitada cobertura desde un punto central de la red (como una estación base), en cambio las WSN son implementadas para la consecución de un objetivo específico de una aplicación por medio de un trabajo colaborativo entre todos los nodos sensores. Otra diferencia es que el número de nodos sensores y la densidad que puede existir en una red puede ser de varios órdenes de magnitud superior a la de los nodos en redes ad hoc. Además, los nodos sensores poseen limitaciones tanto en la capacidad de comunicación, energía, memoria y procesamiento, aspectos que los nodos en las redes ad hoc no son limitantes.

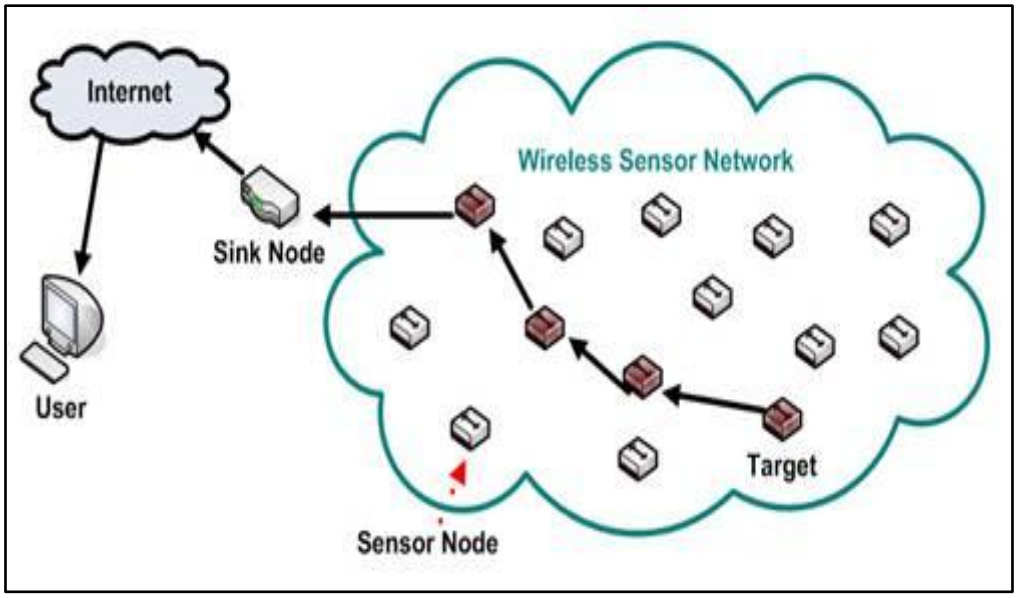

Figura 1.1Típica Arquitectura de red de una WSN
(Fuente:http://www.wireless-sensors.com.cn/ENG/product_BaseStation.htm)

1.1 Características y Limitaciones

Las redes de sensores inalámbricas tienen características que las convierten en un caso particular de los demás tipos de redes inalámbricas existentes las cuales pueden ser:

1. **Topología de red única:** los nodos en estas redes pueden formar topologías muy distintas dependiendo de las condiciones del entorno en que se desplieguen. Por ejemplo en aplicaciones militares los nodos pueden ser distribuidos a lo largo de un área montañosa o irregular y gracias a la capacidad de auto-organización que poseen los nodos pueden crear generalmente topologías de árbol distribuida, la cual además puede ser plana o jerárquica. Generalmente el nodo sumidero (sink) se convierte en la raíz de la topología, figura 1.2a el cual se encarga de la recolección de los datos y actúa como "Gateway" hacia las redes externas. Inclusive, la topología puede ser dinámica debido a que las condiciones de los enlaces y el estado de los nodos pueden variar con el tiempo y formar una nueva topología como se muestra en la figura 1.2b.

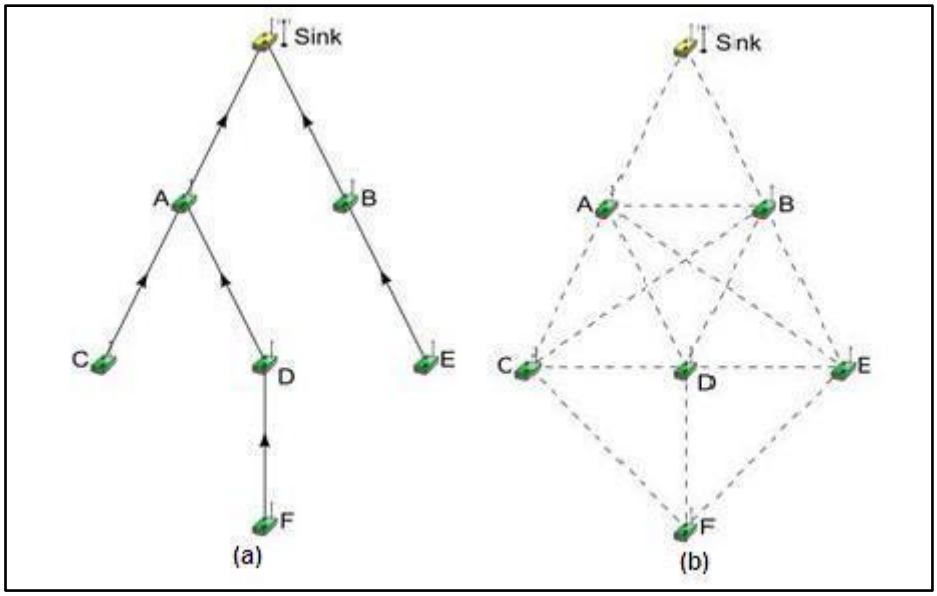

Figura 1.2Topología de árbol distribuida
***(Fuente:** Opportunistic Routing in Wireless Sensor Networks*
*http://lcavwww.epfl.ch/research/topics/opportunistic_routing.html**)***

2. **Diversidad de aplicaciones:** estas redes pueden ser usadas en diferentes entornos brindando soporte a diversas aplicaciones que van desde la monitorización del ambiente, el rastreo y localización de objetos hasta la videovigilancia. Estas aplicaciones demandan diferentes requerimientos en términos de calidad de servicio (QoS) y fiabilidad.

3. **Características de Tráfico**: en las WSN el tráfico principal generalmente fluye en sentido ascendente (upstream) desde los nodos sensores hasta el nodo sumidero (sink), aunque ocasionalmente el "sink" puede generar tráfico en sentido contrario (dowstream) para realizar tareas de gestión y control. En el caso de tráfico ascendente, se realiza una comunicación de muchos a uno. Además, dependiendo de la aplicación el tráfico puede ser entregado al "sink" de forma periódica, cada vez que ocurra un evento, de forma continua en tiempo real, o cualquier combinación de éstas.

4. **Recursos limitados**: los nodos sensores poseen recursos limitados que incluyen una baja capacidad de procesamiento computacional, reducida cantidad de memoria, limitado ancho de banda del orden de los Kbps y una limitada cantidad de energía proporcionada por su batería generalmente no recargable. Actualmente, dependiendo de la aplicación pueden utilizarse métodos externos [1] para alimentar las baterías de los sensores.

5. **Reducido tamaño de mensajes:** usualmente los mensajes en estas redes son reducidos en comparación con las redes tradicionales. Como consecuencia de esto generalmente no es necesaria la fragmentación de los mismos.

1.2 Aplicaciones

Las redes de sensores inalámbricas abarcan un gran abanico de aplicaciones en diversas áreas. Cada una de estas aplicaciones tiene sus requerimientos particulares para que puedan funcionar adecuadamente. A continuación presentamos una breve descripción de algunas de estas aplicaciones de acuerdo al área involucrada:

Transporte:

- **Sistemas Inteligentes de Transporte:** un conjunto de sensores instalados en diferentes partes de un vehículo puede proporcionar al conductor información valiosa de su entorno, como por ejemplo las condiciones del clima, condiciones de la carretera, **(ver figura 1.3)**, de esta manera el conductor puede adaptar manualmente la conducción del vehículo. a su entorno. También los sensores pueden activar automáticamente mecanismos para salvaguardar la vida del conductor, por ejemplo activar el sistema de frenos, el sistema de cinturón de seguridad y las bolsas de aire del vehículo.

Figura 1.3 Sistema de Prevención de colisión y congestión de tráfico de vehículos (Fuente: http://www.intomobile.com/2007/04/17/nissan-developing-test-intelligent-transportation-system-mobile-phones-prevents-autopedestrian-accidents.html)

Militares:

- **Sistemas de detección de minas:** una red de sensores puede ser utilizada para la detección y remoción o desactivación de minas en un campo de enemigo **(Ver figura 1.4)**. De esta manera la WSN permite reducir el riesgo de daños del personal involucrado en esta tarea.

Figura 1.4 Sistema de detección de minas en territorio hostil (Fuente:http://nishantha4u.blogspot.com/)

Sanitarias:

- **Sistemas de Monitorización de Signos vitales en Tiempo Real:** El despliegue de bio-sensores en el cuerpo de un paciente permite el chequeo de sus signos vitales del paciente sin importar en qué lugar del hospital se encuentre. Debido a la larga vida de los dispositivos se puede utilizar los sensores inclusive en implantes.

- **Cuidado de la salud:** dentro del hogar se dotan dispositivos con botones de alarma que pueden ser usadas por personas ancianas o débiles en caso de emergencia.

Ambientales:

- **Situaciones de Emergencia:** una WSN puede ser utilizada para monitorizar el nivel de agua de los ríos en lugares propensos a inundaciones, los movimientos sísmicos o las erupciones volcánicas **(ver figura 1.5)** de manera

que puedan notificar con antelación a los servicios de emergencia para tomar las medidas adecuadas que salvaguarden la vida humana.

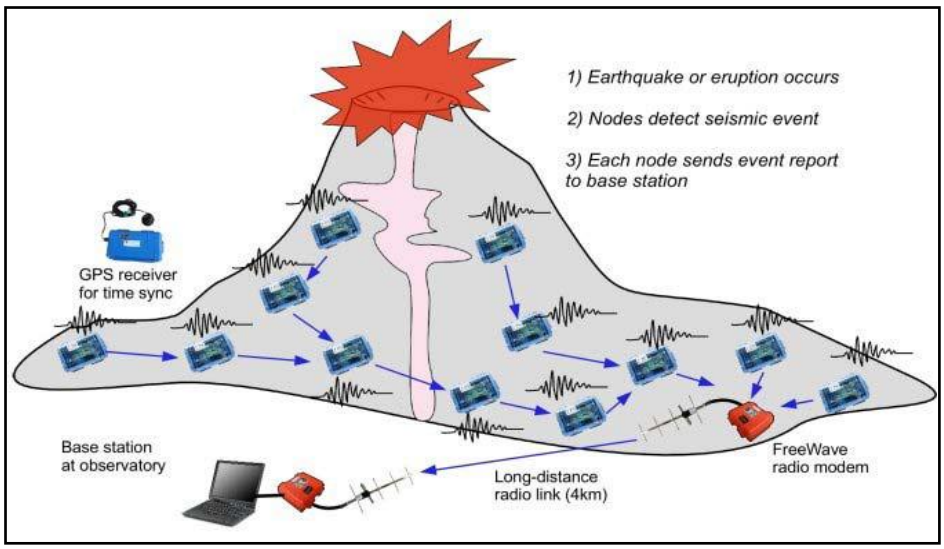

Figura 1.5 Despliegue de una WSN para la detección de actividad volcánica (Fuente: Ubiquitous Sensor Networks (USN), ITU-T Technology Watch Briefing Report Series, No. 4 February 2008)

1.3 Evolución de la arquitectura de protocolos para WSN

Para lograr la comunicación entre los nodos sensores a través del medio inalámbrico, se fueron desarrollando diferentes protocolos que fueron evolucionando con el fin de adaptarse a las características y limitaciones propias de las WSN.

En este apartado describimos los diferentes protocolos que se han ido desarrollando de acuerdo a los diferentes requerimientos que plantean las WSN para su funcionamiento.

1.3.1 Protocolo IEEE 802.15.4

Este estándar surgió en el año 2003 bajo el grupo de trabajo 802.15.4 (LR-WPAN, *Low Rate Wireless Personal Area Network*) de la IEEE, y fue producto de la necesidad de crear un nuevo estándar para redes inalámbricas de bajo consumo y de bajo costo (como las WSN) para aplicaciones domóticas e industriales debido a que se consideraba que las prestaciones de las redes inalámbricas como la 802.11 (WLAN: Wireless Local Area Network) eran muy caras y excesivas para este tipo de aplicaciones.

Las características más importantes del estándar IEEE 802.15.4 son la flexibilidad de la red, bajo costo y bajo consumo de energía; este estándar se puede utilizar para

muchas aplicaciones domóticas, civiles, industriales, médicas, entre muchas otras, donde se requieren una baja tasa de transmisión de datos.

Este estándar se encarga de definir las funcionalidades de la capa física y de control de acceso al medio.

1. **Capa Física (PHY):** consiste en dos capas físicas operando en dos rangos de frecuencias separadas: 868/915 MHz and 2.4 GHz las cuales ofrecen tasas de transmisión de datos de 20/40 kbps y hasta 250 kbps respectivamente.

2. **Capa de Control de Acceso al Medio (MAC):** tiene la responsabilidad de controlar el acceso al canal radio utilizando el CSMA/CA. La capa MAC proporciona soporte para la transmisión de tramas beacon, sincronización de red y transmisión confiable usando CRC y retransmisiones.

1.3.2 Protocolo Zigbee

Este es un protocolo desarrollado por la Zigbee Alliance para las aplicaciones que requieren comunicaciones seguras con baja tasa de envío de datos y maximización de la vida útil de sus baterías.

Zigbee utiliza las capas del estándar IEEE 802.15.4 y agrega otras de nivel superior lo que permiten obtener una pila de protocolos **(ver figura 1.6)** con funcionalidades bastantes completas, lo que lo convierte en un protocolo viable para funcionar en las WSN.
Estas capas son;

1. **Capa de Red (NWK):** se encarga de enviar y recibir datos hacia y desde la capa de aplicación. Además ejecuta las tareas de asociación y des asociación de la red, aplicación de la seguridad y asignación del direccionamiento de hasta 65,536 dispositivos. Esta limitante en el direccionamiento puede traer problemas de escalabilidad en aplicaciones de WSN, como por ejemplo, en la detección de incendios forestales, donde se necesita una alta densidad de estos nodos para esta finalidad.

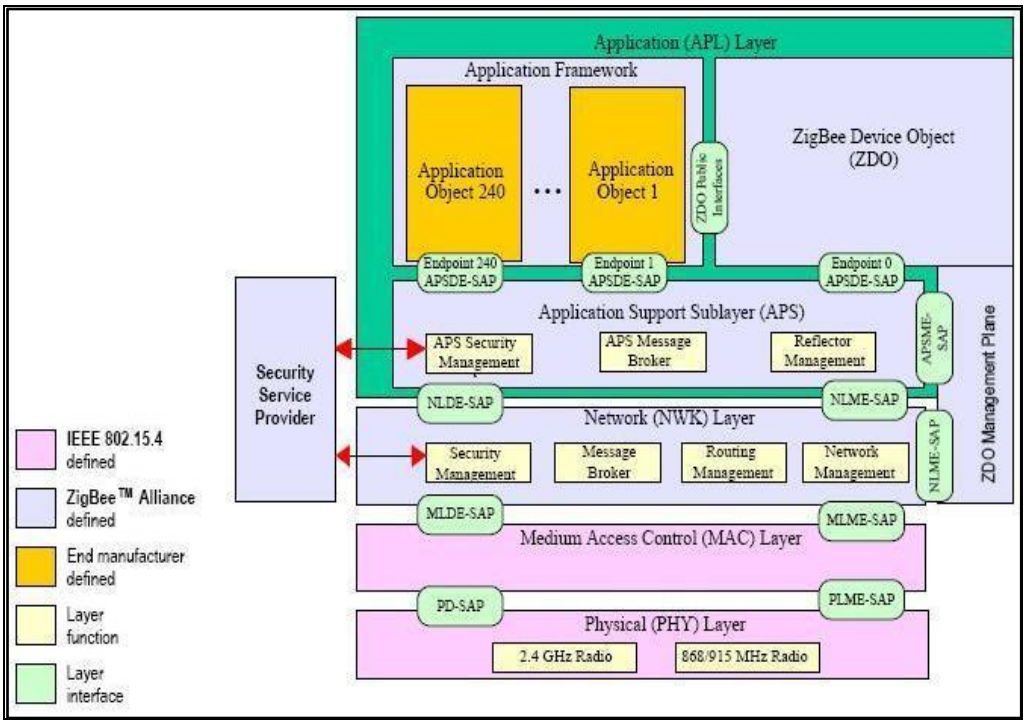

Figura 1.6 Pila de protocolo Zigbee
(Fuente: http://zigbee.egloos.com/)

2. **Capa de Aplicación (APL):** está formada por la subcapa Application Support (APS), la Zigbee Device Object (ZDO) y la Application Object definidas por el fabricante e implementadas en un dispositivo dado.

   La APS se encarga de establecer la comunicación entre dos o más dispositivos según sus servicios y necesidades; y enviar mensajes entre ellos.

   Además, esta subcapa proporciona dos interfaces: APSME-SAP(APS Management Entity Service Access Point) y la APSDE-SAP(APS Data Entity Service Access Point). La primera proporciona los servicios de descubrimiento y unión de dispositivos y mantiene una base de datos del manejo de objetos mientras que la última, provee el servicio de transmisión de datos entre dos o más dispositivos localizados en la misma red

   La subcapa ZDO se encarga de definir el rol del dispositivo dentro de la red (ya sea de coordinador o de dispositivo final), iniciando o respondiendo a las peticiones y estableciendo una conexión segura entre los dispositivos de la red.

   Finalmente, la subcapa Application Object brinda soporte a las aplicaciones de los fabricantes que se ejecutan sobre la pila de protocolo Zigbee a través de

elementos llamados objetos de aplicación. Cada objeto de aplicación es direccionado a través de su correspondiente identificador de "endpoint" numerados de 1 al 240.

La unión de un identificador de "endpoint" con la dirección del dispositivo proporciona una manera uniforme y única de direccionamiento individual de los objetos de aplicación en la red Zigbee.

1.3.2.1 Tipos de dispositivos Zigbee/802.15.4

Una red típica Zigbee, consiste de un nodo coordinador, uno o más dispositivos finales y opcionalmente uno o más routers como se muestra en la figura 1.7

- El nodo coordinador es un dispositivo con funcionalidades completas (FFD: Full Function Device), responsable por el funcionamiento interno de la red Zigbee. Un nodo coordinador configura una red con un identificador PAN (Personal Area Network) dado, al cual los dispositivos finales pueden unirse.

- Estos dispositivos finales son generalmente dispositivos con funcionalidades reducidas (RFD: Reduce Function Device) para lograr una implementación lo más barata posible.

- Los routers pueden ser usados por el nodo coordinador en la PAN como nodos intermediarios, de esta manera permiten que la red se pueda extender más allá del rango de señal radio del nodo coordinador.

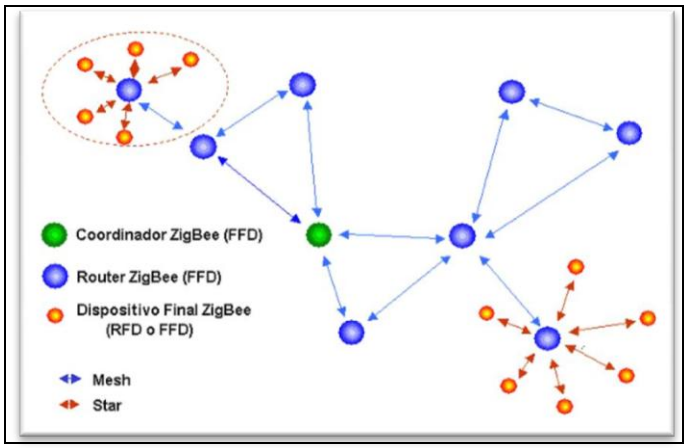

Figura 1.7 Arquitectura de red Zigbee
(Fuente: http://www.zigbee.org/imwp/idms/popups/pop_download.asp?contentID=5162)

Generalmente, los nodos coordinadores y routers poseen alimentación principal y en la mayoría de los casos sus radios están encendidos todo el tiempo. Los dispositivos finales, por otro lado, pueden ser diseñados con muy bajo ciclos de procesamiento, permitiéndoles una expectativa de vida más larga cuando son alimentados por batería.

1.3.3 Estándar 6LowPAN (IPv6 over Low Power Wireless Personal Area Network)

Aunque la combinación del protocolo Zigbee sobre IEEE 802.15.4 proporcionan una arquitectura de protocolo bastante completa para trabajar en WSN, Zigbee presenta el inconveniente de ser incompatible a la hora de lograr una interconexión con redes externas debido a que su esquema de direccionamiento no es compatible con el protocolo IP lo que impide una conexión directa entre los dispositivos de estas redes.

Por otro lado, el desarrollo del protocolo IP versión 6 ha demostrado ofrecer mayores y mejores ventajas que IPv4, como por ejemplo, un espacio de direccionamiento mucho más escalable, autoconfiguración sin estado, entre otras, y el cual pretende convertirse en el protocolo estándar para lograr la interconexión directa de las WSN con las redes externas, resolviendo de esta manera la limitante que posee Zigbee en este aspecto.

Sin embargo el uso de IPv6 en este tipo de redes impone ciertos requerimientos como el incremento de tamaño de las direcciones IPv6 y del MTU en 1280 bytes.

Por tal motivo, se dio origen a 6LowPAN con el fin de eliminar los inconvenientes que tenían los paquetes IPv6 para ser transportados sobre las redes inalámbricas de bajo consumo (LowPAN), específicamente en las redes basadas en IEEE 802.15.4.

Para lograr este objetivo, se definió una capa de adaptación, **(Ver Figura 1.8)**, que tratara los requerimientos impuestos por IPv6 como el incremento de tamaño de las direcciones IPv6 y del MTU en 1280 bytes y se crearon un conjunto de encabezados que permitieron una codificación eficiente de los encabezados y direcciones IPv6 dentro de encabezados comprimidos más pequeños hasta alcanzar en algunas ocasiones los 4 bytes.

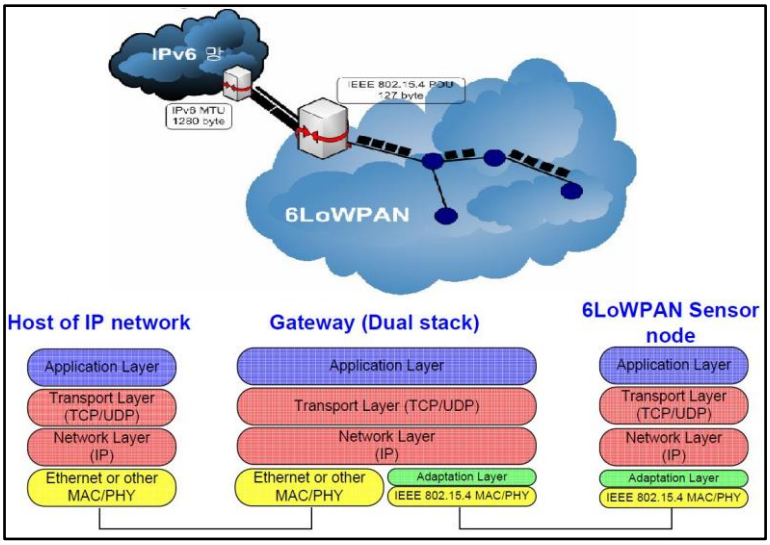

Figura 1.8 Transporte de IPv6 sobre 802.15.4 utilizando 6lowPAN
(Fuente: Interworking between Sensor Networks and IPv6 Network, Department of Computer Engineering, Kyung Hee University)

En la figura 1.9 se presenta la forma en que una red Zigbee podría conectarse indirectamente a una red externa, y la forma como una WSN IPv6 con 6lowPAN logra esta interconexión directa con la red externa.

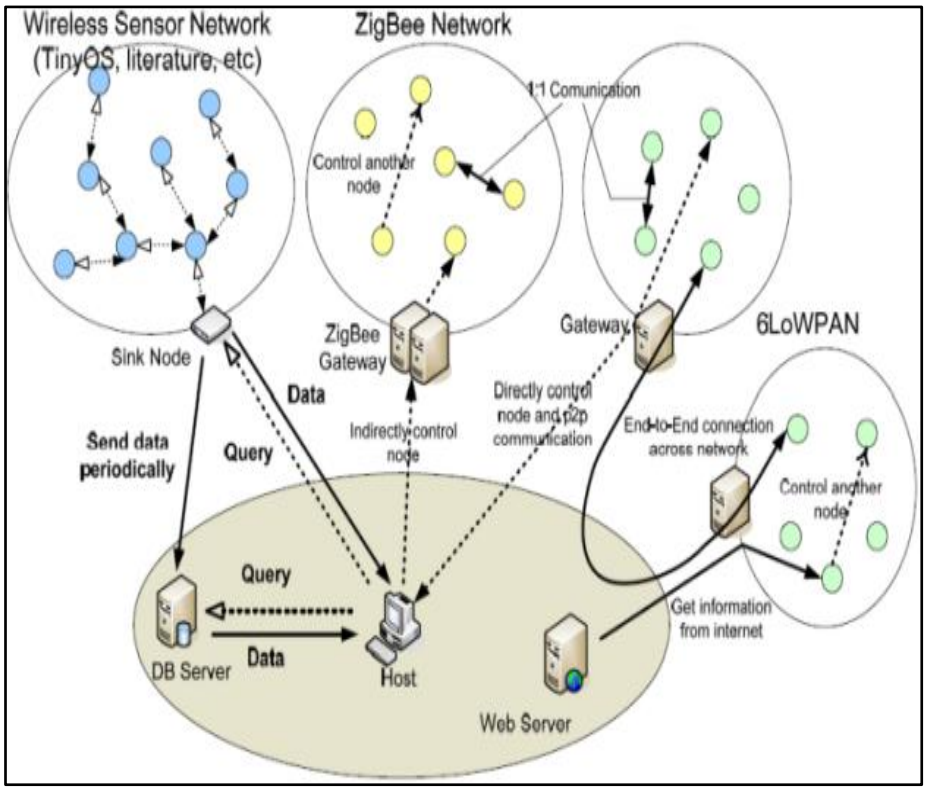

Figura 1.9. Esquema de interconexión de una WSN
(Fuente: Interworking between Sensor Networks and IPv6 Network, Department of Computer Engineering, Kyung Hee University)

1.3.4 Protocolo CAP (Compact Application Protocol)

Esta iniciativa es una combinación de las bondades que ofrecen ambas protocolos antes mencionados, por un lado IPv6 y su versión para bajo consumo 6lowpan que es el estándar de facto con miras a lograr la interconexión con el mundo Internet y por otro lado Zigbee que ha logrado una amplia estandarización de los perfiles necesarios para operar con la muchos fabricantes del mundo de la electrónica y los dispositivos integrados.

Hasta el momento es una propuesta borrador que ha sido sometida a la IEEE para su evaluación, la cual describe una adaptación UDP/IP del IEEE 802.15.4 basado en la capa de aplicación del protocolo Zigbee **(ver figura 1.10)**. Esto permitiría que los hosts IP puedan comunicarse usando los perfiles de aplicación de Zigbee y modelos de datos descritos por el protocolo sobre una amplia gama de dispositivos de diversos fabricantes.

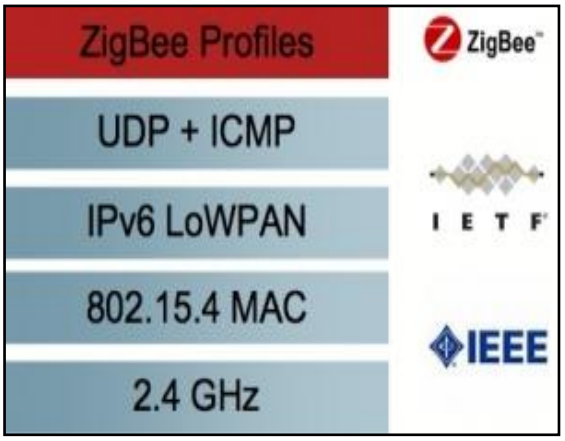

Figura 1.10 Propuesta de la Arquitectura de protocolo CAP
(Fuente http://zachshelby.org/2009/02/20/zigbee-vs-ipv6/)

Este protocolo sería una modificación del protocolo de aplicación Zigbee llamado CAP (Compact Application Protocol) lo que proporcionará una completa pila de perfiles de aplicación, intercambio de datos, operaciones de asociación, protocolos de seguridad y descubrimiento de host con protocolo IP y dispositivos integrados.

1.4 Interconexión con redes TCP/IP

En el pasado era común observar que las aplicaciones de las redes de sensores trabajaran de manera aislada, sin embargo en la actualidad la aplicación de redes de sensores en áreas como la videovigilancia, el rastreo de objetos, medición de parámetros ambientales, por ejemplo en campos de cultivos, conducen cada vez más a la necesidad de interconectar estas redes a otros dominios ya sean redes de área

local, o a Internet, permitiendo de esta manera que las tareas de gestión y control de los datos suministrados por los nodos sensores se puedan realizar de forma remota.

Como una consecuencia de la operación aislada en que se encontraban las redes de sensores en el pasado, los fabricantes desarrollaron protocolos especializados y acorde con las limitaciones particulares de los dispositivos de este tipo de redes, esto trajo como consecuencia que hoy en día exista una incompatibilidad entre estos protocolos (como por ejemplo Zigbee) y los utilizados en la mayoría de las redes, como es el caso del protocolo TCP/IP utilizado como estándar de-facto en Internet.

Para resolver estos inconvenientes se han propuesto varios mecanismos los cuales pueden ser clasificados en: los basados en Proxy, los enfocados en la arquitectura DTN (Delay Tolerant Network) y los basados en recubrimiento (overlay networks).

1.4.1 Mecanismos basados en Proxy

Esta es la alternativa más sencilla y consiste en colocar entre la red de sensores y la red TCP/IP un servidor proxy el cual mantendrá la separación entre ambas redes y realizará las operaciones de traducción de un protocolo a otro **(ver figura 1.11)**. De esta manera ninguno de los dos tipos de redes necesita modificar su funcionamiento logrando así la interconexión entre redes heterogéneas.

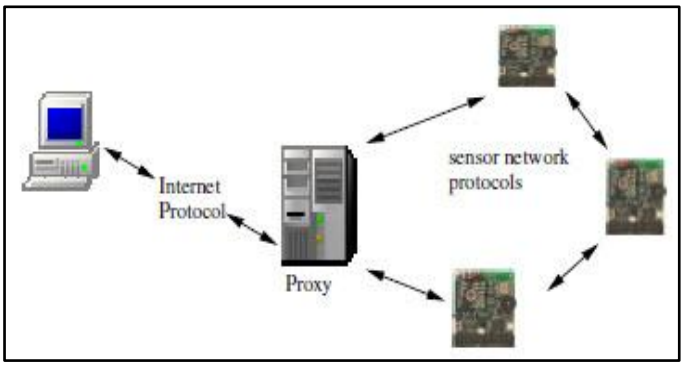

Figura 1.11 Esquema de Interconexión basado en Proxy
(Fuente: Connecting Wireless Sensornets with TCP/IP Networks)

A pesar de lo simple y fácil que puede resultar esta alternativa, cabe señalar que la ubicación del servidor proxy rompe la comunicación extremo a extremo entre ambas redes y por lo tanto trae como desventaja que se cree un solo punto de fallo, es decir, toda la comunicación entre ambas redes dependerá de lo robusto y estable que sea el funcionamiento del servidor proxy. Aunque existen alternativas que sugieren la

implementación de servidores proxy de respaldo, estos adicionarían complejidad al sistema.

También se debe considerar que existe un servidor proxy específico para cada tipo de tarea a realizar o para un conjunto de protocolos en particular, por lo tanto dependiendo del número de aplicaciones que deseamos ejecutar así mismo se incrementará la cantidad de servidores proxy que se necesitarán lo que adiciona complejidad a esta alternativa.

1.4.2 Mecanismos basados en Arquitectura DTN.

Se utiliza la arquitectura DTN que consiste en un conjunto de regiones las cuales comparten una capa en común llamada “bundle” o capa de mensaje que opera entre la capa de aplicación y la capa de transporte

La figura 1.12 describe las dos regiones DTN para una interconexión de una red de sensores y una red TCP/IP. La región 1 (zona 1) está funcionando con un protocolo especializado para redes de sensores y la región 2 (zona 2) está utilizando la pila de protocolo TCP/IP. Un DTN “Gateway” se ubica entre la red de sensores y la red TCIP/IP muy similar a un servidor proxy, sin embargo la arquitectura DTN es mucho más general que el mecanismo de proxy ya que no se necesita de un DTN “Gateway” específico para cada aplicación o protocolo que se desea utilizar, y esto se debe a que la capa “bundle” es compartida entre los dos tipos de redes.

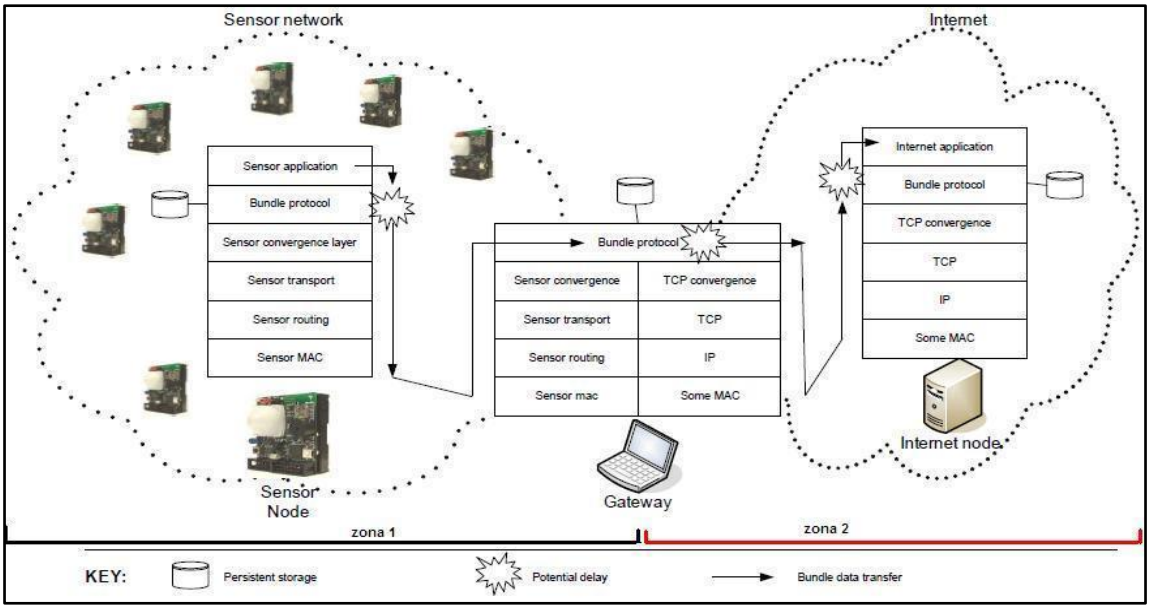

Figura 1.12 Interconexión basada en Arquitectura DTN
(Fuente: Delay Tolerant Networking for Sensor Networks)

Además esta capa utiliza un mecanismo de conmutación de mensajes con almacenamiento y reenvío. Estos mensajes llamados “bundles” son reenviados y almacenados de nodo en nodo hasta llegar a su destino y contienen tanto información del usuario como metadatos relevantes.

Una de las diferencias de este mecanismo con respecto al mecanismo proxy, anteriormente presentado, es que en cada región existe uno o más DTN “Gateway” los cuales reenvían los mensajes entre regiones y también realizan la entrega de mensajes de otras regiones a los nodos correspondientes dentro de cada región local.

La principal ventaja de los sistemas DTN es la sencillez con la que se pueden integrar las redes de sensores a otras redes mediante la capa ‘bundle layer’. Sin embargo, la desventaja que se presenta es la dificultad de desarrollar esta capa dentro de los protocolos ya existentes.

Una solución de este tipo es apropiada para aquellas redes de sensores en las que las particiones de red sean frecuentes o las comunicaciones extremo a extremo lleguen a ser imposibles debido a que el sistema DTN almacenará los mensajes mientras la conexión no esté disponible y los reenviará una vez se restablezca lo que se puede considerar una forma de brindar fiabilidad en la entrega de mensajes.

1.4.3 Mecanismo de Recubrimiento (Overlay Network)

Este tipo de interconexión permite la comunicación directa entre nodos de diferentes redes. Existen dos variantes:

- TCP/IP funcionando sobre el protocolo de la red de sensores
- Protocolo de la red de sensores funcionando sobre TCP/IP.

En el caso de TCP/IP funcionando sobre el protocolo de la red de sensores, esto permite la integración de estas redes con cualquier otra red basada en el protocolo TCP/IP por medio de una conexión directa. Además por medio de una dirección IP se ofrece la posibilidad que cualquier nodo de la red de sensores se encuentre accesible desde Internet, tal y como se muestra en la figura 1.13.

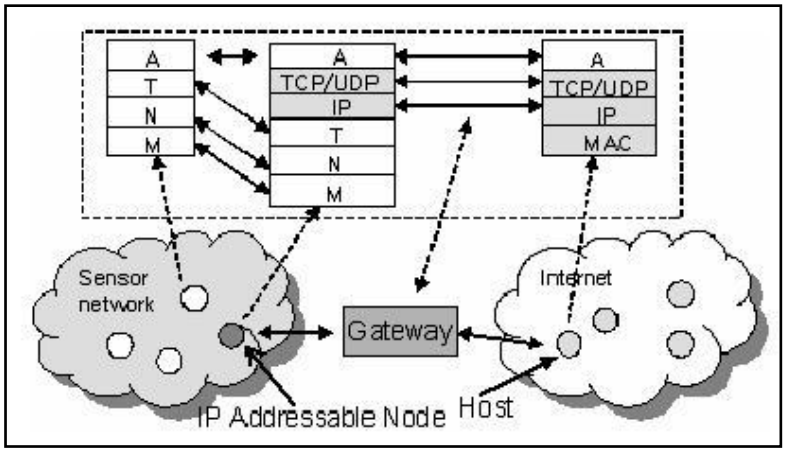

Figura 1.13 TCP/IP sobre redes de sensores
(Fuente: Connecting Heterogeneous Sensor Networks with IP Based Wire/Wireless Networks)

Si comparamos esta variante con el mecanismo de proxy, esta es posiblemente la más eficiente debido a que no crea dependencia en un punto de la red y además mantiene la conexión extremo a extremo.

Las primeras implementaciones de TCP/IP en redes de sensores se atribuyen a Adam Dunkels y el grupo de desarrollo del SICS (Sweden Institute of Computer Science), quienes desarrollaron las pilas LwIP [2] y uIP [3], demostrando que a pesar de las limitaciones y características particulares de las WSN antes mencionadas, es posible llevar a cabo esta interconexión con TCP/IP.

La pila LwIP es una versión simplificada de TCP/IP manteniendo la mayoría de las funcionalidades principales de TCP/IP y soporta los protocolos IP, ICMP, UDP y TCP además que posee una estructura modular que permite añadir nuevos protocolos. Es capaz de gestionar múltiples interfaces de red local e incorpora opciones de configuración que le convierten en un sistema apropiado para conectar una gran variedad de dispositivos.

La pila uIP está orientada a satisfacer los requisitos mínimamente necesarios para cumplir con las especificaciones TCP/IP limitando de cierta manera su funcionalidad; sólo puede gestionar una única interfaz de red y soporta los protocolos IP, ICMP, TCP y UDP. El detalle del funcionamiento de ambas pilas será tratado en un apartado más adelante en este proyecto.

Existen otras particularidades de las redes de sensores que deben tomarse en consideración a la hora de realizar su integración con TCP/IP. Estas pueden ser:

- La comunicación basada en el uso de direcciones IP dentro de la red de sensores puede generar una carga adicional (overhead) en el protocolo, por ejemplo en el caso de tener que hacer procesos de entunelado,

- Además, el esquema de enrutamiento generalmente en las redes de sensores es “data-centric” en cambio en IP está basado en host o en red.

- Adicionalmente, los encabezados del protocolo TCP/IP (40 bytes) son muy grandes para el tamaño tan pequeño del paquete en las redes de sensores, el protocolo TCP no funciona muy bien en condiciones con alta incidencia de errores como generalmente puede suceder en las redes de sensores

- Finalmente, el esquema de retransmisión usado en TCP consume mucha energía en cada salto de la ruta hacia el destino lo que para la red de sensores es un factor crítico para la duración de la red.

En el segundo caso, en el que el protocolo de la red de sensores funciona sobre TCP/IP, cada host es considerado como un sensor virtual y procesa los paquetes como si fuera un sensor, de esta manera se logra una conexión directa entre estas dos redes, tal y como se muestra en la figura 1.14.

El inconveniente de esta variante, es que estamos adicionando una nueva pila de protocolos sobre la ya existente (TCP/IP), lo que agrega una mayor carga adicional a la red TCP/IP.

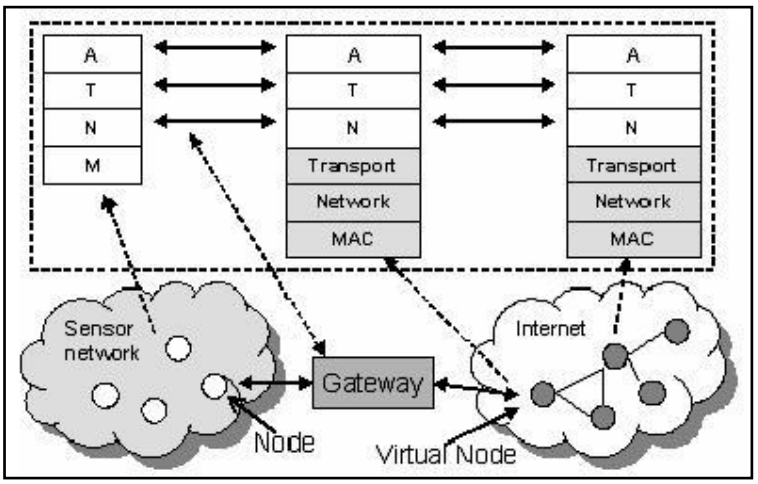

Figura 1.14. Protocolo de redes de sensores sobre TCP/IP
(Fuente: Connecting Heterogeneous Sensor Networks with IP Based Wire/Wireless Networks)

1.5 Pilas TCP/IP para WSN

Como era de esperarse, una vez que las redes de sensores inalámbricas fueran evolucionando, surgiría cada vez más la tendencia a que se interconectasen al resto de tipo de redes, y para lo cual el protocolo defacto para lograr esto sería TCP/IP.

Sin embargo, anteriormente se consideraba que el protocolo TCP/IP no podía ser implementado en los nodos sensores debido a que estos poseían capacidades muy limitadas en memoria y procesamiento. Sin embargo, las primeras implementaciones que demostraron que esto era viable se les atribuyen a Adam Dunkels y al equipo de desarrollo del SICS quienes desarrollaron las pilas LwIP y uIP. Una vez sucedió esto fueron apareciendo distintas versiones y evoluciones de diferentes pilas TCP/IP.

En esta sección presentamos una breve descripción de las pilas LwIP y uIP, haciendo un énfasis especial en esta última la cual se ha utilizado para la realización de este proyecto y finalmente un breve resumen de otras pilas TCP/IP relevantes. Además un resumen de las pilas TCP/IP se presentan en el ANEXO A.

1.5.1 Pilas LwIP (Lighweight IP) y uIP (Micro IP)

A continuación se presentan los protocolos que se implementan en ambas pilas y se identifican las particularidades o diferencias que existen entre ellas.

- **Protocolo IP**

Ambas pilas descartan las opciones del encabezado IP, debido a que no se consideran necesarias para el funcionamiento.

- **Reensamble de Paquete IP**

Tanto la pila LwIP como uIP solamente constan de un único buffer que mantiene el paquete a ser reensamblado, por lo tanto no soporta el reemsamble simultáneo de más de un paquete. Esto es debido a que se supone que la fragmentación no es algo que ocurra muy frecuente, sin embargo se espera que para futuras implementaciones se permita el soporte de múltiples búferes.

- **Broadcast y Multicast a nivel IP**

Actualmente la pila uIP tiene soporte de direcciones broadcast, así como el envío de paquetes multicast, sin embargo no tiene soporte para el protocolo IGMP para unirse a grupos multicast.

- **Protocolo ICMP**

Ambas pilas solamente tienen implementado los mensajes de echo ICMP. Las respuestas a los mensajes “echo” son construidas simplemente intercambiando las direcciones IP destino y origen de la petición ICMP entrante y rescribiendo el encabezado ICMP con el mensaje Echo-Reply.

- **Protocolo TCP**

Las pilas uIP y LwIP implementan las funcionalidades tradicionales de TCP (como por ejemplo el cálculo de RTT, la escucha de conexiones), sin embargo existen ciertos mecanismos de TCP que cada pila implementa de forma distinta y que presentamos a continuación.

  - **Mecanismo de Ventana Deslizante**

La pila uIP no implementa este mecanismos debido, que el algoritmo de dicho mecanismo hace uso de muchas operaciones de 32 bits, las cuales, el autor considera que son muy costosas para la mayoría de CPU’s de 8 y 16-bits que existían al momento del desarrollo de esta pila.

Actualmente ya existen procesadores de 32 bits para los cuales estas operaciones no representan una dificultad.

Como consecuencia de lo anterior la pila uIP solo permite el envío de un solo segmento TCP para ser reconocido, lo que se considera un mecanismo Stop & Wait, además tampoco almacena los paquetes enviados en un búfer, dejando la responsabilidad de esto a la capa de aplicación con la complejidad adicional que esto conlleva.

En cambio la pila lwIP sí permite el mecanismo de ventana deslizante, permitiendo múltiples segmentos TCP en vuelo por cada conexión mediante el uso de colas en el buffer de salida, por lo tanto no agrega mayor complejidad a la capa de aplicación.

  - **Control de Congestión**

La pila uIP no implementa este mecanismo ya que solo maneja un segmento TCP en vuelo por conexión.

En contraste con lo anterior, como la pila lwIP puede manejar múltiples segmentos TCP en vuelo, ésta implementa el mecanismo de control de congestión de TCP estándar Fast Retransmit/Fast Recovery.

- **Retransmisiones**

Las retransmisiones son manejadas de manera diferente en ambas pilas:

1. lwIP mantiene dos colas de salida: una mantiene los segmentos que no han sido enviados todavía y la otra mantiene los segmentos que han sido enviados pero que no se han recibido reconocimiento (ACK) por el receptor. Cuando una retransmisión es requerida el primer segmento en la cola de segmentos que no han recibido reconocimiento (ACK) es enviado. Todos los demás segmentos en la cola son movidos a la cola con segmentos que no se han enviado.

2. uIP no mantiene registro del contenido del paquete después que ha sido enviados por el interfaz de red, por lo tanto requiere que la aplicación juegue un papel activo en la ejecución de la retransmisión. Cuando uIP decide que un segmento debe ser retransmitido, este llama a la aplicación con una marca (flag) indicando que una retransmisión es requerida. La aplicación verifica la marca de retransmisión y produce el mismo dato que fue previamente enviado.

   Cabe señalar, que esta forma de proceder es muy parecida al funcionamiento de una aplicación utilizando el protocolo UDP lo que no garantiza una fiabilidad extremo a extremo.

1.5.2 Otras pilas TCP/IP relevantes

En este apartado pretendemos presentar una descripción de las iniciativas de desarrollo de pilas basadas en el protocolo IPv6 que actualmente están siendo impulsadas por diferentes grupos de fabricantes con la finalidad de hacer del protocolo IPv6 el estándar aceptado para las WSN.

- **Pila uIPv6 SICSlowPAN**

Es una pila que provee las mismas capacidades que la pila uIP versión 4, utilizando tanto el protocolo TCP como UDP, sin embargo se diferencia de las demás pilas IPv6 debido a que crea un nivel de abstracción que permite la integración de la pila uIPv6 con diversas tecnologías de la capa MAC (como 802.11, 802.15.4, Ethernet), como se muestra en la figura 1.15.

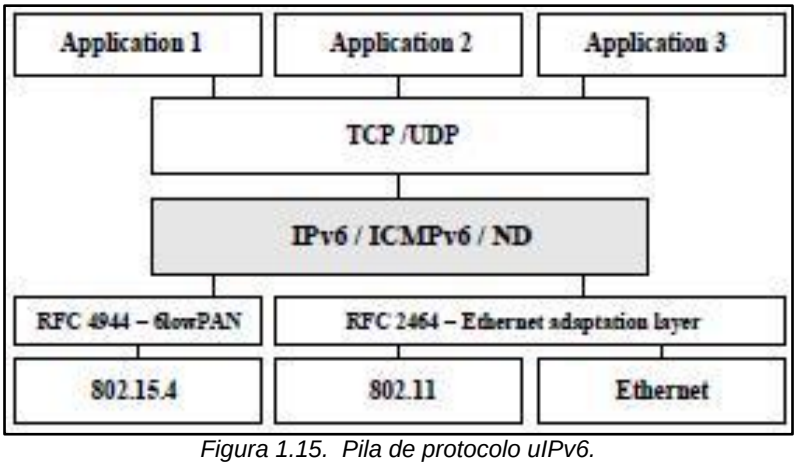

Figura 1.15. Pila de protocolo uIPv6.
(Fuente: Poster Abstract: Making Sensor Networks IPv6 Ready)

- **Pila NanoStack**

Es una pila de protocolos IPv6 con especificaciones 6LoWPAN y la cual puede ser implementada sobre redes IEEE 802.15.4 y fue desarrollada por la empresa Sensinode.

NanoStack incluye los siguientes protocolos: 6LoWPAN IPv6, UDP, ICMP y la capa MAC conforme al estándar IEEE 802.15.4. Adicionalmente cuenta con el protocolo NanoMesh que provee capacidades de comunicación multi-salto.

- **Pila b6lowpan o BLIP**

Esta es la implementación de la pila IPv6 para el sistema operativo TinyOS desarrollada por la Universidad de Berkeley. Esta pila utiliza la compresión de encabezado 6lowpan/HC-01 e incluye los mecanismos de descubrimiento de vecino, selección de ruta por defecto, enrutamiento punto a punto, entre otros.

2. ESTADO DEL ARTE DE PROTOCOLOS DE TRANSPORTE EN REDES DE SENSORES

Las características particulares que poseen las redes de sensores inalámbricas establecen nuevos retos de diseño como son el uso eficiente de la energía, la fiabilidad, el control de congestión y la calidad de servicio. Estos factores deben ser satisfechos para cumplir con los requerimientos que demandan las aplicaciones en este tipo de redes.

En este sentido, uno de los retos que cada vez cobra mayor importancia es la necesidad de implementar mecanismos que proporcionen fiabilidad en la entrega de los datos extremos a extremo, reduzcan la congestión y la pérdida de paquetes, además de proveer justicia en la asignación de ancho de banda. Esto aunado a la tendencia de interconectar las redes de sensores inalámbricas a otro tipo de redes como Internet, redes de área local o intranets para la recepción de forma remota de los datos generados por los sensores, hacen que la elección del protocolo de transporte adecuado sea un aspecto esencial en el desarrollo de aplicaciones para este tipo de redes.

Es por esto que en este capítulo haremos un análisis de los protocolos de transporte tradicionales como TCP y UDP y su viabilidad en las redes de sensores, además de identificar los requisitos que deben cumplir los protocolos de transporte para WSN y acabaremos presentando las distintos tipos de protocolos de transporte existentes para este tipo de redes.

2.1 Limitaciones de los protocolos de transporte tradicionales

Los protocolos de transporte tradicionales utilizados en Internet como lo son UDP y TCP presentan varios inconvenientes para ser implementados en las redes de sensores.

En el caso del protocolo UDP, por ejemplo, este no provee fiabilidad en la entrega de datos, que en la mayoría de los casos es un requisito de las aplicaciones en este tipo de redes, ni tampoco proporciona un mecanismos de control de flujo ni de congestión lo que puede conducir a la pérdida de paquetes y el gasto innecesario de energía de los nodos sensores.

Por otro lado, si analizamos el sistema de comunicación fiable extremo a extremo proporcionado tradicionalmente por TCP encontramos que este tiene serios problemas de rendimiento en las redes inalámbricas, tanto en términos de tasas de transferencia como de eficiencia energética.

Los principales problemas que presenta el protocolo TCP en las WSN se describen a continuación [4]:

- **Pérdida de paquetes no debida a la congestión: se** refiere a que TCP utiliza como mecanismo de detección de congestión la pérdida de paquetes lo que conlleva a que TCP reduzca la tasa de transferencia con la finalidad de no colapsar aún más los enlaces. Sin embargo, en una WSN las pérdidas de paquetes ocurren generalmente por errores de transmisión del medio inalámbrico de manera que la reducción de la tasa de transferencia lo que trae consigo es una reducción innecesaria de la utilización del ancho de banda del enlace y por ende a una degradación en el throughput y un retardo mayor en la comunicación. Una posible solución es la utilización de mecanismos (implícitos o explícitos) de retroalimentación entre los nodos que ayuden a detectar las diferentes causas por las cuales la pérdida de paquetes ha ocurrido (calidad del enlace inalámbrico, fallos en el nodo sensor, y/o congestión) y de acuerdo a esto tomar la decisión más conveniente.

- **Retransmisiones costosas:** TCP confía en las retransmisiones extremo a extremo para proveer una entrega de datos fiable, sin embargo teniendo en cuenta las limitaciones de energía de los sensores y las rutas multi-salto, este mecanismo conllevaría a un mayor consumo de energía y ancho de banda en las WSN. Además el mecanismo de control de congestión extremo a extremo utilizado por TCP ocasiona que se responda muy tarde a una situación de congestión lo que resulta en una gran cantidad de paquetes perdidos lo que se convierte en un gasto de energía adicional en retransmisiones y los tiempos de respuestas tan largos ocasionan un bajo throughput y baja utilización del enlace inalámbrico.

- **Topología Dinámica de la red:** los cambios de topología que caracterizan a las WSN debido a las condiciones del entorno (baja calidad del enlace inalámbrico, interferencias de señal producidas por agentes externos) y a la

propia situación del nodo sensor (el nivel de energía que posea) conllevan a que en un momento dado la ruta entre dos puntos extremos de la red se vea interrumpida. Tal comportamiento no es compatible con el funcionamiento de TCP el cual considera una conectividad permanente extremo a extremo.

- **Red asimétrica:** Se define como aquella en la que el camino utilizado para transportar datos hacia el destino es diferente del camino utilizado para retornarlos hacia el origen tanto en términos topológicos como de latencia, ancho de banda o tasa de pérdida de paquetes. Las WSN son asimétricas en la mayoría de los casos, aspecto que afecta directamente a la transmisión de los ACK's del protocolo TCP, cuyo rendimiento puede verse afectado.

- **Grandes variaciones del RTT:** Debido a la variabilidad de la calidad de los enlaces, la movilidad o la carga de tráfico, las rutas de encaminamiento se ven modificadas a lo largo del despliegue de las redes de sensores. Esto puede generar variaciones en el RTT, degradando el rendimiento de TCP.

- **Transmisión en tiempo real:** Junto a UDP, en las redes de sensores deben implementarse protocolos semejantes a RCP (Rate Control Protocol), de forma que éstas puedan soportar la transmisión extremo a extremo del tráfico en tiempo real.

2.2 Requisitos de los protocolos de transporte

Tomando en cuenta las limitaciones presentadas en el apartado anterior, el diseño de un protocolo de transporte para WSN debería cumplir con los siguientes requisitos:

2.2.1 Fiabilidad en la entrega de datos

Dependiendo del tipo de aplicación la fiabilidad se puede clasificar en: **fiabilidad basada en paquetes:** que se refiere a que todos los paquetes enviados por la fuente deben llegar al destino, **fiabilidad basada en eventos:** que se enfoca en la necesidad de notificar al destino sobre la ocurrencia de un evento más que en la entrega de todos los mensajes al destino. En este tipo de fiabilidad se debe considerar la calidad de los datos recibidos para considerar que verdaderamente un evento ha ocurrido, es decir que los datos recolectados deberían ser representativos de la región consultada o del evento medido.

También de acuerdo a la dirección del tráfico a la que se le ofrece fiabilidad ésta se puede clasificar en:

- **Fiabilidad Descendente (downstream):** es la fiabilidad proporcionada al tráfico originado en el nodo "sink" y dirigido hacia los nodos sensores. Generalmente este tráfico contiene mensajes de consultas o mensajes de control tales como, los relacionados a tareas de control/notificación de congestión, reprogramación, configuración. Por lo tanto, el protocolo de transporte debería ofrecer el más alto nivel de fiabilidad a este tráfico ya que la pérdida de estos mensajes puede ir en perjuicio de la correcta operación de la red.
- **Fiabilidad Ascendente (upstream):** se refiere a la fiabilidad ofrecida al tráfico que va desde los nodos sensores hacia el nodo "sink" o hacia un nodo que haga agregación de datos. La agregación de datos es la capacidad que tienen los nodos de procesar información recibida de otros nodos sobre un evento observado. Este proceso permite a los nodos reducir el volumen de información que debe ser transmitido finalmente hacia el nodo "sink", y esta reducción redunda en beneficio de un ahorro de energía en los nodos.

Tomando en cuenta los diferentes tipos de fiabilidad presentados, los protocolos de transporte deberían considerar el uso de un mecanismo que adapte su comportamiento de acuerdo por ejemplo a los requisitos de las aplicaciones que se ejecutan en los sensores o a la dirección del tráfico que maneja.

Otras consideraciones a tomar en cuenta en la fiabilidad, es si ésta debe ser extremo a extremo o salto a salto. La primera es el tipo de fiabilidad ofrecida tradicionalmente por el protocolo TCP la cual se podría considerar inapropiada para las WSN ya que implica un mayor consumo de energía en los nodos intermedios cada vez que ocurre una transmisión de reconocimientos TCP (ACK) por parte del destino con la finalidad de confirmar la recepción de los datos. En cambio la fiabilidad salto a salto podría resultar más eficiente en términos de energía ya que a medida que el paquete pasa por los nodos, estos pueden recibir un mensaje (generalmente de capa de enlace) confirmando la recepción del paquete por el siguiente nodo.

2.2.2 Eficiencia energética

Debido a que generalmente los nodos tendrán una fuente de energía limitada, el protocolo de transporte debe ser diseñado de manera que mantenga un alto grado de eficiencia energética con el fin de maximizar la vida útil de la red. En este sentido se

deben considerar factores como el número de retransmisiones de paquetes, el número de saltos (distancia) que conlleva cada retransmisión, y la carga adicional (overhead) asociado a los mensajes de control ya que son factores que tendrán un impacto directo en el consumo de energía del nodo sensor y por ende en el tiempo de vida de la WSN.

2.2.3 Parámetros de calidad de servicio (QoS)

En este sentido el protocolo de transporte debería garantizar diferentes niveles de retardo, ancho de banda, throughput, entre otros, dependiendo del tipo de aplicación. Por ejemplo aquellas aplicaciones que transmitan continuamente imágenes de seguimiento de objetos requerirán un mayor ancho de banda en comparación con las basadas en eventos, de igual manera aquellas aplicaciones "critical-time" requerirán de un menor nivel de retardo que otras que solamente realizan transmisiones periódicas.

2.2.4 Control de Congestión:

Las principales causas de congestión en las WSN se deben a:

- La tasa de llegada de paquetes excede la tasa de servicio de un nodo, generalmente esto suele ocurrir en los nodos que se encuentran más cerca del nodo "sink" ya que estos nodos recibirán un mayor tráfico combinado producto de las diferentes aplicaciones que se ejecutan en la WSN.

- Factores que afectan el rendimiento del enlace inalámbrico como: contención MAC, o interferencia que pueden ocasionar que los paquetes permanezcan más tiempo en el búfer del nodo llegando a ocupar todo el espacio disponible.

La congestión en la WSN tiene un fuerte impacto en la eficiencia energética de los nodos, inclusive puede generar un gran nivel de retardo y altos niveles de pérdidas de paquetes en las WSN.

Existen tres mecanismos para hacer frente a la congestión en la WSN, estos son:

1. **Detección de Congestión**

En las WSN, se recomienda la utilización de mecanismos proactivos como por ejemplo medir la ocupación del búfer en los nodos, el tiempo de servicio de un paquete en los nodos. Además la utilización de mecanismos de "cross-layer" de nivel de enlace que

permitan detectar por ejemplo, el nivel de ocupación del medio inalámbrico, la calidad del enlace inalámbrico.

2. **Notificación de Congestión:**

Los protocolos de transporte en las WSN deben ser capaces de transmitir información sobre el nodo congestionado a los nodos involucrados y sobre todo a la fuente que genera la congestión. Esta notificación puede ser explícita la cual consiste en que los nodos envían mensajes de control en la que solo avisan que existe una situación de congestión, por ejemplo utilizando el bit CN (Congestion Notification) de TCP, o mayor información como el nivel de congestión. También puede ser una notificación implícita en la que la información de congestión va añadida (piggyback) en los mensajes de datos recibidos o en la que dicha información se escucha a través del enlace inalámbrico.

3. **Control de la tasa de transmisión:**

De acuerdo a la información de congestión recibida los protocolos de transporte podrían ajustar la tasa de transmisión de datos en el nodo que está causando la congestión.

En contraste con esta situación, también se podría optar por la utilización de mecanismos de control de congestión activa (ACC) el cual puede hacer que el emisor o nodo intermedio reduzca su tasa de transmisión cuando el tamaño del búfer de los nodos vecinos en el sentido descendente (downstream) exceda un límite, para evitar una situación de congestión, cabe señalar que este mecanismo tiene como inconveniente la reducción en la utilización del enlace inalámbrico

2.2.5 Recuperación de Paquetes Perdidos

La pérdida de paquetes usualmente es causada por la calidad del enlace inalámbrico, fallos propios del nodo y/o por la congestión. Además la pérdida de paquetes afecta la fiabilidad en la entrega de datos y los parámetros de QoS de la WSN.

Existen tres mecanismos para realizar la recuperación de paquetes perdidos, estos son:

1. **Detección/Notificación**

El método de asignar a cada paquete un número de secuencia de manera que el receptor pueda detectar con ello la pérdida de paquetes es generalmente el utilizado por los protocolos de transporte fiables. Una vez el receptor detecta que un paquete se ha perdido envía mensajes de ACK o NACK para notificar al emisor de los paquetes perdidos para que ésta pueda iniciar su recuperación, este modo de funcionamiento se considera de extremo a extremo.

Debido a que los nodos sensores son dispositivos con limitados recursos de energía, la detección/notificación extremo a extremo no es la más conveniente para este tipo de redes por las siguientes razones:

- Los mensajes de notificación (ACK o NACK) que son enviados por el receptor deben utilizar una ruta de retorno hacia el emisor consistente en varios saltos, lo que conduce a un mayor gasto de energía en cada uno de los nodos.

- De igual manera, ya que la notificación llega al emisor, éste es responsable del realizar la retransmisión del paquete perdido, por lo tanto el paquete debe atravesar todos los nodos en la ruta hacia el destino lo que conlleva a un gasto de energía en cada nodo.

- Los mensajes de notificación también están expuestos a perderse debido a las condiciones del medio inalámbrico o a la congestión que pueda surgir en la ruta hacia el emisor.

En contraste con lo anterior, un esquema salto a salto permite que cada nodo en la ruta hacia el destino pueda detectar y notificar la pérdida de paquete a su vecino al contrario de dejarle esta responsabilidad al receptor, además el hecho que un nodo detecte o sea notificado de una pérdida de paquete por su vecino, permite la retransmisión desde dicho nodo, al contrario de realizarla desde el emisor. Estas características conducen a un uso más eficiente de la energía en la red y por lo tanto este esquema se considera el más apropiado para las WSN.

Por otra parte, la detección de un paquete perdido basado en el esquema salto a salto, puede darse de manera implícita o explícita. En la primera, el nodo sensor puede aprovecharse de la característica “broadcast” del medio inalámbrico para escuchar la

transmisión del paquete por parte de su nodo vecino para determinar la pérdida de paquetes. Cabe señalar que esta manera puede no ser viable en situaciones donde se estén utilizando mecanismos MAC como TDMA.

En la segunda forma, el nodo puede recibir un mensaje de control, haciendo uso, generalmente, de ACK's a nivel de enlace, del nodo vecino que confirme la recepción del paquete.

2. Retransmisión de paquetes

De forma similar a la detección/notificación de pérdida de paquetes, la retransmisión de paquetes también puede ser extremo a extremo o salto a salto. Existen varios factores que hacen al esquema salto a salto más ventajoso frente al esquema extremo a extremo, los cuales detallamos a continuación:

1. El esquema salto a salto permite reaccionar de forma más rápida a la pérdida de paquetes que el esquema extremo a extremo, lo que evita la degradación del "throughput" en los nodos.

2. La distancia (medida en número de saltos) que existe entre el nodo donde se detecta la pérdida de paquete y el punto que realiza la retransmisión del paquete se denomina distancia de retransmisión. Esta distancia se puede tomar como un factor de eficiencia en términos de energía consumida en el proceso de retransmisión.

   En este sentido un esquema salto a salto en el que los nodos intermedios almacenan los paquetes en un búfer para retransmitirlos en caso de detectar pérdida en su nodo vecino, la distancia de retransmisión será generalmente de un salto comparada con el esquema extremo a extremo en el cual esta distancia siempre será medida desde el emisor al punto donde ocurrió la pérdida de paquete en cuyo caso usualmente resultará mayor. **(ver Figura 2.1)**

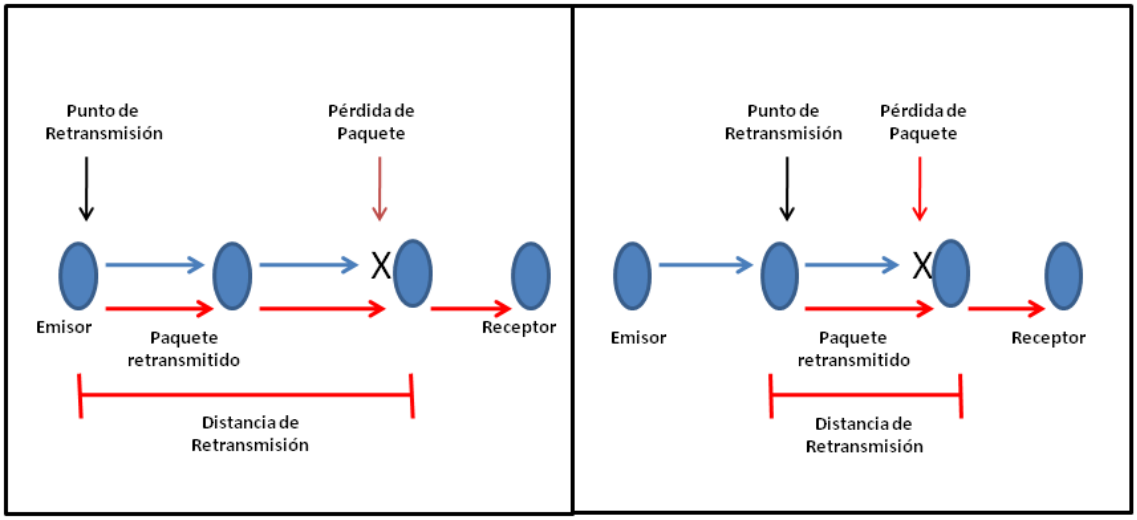

Figura 2.1Tipos de esquemas de recuperación de errores.

Finalmente, existen factores adicionales que deben tomarse en cuenta en el momento de diseñar mecanismos de recuperación de errores, como lo son:

- La cantidad de tiempo que debe mantenerse el paquete almacenado en el búfer de un nodo.

- La velocidad con que se debe realizar una retransmisión una vez detectada la pérdida de paquete, Esto es debido a que si la pérdida de paquete es causada por congestión, si se retransmite inmediatamente se puede agravar aún más la situación del nodo congestionado lo que resultará en un aumento en la cantidad de paquetes perdidos.

- Determinar los nodos que puedan tener capacidad para almacenar los paquetes considerando sus limitaciones de memoria Pueden haber dos alternativas que son: distribuir de forma probabilística el almacenamiento de los paquetes en los nodos de la red o almacenar los paquetes en los nodos más cercanos a los nodos potencialmente congestionados donde generalmente ocurrirá la pérdida de paquetes.

2.3 Protocolos de transporte existentes en WSN.

Los mecanismos de transporte que existen para redes de sensores inalámbricas se pueden dividir en aquellos que proporcionan en alguno o en ambos sentidos

(sea ascendente o descendente) y una o la combinación de las siguientes funciones **(Ver figura 2.2)**:

- Fiabilidad en la entrega de mensajes (incluyendo la recuperación de errores)
- Control de congestión
- Conservación de la energía

Además también pueden ser clasificados en:

- Protocolos no basados en TCP
- Protocolos basados en TCP

En este proyecto se utilizará esta última clasificación de protocolos de transporte, identificando además, las funciones y objetivos para los que fueron diseñados.

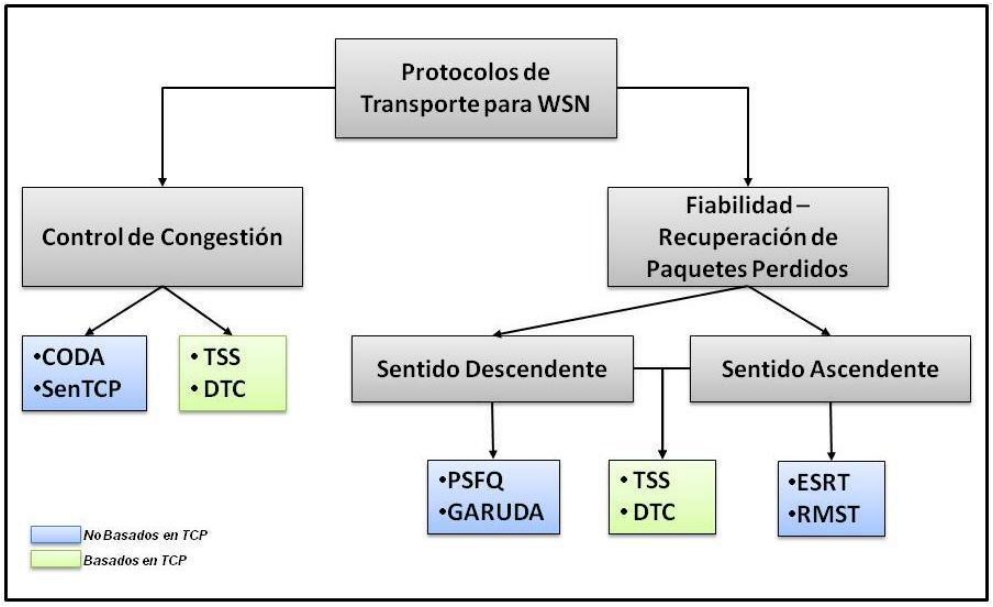

Figura 2.2Clasificación de los protocolos de transportes existentes para WSN.

2.3.1 Protocolos no basados en TCP

Estos protocolos han sido desarrollados buscando resolver las limitaciones que posee TCP, en este apartado presentamos algunos de ellos.

- **CODA (Congestion Detection and Avoidance)** [5]

Este es un protocolo que proporciona control de congestión en sentido ascendente (upstream). CODA detecta la congestión en la red mediante la monitorización de la ocupación de memoria de los nodos y la carga del canal inalámbrico. Si alguna de

estas dos variables supera un cierto umbral, se considera que existe congestión y se notifica de la congestión a los nodos que deben reducir su tasa de transferencia de datos a la red.

Las principales desventajas de CODA son: solo proporciona control de congestión en dirección ascendente, no provee ningún tipo de fiabilidad en la entrega de mensajes, y debido a esto también, su tiempo de respuesta a la congestión puede incrementar ya que los mensajes ACK utilizados para notificar la congestión pueden perderse en condiciones de altas tasas de error.

- **SenTCP** [6]

Es un protocolo enfocado en el control de congestión y el cual utiliza el tiempo medio de servicio de un paquete a nivel local y el tiempo medio de llegadas entre paquetes también a nivel local para estimar el grado de congestión local en cada nodo intermedio. Además realiza un control de congestión salto a salto. En SenTCP cada nodo intermedio notifica a sus vecinos, en dirección a la fuente, los paquetes recibidos. La notificación de congestión, la cual incluye el grado de congestión local y el ratio de ocupación de memoria, será utilizada por los nodos vecinos para ajustar su tasa de transferencia en la capa de transporte.

La ventaja principal de este protocolo, radica en el control de congestión salto a salto que permite responder y eliminar la situación de congestión de forma rápida, este mecanismo también reducirá la pérdida de paquetes lo que resultará en un mayor throughput.

- **ESRT (Event to sink Reliable Transport**) [7]

Este protocolo está orientado a proporcionar fiabilidad extremo a extremo y control de congestión en dirección ascendente. Su funcionamiento radica en calcular periódicamente la fiabilidad de acuerdo a la cantidad de paquetes recibidos en un intervalo de tiempo y de esta manera deduce la frecuencia con que los nodos sensores deben transmitir sus paquetes. De esta manera ESRT ajusta las propiedades de los sensores de forma que se consigan cuanto antes los niveles de fiabilidad que se han marcado como objetivo. En cambio, si la fiabilidad es superior a la requerida, ESRT procura que los nodos ahorren energía. Esta autoconfiguración del protocolo ESRT dota al sistema de robustez ante la topología dinámica de las WSN. Además ESRT proporciona fiabilidad basada en eventos.

Los algoritmos de ESRT son implementados principalmente en el nodo destino, con mínimos requerimientos en los nodos sensores.

Sin embargo, la implementación de esta técnica no evita todas las pérdidas ni garantiza la entrega de todos los paquetes; aunque sí proporciona la frecuencia óptima para enviar mensajes.

- **PSFQ (*Pump Slowly Fetch Quickly)*** [8]

Es un protocolo diseñado específicamente para resolver los problemas de fiabilidad basada en paquetes y en dirección descendente.

PSFQ basa su funcionamiento en tres modos de operación: el primero consiste en que los datos son transmitidos lentamente (“slow pump) desde el nodo “sink” al resto de la red. El segundo modo de operación se activa cuando los nodos sensores experimentan pérdidas de paquetes las cuales pueden recuperarse mediante rápidas peticiones (“fetch quickly”) a sus vecinos más cercanos. Para reducir el exceso de datos en la red, los nodos notifican la pérdida de paquetes usando reconocimientos negativos (NACK’s), en vez de usar reconocimientos positivos cada vez que reciben un paquete. El último modo de operación se activa cuando el nodo “sink”, si así lo requiere, solicita a los nodos sensores enviarle información (“report”) de estado de la entrega de paquetes.

Este protocolo está destinado a redes que generen poco volumen de tráfico, de forma que la pérdida de paquetes sea debida principalmente a errores en el medio inalámbrico más que a una posible congestión del sistema. Es por esta razón que PSFQ no tiene implementado ningún mecanismo de control de congestión. Además tiene como desventaja que el modo de transmitir lentamente los datos hacia los nodos genera un aumento en el retardo.

Algunas de las aplicaciones que se podrían aprovechar de estas ventajas podrían ser, por ejemplo, aquellas que transmitan imágenes en formato binario o las que se encarguen de reconfigurar los nodos de la red.

- **RMST (Reliable Multi-Segment Transport)** [9]

Este protocolo pertenece a aquellos que se encargan de asegurar fiabilidad basada en paquete en sentido ascendente. También, RMST puede realizar la recuperación de paquetes tanto en el esquema extremo a extremo como en el esquema salto a salto.

Su utilización está dirigida a tareas de reprogramación de sensores y de transferencia de objetos binarios, donde la pérdida de un único elemento podría suponer daños irreparables en el mensaje.

RMST, está diseñado para funcionar junto con el protocolo de enrutamiento de Difusión Directa [10], del que se ayudará para el descubrimiento de caminos desde los sensores hasta el destino final y al que añadirá dos nuevas características como son la fragmentación/reensamblaje de segmentos, y la distribución fiable de mensajes. RMST utiliza NACK's para detectar y recuperar mensajes perdidos de forma similar a como lo hace PSFQ.

- **GARUDA** [11]

GARUDA es un protocolo dedicado a garantizar la fiabilidad en sentido descendente, Este protocolo usa la transmisión de un pulso WFP (Wait for First Packet) para garantizar la correcta recepción de un primer paquete y para construir una arquitectura de dos capas. La primera capa está formada por aquellos nodos que se encuentren a una distancia 3*i (i: es un número entero) saltos del nodo "sink" los cuales son llamados nodos "núcleo", y la segunda capa la forman el resto de nodos. Con esta arquitectura, GARUDA lleva a cabo dos formas diferentes de recuperación de pérdidas; una entre los nodos 'núcleo' y otra entre los nodos "núcleos" y el resto de los nodos mediante el uso de NACK.

2.3.2 Mecanismos basados en TCP

En este apartado se va a analizar algunos de los principales mecanismos que adaptan el protocolo TCP a las necesidades específicas de las redes de sensores inalámbricas: Distributed TCP Caching (DTC) y TCP Support for Sensor networks (TSS).

DTC y TSS permiten el almacenamiento de segmentos TCP en los nodos intermedios y mejoran el mecanismo de retransmisión en caso que se produzcan errores, reduciendo así el número de retransmisiones de paquetes en el sistema.

- **DTC (Distributed TCP Caching)** [12]

DTC proporciona fiabilidad en ambos sentidos tanto ascendente como descendente. Su funcionamiento se basa en la recuperación de paquetes basada en un esquema salto a salto, en el cual los nodos almacenan segmentos TCP que son retransmitidos localmente cuando se producen pérdidas de paquetes. Además DTC utiliza los mensajes ACK de nivel de enlace para confirmar la recepción de los datos en cada

nodo, en caso contrario se considera que ha habido una pérdida de paquete y se procede a la retransmisión.

Debido a las restricciones de memoria de los sensores, en este sistema sólo puede ser almacenado un segmento TCP por sensor. Los nodos almacenan, con una cierta probabilidad, el segmento TCP transmitido con el mayor número de secuencia.

- **TSS (TCP Support for Sensor Networks)** [13]

Este protocolo mantiene muchas similitudes con respecto a DTC. Sus mayores diferencias se basan en que TSS proporciona un mecanismo de control de congestión denominado “backpressure” por medio del cual un nodo puede detener la transmisión de subsecuentes paquetes hasta que este conozca que todos los paquetes anteriores han sido recibidos y transmitidos por el siguiente nodo. Además en TSS, los nodos escuchan la transmisión de paquetes del siguiente nodo para detectar la pérdida de paquetes,

Igual que en DTC, los nodos sólo pueden guardar en memoria un único segmento. No obstante, la decisión del segmento guardado es totalmente determinista. De hecho, cada nodo siempre guarda en memoria el segmento TCP que contiene el primer byte de datos que todavía no ha sido reconocido o transmitido por el siguiente nodo hacia el nodo destino.

Este paquete es almacenado hasta que se asegura que el siguiente nodo ha recibido el segmento. TSS necesita, además, un búfer adicional para guardar temporalmente paquetes a la espera de ser enviados al siguiente nodo.

Este proyecto se centra en el análisis exhaustivo de este protocolo, su implementación y mejoras, las cuales se presentan en el capítulo 3 de este trabajo.

3. ANÁLISIS E IMPLEMENTACIÓN DE TSS (TCP SUPPORT FOR SENSOR NETWORKS)

En este capítulo primeramente analizaremos en detalle el funcionamiento y características del protocolo TSS propuesto por Adam Dunkels y a continuación detallaremos su proceso de implementación en una red real de sensores inalámbricos identificando las limitaciones y problemáticas encontradas y cómo éstas fueron superadas para la puesta en funcionamiento considerando que hasta la fecha no se conoce otra implementación del protocolo.

Finalmente se presentan un conjunto de pruebas realizadas a la implementación y se discuten y analizan los resultados obtenidos.

3.1 Funcionamiento de TSS

TSS es un protocolo que busca superar las limitaciones que posee TCP en cuanto a su eficiencia energética y rendimiento en las redes de sensores inalámbricas.

En términos generales TSS se basa en un mecanismo que almacena los segmentos TCP en los nodos intermedios hacia el destino, de manera que puedan ser retransmitidos lo más cercano al destino en caso de que se detecte pérdida de los mismos, utiliza un mecanismo de recuperación de reconocimientos (ACK) TCP y utiliza un mecanismo cross-layer de confirmación implícita de la capa de enlace para detectar pérdidas. Además TSS implementa un mecanismo con el cual detiene la transmisión de nuevos segmentos TCP si detecta que existe congestión en el próximo nodo hacia el destino, llamado “backpressure congestion”.

TSS no requiere ningún tipo de cambio en la implementación TCP del nodo origen ni destino de la conexión, por lo tanto su funcionamiento estará limitado a los nodos intermedios.

Podemos considerar TSS como una evolución de DTC (Distributed TCP Caching). TSS se diferencia de DTC principalmente en que el mecanismo de “backpressure congestion” permite la posibilidad de implementar en el nodo emisor un control de congestión de manera que ajuste la ventana de congestión sin conocer la cantidad de saltos (topología) hacia el destino, a diferencia de DTC que mantiene un número fijo de segmentos por ventana, además este mecanismo mantiene los paquetes almacenados en un nodo intermedio hasta que el nodo siguiente haya transmitido los paquetes anteriores. Además TSS no utiliza las opciones TCP como los reconocimientos selectivo (SACK) y asegura la secuencia de llegada de los segmentos

de datos TCP al destino por lo tanto evita los procesos de reordenamiento en el destino.

A continuación detallamos las funciones que realiza TSS para lograr una eficiencia energética y mejorar el rendimiento de TCP en las redes de sensores inalámbricas:

3.1.1 Almacenamiento "Caching"

Un nodo intermedio almacenará un segmento hasta que esté seguro que el nodo sucesor hacia el destino ha recibido el segmento. Un nodo conoce esto, cuando detecta (escuchando el medio inalámbrico) que el nodo sucesor ha transmitido el segmento o cuando recibe un reconocimiento TCP (TCP ACK) que ha sido enviado desde el destino hacia el origen.

Por otra parte, además de almacenar los segmentos en la caché, TSS requiere de un búfer adicional para almacenar temporalmente el próximo paquete que está esperando a ser transmitido al nodo sucesor. Este búfer es requerido por el mecanismo de "backpressure congestion".

3.1.2 Retransmisiones locales de segmentos de datos TCP

Todos los nodos intermedios ejecutarán retransmisiones locales cuando ellos detectan que un segmento almacenado no ha sido recibido por el nodo sucesor hacia el destino.

El proceso consta de 2 fases:

a. Fase 1: Mecanismo Cross-layer de confirmación implícita.

Se establece un primer temporizador en el momento de enviar un segmento de datos durante el cual se espera que el nodo pueda escuchar y detectar que su nodo vecino ha enviado el paquete recibido al nodo sucesor **(ver figura 3.1)**. Si la detección es exitosa, el paquete almacenado en la caché es removido, de lo contrario se pasa a la fase 2 de reconocimiento TCP ACK

Aunque este mecanismo cross-layer de confirmación implícita pudiera parecer una operación que consume energía de manera ineficiente, según el autor es más factible que utilizar confirmación explicita del nivel de enlace, tal y como lo hace DTC, debido a que este requiere que el nodo escuche, reciba y que el nodo sucesor transmita paquetes de confirmación adicionales.

En TSS un nodo solo escucha las demás transmisiones por un periodo de tiempo muy corto. Generalmente un paquete será transmitido inmediatamente

por el nodo sucesor y solamente en caso de pérdida del paquete un nodo debe mantenerse escuchando durante el intervalo completo que dure el temporizador de expiración para retransmisión.

Es importante recalcar que este mecanismo de confirmación implícita solo es viable entre los nodos intermedios ya que cada nodo escuchará a su nodo intermedio sucesor, sin embargo este mecanismo no podría utilizarse entre el nodo origen y el primer nodo intermedio y entre el último nodo intermedio y el nodo destino ya que tanto el nodo origen como el nodo destino son los extremos de la conexión y no se transmitirá más allá de estos límites.

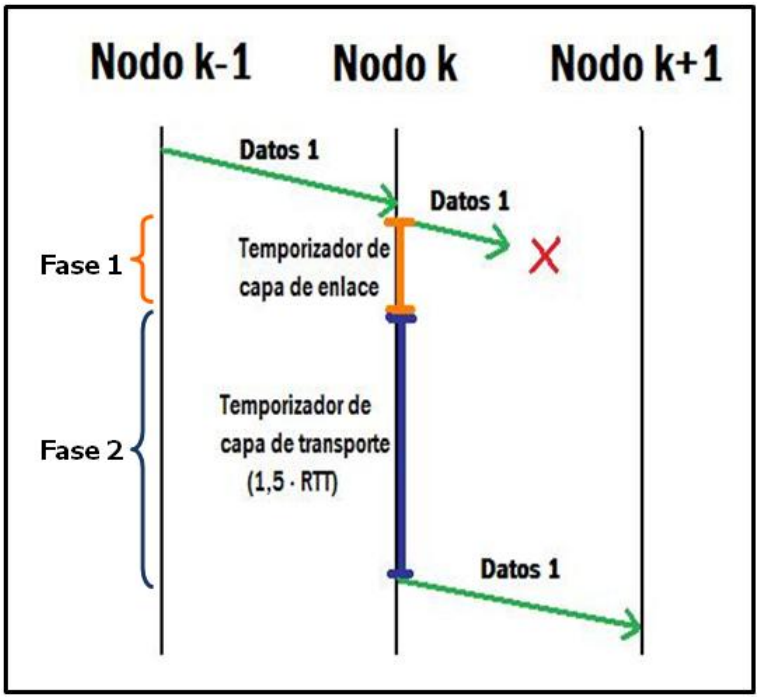

Figura 3.1Esquema de funcionamiento de los temporizadores en TSS

b. Fase 2: Mecanismo de Reconocimiento TCP

Si el nodo no detecta la transmisión del paquete de su nodo vecino al nodo sucesor se considera que el paquete se ha perdido y por lo tanto se activa un temporizador de valor igual a 1,5*RTT **(ver figura 3.1)**, de modo que durante ese instante de tiempo sólo se aceptan reconocimientos TCP ACK. Si esto sucede el paquete almacenado será removido de la caché, si por el contrario este temporizador expira, se procede a la retransmisión del segmento almacenado en la caché y se repite la fase 1 e igualmente si no se cumple entramos en la fase 2.

El valor de temporizador de retransmisión 1.5*RTT debería permitir la recuperación de pérdida de paquete antes que el temporizador de retransmisión de la fuente expire de lo contrario obligaría a una retransmisión extremo a extremo, resultando en un costo energético en todos los nodos que se encuentran en la ruta hacia el nodo destino.

3.1.3 Regeneración y Recuperación de Reconocimientos TCP

Los reconocimientos de TCP (ACK) juegan un papel muy importante para TSS debido a que varios mecanismos como la estimación del RTT, las retransmisiones y el almacenamiento dependen de éstos. La falta o pérdida de los reconocimientos de TCP pueden causar un serio impacto en la cantidad de transmisiones de segmentos de datos TCP. Por lo tanto TSS implementa dos mecanismos que ayudan a disminuir el número de transmisiones de segmentos de datos TCP significativamente.

- Mecanismo de regeneración local de reconocimientos TCP
- Mecanismo de recuperación de reconocimientos TCP

El primer mecanismo se activa cuando un nodo recibe un segmento de datos TCP el cual ya ha sido reconocido por el destino. El segmento de datos TCP es descartado y un reconocimiento TCP con el número de reconocimiento más alto que se haya recibido es regenerado y transmitido hacia la fuente. El mecanismo agresivo de recuperación de reconocimientos TCP retransmite los reconocimientos TCP si un nodo no ha descubierto el envío del reconocimiento TCP por el nodo sucesor. Debido a que los reconocimientos TCP deben usualmente ser enviados sin un retardo significante hacia el emisor de los segmentos de datos TCP, cada nodo mide el tiempo entre su propia transmisión de reconocimiento TCP al nodo sucesor y escucha la transmisión del reconocimiento TCP del nodo sucesor hacia el emisor del segmento de datos TCP. El tiempo de expiración para retransmisión de reconocimientos es configurado al doble valor promedio (utilizando promedio exponencial). Después que este tiempo expira, un reconocimiento TCP es retransmitido utilizando el número de reconocimiento más alto que se haya visto.

3.1.4 Control de Congestion "Backpressure"

Si el sucesor de un nodo no ha enviado todos los paquetes recibidos, puede existir un problema en la red. Por ejemplo que la red puede estar congestionada o el envío de un paquete no progresa debido a que un segmento de datos TCP perdido necesita ser recuperado primero.

Si un nodo continuara con el envío del paquete en caso de congestión, el riesgo de transmisiones innecesarias sería muy alto y por lo tanto un segmento enviado puede fácilmente perderse. Lo mismo podría suceder en el caso de un paquete perdido debido a errores de bit, en tal situación todos los cachés en los subsecuentes nodos están ocupados y la transmisión de un nuevo paquete no sería protegida por el almacenamiento (caching). Por esta razón un nodo TSS detiene cualquier envío de subsecuentes paquetes hasta que éste conozca que todos los paquetes anteriores han sido recibidos y enviados por su nodo sucesor.

Si el envío del paquete se detiene en algún punto, todos los demás nodos en la cadena detrás del nodo detenido también detendrán sus transmisiones hasta que se detecte el progreso o la continuación del envío de paquetes en sus respectivos nodos sucesores.

También se puede optar por implementar el mecanismo de “backpressure congestion” en el nodo emisor de manera que éste pueda ajustar el tamaño de la ventana en caso de que se detecte que los paquetes anteriores no han sido transmitidos por su nodo sucesor, además se recomienda no incrementar la ventana de congestión de TCP siempre y cuando existan un cierto número de paquetes esperando en el emisor para ser transmitidos.

Cabe señalar que en caso que ocurra esta situación, el algoritmo de TSS no especifica la forma en que se comunica al nodo emisor que debe ajustar la ventana o detener la transmisión de los paquetes subsecuentes.

Con todo esto, podemos resumir que la utilización de TSS en la red de sensores inalámbricas proporciona las siguientes ventajas:

1. Reducción del número total de segmentos de datos TCP transmitidos por los nodos de la red. Esta disminución se debe a que en TSS las retransmisiones pueden generarse localmente en los nodos intermedios y no sólo en el nodo cliente.

2. Disminución del número de retransmisiones extremo a extremo, gracias a que los temporizadores de retransmisión en los nodos intermedios permiten que se recuperen la mayoría de los paquetes perdidos, antes que expire el temporizador de retransmisión del origen.

3. Reparto más equitativo del consumo de energía asociado a la transmisión de paquetes, el cual pasa de concentrarse en los nodos más cercanos al cliente a distribuirse por toda la red.

3.2 Implementación de TSS

Para la implementación de TSS se utilizó la pila uIP, la cual ofrece las principales funcionalidades TCP/IP y ofrece soporte a los protocolos IP, ICMP, TCP y UDP, como ya se ha mencionado en el capítulo 2. Esta pila viene incluida en el sistema Operativo Contiki también desarrollado por Adam Dunkels y su equipo del SICS (Swedish Institute of Computer Science) el cual ha sido escrito completamente en C++ con licencia de código abierto y con características multitarea, carga dinámica de módulos y alta portabilidad. Un resumen de las características de Contiki y su comparación con otros sistemas operativos se presenta en el ANEXO B.

Esta pila ha sido acondicionada con una serie de modificaciones basadas en el algoritmo de TSS **(ver ANEXO C)**, con el objetivo de lograr una correcta operación del mecanismo de TSS, las cuales se presentan a continuación:

1. TSS está orientado para trabajar en una topología de red en la que solo existe una única ruta multi-salto para comunicar al origen con el destino y viceversa **(ver figura 3.2)**, Por lo tanto, se han modificado las funciones de descubrimiento de rutas dinámicas para que éstas sean estáticas, de manera que cada nodo previamente conoce el camino que debe seguir para llegar tanto al origen como al destino a través de su nodo vecino. Con esto creamos una topología simétrica donde los paquetes que se transmitan hacia el destino o hacia el origen pasarán por los mismos nodos.

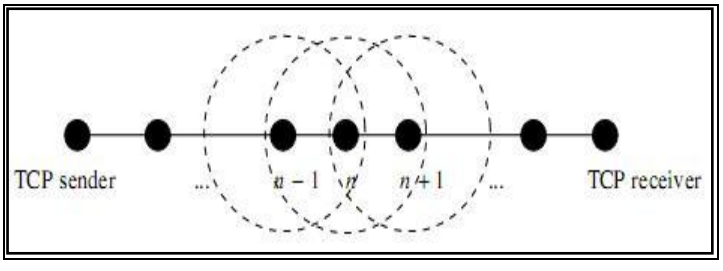

Figura 3.2 Topología de la red de 7 sensores donde se ha implementado TSS
Tomado de [12]

2. Otra modificación a la pila uIP que se ha realizado para lograr una adecuada implementación de TSS, es dotarla de un mecanismo Go-Back-N similar a la de TCP (donde N es un parámetro configurable del sistema) con lo que se eliminaría la limitación que posee la pila original de utilizar un mecanismo Stop & Wait como se ha explicado en el capítulo 1.

   El funcionamiento del mecanismo Go-Back-N permite la transmisión por parte del cliente de N segmentos TCP de forma secuencial y la posterior activación de un temporizador con el que se espera recibir confirmación del último segmento enviado antes que expire, como se muestra en la figura 3.3.

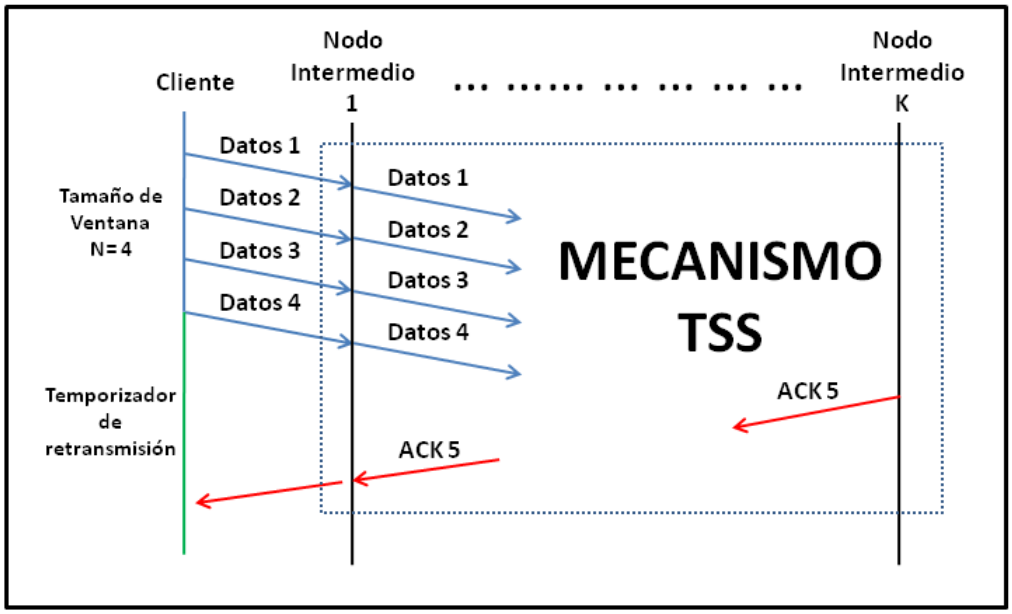

Figura 3.3 Esquema de funcionamiento de TSS con un mecanismo de transmisión Go-Back-N con valor N=4

En caso que el temporizador expire, y el nodo emisor no haya recibido la notificación del último segmento, se procederá a la retransmisión de toda la ventana de N segmentos y a una nueva activación del temporizador. En caso que se reciba un reconocimiento del último paquete, el emisor procederá a la transmisión de los siguientes N segmentos de datos.

Debido a que la pila uIP no mantiene un registro de los paquetes enviados, su funcionamiento no sería compatible para que el sistema de Go-Back-N pudiera realizar la retransmisión de los N segmentos de la ventana por lo tanto, hemos modificado esta función de manera que cada vez que el nodo origen transmita un paquete de la ventana éste se almacene en un búfer desde donde se

gestionarán las retransmisiones que sean necesarias. Cabe señalar que creando un búfer estamos consumiendo mayores recursos de memoria del dispositivo, de esto se puede concluir que el número de segmentos a transmitir por ventana guarda una estrecha relación con la cantidad de memoria que se debe reservar para almacenar los segmentos que se envían por lo tanto, el parámetro N del sistema Go-Back-N debe ser ajustado tomando en consideración los recursos de memoria con que cuenta el dispositivo.

Finalmente, al igual que DTC, las variables temporales asociadas a la transmisión de ráfagas (ventana de paquetes) TCP deben ser ajustadas correctamente para minimizar pérdidas de paquetes que pueden ocasionarse si tanto la caché como el búfer temporal de un nodo intermedio estuvieran ocupados.

3. Como ya se ha mencionado antes, el mecanismo de "backpressure congestion" requiere que TSS no transmita inmediatamente un paquete, como lo hace DTC, sino que lo mantenga almacenado hasta que detecte que los paquetes transmitidos anteriormente hayan sido recibidos y transmitidos por el nodo sucesor. Por lo tanto se ha dotado a la pila uIP de un búfer adicional para el almacenamiento temporal de un paquete que esté a espera de ser transmitido. Este requerimiento, añade una mayor demanda de recursos de memoria del dispositivo.

4. Aunque la pila uIP ya hace cálculos de RTT, en este proyecto se ha utilizado un mecanismo más simple y con menos consumo en términos de procesado para los nodos intermedios.

   Su funcionamiento se basa en calcular el RTT en el establecimiento de la conexión TCP. Se toman marcas de tiempo en cada nodo intermedio en el momento de transmitir el mensaje SYN proveniente del nodo cliente a través de la red y al recibir el SYN/ACK del nodo servidor **(ver figura 3.4)**

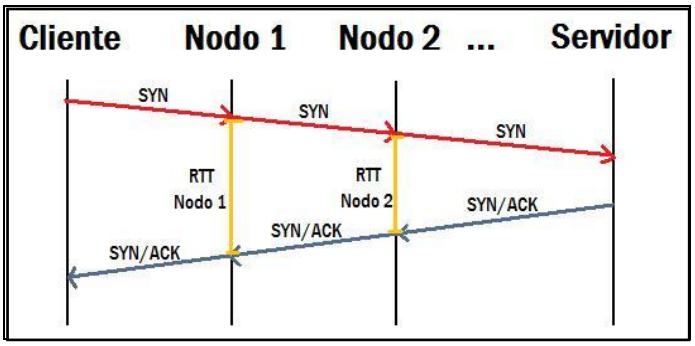

Figura 3.4 Obtención del RTT (Round Trip Time) por parte de los nodos intermedios

La determinación del RTT sólo se realiza al inicio de la conexión para tener así una estimación real. No se ha implementado un mecanismo adaptativo de cálculo del RTT debido a que añadiría complejidad y supondría un mayor coste computacional al sistema, lo que repercutiría en un mayor consumo de energía.

El impacto del cálculo del RTT en los nodos intermedios debe ser estudiado y analizado con mayor detalle, por lo tanto se propone como una línea futura de investigación, tal y como se recoge en el capítulo 4 del presente trabajo, dedicado a conclusiones y líneas futuras.

5. Como TSS utiliza un mecanismo de recuperación de reconocimiento TCP, a los nodos intermedios se les ha dotado de un búfer adicional donde se almacenen los segmentos TCP ACK y desde donde se gestione la retransmisión de estos segmentos cuando detecte que el nodo vecino en la ruta hacia el nodo origen no ha transmitido el segmento ACK hacia su nodo sucesor.

6. Tomando en cuenta que TSS utiliza un mecanismo cross-layer de confirmación implícita de capa de enlace, se han realizado modificaciones al mecanismo de transmisión unicast, de manera que cada nodo esté capacitado para escuchar la transmisión de paquete de su vecino y detectar cuando el paquete ha sido transmitido al nodo sucesor de éste.

   De esto se concluye que un nodo debe mantenerse más tiempo activo (despierto) del que necesita para transmitir o recibir un paquete, debido a esto, se ha tenido que utilizar el protocolo NULLMAC el cual mantiene siempre encendida la interface radio del dispositivo lo que conllevará a un mayor consumo de energía. El estudio de un protocolo de capa de enlace adecuado

con este tipo de comportamiento no ha sido considerado dentro de este proyecto y queda para investigaciones futuras.

Como se ha señalado anteriormente, este mecanismo de cross-layer de confirmación implícita solo es viable en los nodos intermedios, el algoritmo de TSS no contempla la manera en que el nodo intermedio cercano a la fuente tiene conocimiento que ésta ha recibido el ACK de nivel de transporte. Ésta situación se repite de manera similar en el nodo intermedio más cercano al destino, debido a que el algoritmo de TSS tampoco especifica cómo este nodo intermedio determina que el nodo destino ha recibido el segmento TCP.
Para resolver estos inconvenientes se han realizado, como aporte del proyecto, las siguientes adiciones al funcionamiento del algoritmo TSS:

a) Para el primer caso se hace uso de un mecanismo de cross-layer de confirmación explícita de capa de enlace, que se basa en que la fuente una vez reciba un ACK de nivel de transporte, transmitirá un mensaje de confirmación (ACK) de capa de enlace hacia el nodo intermedio. Este nodo ya habrá activado un temporizador de capa de enlace durante el cual espera recibir esta confirmación explícita de lo contrario retransmitirá el ACK de capa de transporte, activando nuevamente el temporizador de capa de enlace **(ver figura 3.5a).**

b) En cambio para el segundo caso, el nodo cercano al destino una vez transmita el paquete al destino iniciará un temporizador con valor 1.5 * RTT durante el cual esperará recibir un ACK TCP, que debe ser generado por el destino debido a que ha recibido un segmento de datos TCP, si esto no ocurriera el segmento almacenado se retransmite **(ver figura 3.5b)**. Note que este nodo cercano al destino realiza solo la fase 2 descrita en el apartado “Retransmisiones Locales” de la sección 3.1

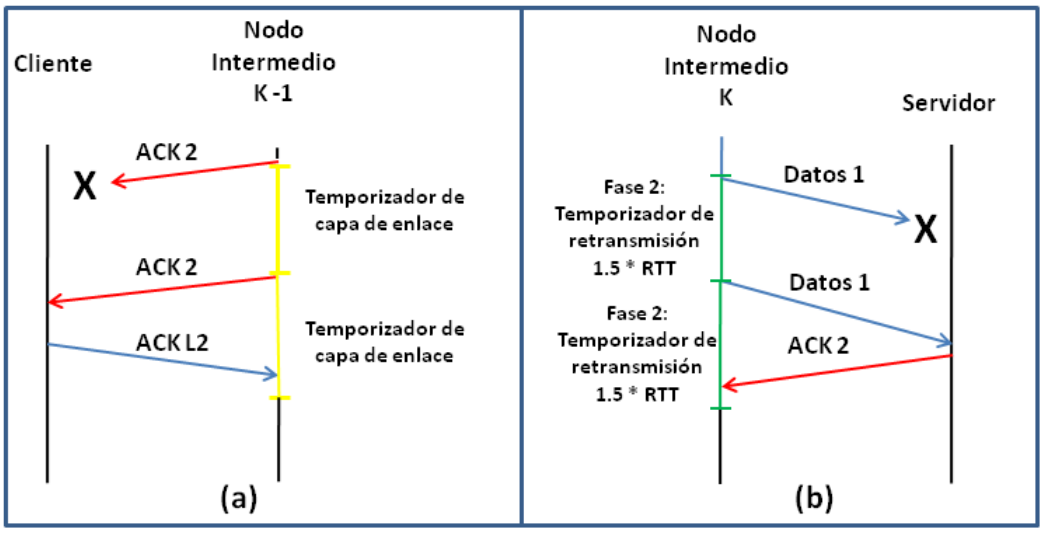

Figura 3.5 Esquema de detección y recuperación de errores de los nodos intermedios cercanos a la fuente y al destino respectivamente.

7. El algoritmo de TSS requiere que un nodo intermedio recupere en caso de pérdida tanto los reconocimientos como los segmentos de datos TCP, por medio del mecanismo cross-layer de confirmación implícita. En las pruebas que hemos realizado hemos encontrado que existen ocasiones en que la llegada de un segmento TCP y un reconocimiento TCP ocurre de manera concurrente en un nodo, en este caso el nodo enfrenta problemas en el proceso de escucha de la transmisión ya sea de su nodo vecino hacia el origen o de su nodo vecino hacia el destino. Esto trae consigo que se retransmita un segmento o un reconocimiento TCP adicional o en circunstancias de alta probabilidad de error la inestabilidad en el funcionamiento del algoritmo de TSS.

   Considerando este caso hemos realizado la siguiente modificación del comportamiento del algoritmo como aporte de este proyecto:

   - Solo se enviará a la fuente el reconocimiento TCP correspondiente al último segmento de la ventana por medio de una función de filtrado ubicada en el nodo más cercano al destino. Esta solución se considera viable aprovechando que el mecanismo de Go-Back-N con que funciona la fuente permite este comportamiento. De esta manera reducimos la posibilidad de una concurrencia de segmentos de datos y de reconocimientos TCP en un nodo. Además debido a que estamos utilizando el protocolo NULLMAC el cual no implementa ningún mecanismo de colisión en el medio, con esta solución disminuimos en gran medida la posibilidad de colisiones en el medio.

8. Otro aporte particular muy similar al mencionado en el punto anterior ha sido implementado en el nodo más cercano a la fuente, se basa en que este nodo solo enviará los segmentos TCP que sean consecutivos al último segmento confirmado por el nodo sucesor. Esto eliminaría la introducción de paquetes redundantes en la red **(ver figura 3.6)**.

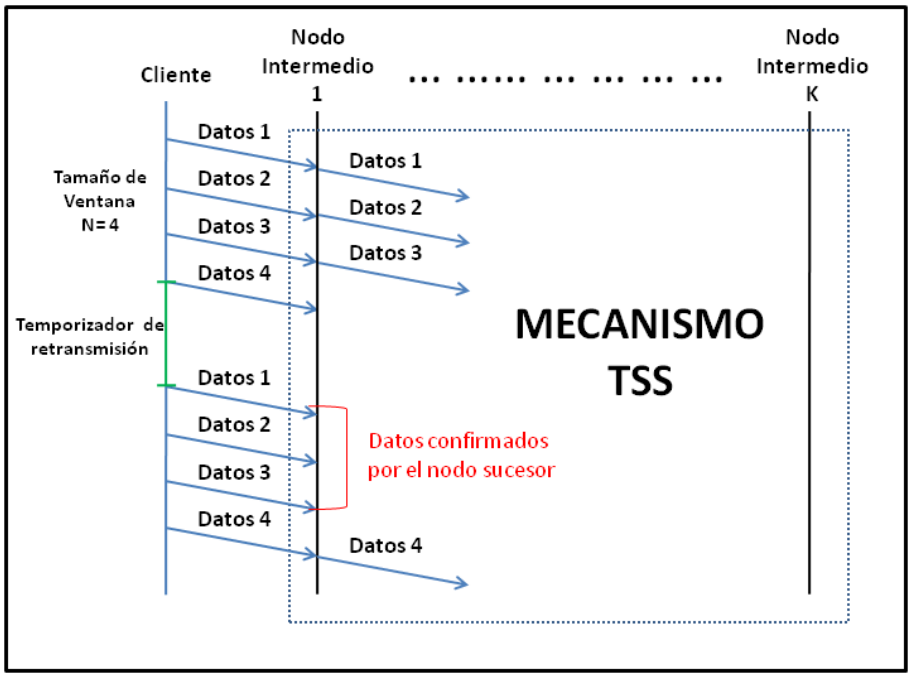

Figura 3.6 Mecanismo de filtrado del nodo intermedio más cercano al nodo cliente.

9. Se ha necesitado la utilización de un nuevo canal de comunicación para el envío y la recepción de los reconocimientos a nivel de enlace entre la fuente y su nodo vecino más cercano debido a que se necesita para el funcionamiento del mecanismo de confirmación explícita propuesto de manera particular para este proyecto.

10. Se ha diseñado una función de probabilidad con la finalidad de simular las condiciones de error que pueden existir en una aplicación dentro de un entorno real. Estas probabilidades de error afectan la recepción de un segmento TCP, un reconocimiento TCP o un reconocimiento a nivel de enlace en un nodo tanto intermedio como en la fuente y en el destino.

3.3 Implementación de TSS en CONTIKI

Además de las modificaciones realizadas concretamente en la pila uIP, también se han hecho necesarios algunos cambios y adiciones en otras funcionalidades que son proporcionadas por el sistema Operativo Contiki y que describiremos en esta sección.

1. Funcionamiento de la transmisión unicast entre los nodos.

La transmisión de paquetes de forma unicast utilizada en Contiki radica en el envío de un paquete que contiene la dirección del nodo destino de capa de enlace, sin embargo esta transmisión se realiza enviando este paquete en un mensaje broadcast a toda la red. Todos los nodos escucharán el paquete pero solo aquel al cual está dirigido lo procesará.

Este funcionamiento ha sido ligeramente modificado para poder implementar el mecanismo cross-layer de confirmación implícita. Cuando el paquete es escuchado por un nodo este puede analizar los campos de origen y destino de la capa de enlace y de acuerdo a esto detectar si el paquete pertenece a una transmisión que realiza su nodo vecino hacia el nodo sucesor.

El archivo que rige el funcionamiento de la transmisión unicast en Contiki se encuentra bajo la ruta ***core/net/rime/unicast.c*** del directorio raíz del sistema operativo.

2. Funcionamiento del descubrimiento de rutas

La forma en que se realiza el descubrimiento de rutas proporcionado por Contiki está basada en el protocolo de enrutamiento AODV. Para este proyecto se hace necesario que las rutas sean simétricas y estáticas, por lo tanto, se ha asignado a cada nodo unas entradas estáticas en la tabla de enrutamiento de este protocolo para lograr este objetivo.

El archivo que rige el funcionamiento de este protocolo de enrutamiento en Contiki se encuentra bajo la ruta ***core/net/uip-over-mesh.c*** del directorio raíz del sistema operativo.

3. Protocolo MAC

Por defecto el sistema operativo Contiki trabaja con el protocolo X-MAC el cual está orientado a reducir los periodos de escucha con el fin de reducir los costos de energía asociados a los periodos de inactividad.

El protocolo X-MAC sería incompatible para la implementación de TSS debido a la necesidad del mecanismo de cross-layer de confirmación implícita que se requiere en los nodos intermedios. Es por ello que se ha utilizado el protocolo NULLMAC que radica su funcionamiento básicamente en mantener la interfaz radio encendida para cumplir una de las funciones requeridas por TSS.

El funcionamiento de este protocolo debe ser activado en el archivo principal de configuración correspondiente al hardware que se utiliza. En nuestro caso hemos utilizado los motes Crossbow TelosB Mote TPR2420, por lo tanto este archivo se encuentra en la ruta ***platform/sky/contiki-sky-main.c*** del directorio raíz del sistema operativo.

4. Protocolo TCP/IP

Por defecto, el sistema operativo Contiki no tiene activada la pila uIP, esto debe ser realizado en el archivo de configuración correspondiente al hardware que se utiliza. Igualmente que en el punto anterior, este archivo se encuentra en la ruta platform/sky/contiki-sky-main.c del directorio raíz del sistema operativo.

Además las modificaciones que se realizaron en el funcionamiento para adecuarla a TSS, fueron realizadas en el archivo que se encuentra en la ruta core/net/uip.c del directorio raíz del sistema operativo.

5. Mecanismos de TSS adicionales

Las funcionalidades particulares de TSS como son el almacenamiento en los búferes de acuerdo al tipo de paquetes y su correspondiente tratamiento y gestión ha sido programadas utilizando las características multitarea que proporciona Contiki, las cuales se encuentran manejadas por dos procesos y se encuentran igualmente en la ruta ***core/net/uip-over-mesh.c*** del directorio raíz del sistema operativo.

3.4 Resultados

Una vez que hemos analizado cada uno de los mecanismos que proporciona TSS para mejorar el comportamiento de TCP en las redes de sensores inalámbricas, nos proponemos realizar una serie de pruebas en hardware con el objetivo validar el funcionamiento de TSS bajo diferentes condiciones de errores y compararlas con los resultados obtenidos en las simulaciones realizadas por Adam Dunkels con el simulador OMNet++. De esta forma podemos analizar los retos de la implementación de éste protocolo en este tipo de entornos.

Cabe señalar que los resultados obtenidos de las simulaciones realizadas por Adam Dunkels se basaron en una topología de 11 nodos y una transmisión de 1000 segmentos TCP en cambio los resultados que presentamos en este apartado fueron obtenidos de las diferentes pruebas que se realizaron en una implementación real que consistió en una red de 7 nodos sensores inalámbricos y en todas las pruebas se transmitieron 100 segmentos TCP con una carga de datos de 70 bytes y el hardware utilizado fue Crossbow TelosB Mote TPR2420 **(ver figura 3.7)**.

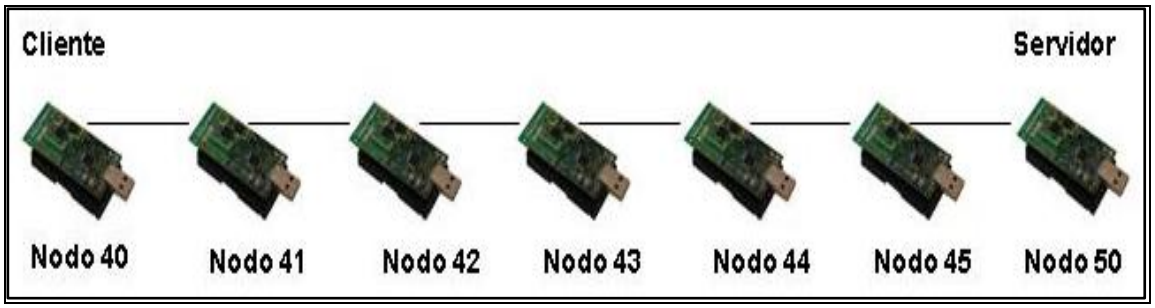

Figura 3.7Escenario de pruebas y nomenclatura de los nodos utilizados en la red

Además, los datos recogidos corresponden al periodo comprendido entre la transmisión del primer paquete de datos por parte del nodo cliente y la recepción del último reconocimiento por parte de este mismo nodo. Los valores mostrados equivalen a la media aritmética de los resultados obtenidos tras 10 repeticiones de la misma ejecución.

Las pruebas se han llevado a cabo con diferentes probabilidades de error, de forma que una vez fijada la probabilidad de error para el envío de un segmento de datos, ésta se ve reducida a la mitad para un reconocimiento a nivel TCP y a un cuarto para un reconocimiento a nivel de enlace. A continuación se muestra en la tabla 3.1 las

diferentes configuraciones utilizadas así como la nomenclatura que se utilizará en la presentación de los resultados en las tablas y gráficos de este apartado.

P_{error} paquete	P_{error} ACK	P_{error} ACK (Link Layer)	Nomenclatura
0 %	0 %	0 %	0/0/0
1,25 %	2,5 %	5 %	1,25/2,5/5
2,5 %	5 %	10 %	2,5/5/10
3,75 %	7,5 %	15 %	3,75/7,5/15

Tabla 3.1Configuración de las probabilidades de error

Otro objetivo de este apartado es comparar los resultados obtenidos con TSS en este trabajo, con los (resultados) obtenidos por las experimentaciones reales con DTC realizadas en [14] y de esta forma verificar cuál de estas dos propuestas tiene un comportamiento en términos de eficiencia energética y rendimiento en las redes de sensores inalámbricas.

La tabla 3.2 presenta un resumen de los resultados obtenidos en ambos trabajos y en la cual se puede observar que la disminución en la transmisión de paquetes es mayor con TSS que con DTC a medida que se van incrementando las probabilidades de errores, esta diferencia radica en el mecanismo de recuperación de ACK que implementa TSS y que ayuda a disminuir las retransmisiones extremo a extremo. Además la cantidad de reconocimientos TCP transmitidos también es considerablemente menor con TSS debido al mecanismo de filtrado implementado en el nodo intermedio más cercano al destino. Así mismo, la cantidad de reconocimientos de enlace de datos (ACK L2) transmitidos por TSS es muy inferior a DTC, debido a la implementación particular del mecanismo de confirmación explícita entre la fuente y su nodo intermedio más cercano.

	0/0/0		1,25/2,5/5		2,5/5/10		3,75/7,5/15	
	Con TSS	Con DTC	Con TSS	Con DTC	Con TSS	Con DTC	Con TSS	Con DTC
Paquetes Enviados	600	600	668	679,86	733,1	776,60	827,1	952,75
ACK enviados	225	525	237,9	519,29	240,8	524,20	257,5	536,00
ACK (Link Layer) enviados	25	600	28,4	647,29	30,8	704,50	35,4	814,25
Retransmisiones TCP end to end	0	0	8,4	10,14	15.8	20,90	26,6	40,00

Tabla 3.2Comparación de resultados TSS y DTC

Una explicación más detallada de estas comparaciones se presenta en los apartados 3.4.1.3 y 3.4.2.1 de este capítulo.

3.4.1 Transmisión de Paquetes

Una de las principales ventajas que ofrece TSS, es la reducción de la cantidad de paquetes transmitidos por cada uno de los nodos en la red, de esta manera se hace uso eficiente del consumo de energía en cada nodo.

En las pruebas realizadas, podemos señalar que en ausencia de errores TSS se comporta igual que TCP sin TSS, como se muestra en la figura 3.8, debido a que no se hace uso de los mecanismos de recuperación de pérdida de paquetes.

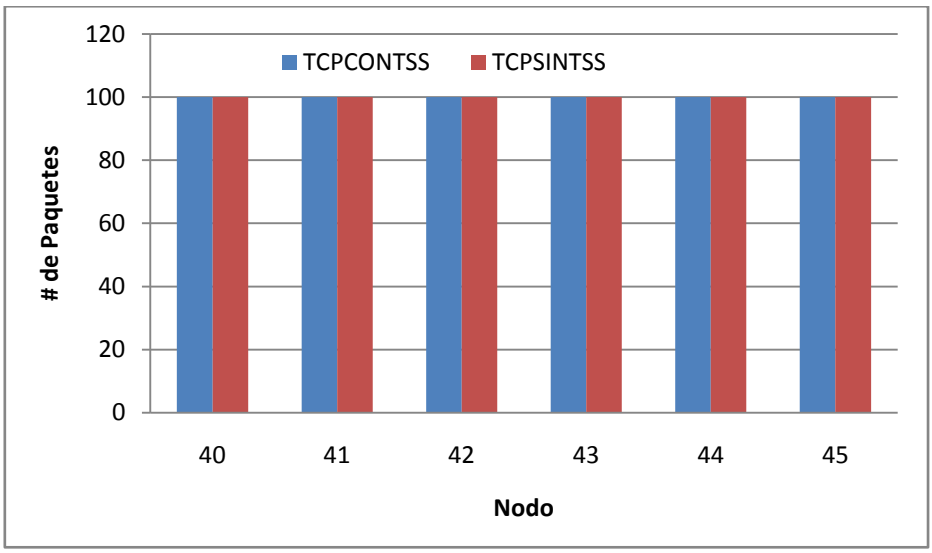

Figura 3.8 Transmisión de Paquetes en ausencia de errores

A medida que las condiciones de errores van en aumento debido a las diferentes configuraciones de probabilidades de error distribuidas en cada nodo de la red, **(ver figuras 3.9, 3.10, 3.11)**, se puede observar que TSS mantiene la cantidad de paquetes transmitidos casi equitativamente distribuidos entre los nodos, esto se debe a que solo los paquetes necesarios (los aceptados según el algoritmo de TSS **(Ver Anexo C)** y los retransmitidos por pérdida) son transmitidos hacia el siguiente nodo.
Una situación contraria se puede observar si utilizamos TCP sin TSS, donde el aumento en las probabilidades de error resulta en una mayor cantidad de paquetes transmitidos por todos los nodos de la red. Esta situación se refleja más en el nodo fuente, en el que esta cantidad puede llegar a ser casi un 70% más que TSS, en el peor de los casos (configuración 3,75/7,5/15) para el mismo nodo, dando como resultado un mayor consumo de energía **(ver figura 3.11).**

Otro aspecto importante a señalar en cada una de estas gráficas con diferentes configuraciones de probabilidades de error, es que con TSS, las retransmisiones se

realizan desde el nodo intermedio que detecta la pérdida de paquete y no desde el nodo origen como lo hace TCP sin TSS, por lo que disminuye la cantidad de paquetes transmitidos por el nodo origen, como también se puede notar en las figuras anteriormente mencionadas. De esta manera se cumple con uno de los objetivos que intenta resolver TSS según el estudio teórico realizado por Adam Dunkels en [14].

También se puede observar que con la máxima configuración de probabilidad de error (3,75/7,5/15), el protocolo TSS incrementa solo en un 28 % la transmisión de paquetes en cada nodo intermedio, comparado con TCP sin TSS que la incrementa en un 80%, por lo tanto esta situación demuestra que TSS siempre busca reducir las transmisiones de paquetes de los nodos intermedios, favoreciendo el ahorro de energía en los nodos.

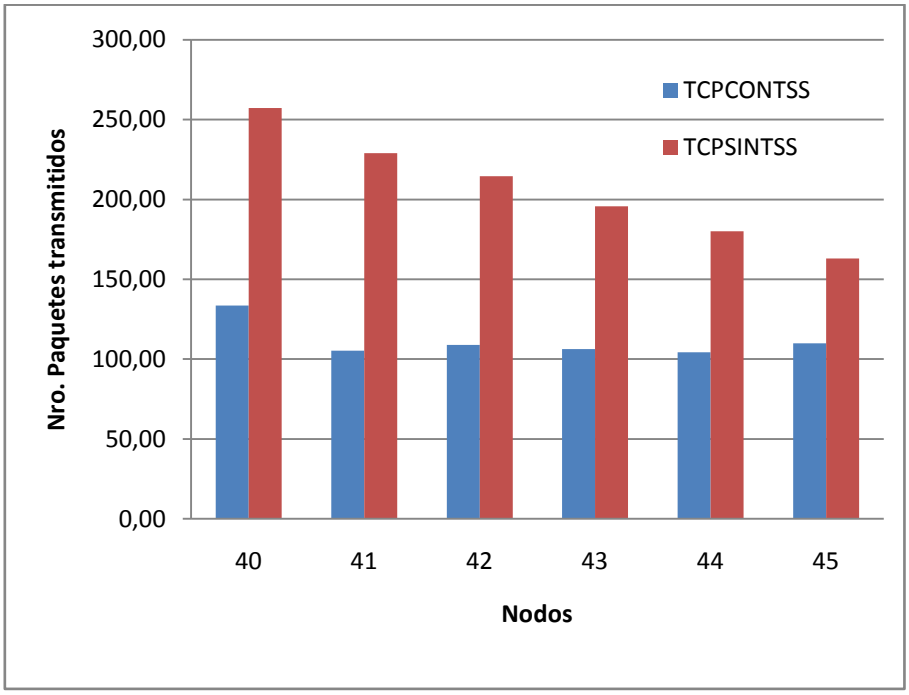

Figura 3.9 Transmisión de Paquetes con 1,25/2,5/5 % de error

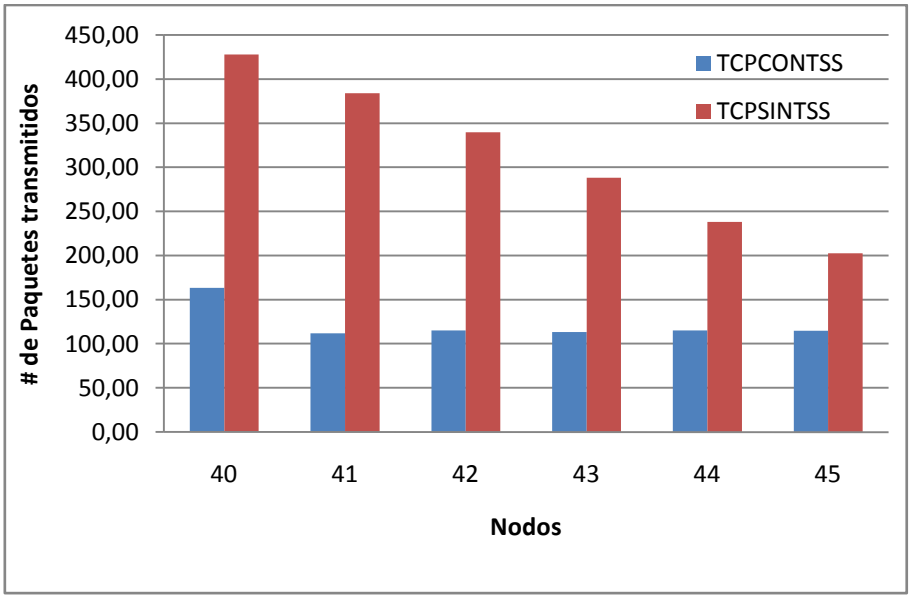

Figura 3.10 Transmisión de Paquetes con 2,5/5/10 % de error

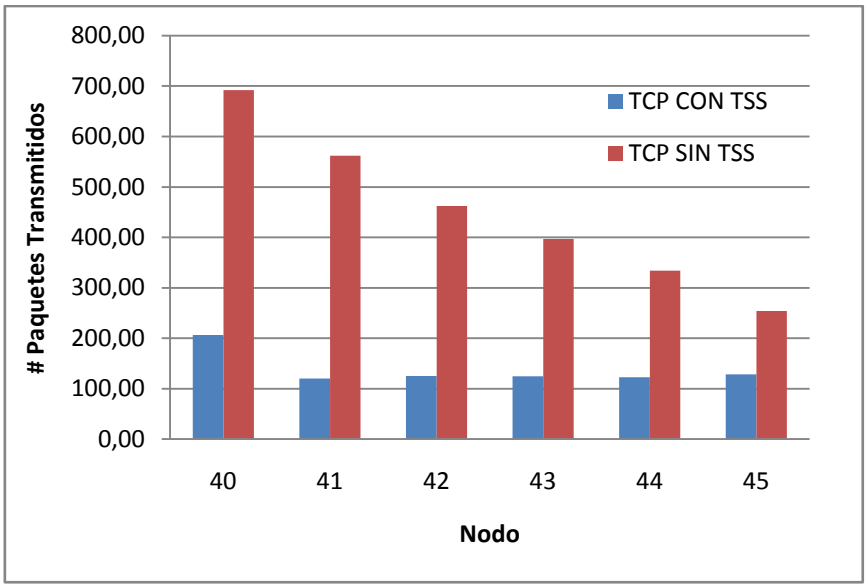

Figura 3.11 Transmisión de Paquetes con 3,75/7,5/15 % de error

En la figura 3.12 se puede observar que a pesar del aumento en las probabilidades de error, la cantidad de paquetes transmitidos de los nodos intermedios es bien reducida en comparación con el nodo origen, llegando a alcanzar una disminución relativa de entre un 20% y 40%, para las configuraciones (1,25/2,5/5)% y (3,75/7,5/15) respectivamente, esto se debe también al efecto que tiene el filtrado que se realiza en el nodo intermedio más cercano al nodo origen (nodo 41), que evita que los paquetes que ya han sido confirmados por el nodo sucesor (del nodo 41) sean introducidos a la red.

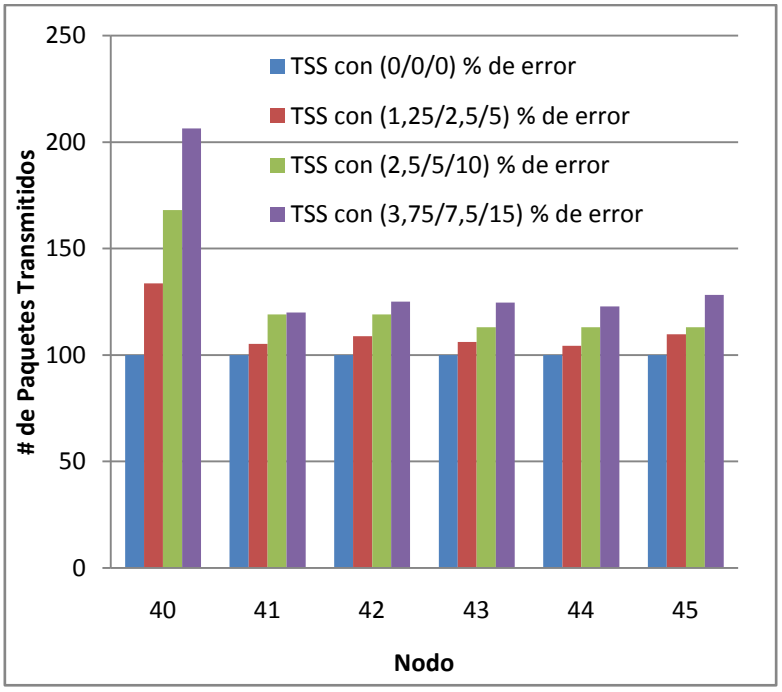

Figura 3.12 . Transmisión de Paquetes con TSS, mostrando las diferentes probabilidades de error distribuida en todos los nodos.

3.4.1.1 Medición del RTT y Cálculo del RTO

Otra de las fortalezas que presenta TSS es que cada nodo intermedio tiene la capacidad de realizar las retransmisiones de los paquetes que se detecten como perdidos, antes que el temporizador de retransmisión del nodo origen expire, lo que evita que la fuente tenga que realizar una retransmisión extremo a extremo.

Esto es posible debido a que el temporizador de retransmisiones en cada nodo intermedio es calculado como RTO =1.5*RTT, donde RTT es el tiempo de ida y vuelta de un paquete medido desde cada nodo intermedio hasta el nodo destino, lo que implica que a medida que la distancia entre estos nodos sea más corta el valor de RTT sea menor y por lo tanto cada valor de RTO también sea menor que el RTO del nodo origen.

La figura 3.13 muestra los valores de RTT y su correspondiente valor de RTO calculado en cada nodo intermedio. En las pruebas realizadas el valor inicial del RTO del nodo origen es de 3 segundos que es el establecido por defecto en la pila uIP y ha resultado adecuado para las pruebas realizadas.

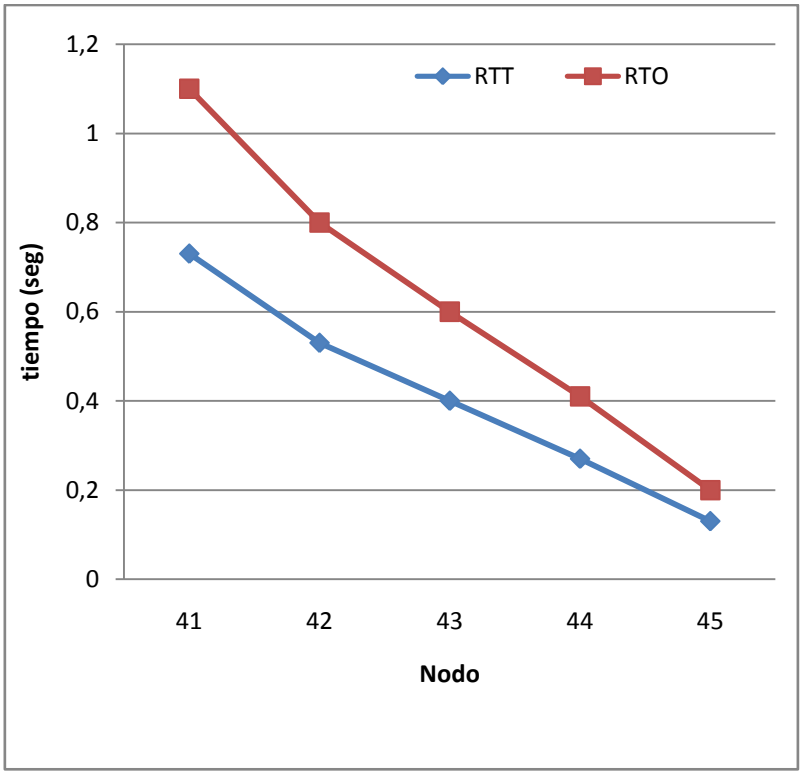

Figura 3.13 Medición del RTT y Cálculo del RTO en cada nodo

3.4.1.2 Efecto de la transmisión ordenada de paquetes.

Otro aspecto a tener en cuenta es que TSS asegura la llegada ordenada de los paquetes al destino, lo que evita el re-secuenciamiento de búfer en el destino. Sin embargo, para que esto suceda los nodos intermedios deberán descartar los paquetes que no sigan la secuencia de los paquetes transmitidos. Esto trae como consecuencia retransmisiones adicionales de paquetes y por ende un mayor consumo de energía en los nodos.

En las pruebas realizadas hemos encontrado que esta situación se refleja cuando se produce una pérdida de paquete en el nodo más cercano a la fuente (nodo 41) lo que conduce al descarte de los subsecuentes paquetes de la ventana por parte de este nodo como se puede observar en la figura 3.14. Esta cantidad de paquetes descartados representa entre 7% y 10% de los paquetes transmitidos por la fuente, dependiendo de la configuración de probabilidad de error.

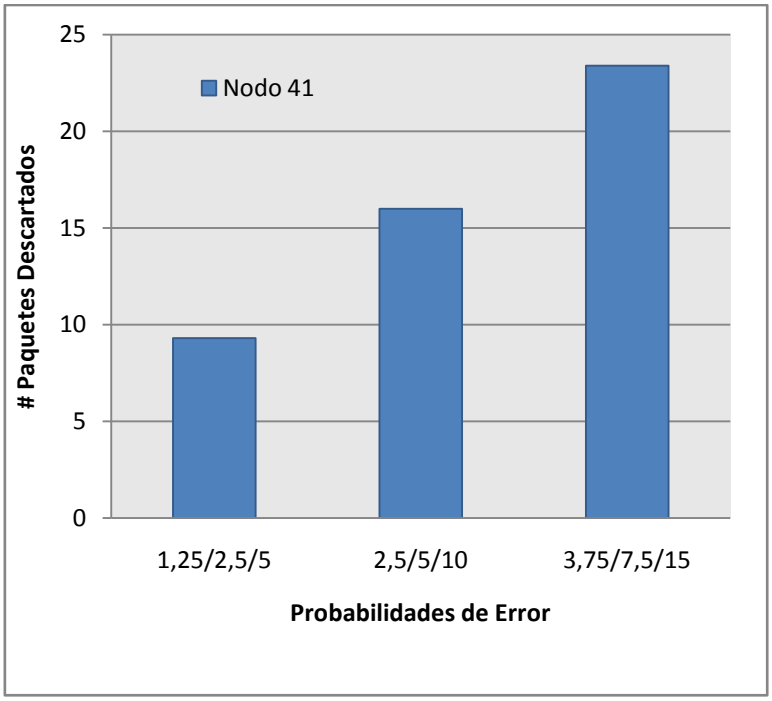

Figura 3.14 Efecto del número de paquetes descartados en el nodo 41 debido a la llegada de paquetes fuera de orden con diferentes probabilidades de pérdida

Además esta situación origina la retransmisión de la ventana por parte del nodo origen. Una vez se realice esta retransmisión habrán paquetes que ya fueron recibidos por el nodo intermedio anteriormente, por lo que también serán descartados, tal y como se muestra en la figura 3.15.

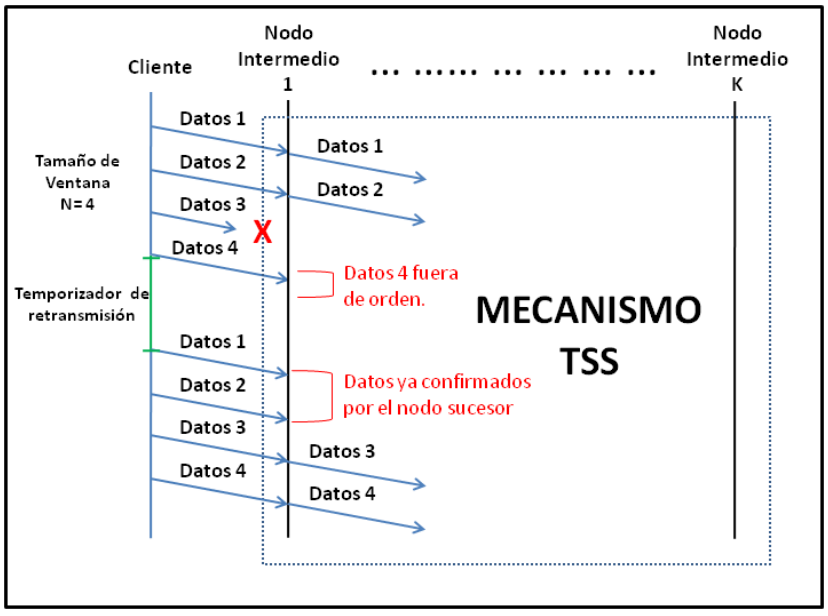

Figura 3.15 Efecto de la transmisión de paquetes en fuera de orden

Una posible solución a este inconveniente podría ser la incorporación del mecanismo cross-layer de confirmación implícita en la fuente con el cual pueda detectar la pérdida de paquete de su nodo sucesor y retransmitirlo antes de continuar con el siguiente paquete de la ventana, sin embargo este tema debe ser objeto de un estudio más detallado y se presenta como una línea de investigación futura.

3.4.1.3 Eficiencia de TSS versus DTC en la transmisión de paquetes

El objetivo de este apartado es comparar la eficiencia en la transmisión de paquetes entre los protocolos de transporte para WSN basados en TCP que se conocen hasta el momento, como lo son DTC y TSS. En este sentido comparamos los resultados de la implementación real de DTC realizada en [14] con los obtenidos en este proyecto.

En las figuras 3.16, 3.17 y 3.18 se hace una comparación de la cantidad de paquetes transmitidos por ambos protocolos utilizando las diferentes configuraciones de probabilidades de errores de manera distribuida por igual en todos los nodos. Se puede observar que la cantidad de transmisiones de paquetes realizada en TSS es ligeramente menor (alrededor de un 4%) que DTC en las configuraciones con probabilidades de error 1,25/2,5/5 % y 2,5/5/10 % **(ver figuras 3.16 y 3.17)**, sin embargo con una mayor probabilidad de errores (configuración 3,75/7,5/15%) la diferencia se hace más notable (alrededor de un 11 %), a favor de TSS. **(Ver figura 3.18)**.

Esta diferencia se debe a que TSS activa su mecanismo de recuperación de reconocimientos TCP en caso que se detecte la pérdida de éste, lo que evita las retransmisiones extremo a extremo por parte de la fuente. En cambio DTC, carece de este mecanismo y por lo tanto la fuente realizará retransmisiones extremo a extremo en caso que no reciba el reconocimiento TCP esperado y por ende se transmitirá una cantidad adicional de paquetes.

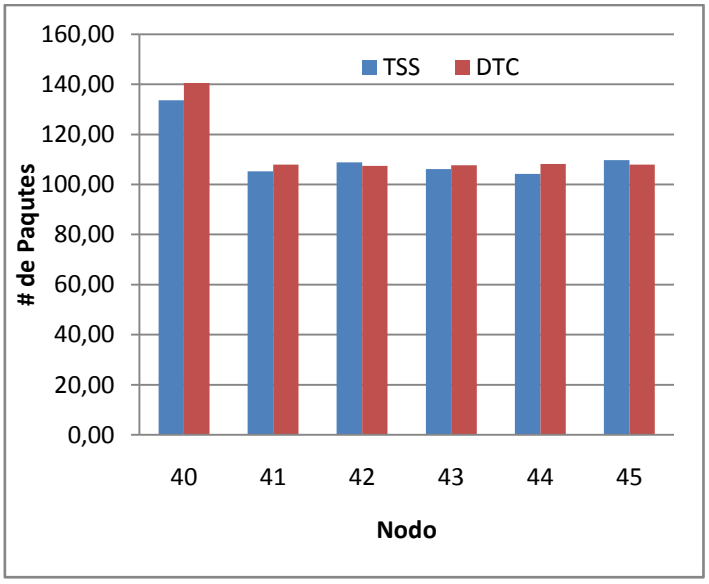

Figura 3.16 Transmisión de Paquetes con 1,25/2,5/5 % de error

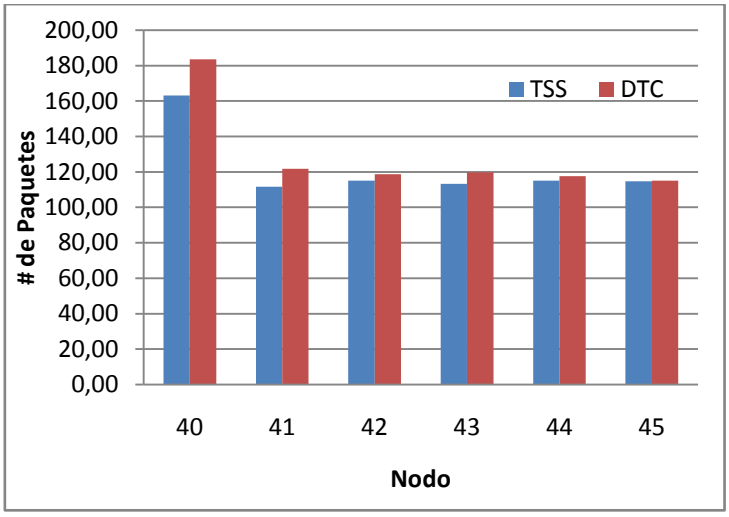

Figura 3.17 Transmisión de Paquetes con 2,5/5/10 % de error

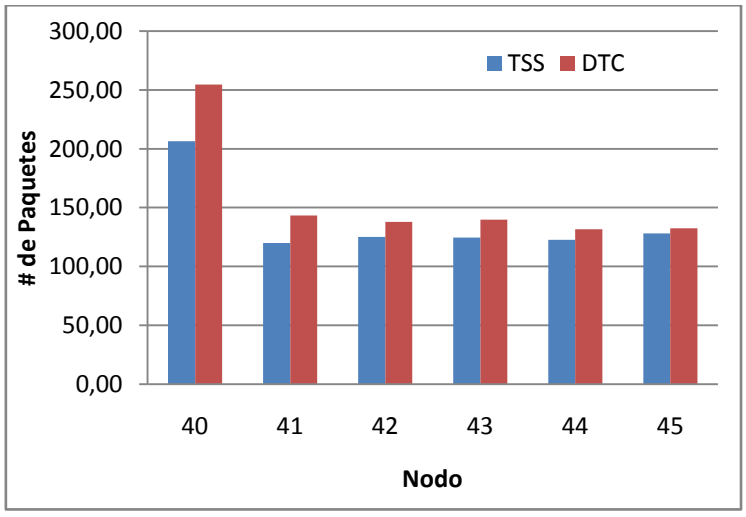

Figura 3.18 Transmisión de Paquetes con 3,75/7,5/15 % de error

3.4.1.4 Throughput de TSS

TSS no fue desarrollado para optimizar el throughput de TCP, ya que su principal objetivo era mantener el número de transmisiones tan bajo como fuera posible. A pesar de esto, en las pruebas realizadas ha quedado demostrado que TSS proporciona un mayor throughput que TCP sin TSS, como se muestra en la figura 3.19.

Para calcular el throughput con TSS y TCP sin TSS, se tomó en cuenta los siguientes aspectos:

- la transmisión de 100 paquetes con una carga útil total por paquete de 127 bytes.
- velocidad de transmisión de 250 Kbps (proporcionada por el hardware TelosB).
- El tiempo total de transmisión es equivalente a la transmisión de 100 paquetes netos (sin contar retransmisiones) hasta recibir el reconocimiento TCP (ACK) del último paquete.
- Tiempo de transmisión de un paquete es de 4 ms aproximadamente.
- Tiempo mínimo total de transmisión teórico de 100 paquetes es de 297,4 segundos
- Throughput máximo teórico es de 0,19 Kbps

La tabla 3.3 resume los tiempos de transmisión total y el throughput resultante para cada una de las configuraciones de probabilidades de errores utilizadas para TCP sin TSS y TCP con TSS.

	0/0/0/		1,25/2,5/5		2,5/5/10		3,75/7,5/15	
	Sin TSS	Con TSS	Sin TSS	Con TSS	Sin TSS	Con TSS	Sin TSS	Con TSS
Tiempo total de transmisión (seg)	412,72	310.15	487,7	331,2	995,5	347,3	1478,3	510,8
Throughput	0,14	0,18	0,11	0,17	0,06	0,16	0,04	0,11

Tabla 3.3 Resumen de Tiempo total de Transmisión y Throughput

Como se puede observar en la gráfica 3.19, el throughput de TSS resultante va desde un 0,18 Kbps (en ausencia de errores) hasta un 0,11 Kbps en el peor de los casos estudiados (configuración 3,75,/7,5/15) comparado con TCP sin TSS que pasa de un 0,14 Kbps a un 0,04 Kbps respectivamente.

Cabe señalar que el motivo de que el throughput sea tan bajo se debe al efecto ocasionado por los intervalos de tiempo de 3 segundos entre paquete, que realiza el nodo origen en la transmisión de cada ventana. Este tiempo de inactividad lleva a una reducción de la utilización del ancho de banda del enlace y por ende a una degradación en el throughput y a un retardo mayor en la comunicación.

Sin embargo este intervalo de tiempo, fue necesario para reducir la pérdida de paquetes que puede ocurrir, en caso de que la caché y el búfer del nodo intermedio estén ocupados, como ya se ha mencionado anteriormente.

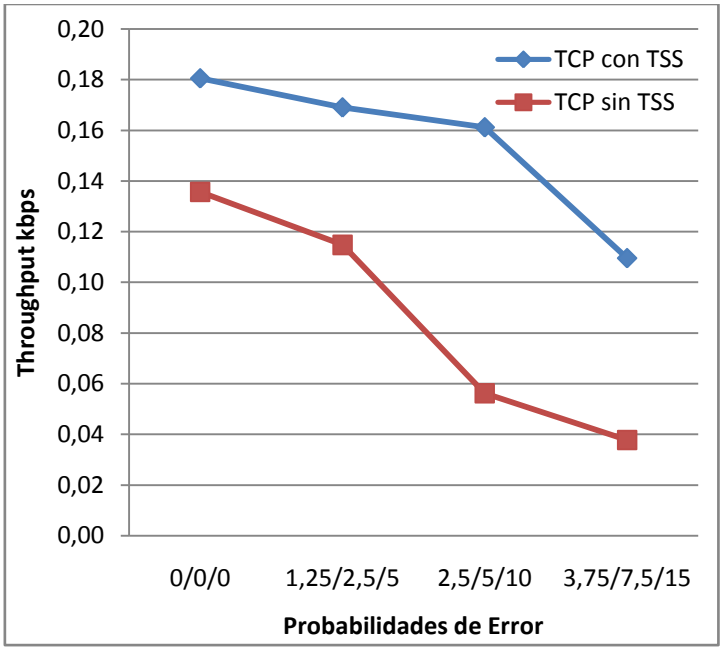

Figura 3.19 Throughput de TSS y TCP sin TSS

3.4.2 Transmisión de ACK'S (Reconocimientos TCP)

De forma similar a lo que se ha comentado en el apartado 3.4.1, TSS se comporta igual y no añade ventaja con respecto a la transmisión de reconocimientos TCP cuando se encuentra en ausencia de errores.

Solo cabe resaltar que en nuestra implementación tanto de TSS como de TCP sin TSS se ha utilizado un filtro en el nodo más cercano al nodo destino (nodo 45), por medio del cual solo se transmitirán los reconocimientos TCP que correspondan al último segmento de la ventana ya que es lo que espera recibir el mecanismo de Go-Back-N implementado en el nodo origen **(ver figura 3.20)**. Con este filtro se trata de disminuir las colisiones en el medio inalámbrico y la congestión en los nodos de la WSN.

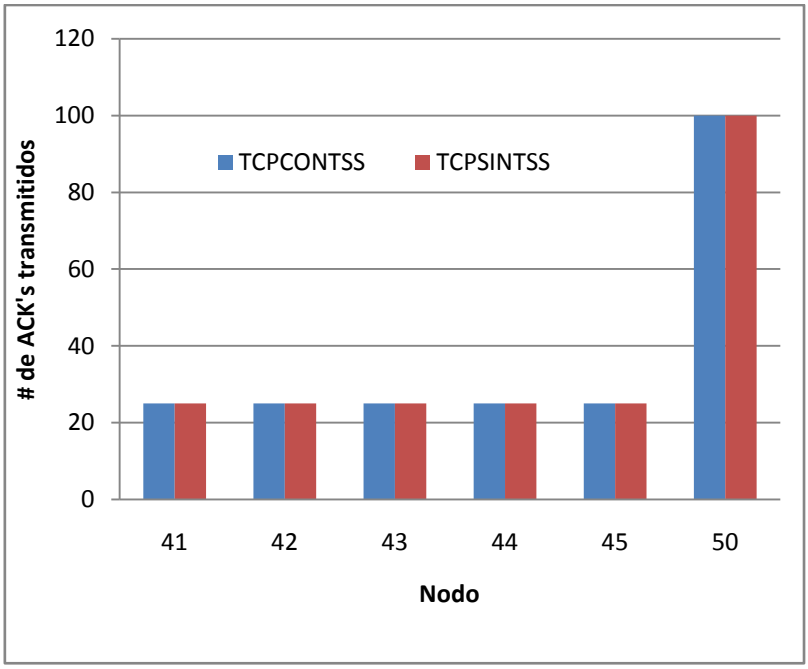

Figura 3.20 Transmisión de ACK's en ausencia de errores

Cuando las condiciones de errores van en aumento, la pérdida de los reconocimientos TCP se convierte en un factor importante a considerar debido a que esta también es causante de la transmisión de paquetes TCP adicionales. Para hacer frente a esto, TSS activa un mecanismo que permite la recuperación de reconocimientos TCP de una forma muy similar a la que se realiza con los segmentos de datos TCP.

Adicionalmente TSS descarta aquellos reconocimientos TCP que ya han sido recibidos por el nodo sucesor lo que ayuda a reducir y a distribuir casi equitativamente el número de reconocimientos TCP que deben ser transmitidos por cada nodo, tal y como se muestra en las figuras 3.21, 3.22 y 3.23
Además, en comparación con TCP sin TSS, se logra una reducción de hasta casi un 50% en el nodo servidor (nodo 50), en situaciones con altas probabilidades de error (configuración 3,75/7,5/15), como se puede apreciar en la figura 3.23.

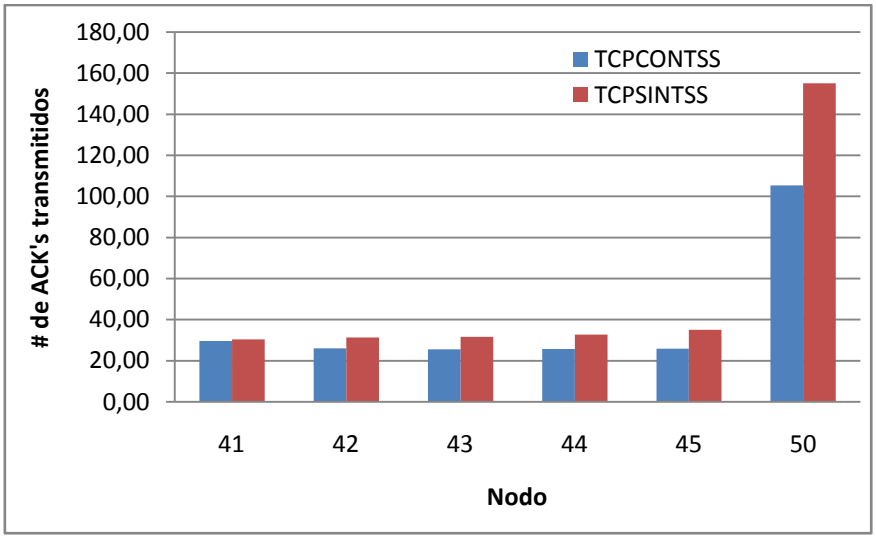

Figura 3.21 Transmisión de ACK's con 1,25/2,5/5 % de error

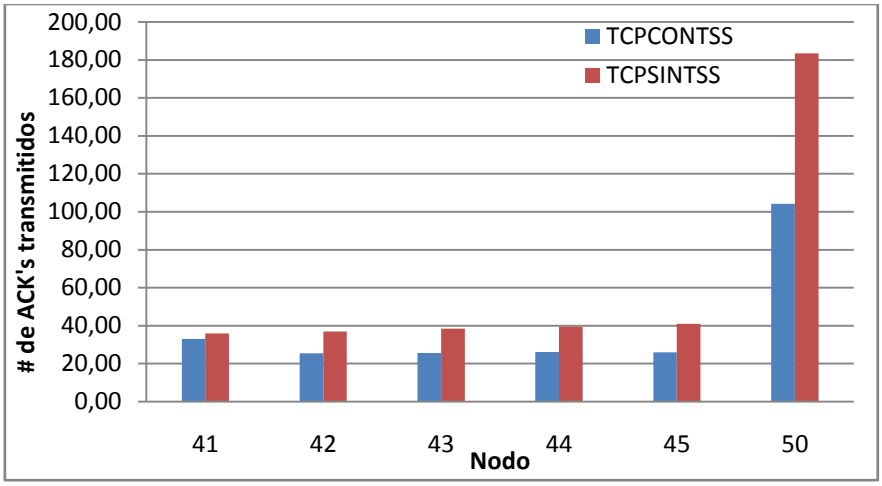

Figura 3.22 Transmisión de ACK's con 2,5/5/10 % de error

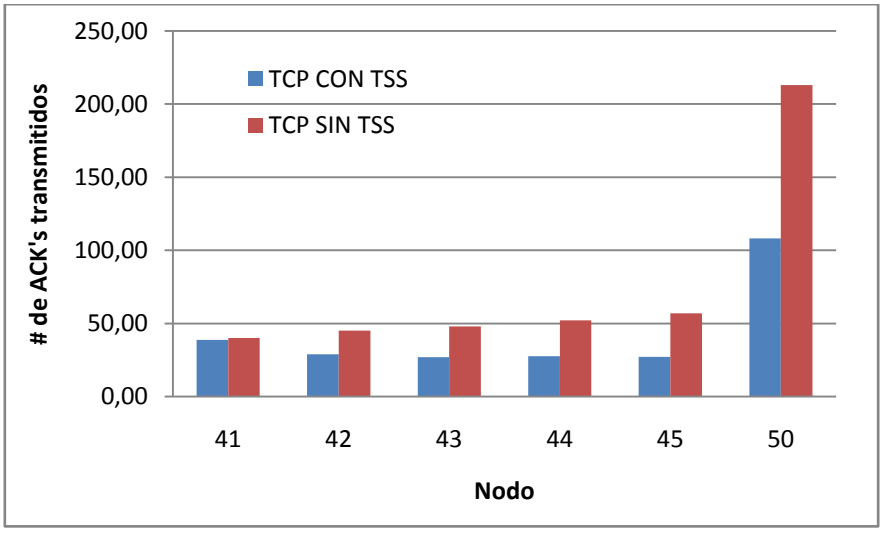

Figura 3.23 Transmisión de ACK's con 3.75/7.5/15 % de error

En la figura, 3.24 se resume el comportamiento de la transmisión de ACK's utilizando TSS con las diferentes configuraciones de probabilidades de error. En la misma se puede observar que el nodo más cercano a la fuente (nodo 41) transmite una mayor cantidad de reconocimientos TCP (ACK) con respecto a los demás nodos intermedios. Esto se debe al mecanismo de confirmación explicita de capa de enlace (explicado en el apartado 3.2), en el que el nodo cercano a la fuente necesita recibir un mensaje de confirmación de nivel de enlace de la fuente, en caso que esto no suceda (debido a las probabilidades de error) el nodo intermedio retransmite el ACK.

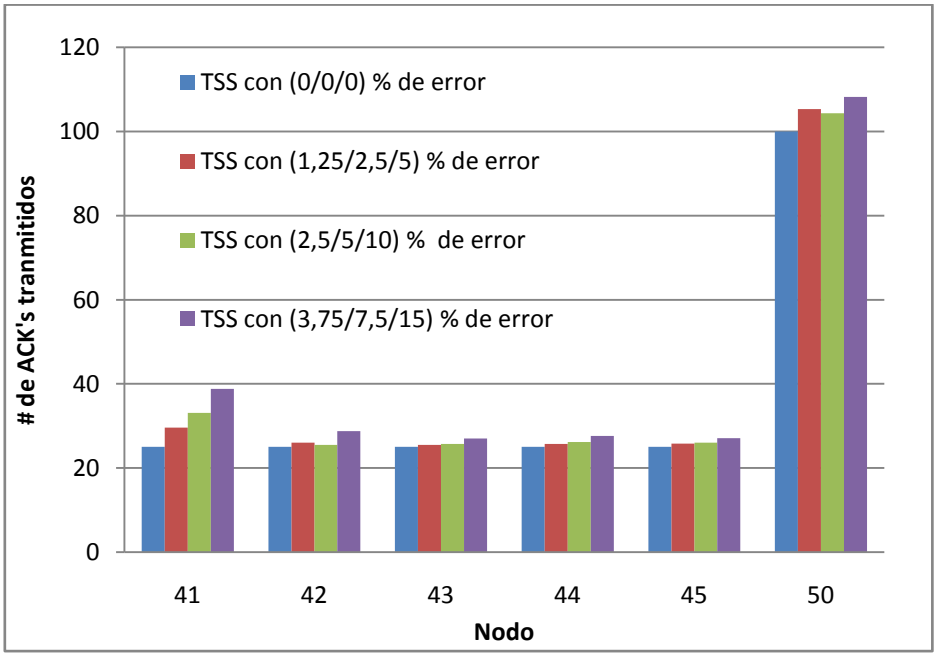

Figura 3.24 Transmisión de ACK's con TSS, con las diferentes configuraciones de probabilidades de error distribuida en todos los nodos.

3.4.2.1 Eficiencia de TSS versus DTC en la transmisión de reconocimientos TCP

En cuanto a las transmisiones de reconocimientos TCP, hay que señalar que TSS siempre intentará recuperar los reconocimientos TCP (ACK) a diferencia de DTC que no implementa este mecanismo, por lo tanto la cantidad de ACK TCP transmitidos debería ser mayor en TSS.

Sin embargo, esta premisa solo se cumple en el nodo destino, ya que en los demás nodos (intermedios y origen), la cantidad de reconocimientos TCP (ACK) es mucho menor en TSS que en DTC, y se debe al filtrado que se ha implementado en el nodo más cercano al destino, que ya se ha comentado anteriormente **(ver figuras 3.25, 3.26 y 3.27).**

Esta disminución puede representar entre un 57% para condiciones sin errores y un 52% para condiciones con altas probabilidades de errores (configuración 3.75/7,5/15), tomando en cuenta todo el conjunto de nodos.

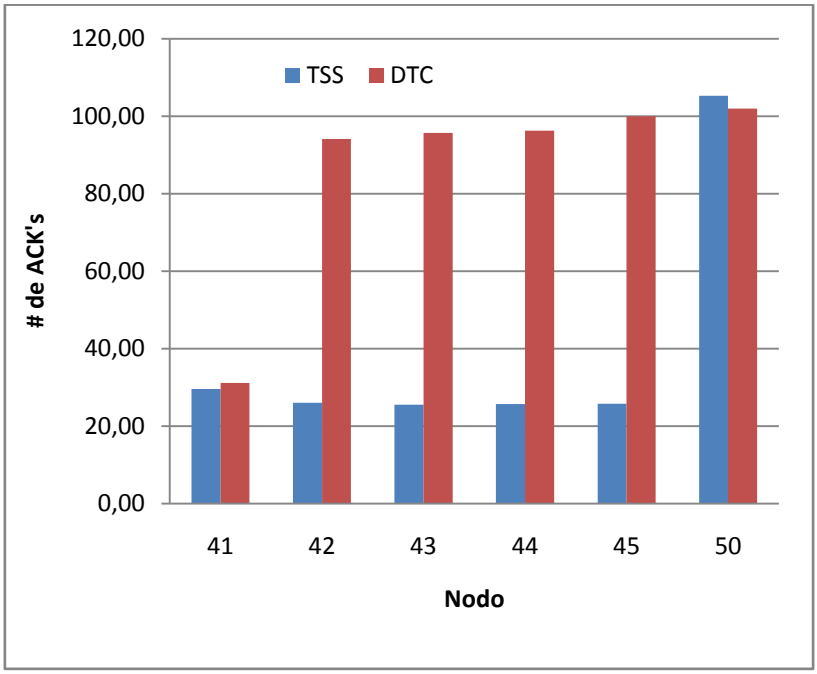

Figura 3.25 Transmisión de ACK's con 1,25/2,5/5 % de error

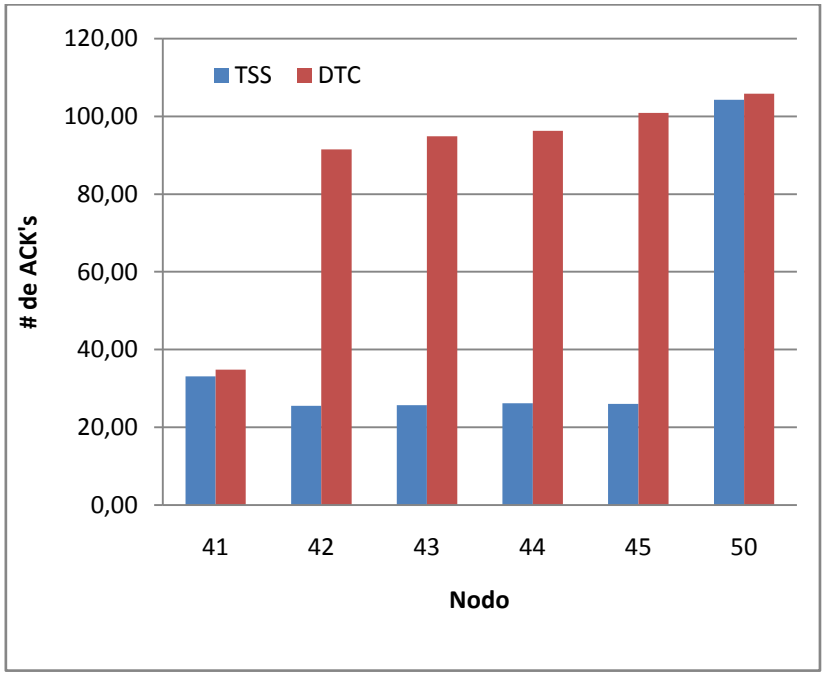

Figura 3.26 Transmisión de ACK's con 2,5/5/10 % de error

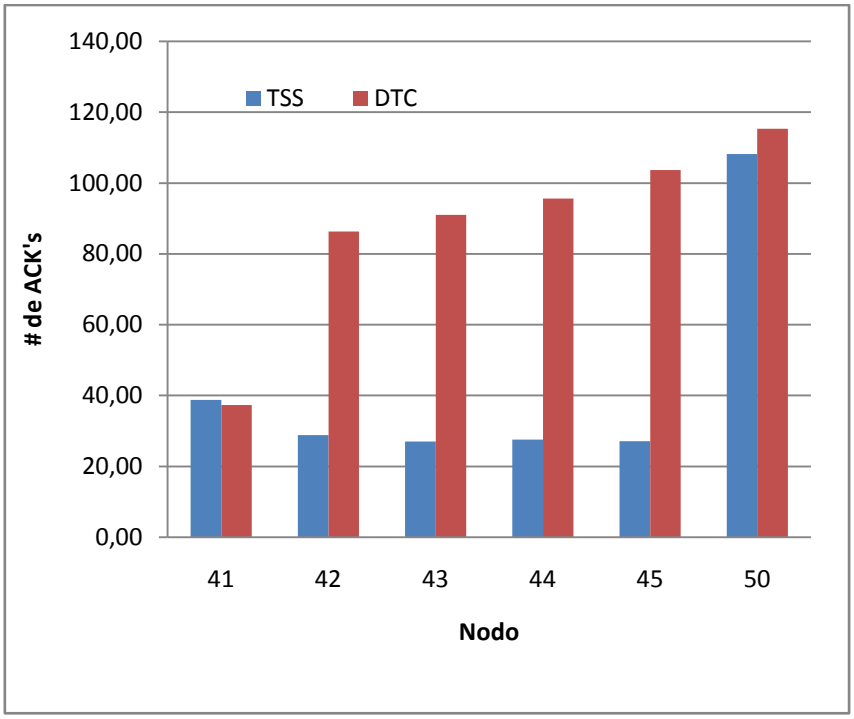

Figura 3.27 Transmisión de ACK's con 3.75/7.5/15 % de error

3.4.3 Ocupación de Memoria

La memoria es otro de los recursos limitados con que cuentan los nodos sensores. Es por eso, que debe ser un punto importante a considerar a la hora de programar estos dispositivos. El hardware que se ha utilizado en este proyecto es un TelosB TPR2420 el cual tiene 48 Kbytes de memoria Flash.

La tabla 3.4 se presenta un resumen del espacio en (Kbytes) que ocupa la pila uIP original del sistema operativo Contiki 2.2.2 y las diferentes implementaciones de uIP utilizadas para este proyecto.

Implementación	Ocupación de Memoria						
	Tamaño en kbytes						
	nodo 40 cliente	nodo 41	nodo 42	nodo 43	nodo 44	nodo 45	nodo 50 servidor
uIP original proporcionada por Contiki 2.2.2	7,2	7,2	7,2	7,2	7,2	7,2	7,2
uIP + Go Back N	8,3	7,9	7,9	7,9	7,9	7,9	7,9
uIP + Go Back N + TSS	8,8	8,6	8,6	8,6	8,6	8,6	8,6

Tabla 3.4 Ocupación de Memoria de las diferentes implementaciones de la pila uIP

De la información presentada en la Tabla 3.4 se puede notar que la implementación de nuevos mecanismos como el Go-Back-N y los mecanismos utilizados por TSS aumenta el tamaño original de la pila uIP original en memoria (alrededor de un 20%) a pesar de esto, no se considera que esto limite la capacidad de memoria del hardware, ya que este incremento solo representa un 3% aproximadamente de la memoria que posee el TelosB.. Además, debido a que el mecanismo de Go-Back-N solamente se implementa en el cliente, el incremento de ocupación de memoria siempre es mayor en este nodo en comparación con los demás (nodos intermedios y servidor).

En el siguiente capítulo presentamos las conclusiones finales de este proyecto y las líneas futuras de investigación que surgen de los hallazgos encontrados en la implementación de TSS en este proyecto.

4. CONCLUSIONES Y LINEAS FUTURAS

En el presente proyecto se han analizado las limitaciones que poseen los protocolos de transporte fiable como TCP en las WSN, y realizado un estado del arte de los requerimientos que deberían cumplir los protocolos de transporte para trabajar eficientemente en las WSN.

Finalmente, en este proyecto se han validado los mecanismos que ofrece el protocolo TSS para resolver las limitaciones que posee TCP en las redes sensores inalámbricas, de acuerdo al trabajo teórico realizado por Adam Dunkels y el grupo de desarrollo del SICS.

En este sentido, se ha realizado una implementación real de este protocolo utilizando la plataforma TelosB y se ha puesto a prueba su funcionamiento en situaciones con diferentes probabilidades de errores en diferentes capas (capa de enlace de datos y capa de transporte).

De los resultados obtenidos se ha demostrado que TSS ofrece las siguientes ventajas en comparación con TCP tradicional:

- Reduce la cantidad de paquetes transmitidos por todos los nodos de la red, entre un 45 y 70% dependiendo de las probabilidades de errores.
- Disminuye las retransmisiones extremo a extremo. Esta disminución comprende entre un 75% y 80 % dependiendo de las probabilidades de errores.
- Distribuye casi equitativamente la cantidad de paquetes transmitidos por cada nodo sensor. Debido a esto TSS asegura la conservación de la energía de cada uno de los nodos sensores cumpliendo con uno de los requisitos de los protocolos de transporte para WSN.

El mecanismo de “backpressure congestion” disminuye la posible congestión en los nodos, manteniendo almacenado los paquetes subsecuentes hasta que los paquetes transmitidos anteriormente hayan sido transmitidos por el nodo sucesor.

Por otra parte, se ha realizado una comparación entre TSS y DTC demostrando con ella que la cantidad de paquetes transmitidos en situaciones con mayores probabilidades de error, se reduce en TSS en un 13% con respecto a DTC, lo que se traduce en un menor consumo de energía en este aspecto.

Adicionalmente, el protocolo TSS no contempla la transmisión de reconocimientos de enlace de datos, sin embargo éstos reconocimientos, han sido utilizados en este proyecto para implementar el mecanismo de confirmación explicita entre la fuente y el nodo intermedio más cercano a ésta. A pesar de ello se ha demostrado que éste no representa un mayor incremento (alrededor de 3%) en la cantidad de transmisiones totales (paquetes y reconocimientos TCP).

También fue necesaria la modificación de ciertas funcionalidades de TSS para lograr la implementación en el hardware, y que al final aportaron beneficios al rendimiento del protocolo, estos fueron:

- El mecanismo de filtrado en el nodo más cercano al destino, permite una reducción de la cantidad de reconocimientos TCP (ACK) transmitidos por los nodos intermedios (alrededor de un 25% entre todos los nodos intermedios), y además disminuye la probabilidad de colisión en el medio inalámbrico y la congestión en los nodos de la red.

- Igualmente el mecanismo de filtrado en el nodo más cercano a la fuente, evita que se introduzcan paquetes redundantes a la red, aproximadamente entre un 13% y un 23 %, dependiendo de las probabilidades de errores.

- El mecanismo de confirmación explicita de capa de enlace entre la fuente y el nodo más cercano permite que se recuperen los reconocimientos TCP (ACK) en caso de pérdida, lo que evita que hayan retransmisiones por parte de la fuente.

- De forma similar, el uso del reconocimiento de TCP (ACK) que envía el destino como mecanismo de detección y recuperación de paquetes por el nodo intermedio cercano al destino, evita la pérdida de paquetes y por lo tanto retransmisiones extremo a extremo.

- Adicionalmente se ha demostrado que la incorporación de los mecanismos requeridos para el funcionamiento de TSS, así como los aportes particulares realizados a la pila uIP original incrementan ligeramente la ocupación de memoria en un 3% en relación con la memoria total de 48kb que posee la plataforma TelosB. Por lo tanto no constituye un mayor problema para su implementación en este hardware.

Con todo esto, se puede considerar a TSS como una evolución de DTC ya que adiciona un mecanismo de control de congestión, recuperación de reconocimientos de TCP y mecanismo cross-layer de confirmación implícita, convirtiéndose en una solución prácticamente completa para el transporte fiable de los datos en las WSN.

Sin embargo, en la implementación real del protocolo TSS se detectaron algunos aspectos que deben considerarse para mejorar su funcionamiento, y que presentamos como líneas futuras de investigación:

1 La transmisión ordenada de paquetes que garantiza TSS, conlleva a que el nodo intermedio cercano a la fuente descarte, en caso de pérdida, los subsecuentes paquetes de la ventana, lo que trae consigo la retransmisión de la ventana por parte de la fuente. Esta situación, lleva un costo asociado de consumo de energía y un desaprovechamiento de los paquetes que se transmitieron.

   Una posible alternativa a este inconveniente podría ser la incorporación del mecanismo cross-layer de confirmación implícita en la fuente con el cual pueda detectar la pérdida de paquete de su nodo sucesor y retransmitirlo antes de continuar con el siguiente paquete de la ventana.

2 El cálculo del valor de RTT fijo calculado al momento de establecer la conexión TCP no es el más conveniente para las WSN debido a las condiciones cambiantes de su entorno. Sin embargo el cálculo adaptativo del RTT propuesto por TSS aunque puede ser adecuado, se debe considerar el coste de proceso computacional en los nodos que implicaría implementarlo, ya que se necesitaría mantener un temporizador por cada paquete transmitido y detectar su correspondiente reconocimiento TCP (ACK) para detener el temporizador.

   Una medida para resolver esta situación podría ser el cálculo de RTT por ventana, iniciando el temporizador con la transmisión del primer paquete y cancelándolo cuando se reciba el reconocimiento TCP (ACK) del último paquete de la ventana. Sin embargo, tendría que considerarse el efecto que tiene el tamaño de la ventana en el valor del RTT, ya que entre más grande sea la ventana, el valor del RTT será mucho mayor.

3 A pesar de que el protocolo MAC (NULLMAC) utilizado por TSS en este proyecto permite la implementación del mecanismo cross-layer de confirmación implícita, éste lleva asociado un gran consumo de energía ya que mantiene siempre

encendida la interfaz radio. Por esta razón, se hace necesario el estudio e implementación de un protocolo MAC que sea compatible con el funcionamiento de TSS y que tenga en cuenta el ahorro de energía.

Una posible solución inmediata sería aprovechar uno de los protocolos MAC existente para WSN, como X-MAC, y extenderle la cantidad de tiempo que debe mantener la radio encendida a un valor adecuado (temporizador de capa de enlace) para que se pueda detectar la transmisión del paquete del siguiente nodo hacia su nodo sucesor.

4 TSS fue pensado para trabajar en topologías de red donde existe una única ruta entre el origen y el destino y viceversa. Sin embargo, en las WSN esta situación no es muy común debido a la topología dinámica de las WSN en la que, por ejemplo, el agotamiento de energía en un nodo puede generar cambios dinámicos en el enrutamiento del tráfico. Además para mejorar la eficiencia de la red, los protocolos de enrutamiento para WSN deben buscar siempre la mejor ruta hacia el destino basada en los recursos disponibles de los nodos, lo que haría a TSS incompatible con este funcionamiento. Por lo tanto convendría realizar un estudio y análisis para adaptar TSS a topologías con comportamiento dinámico y con múltiples rutas disponibles.

5 Estudiar y aprovechar los mecanismos que ofrece tanto DTC (SACK) como TSS (“backpressure congestion control”) con el objetivo de obtener una implementación híbrida y adaptativa a las condiciones del entorno o al tipo de fiabilidad que requiera la aplicación.

Finalmente, tanto TSS como DTC han demostrado que el protocolo TCP puede lograr un funcionamiento eficiente del protocolo TCP en redes de sensores inalámbricas, sin embargo para ello se hace necesaria, la compartición de información con las demás capas (por ejemplo MAC). Esto demuestra una evidente tendencia hacia el diseño y optimización de mecanismos cross-layer para tomar las mejores decisiones que proporcionen fiabilidad, control de congestión y a la vez conserven al máximo la energía de las WSN.

Bibliografía

1. **Chalasani, Sravanthi and Conrad, James M.** *A Survey of Energy Harvesting Sources for Embedded Systems.* 2008.

2. **Dunkels, A.** Full TCP/IP for 8-bit Architectures. *ACM MobiSys.* Mayo 2003. pp. 85-98.

3. **Dunkels, Adam.** [Online] Octubre 14, 2002. [Cited: febrero 5, 2009.] http://dunkels.com/adam/uip/.

4. **Akyildiz, Ian F. and Wang, Xudong.** A Survey on Wireless Mesh Networks. *IEEE Radio Communications.* Septiembre 2005.

5. **Wan, C. Y., Eisenman, S. B. and Campbell, A. T.** CODA: Congestion Detection and Avoidance in Sensor Networks. *Conference On Embedded Networked Sensor Systems Proceedings of the 1st international conference on Embedded networked sensor systems.* 2003. pp. 266 - 279 . 1-58113-707-9 .

6. **Wang, C., Sohraby, K. and Li, B.** SenTCP: A Hop-by-Hop Congestion Control Protocol for Wireless Sensor Networks. *Proceedings of the 2nd annual international workshop on Wireless internet.* 2006. 1-59593-510-X .

7. **Sankarasubramaniam, Y., Akan, O. B. and Akyildiz, I. F.** ESRT: Event-to-Sink Reliable Transport in Wireless Sensor Networks". *MobiHoc 2003 4th Symposium on Mobile Ad Hoc Networking & Computing.* Junio 2003. pp. 177-188.

8. **Wan, C. Y., Campbell, A. T. and Krishnamurthy, L.** PSFQ: A Reliable Transport Protocol for Wireless Sensor Networks. *In Proc. of First ACM International Workshop on Wireless Sensor Networks.* Septiembre 2002. pp. 1-11.

9. **Stann, F. and Heidemann, J.** RMST: Reliable Data Transport in Sensor Networks. *IEEE International Workshop on Sensor Net Protocols and Applications.* Mayo 11, 2003. pp. 1-11.

10. **Intanagonwiwat, C., Govindan, R. and Estrin, D.** Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Network. *Mobicon 00.* Agosto 2000. pp. 56-67.

11. **Seung-Jong, P, et al.** GARUDA: Achieving Effective Reliability for Downstream Communication in Wireless Sensor Networks. Febrero 2008.

12. **Dunkels, A., et al.** Distributed TCP Caching for Wireless Sensor Networks. *Annual Mediterranean Ad Hoc Networking Workshop.* Junio 2004.

13. **Braun, T., Voigt, T. and Dunkels, A.** TCP support for sensor networks. *Wireless on Demand Network Systems and Services, 2007. WONS '07.* [Enero]. 2007. pp. 162-169.

14. **Adame Vázquez, Antonio.** ESTUDIO DE LAS INICIATIVAS DEL INTERNET DEL FUTURO: Implementación y evaluación de DTC en redes de sensores. *Proyecto Final de Carrera.* 2009.

15. **Seah, Winston K.G., Euy, Zhi Ang and Tan, Hwee-Pink.** *Wireless Sensor Networks Powered by Ambient Energy Harvesting - Survey and Challenges.*

16. **Acosta Ponce, Maria Catalina.** Estudio del estándar IEEE 80.15.4 "ZIGBEE" para comunicaciones inalámbricas de área personal de bajo consumo de energía y comparación con el estándar IEEE 802.15.1 "BLUETOOTH". *Tesis Doctoral.* 2006.

17. **Dunkels, A., Alonso, J. and Voigt, T.** Making TCP/IP Viable for Wireless Sensor Networks. *First European Workshop on Wireless Sensor Networks (EWSN'04).* Berlín, Alemania : s.n., Enero 2004.

18. **Dunkels, A., et al.** Connecting Wireless Sensornets with TCP/IP Netowrks. *Second International Conference on Wired/Wireless Internet Communications (WWWIC2004).* Febrero 2004. pp. 143-152.

19. **Loubser, M.** Delay Tolerant Networking for Sensor Networks. *SICS Technical Report T2006:01.* Enero 2006. 1100-3154.

20. **Jones, J. and Atiquzzaman, M.** Transport Protocols for Wireless Sensor Networks: State-of-the-Art and Future Directions. *International Journal of Distributed Sensor Networks.* 2007. Vol. 3, 1, pp. 119-133. 1550-1477.

21. **Balakrishnan, H., et al.** Improving TCP/IP performance over wireless networks. *ACM Mobicom,.* Noviembre 1995. pp. 2-11.

22. **Braun, T., Voigt, T. and Dunkels, A.** Energy-Efficient TCP Operation in Wireless Sensor Networks. *Praxis der Informationsverarbeitung und Kommunikation (PIK).* 2005. 2.

23. **Zorzi, M. and Rao., R.** Is TCP energy efficient? *International Workshop on Mobile Multimedia Communications.* 1999.

24. **Karl, Holger and Willig, Andreas.** *Protocols and Architectures for Wireless Sensor Networks.* s.l. : Wiley, 2005. 0-470-09510-5.

25. **Sohraby, K., Minoli, D. and Znati, T.** *Wireless Sensor Networks: Technology, Protocols and Applications.* s.l. : Wiley Interscience. 978-0-471-74300-2.

ANEXO A: PiLAS IP PARA WSN

X: no posee la característica. N/E: no especificada por el fabricante o autor. N/A: no aplica

Descripción	nanoIP	uIP	LwIP	b6lowpan/ TinyOS 2.0	uIPv6/ SICSlowpan	NanoStack/ Sensinode
Licencia	BSD, GPL	GPL	GPL	BSD	BSD	BSD
Requerimientos en RAM	211 Bytes	2kb	2kb	3.5 Kbytes	1.8 KB	N/E
Requerimientos en ROM	3.9 Kbytes	~5 Kbytes	5 Kbytes	24 Kbytes	11 Kbytes	N/E
Sistema Operativo	TinyOS	Contiki, FreeRTOS	Contiki, FreeRTOS	TinyOS	Contiki,	FreeRTOS
Protocolos que implementa	nanoTCP, nanoUDP, nanoIP	IP, TCP, UDP, ICMP	IP,TCP,UDP,ICMP	ICMPv6 reducida, DHCPv6, UDP,TCP	ICMPv6, DHCPv6, UDP,TCP	ICMPv6, DHCPv6, UDP
Direccionamiento	MAC	IP	IP	IP	IP	IP
Versión de Protocolo IP	4	4	4	6	6	6
IP y TCP Checksum	X	√	√	X	X	X
Re ensamblado de Fragmento IP	√	√	√	X	X	X
Opciones IP	X	X	√	X	X	X
Múltiples Interfaces	N/E	X	√	N/E	X	N/E
UDP	√	√	√	√	√	√
Múltiples conexiones TCP	N/E	X	√	X	X	X
Opciones TCP	X	√	√	X	X	X
Estimación del RTT	√	√	√	X	X	X
Control de Flujo TCP	√	√	√	X	X	X
Ventana Deslizante TCP	√ Tamaño Fijo	X	√	X	X	X
Control de Congestión TCP	X	X	√	X	X	X

Datos TCP desordenados	√	X	√	X	X	X
Datos Urgente TCP	X	√	√	X	X	X
Retransmisión de datos	√	X	√	X	X	X
Fragmentación IP	√	√	X	X	√	√
Compresión de Encabezados	N/E	√	X	√	√	√
Descubrimiento de Vecino	N/A	N/A	N/A	X	√	√
Autoconfiguración sin estado	N/A	N/A	N/A	√	√	√
Autoconfiguración sin estado con extensiones de privacidad	N/A	N/A	N/A	√	√	√
Selección de dirección IP por defecto	N/A	N/A	N/A	√	√	√
Autoconfiguración con estado	N/A	N/A	N/A	√	√	√
Descubrimiento de “listeners” de multicast	N/A	N/A	N/A	√	X	√
Seguridad	N/E	X	X	√	X (futuras versiones)	√ (AES-128)

ANEXO B: SISTEMAS OPERATIVOS PARA WSN

X: no posee la característica. N/E: no especificada por el fabricante o autor.

	TinyOS	Contiki	Mantis OS	FreeRTOS
Licencia	BSD	BSD	BSD	GPL
Sitio Web y del código	http://www.tinyos.net	http://www.sics.se/contiki	http://mantis.cs.colorado.edu/index.php/	http://www.freertos.org/
Req. RAM	512 Bytes	2 Kbytes	Menos de 500 Bytes	N/E
Req. ROM	8 Kbytes	40 Kbytes	14 KBytes	3.3 Kbytes
Características de Diseño	Modularidad Abstracción de componentes Concurrencia Planeamiento (Scheduling)	Alta portabilidad Multitarea Carga dinámica de servicios	Multitarea Programador de tareas con prioridad	"Kernel" de tiempo real para proveer respuesta oportuna (tiempo real) a los eventos del mundo real.
Tipo de Kernel (funcionamiento)	Manejado por eventos	Manejado por eventos y protothreads	Manejado por multithreads	Manejado por "multithreads" con "scheduler" con soporte de prioridades entre Co-rutinas y tareas.
Pila implementada	6lowpan, Zigbee	uIP, uIPv6, Rime*	implementa una capa Comm	NanoStack, uIP, lwIP
Ventajas	Consume 30% menos memoria que MantisOS Soporte de librería multithreads (TinyOS 2.1)	Soporte de librería multithreads Flexibilidad:	Ejecución programada de multithreads incluyendo sincronización.	Poderosa funcionalidad de registro (tracing) que permite obtener datos del comportamiento de la aplicación.
Lenguaje de Programación	NesC	C, C++	C	C
Herramientas de Simulación	TOSSIM	COOJA	X	X
Hardware implementado	MICAz, TelosB	Sky/TelosB, ATMEL y AVR,	MICA2,	ARM7, ARM CORTEX M3,

ANEXO C: ALGORITMO DE FUNCIONAMIENTO DE TSS

```
switch(event){
case ack_timeout: // -1-
retransmit_ack(acknowledged);
start(ack_timer, acknowledged,
γ^attempts++ * ack_forwarding_time);
break;
case retransmission_timeout: // -2-
sequence_no =
sequence_number_of_packet_to_be_retransmitte
d;
if ((sequence_no + length > confirmed){
retransmit_data(sequence_no);
if (number_of_retransmissions > limit)
delete(cache);}
break;

default: // -3-
if (packet_has_bit_error || ttl_expired ||
(own_address != next_address) &&
(own_address != previous_address))
delete(packet);
else if (next_address == own_address) { // -3a

switch(type_of_packet){
case ack:
acknowledged=max(ack_no- 1, acknowledged);
if ((acknowledged > confirmed) &&
((byte[acknowledged+1]∩buffered_packet)
≠ Ø)){
forward(buffered_packet);
move(buffered_packet, cache);
transmitted=
sequence_number_of_buffered_packet +
length-1;
start_timer(retransmission_timer,
sequence_no, β * rtt);
confirmed = acknowledged;}
if (ongoing_rtt_measurement &&
(ack_no > rtt_sequence_no)){
rtt= (1–α) * rtt + α *
(current_time–start_of_measurement);
ongoing_rtt_measurement = FALSE;}
if (ack_no <= ack_forwarded)
delete(packet);
else {
forward(packet);
start(ack_timer, ackno,
γ * ack_forwarding_time);
attempts = 1;}
break;

case data:
if (sequence_no > transmitted + 1)
delete(packet);
else if ((sequence_no > confirmed + 1) &&
(buffer_is_empty ||
(sequence_no < seqno_of_buffer)))
move(packet, buffer);
else if (sequence_no + length – 1
<= acknowledged){
retransmit_ack(acknowledged);
start_timer(ack_timer,acknowledged,
γ*ack_forwarding_time);
attempts = 1;
delete(packet);}
else if ((transmitted == confirmed) &&
(byte[confirmed + 1] ∩ packet) ≠ Ø)){
if (! ongoing_rtt_measurement){
ongoing_rtt_measurement = TRUE;
rtt_sequence_no = sequence_no;
start_of_measurement = current_time;}
forward(packet);
transmitted = sequence_no + length - 1;
move(packet, cache);
start_timer(retransmission_timer,
sequence_no, β * rtt);}
else
delete(packet);}
else if (own_address == previous_address){ //-3b
switch(type_of_packet){
case ack:
ack_forwarding_time = (1 – α) *
ack_forwarding_time + α *
(current_time – transmission_time(ack_no));
cancel(ack_timer, ack_no);
ack_forwarded = ackno;
break;
case data:
if (sequence_no + length – 1 > confirmed){
cancel(retransmission_timer,
sequence_no);
delete(cache);
confirmed = sequence_no + length - 1;
if (byte[confirmed + 1]∩buffered_packet≠Ø){
forward_delayed(buffer);
transmitted = sequence_no_of_buffered_packet
+ length - 1;
move(buffer, cache);
start(retransmission_timer,
sequence_no_of_buffer, β * rtt);}}}
delete(packet);}}
```

ANEXO D: CÓDIGO FUENTE DE TSS

Archivo uip-over-mesh.c

```
/*
 * Copyright (c) 2007, Swedish Institute of Computer Science.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of the Institute nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE INSTITUTE AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE INSTITUTE OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * This file is part of the Contiki operating system.
 *
 * $Id: uip-over-mesh.c,v 1.10 2008/11/09 12:20:56 adamdunkels Exp $
 */

/**
 * \file
 *         Code for tunnelling uIP packets over the Rime mesh routing module
 * \author
 *         Adam Dunkels <adam@sics.se>
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "contiki.h"
#include "net/hc.h"
#include "net/uip-fw.h"
#include "net/uip-over-mesh.h"
#include "net/rime/route-discovery.h"
#include "net/rime/route.h"
#include "net/rime/trickle.h"
#include "dev/leds.h"

#define ROUTE_TIMEOUT CLOCK_SECOND * 4

static struct queuebuf *queued_packet;
static rimeaddr_t queued_receiver;

 /* Connection for route discovery: */
static struct route_discovery_conn route_discovery;

/* Connection for sending data packets to the next hop node: */
static struct unicast_conn dataconn;

/* Connection for sending gateway announcement message to the entire
   network: */
static struct trickle_conn gateway_announce_conn;

/*Conexion a nivel de enlace para los reconocimientos TCP*/
static struct unicast_conn ackconn;
```

```
/*Definiciones TSS*/
#define BUF ((struct uip_tcpip_hdr *)&uip_buf[UIP_LLH_LEN])
#define NO_RET               0
#define RET           1
#define RET_ACK              2
#define NO_RET_ACK    3
#define NO_RETBUF     4

/*FIN Definiciones TSS*/

#define DEBUG 0
#if DEBUG
#include <stdio.h>
#define PRINTF(...) PRINTF(__VA_ARGS__)
#else
#define PRINTF(...)
#endif

#define BUF ((struct uip_tcpip_hdr *)&uip_buf[UIP_LLH_LEN])
#define LL_TIMER  CLOCK_SECOND/3
#define ACK_TIMER CLOCK_SECOND/3
#define DELAY_TIMER CLOCK_SECOND/500
static struct uip_fw_netif *gw_netif;
static rimeaddr_t gateway;
static uip_ipaddr_t netaddr, netmask;
static rimeaddr_t nodo_sig_der;
static rimeaddr_t nodo_sig_izq;
static rimeaddr_t nodo_oculto_der;
static rimeaddr_t nodo_oculto_izq;
static rimeaddr_t nodo_dest;

/* VARIABLES MECANISMO TSS*/
static u8_t buffer_tss[110];
static u8_t buffer_ack[110];
static u8_t buffer_queue[110]; /*buffer para paquetes entrantes en espera*/


static unsigned int transmitted=0;
unsigned int confirmed=0;
static unsigned int acknowledged=0;
static unsigned int ack_forwarded=0;
static unsigned int rtt_seqno;

int ret_ll,ret_ack;
static int ret_rto = 0;
u16_t uip_len_buf;
static struct etimer ll_timer, rto_timer, ack_timer;
static struct timestamp temps;
static unsigned short tiempo, rtt1, rtt2;
static unsigned long rtt;
static int attempts=0;
static int rtxw=0;
static int w=0;
static int i;
static int n=0;
static int k=0;
static int tbq=0;
static int out_rtt_measure=0;  //0 falso, 1 verdadero
process_event_t ll_event;
process_event_t rto_event;
process_event_t ack_event;
process_event_t forward_event;
int paq_filter=0;
int n_ll = 0;
int n_ack = 0;
int n_paq = 0;
int rtx_data = 0; //numero de TCP DATA retransmitidos
int rtx_ack = 0; //numero de TCP ACK retransmitidos
int no_data = 0; //numero de TCP DATA no recibida
int no_ack = 0;  //numero de TCP ACK no recibida
int no_l2 = 0;
int mayor_txed = 0;
int borro_paq = 0;
```

```
PROCESS(tss_process, "Proceso TSS");

PROCESS_THREAD(tss_process, ev, data)
{
  static rimeaddr_t nodo_dest;

  PROCESS_BEGIN();
  n=1;

if ((rimeaddr_cmp(&rimeaddr_node_addr,&nodo_40)==0)&&
(rimeaddr_cmp(&rimeaddr_node_addr,&nodo_50)==0)) /*NO ERES NI CLIENTE NI SERVIDOR (NI 40
NI 50)*/
  {
/* MECANISMO TSS */
       if (((BUF->flags & 0x10)==16) && (uip_buf[14] == 50) && (uip_buf[15] == 0))
/*COMPROBAR SI SE TRATA DE UN MENSAJE ACK DEL SERVIDOR*/
       {
         leds_on(LEDS_BLUE);
         if ( (calcular_seq(&uip_buf[28])-1) > acknowledged )
         {
         ret_rto = 0;
          printf("RET_RTO = %d\n",ret_rto);
          rto_event = process_alloc_event();
          rimebuf_copyto(buffer_ack);
          acknowledged = (calcular_seq(&buffer_ack[28])-1);
         }
         if ( (acknowledged > confirmed) && (cache_vacio(buffer_queue)==1) && (
((acknowledged+1)>=calcular_seq(&buffer_queue[24])) &&
((acknowledged+1)<=(calcular_seq(&buffer_queue[24]) + 70 -1))) )
         {
           tbq=1;
           rimebuf_copyfrom(&buffer_queue[UIP_LLH_LEN], uip_len_buf);
           rimebuf_copyto(buffer_tss);
           memset(buffer_queue,0,110);
           transmitted= (calcular_seq(&buffer_tss[24]) + 70 -1);
           nodo_dest = det_routing(NO_RET);
           if (tbq==1)
           {
            for (i=0;i<2000;i++)
           {
            if ((i%1000)==0)
            {
             PRINTF("%d...",i);
            }
           }
            unicast_send(&dataconn,&nodo_dest);
            n_paq++;
            temps = calcular_tiempo();
           tbq=0;
          }
          else
          {
          unicast_send(&dataconn,&nodo_dest);
          n_paq++;
          temps = calcular_tiempo();
          }
          ret_ll = 1;
          ret_rto = 1;
          while ((ret_ll==1) || (ret_rto==1))
          {
           if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_45)!=0)
           {
            goto timertcp1;
           }
           etimer_set(&ll_timer,LL_TIMER);
           PROCESS_WAIT_EVENT_UNTIL((etimer_expired(&ll_timer)) || (ev==ll_event));
          if (ret_ll==1)
          {
          leds_on(LEDS_RED);
          timertcp1:
          etimer_set(&rto_timer, CLOCK_SECOND * rtt/4096);
           PROCESS_WAIT_EVENT_UNTIL((etimer_expired(&rto_timer)) || (ev == rto_event));
          if (ret_rto == 1)
          {
           enviar_paq_buf(0);
          }
           else
```

```
                {
                          ret_ll = 0;
                         confirmed= (calcular_seq(&buffer_tss[24]) + 70 -1);
                             if ( (cache_vacio(buffer_queue)==0) && ( (confirmed+1) >=
(calcular_seq (&buffer_queue[24]) ) ) && ( (confirmed+1) <=
(abs(calcular_seq(&buffer_queue[24]) + 70 -1)) ) )
                        {
                         rimebuf_copyfrom(&buffer_queue[UIP_LLH_LEN], uip_len_buf);
                         rimebuf_copyto(buffer_tss);
                         memset(buffer_queue,0,110);
                         tbq=1;
                         goto tx_buffer_queue1;
                        }/*fin del if de transmitir paquete almacenado*/
                        break;
              }
            }
            else
            {
                        ret_rto = 0;
              confirmed= (calcular_seq(&buffer_tss[24]) + 70 -1);
              if ( (cache_vacio(buffer_queue)==0) && ( (confirmed+1) >= (calcular_seq
(&buffer_queue[24]) ) ) && ( (confirmed+1) <= (abs(calcular_seq(&buffer_queue[24]) + 70
-1)) ) )
                        {
                         rimebuf_copyfrom(&buffer_queue[UIP_LLH_LEN], uip_len_buf);
                         rimebuf_copyto(buffer_tss);
                         memset(buffer_queue,0,110);
                         tbq=1;
                         goto tx_buffer_queue1;
                        }/*fin del if de transmitir paquete almacenado*/
               break;
             }
         }/*fin del while*/
               memset(buffer_tss,0,110);
               printf("\n++++++++++HEMOS BORRADO EL PAQUETE %d EN
MEMORIA+++++\n",calcular_seq(&uip_buf[24]));
               confirmed=acknowledged;
               }
               if ( calcular_seq(&uip_buf[28]) <= ack_forwarded )
               {
                goto drop;
               }
               else
               {
                 if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_45)!=0)  //eres nodo 45
                  {
                   if (acknowledged%(MODULO*70)==0)
                   {
                    goto tx_last_ack;
                   }
                   else
                   {
                    goto drop;
                   }
                  }
tx_last_ack:
                 rimebuf_copyfrom(&uip_buf[UIP_LLH_LEN], uip_len);
                 nodo_dest = det_routing(NO_RET);
                 unicast_send(&dataconn,&nodo_dest);
                 n_ack++;
                 ret_ack=1;
                 if ( acknowledged%(MODULO*70)==0 )
                 {
                 while (ret_ack==1)
                  {
                  etimer_set(&ack_timer,ACK_TIMER);
                  PROCESS_WAIT_EVENT_UNTIL((etimer_expired(&ack_timer)) ||
(ev==ack_event));
                  if (ret_ack==1)
                   {
                    enviar_paq_buf(1);
                          }
                   else
                   {
                    ack_forwarded = calcular_seq(&buffer_ack[28]);
                   }
                  }
```

```
                }
              }
            attempts=1;
       drop:
       leds_off(LEDS_BLUE);
       }
       else /*DATOS O EL SYN QUE ENVIA EL CLIENTE EN EL 1er PASO DEL ESTABLECIMIENTO DE
LA COMUNICACION*/
       {
       leds_on(LEDS_GREEN);


/*codigo tss para data:soy proxima direccion*/
        if (calcular_seq(&uip_buf[24]) > (transmitted+1))
        {
         mayor_txed++;
         goto drop2;
        }
        else if   ( (calcular_seq(&uip_buf[24]) > (confirmed +1)) && (
(cache_vacio(buffer_queue)==1) || (calcular_seq(&uip_buf[24]) <
calcular_seq(&buffer_queue[24])) ) )
        {
         rimebuf_copyto(buffer_queue);
         uip_len_buf = uip_len;
        }
        else if ((calcular_seq(&uip_buf[24]) + 70 -1) <= confirmed)
        {
         if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_41)!=0)
         {
          paq_filter++;
          if (acknowledged == confirmed)
          {
           rtxw++;
           if ( (calcular_seq(&uip_buf[24]) + 70 -1)%(MODULO*70)==0 )
           {
             enviar_paq_buf (1);
             ret_ack=1;
             while (ret_ack==1)
             {
              etimer_set(&ack_timer,ACK_TIMER);
              PROCESS_WAIT_EVENT_UNTIL((etimer_expired(&ack_timer)) ||
(ev==ack_event));
               if (ret_ack==1) {
                enviar_paq_buf(1);
               }else
               {
                ack_forwarded = calcular_seq(&buffer_ack[28]);
               }
             }
             rtxw=0;
           }
          }
          goto drop2;
         }
        }
        else if  ( ((calcular_seq(&uip_buf[24]) + 70 -1) <= acknowledged) )
        {
          /*  Note: 70 is the data lenght
            1. retransmitir_ack;
            2. iniciar temporizador Y*ack_forwarding_time;
            3. attempts=1;
            4. borro el paquete; */
          /*1*/
        if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_45)!=0)
         {
          enviar_paq_buf (1); //asumo que el ack acknowledged lo guardo en buffer_ack
          ret_ack=1;
          if ( (calcular_seq(&buffer_ack[28]) -1)%(MODULO*70)==0 )
          {
            while (ret_ack==1)
            {
              etimer_set(&ack_timer,ACK_TIMER);
              PROCESS_WAIT_EVENT_UNTIL((etimer_expired(&ack_timer)) ||
(ev==ack_event));
               if (ret_ack==1) {
                enviar_paq_buf(1);
               }else
```

```
                {
                 ack_forwarded = calcular_seq(&buffer_ack[28]);
                }
              }
            }
           }
              printf("****** BORRO PAQUETE PORQUE YA FUE ENVIADO******\n");
              /*3*/  attempts=1;
              goto drop2;
         }
          else if   ((transmitted == confirmed) &&
(((confirmed+1)>=(calcular_seq(&uip_buf[24]))) &&
((confirmed+1)<=(calcular_seq(&uip_buf[24]) + uip_len -1))))
          {
                rimebuf_copyto(buffer_tss);
               tbq=0;
tx_buffer_queue:
               transmitted= (calcular_seq(&buffer_tss[24])  + 70 -1);
               if (tbq==1)
               {
                 nodo_dest = det_routing(NO_RETBUF);
                }
                else
                {
                  nodo_dest = det_routing(NO_RET);
                }
                if (tbq==1)
                {
                  for (i=0;i<2000;i++)
                  {
                  if ((i%1000)==0)
                  {
                   PRINTF("%d...",i);
                  }
                  }
                  unicast_send(&dataconn,&nodo_dest);
                  n_paq++;
                  temps = calcular_tiempo();
                   memset(buffer_queue,0,110);
                   tbq=0;
                }
                 else
                 {
                 unicast_send(&dataconn,&nodo_dest);
                 n_paq++;
                 temps = calcular_tiempo();
                 }
               ret_ll = 1;
               ret_rto = 1;
               while ((ret_ll==1) || (ret_rto==1))
               {
                 if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_45)!=0)
                 {
                  goto timertcp;
                }
                etimer_set(&ll_timer,LL_TIMER); /*CLOCK_SECOND es un parametro que debe
cambiarse para optimizar el proceso*/
                PROCESS_WAIT_EVENT_UNTIL((etimer_expired(&ll_timer)) || (ev ==
ll_event));
               if (ret_ll==1)
               {
                leds_on(LEDS_RED);
                timertcp:
                 etimer_set(&rto_timer, CLOCK_SECOND * rtt/4096);
                PROCESS_WAIT_EVENT_UNTIL((etimer_expired(&rto_timer)) || (ev ==
rto_event));
                 if (ret_rto == 1)
                 {
                  enviar_paq_buf(0); /*La funcion de probabilidad de envio esta incluida
en enviar_paq_buf*/
                 }
                 else
                 {
                  ret_ll = 0;
                  if ( (calcular_seq(&buffer_tss[24]) + 70 -1) > confirmed )
                 {
                  confirmed= (calcular_seq(&buffer_tss[24]) + 70 -1);
```

```
                }
                 else
                 {
                }
                 if ( (cache_vacio(buffer_queue)==0) && ( (confirmed+1) >= (calcular_seq
(&buffer_queue[24]) ) ) && ( (confirmed+1) <= (abs(calcular_seq(&buffer_queue[24]) + 70
-1)) ) )
                 {
                   rimebuf_copyfrom(&buffer_queue[UIP_LLH_LEN], uip_len_buf);
                   rimebuf_copyto(buffer_tss);
                   tbq=1;
                   goto tx_buffer_queue;
                 }/*fin del if de transmitir paquete almacenado*/
                 break;
                }
               }
               else
               {
                 if ( (calcular_seq(&buffer_tss[24]) + 70 -1) > confirmed )
                 {
                   confirmed= (calcular_seq(&buffer_tss[24]) + 70 -1);
                 }
        ret_rto = 0;
                 if ( (cache_vacio(buffer_queue)==0) && ( (confirmed+1) >= (calcular_seq
(&buffer_queue[24]) ) ) && ( (confirmed+1) <= (abs(calcular_seq(&buffer_queue[24]) + 70
-1)) ) )
                 {
                   rimebuf_copyfrom(&buffer_queue[UIP_LLH_LEN], uip_len_buf);
/*-transmitimos el paquete almacenado en buffer_queue pero con un random delay------*/
                   rimebuf_copyto(buffer_tss);
                   tbq=1;
                   goto tx_buffer_queue;
                 }/*fin del if de transmitir paquete almacenado*/
                break;
               }
              }/*fin del while*/
              memset(buffer_tss,0,110);
          leds_off(LEDS_RED);
         }
         else
         {
          borro_paq++;
          printf ("borro el paquete\n");
         }
       drop2:
       leds_off(LEDS_GREEN);
       }
  }
  //la parte cliente y servidor se encuentra en la funcion recv_data

  n=0;
  printf(">>>< <TERMINA PROCESO 1 ><<\n");
  PROCESS_END();
}


PROCESS(tss_process2, "Proceso TSS");
PROCESS_THREAD(tss_process2, ev, data)
{
  static rimeaddr_t nodo_dest;

  PROCESS_BEGIN();
  n=1;

  if
((rimeaddr_cmp(&rimeaddr_node_addr,&nodo_40)==0)&&(rimeaddr_cmp(&rimeaddr_node_addr,&nod
o_50)==0)) /*NO ERES NI CLIENTE NI SERVIDOR (NI 40 NI 50)*/
  {
/* MECANISMO TSS */

/*3*/  if (((BUF->flags & 0x10)==16) && (uip_buf[14] == 50) && (uip_buf[15] == 0))
/*COMPROBAR SI SE TRATA DE UN MENSAJE ACK DEL SERVIDOR*/
       {
        leds_on(LEDS_BLUE);
                 If ( (calcular_seq(&uip_buf[28])-1) > acknowledged )
                 {
                  ret_rto = 0;
```

```
                    printf("RET_RTO = %d\n",ret_rto);
                    rto_event = process_alloc_event();
                    rimebuf_copyto(buffer_ack);
                    acknowledged = (calcular_seq(&buffer_ack[28])-1);
                  }

                  if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_45)!=0)
                  {
                      PROCESS_WAIT_UNTIL(process_is_running(&tss_process)==0);
        }

                  if ( (acknowledged > confirmed) && (cache_vacio(buffer_queue)==1) && (
((acknowledged+1)>=calcular_seq(&buffer_queue[24])) &&
((acknowledged+1)<=(calcular_seq(&buffer_queue[24]) + 70 -1)) ) )
                  {
                   tbq=1;
          rimebuf_copyfrom(&buffer_queue[UIP_LLH_LEN], uip_len_buf);
                  rimebuf_copyto(buffer_tss);
                  memset(buffer_queue,0,110);
                  printf("\n+++++HEMOS BORRADO EL PAQUETE %d de
BUFFER_QUEUE++++\n",calcular_seq(&buffer_queue[24]));
tx_buffer_queue2:
                  transmitted= (calcular_seq(&buffer_tss[24]) + 70 -1);
         nodo_dest = det_routing(NO_RET);
                  if (tbq==1)
                  {
                   for (i=0;i<2000;i++)
                   {
                    if ((i%1000)==0)
                   {
                    PRINTF("%d...",i);
                   }
                   }
                  unicast_send(&dataconn,&nodo_dest);
                  n_paq++;
                  temps = calcular_tiempo();
                            printf("\n+++++++ HEMOS BORRADO EL PAQUETE %d DEL
                      tbq=0;
                   }
                 else
                 {
                  unicast_send(&dataconn,&nodo_dest);
                  n_paq++;
                  temps = calcular_tiempo();
segundos\n",nodo_dest.u8[0],nodo_dest.u8[1],temps.segundos, temps.milesimas);
                 }
                  ret_ll = 1;
                  ret_rto = 1;
                  while ((ret_ll==1) || (ret_rto==1))
                  {
                       if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_45)!=0)
                       {goto timertcp3;
                      }
                      etimer_set(&ll_timer,LL_TIMER);
                       PROCESS_WAIT_EVENT_UNTIL((etimer_expired(&ll_timer)) || (ev ==
ll_event));
                      if (ret_ll==1)
                      {
                         leds_on(LEDS_RED);
                         printf("\nEntrando en la temporizacion RX_ACK/
TX_DE_BUFFER_QUEUE.... ACTUALIZADO 2\n");
                         timertcp3:
                          etimer_set(&rto_timer, CLOCK_SECOND * rtt/4096);
                         PROCESS_WAIT_EVENT_UNTIL((etimer_expired(&rto_timer)) || (ev ==
rto_event));
                          if (ret_rto == 1)
                         {
                            enviar_paq_buf(0);
                         }
                         else
                         {
                            ret_ll = 0;
                             confirmed= (calcular_seq(&buffer_tss[24]) + 70 -1);
                           if ( (cache_vacio(buffer_queue)==0) && ( (confirmed+1) >=
(calcular_seq (&buffer_queue[24]) ) ) && ( (confirmed+1) <=
(abs(calcular_seq(&buffer_queue[24]) + 70 -1)) ) )
                            {
```

```
                        rimebuf_copyfrom(&buffer_queue[UIP_LLH_LEN], uip_len_buf);
                        rimebuf_copyto(buffer_tss);
                        memset(buffer_queue,0,110);
                        tbq=1;
                        goto tx_buffer_queue2;
                       }/*fin del if de transmitir paquete almacenado*/
                  break;
                }
              }
              else
              {
                          ret_rto = 0;
                         confirmed= (calcular_seq(&buffer_tss[24]) + 70 -1);
            printf("$$$$$$$ SALGO DE TEMPORIZACION LINK LAYER $$$$$$\n\n");
                          if ( (cache_vacio(buffer_queue)==0) && ( (confirmed+1) >=
(calcular_seq (&buffer_queue[24]) ) ) && ( (confirmed+1) <=
(abs(calcular_seq(&buffer_queue[24]) + 70 -1)) ) )
                          {
                           rimebuf_copyfrom(&buffer_queue[UIP_LLH_LEN], uip_len_buf);
                           rimebuf_copyto(buffer_tss);
                           memset(buffer_queue,0,110);
                           tbq=1;
                           goto tx_buffer_queue2;
                          }/*fin del if de transmitir paquete almacenado*/
                         break;
                       }
                    }/*fin del while*/
                    memset(buffer_tss,0,110);
                   }


                  if ( calcular_seq(&uip_buf[28]) <= ack_forwarded )
                  {
                   goto drop;
                  }
                  else
                  {
                    if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_45)!=0)  //eres nodo 45
                    {
                      if (acknowledged%(MODULO*70)==0)
                      {
                       goto tx_last_ack;
                      }
                      else
                      {
                       goto drop;
                      }
                    }
tx_last_ack:
                    rimebuf_copyfrom(&uip_buf[UIP_LLH_LEN], uip_len);
                      nodo_dest = det_routing(NO_RET);
                      unicast_send(&dataconn,&nodo_dest);
                     n_ack++;
                      ret_ack=1;
                      if ( acknowledged%(MODULO*70)==0 )
                      {
                       while (ret_ack==1)
                       {
                        etimer_set(&ack_timer,ACK_TIMER);
                        PROCESS_WAIT_EVENT_UNTIL((etimer_expired(&ack_timer)) ||
(ev==ack_event));
                        if (ret_ack==1)
                          {
                           enviar_paq_buf(1);
                                 }
                          else
                          {
                           printf("*****YA SE HA ESCUCHADO LA TRANSMISION ACK DEL NODO
SUCESOR*****\n");
                           ack_forwarded = calcular_seq(&buffer_ack[28]);
                          }
                       }
                      }
                  }
                attempts=1;
        drop:
        leds_off(LEDS_BLUE);
```

```
       }

/*1*/  else /*DATOS O EL SYN QUE ENVIA EL CLIENTE EN EL 1er PASO DEL ESTABLECIMIENTO DE
LA COMUNICACION*/
       {
       leds_on(LEDS_GREEN);

/*codigo tss para data:soy proxima direccion*/
         if (calcular_seq(&uip_buf[24]) > (transmitted+1))
         {
          mayor_txed++;
          printf("borro el paquete se cumple > TXTTED +1\n");
          goto drop2;
         }
         else if   ( (calcular_seq(&uip_buf[24]) > (confirmed +1)) && (
(cache_vacio(buffer_queue)==1) || (calcular_seq(&uip_buf[24]) <
calcular_seq(&buffer_queue[24])) ) )
         {
          rimebuf_copyto(buffer_queue);
          uip_len_buf = uip_len;
         }
         else if ((calcular_seq(&uip_buf[24]) + 70 -1) <= confirmed)
         {
          if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_41)!=0)
          {
           paq_filter++;
           if (acknowledged == confirmed)
           {
            rtxw++;
            if ( (calcular_seq(&uip_buf[24]) + 70 -1)%(MODULO*70)==0 )
            {
              //retransmitir ack y escucharlo
              enviar_paq_buf (1); //asumo que el ack acknowledged lo guardo en
buffer_ack
              ret_ack=1;
              while (ret_ack==1)
              {
               etimer_set(&ack_timer,ACK_TIMER);
               PROCESS_WAIT_EVENT_UNTIL((etimer_expired(&ack_timer)) ||
(ev==ack_event));
               if (ret_ack==1) {
                enviar_paq_buf(1);
                }else
                {
                 ack_forwarded = calcular_seq(&buffer_ack[28]);
                 printf("*****YA SE HA ESCUCHADO LA TRANSMISION DEL NODO
SUCESOR*****\n");
                }
              }
              rtxw=0;
            }
           }
          goto drop2;
         }
         }
         else if  ( ((calcular_seq(&uip_buf[24]) + 70 -1) <= acknowledged) )
         {
           /*  Note: 70 is the data lenght
             1. retransmitir_ack;
             2. iniciar temporizador Y*ack_forwarding_time;
             3. attempts=1;
             4. borro el paquete; */
             /*1*/
          if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_45)!=0)
          {
           enviar_paq_buf (1); //asumo que el ack acknowledged lo guardo en buffer_ack
           ret_ack=1;
           if ( (calcular_seq(&buffer_ack[28]) -1)%(MODULO*70)==0 )
           {
             while (ret_ack==1)
             {
              etimer_set(&ack_timer,ACK_TIMER);
              PROCESS_WAIT_EVENT_UNTIL((etimer_expired(&ack_timer)) || (ev==ack_event));
              if (ret_ack==1) {
               enviar_paq_buf(1);
               }else
               {
```

```
                ack_forwarded = calcular_seq(&buffer_ack[28]);
                printf("*****YA SE HA ESCUCHADO LA TRANSMISION DEL NODO
SUCESOR*****\n");
               }
             }
             /*3*/ // attempts=1;
           }
          }
             printf("****** BORRO PAQUETE PORQUE YA FUE ENVIADO******\n");
             goto drop2;
      }
         else if   ((transmitted == confirmed) &&
(((confirmed+1)>=(calcular_seq(&uip_buf[24]))) &&
((confirmed+1)<=(calcular_seq(&uip_buf[24]) + uip_len -1))))
         {
               rimebuf_copyto(buffer_tss);
               tbq=0;
tx_buffer_queue3:
               transmitted= (calcular_seq(&buffer_tss[24])  + 70 -1);
               if (tbq==1)
               {
                 nodo_dest = det_routing(NO_RETBUF);
                }
                else
                {
                  nodo_dest = det_routing(NO_RET);
                }
                if (tbq==1)
                {
                  for (i=0;i<2000;i++)
                 {
                  if ((i%1000)==0)
                  {
                   PRINTF("%d...",i);
                  }
                 }
          unicast_send(&dataconn,&nodo_dest);
          n_paq++;
          temps = calcular_tiempo();
                    memset(buffer_queue,0,110);
                    tbq=0;
                }
                else
                {
                unicast_send(&dataconn,&nodo_dest);
                n_paq++;
                temps = calcular_tiempo();
                 }
               ret_ll = 1;
               ret_rto = 1;
               while ((ret_ll==1) || (ret_rto==1))
               {
                           if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_45)!=0)
                           {
                             goto timertcp2;
                }
               etimer_set(&ll_timer,LL_TIMER); /*CLOCK_SECOND es un parametro que debe
cambiarse para optimizar el proceso*/
               PROCESS_WAIT_EVENT_UNTIL((etimer_expired(&ll_timer)) || (ev ==
ll_event));
               if (ret_ll==1)
               {
                leds_on(LEDS_RED);
                printf("\nEntrando en la temporizacion.... ACTUALIZADO 2\n");
                printf("\n$$$$ ENTRO EN TEMPORIZACION TCP $$$\n\n");
                timertcp2:
                 etimer_set(&rto_timer, CLOCK_SECOND * rtt/4096);
                PROCESS_WAIT_EVENT_UNTIL((etimer_expired(&rto_timer)) || (ev ==
rto_event));
                 if (ret_rto == 1)
                {
                 enviar_paq_buf(0); /*La funcion de probabilidad de envio esta incluida
en enviar_paq_buf*/
                }
                else
                {
                 ret_ll = 0;
```

```
                if ( (calcular_seq(&buffer_tss[24]) + 70 -1) > confirmed )
               {
                  confirmed= (calcular_seq(&buffer_tss[24]) + 70 -1);
                }
                if ( (cache_vacio(buffer_queue)==0) && ( (confirmed+1) >= (calcular_seq
(&buffer_queue[24]) ) ) && ( (confirmed+1) <= (abs(calcular_seq(&buffer_queue[24]) + 70
-1)) ) )
                {
                  rimebuf_copyfrom(&buffer_queue[UIP_LLH_LEN], uip_len_buf);
                  rimebuf_copyto(buffer_tss);
                  tbq=1;
                  goto tx_buffer_queue3;
                }/*fin del if de transmitir paquete almacenado*/
               break;
        }
       }
       else
       {
                  if ( (calcular_seq(&buffer_tss[24]) + 70 -1) > confirmed )
        {
                   confirmed= (calcular_seq(&buffer_tss[24]) + 70 -1);
                  }
        ret_rto = 0;
                  printf("$ SALGO DE TEMPORIZACION LINK LAYER $$$$$$$\n\n");
                 if ( (cache_vacio(buffer_queue)==0) && ( (confirmed+1) >= (calcular_seq
(&buffer_queue[24]) ) ) && ( (confirmed+1) <= (abs(calcular_seq(&buffer_queue[24]) + 70
-1)) ) )
                {
                  rimebuf_copyfrom(&buffer_queue[UIP_LLH_LEN], uip_len_buf);
                  rimebuf_copyto(buffer_tss);
                  tbq=1;
                  goto tx_buffer_queue3;
                }/*fin del if de transmitir paquete almacenado*/
        break;
       }
       }/*fin del while*/
          memset(buffer_tss,0,110);
          leds_off(LEDS_RED);
        }
         else
         {
          borro_paq++;
         }
       drop2:
       leds_off(LEDS_GREEN);
       }
  }
  //la parte cliente y servidor se encuentra en la funcion recv_data
  n=0;
  printf(">>>>>>>>< <TERMINA PROCESO 2 >>>><<<<<\n");
  PROCESS_END();
}
/*---------------------------------------------------------------------------*/
/*FUNCIONES TSS*/
int cache_vacio (u8_t *cadena)
{
  int i;
  int j=1;

  for(i=0; i<110; i++)
  {
       if (cadena[i]!=0)
       {
          j=0;
       }
  }
  return j;
}

rimeaddr_t det_routing (int c)
{
       static rimeaddr_t direccion;

       if (c==RET) /*Se trata de una retransmisiÃ³n*/
       {
              direccion.u8[0] = buffer_tss[18];
```

```
                direccion.u8[1] = buffer_tss[19];
        }

        if (c==NO_RET) /*NO Se trata de una retransmisiÃ³n*/
        {
                direccion.u8[0] = uip_buf[18];
                direccion.u8[1] = uip_buf[19];
        }

         if (c==NO_RETBUF) /*NO Se trata de una retransmisiÃ³n*/
        {
                direccion.u8[0] = buffer_queue[18];
                direccion.u8[1] = buffer_queue[19];
        }

        if (c==RET_ACK) /*Se trata de una retransmisiÃ³n*/
        {
                direccion.u8[0] = buffer_ack[18];
                direccion.u8[1] = buffer_ack[19];
        }


        if (rimeaddr_cmp(&direccion,&nodo_oculto_izq)!=0) /*diferente de 0 si son iguales
||| 0 si son diferentes */
        {
                rimeaddr_copy(&direccion, &nodo_sig_izq);
         }

        if (rimeaddr_cmp(&direccion,&nodo_oculto_der)!=0) /*diferente de 0 si son iguales
||| 0 si son diferentes */
        {
                rimeaddr_copy(&direccion, &nodo_sig_der);
        }

        return direccion;
}

void enviar_paq_buf (int tipo)
{
  static rimeaddr_t nodo_dest;
switch (tipo)   {
case 0: /* retransmitir paquetes*/
         rtx_data++;
        if (cache_vacio(buffer_tss)==0)
        {
                   rimebuf_copyfrom(buffer_tss,110);
                nodo_dest = det_routing(RET);
                        unicast_send(&dataconn,&nodo_dest);
                        n_paq++;
        }
         break;
 case 1:  /* retransmitir ack*/
        rimebuf_copyfrom(buffer_ack,40);
        nodo_dest = det_routing(RET_ACK);
                    rtx_ack++;
          unicast_send(&dataconn,&nodo_dest);
          n_ack++;
          break;

 }/*fin del switch(tipo)*/
}


int calcular_seq (u8_t cadena[4])
{
        u8_t calc[4];
        u8_t i;

        for (i=0;i<4;i++)
        {
                calc[i]=cadena[3-i];

        }
        return *(int *)calc;
}

void recv_ack_ll (struct unicast_conn *c, rimeaddr_t *from)
```

```
{

  if    (rimeaddr_cmp(from,&nodo_40)!=0) /*eres nodo 40*/
  {
      if (prob(PP_LL)==1)
      {
        ret_ack = 0;
       ack_event = process_alloc_event();
       temps = calcular_tiempo();
       printf("HEMOS RECIBIDO UN ACK A NIVEL DE ENLACE DE  %u.%u a los %u.%u
segundos\n", from->u8[0], from->u8[1],temps.segundos, temps.milesimas);
      }
      else
      {
        no_l2++;
      }
  }
}

int prob(int valor)
{
       unsigned int p, x;

       p = 100 - valor;
       x = abs(rand())%100;

       if (p >= x)
       {
              return 1;
       }
       else
       {
              return 0;
       }
}

struct timestamp calcular_tiempo(void)
{
  struct timestamp t;
  tiempo = timesynch_time();

  if (tiempo < ult_tiempo)
  {
     w++;
  }

  ult_tiempo = tiempo;

  t.segundos = tiempo/4096 + w * 16;
  t.milesimas = (1000*tiempo/4096)%1000;

  return t;
}

unsigned long calcular_rtt (unsigned short t1, unsigned short t2)
{
       unsigned long rtt;

       rtt = 3 * (t2 - t1) / 2;

       return rtt;
}

/*FIN FUNCIONES TSS*/
/*-------------------------------------------------------------------------*/


static void
recv_data(struct unicast_conn *c, rimeaddr_t *from)
{

  uip_len = rimebuf_copyto(&uip_buf[UIP_LLH_LEN]);

  /*  uip_len = hc_inflate(&uip_buf[UIP_LLH_LEN], uip_len);*/

  PRINTF("uip-over-mesh: %d.%d: recv_data with len %d\n",
         rimeaddr_node_addr.u8[0], rimeaddr_node_addr.u8[1], uip_len);
```

```
        temps = calcular_tiempo();
        printf("\n**************************************\n");
        printf("MENSAJE RECIBIDO DE %d.%d A LOS %u.%u segundos.\n",from->u8[0], from-
>u8[1], temps.segundos, temps.milesimas);
        printf("Se han copiado %d bytes de informacion\n",uip_len);
        printf("**************************************\n");
        printf("NUMERO DE SECUENCIA = %d\n",calcular_seq(&uip_buf[24]));
        printf("NUMERO DE ACK = %d\n",calcular_seq(&uip_buf[28]));
        printf("FLAGS del mensaje recibido: %02x\n",uip_buf[33]);
        printf("**********************************\n");
        printf("**********************************\n");

if
((rimeaddr_cmp(&rimeaddr_node_addr,&nodo_40)==0)&&(rimeaddr_cmp(&rimeaddr_node_addr,&nod
o_50)==0)) /*NO ERES NI CLIENTE NI SERVIDOR (NI 40 NI 50)*/
  {
/*3*/  if (((BUF->flags & 0x10)==16) && (uip_buf[14] == 50) && (uip_buf[15] == 0))
/*COMPROBAR SI SE TRATA DE UN MENSAJE ACK DEL SERVIDOR*/
        {
                leds_on(LEDS_BLUE);
                 printf("******valor de flags %d\n",(BUF->flags & 0x10));
                if ((BUF->flags & 0x3f) ==18) /*Mensaje SYN/ACK*/
                {
                        rtt2 = timesynch_time();
                        rtt = calcular_rtt(rtt1,rtt2);
                         nodo_dest = det_routing(NO_RET);
                        unicast_send(&dataconn,&nodo_dest);
                        goto drop;
                }

                  if (prob(PP_ACK)==1)
                  {
                    if (process_is_running(&tss_process)==0)
                    {
                    process_start(&tss_process,NULL);
                    }
                    else
                    {
                     if (process_is_running(&tss_process2)==0)
                     {
                     process_start(&tss_process2,NULL);
                     }
                   }
                   else
                   {
                    no_ack++;
                   }
         drop:
         leds_off(LEDS_BLUE);
        }
        else
        {
         leds_on(LEDS_GREEN);
         if (((BUF->flags & 0x02) == 2) || (((BUF->flags & 0x18) == 16) && (uip_buf[14]
== 40) && (uip_buf[15] == 0))) /*Mensaje SYN o ACK en contestacion al SYN/ACK */
                {
                        if ((BUF->flags & 0x02) == 2)
                        {
                                nodo_dest = det_routing(NO_RET);
                                unicast_send(&dataconn,&nodo_dest);
                                rtt1 = timesynch_time();
                                goto drop2;
                        }
                        else
                        {
                                nodo_dest = det_routing(NO_RET);
                                unicast_send(&dataconn,&nodo_dest);
                                goto drop2;
                        }
                }
         if (prob(PP_PAQ)==1)
         {
           if (process_is_running(&tss_process)==0)
           {
           process_start(&tss_process,NULL);
           }
           else
```

```
          {
          process_start(&tss_process2,NULL);
          }
         }
         else
         {
          no_data++;
         }
         drop2:
        leds_off(LEDS_GREEN);
       }
  }
  else
  {
   if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_50)!=0)
   {
     if ((BUF->flags & 0x18) == 24)
     {

        if (prob(PP_PAQ)==1)
        {
              rimebuf_copyfrom(&uip_buf[UIP_LLH_LEN], uip_len);
              leds_on(LEDS_GREEN);
              tcpip_input();
              leds_off(LEDS_GREEN);
        }
         else
         {
          no_data++;
         }
     }
     else
     {
       leds_on(LEDS_GREEN);
       tcpip_input();
       leds_off(LEDS_GREEN);
     }
   }

      if  (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_40)!=0)
      {
        if ( ((BUF->flags & 0x10)==16) && ((BUF->flags & 0x3f)!=18) && (uip_buf[14] ==
50) && (uip_buf[15] == 0) )
        {
           if (prob(PP_ACK)==1)
           {
            if ( (calcular_seq(&uip_buf[28])-1) > acknowledged )
            {
              acknowledged = (calcular_seq(&uip_buf[28])-1);
              printf("<<<<<<<<<< ACKNOWLEDGED = %d >>>>>>>>>\n",acknowledged);
            }
               rimebuf_copyfrom(NULL,0);
               unicast_send(&ackconn,&nodo_sig_der);
               n_ll++;
               printf("(SOY CLIENTE ACTUALIZADO 1) HEMOS ENVIADO UN ACK A NIVEL DE
ENLACE A %d.%d\n",nodo_sig_der.u8[0],nodo_sig_der.u8[1]);
               rimebuf_copyfrom(&uip_buf[UIP_LLH_LEN], uip_len);
                leds_on(LEDS_GREEN);
               tcpip_input();
               leds_off(LEDS_GREEN);
          }
          else
          {
          no_ack++;
          }
        }
        else
        {
        leds_on(LEDS_GREEN);
        tcpip_input();
        leds_off(LEDS_GREEN);
        }
      }
  }

}
/*---------------------------------------------------------------------------*/
```

```
static void
send_data(rimeaddr_t *next)
{
  PRINTF("*********************************************\n");
  PRINTF("uip-over-mesh: %d.%d: send_data with len %d to %d.%d\n",
         rimeaddr_node_addr.u8[0], rimeaddr_node_addr.u8[1],
         rimebuf_totlen(), next->u8[0], next->u8[1]);
  PRINTF("*********************************************\n");
  unicast_send(&dataconn, next);
}
/*---------------------------------------------------------------------------*/
static void
new_route(struct route_discovery_conn *c, rimeaddr_t *to)
{
  struct route_entry *rt;

  if(queued_packet) {
    PRINTF("uip-over-mesh: new route, sending queued packet\n");
    queuebuf_to_rimebuf(queued_packet);
    queuebuf_free(queued_packet);
    queued_packet = NULL;

    rt = route_lookup(&queued_receiver);
    if(rt) {
      send_data(&queued_receiver);
    }
  }
}
/*---------------------------------------------------------------------------*/
static void
timedout(struct route_discovery_conn *c)
{
  PRINTF("uip-over-mesh: packet timed out\n");
  if(queued_packet) {
    PRINTF("uip-over-mesh: freeing queued packet\n");
    queuebuf_free(queued_packet);
    queued_packet = NULL;
  }
}
/*---------------------------------------------------------------------------*/
static const struct unicast_callbacks data_callbacks = { recv_data };
static const struct route_discovery_callbacks rdc = { new_route, timedout };
static const struct unicast_callbacks ack_callbacks = {recv_ack_ll};
/*---------------------------------------------------------------------------*/
struct gateway_msg {
  rimeaddr_t gateway;
};

static uint8_t is_gateway;

static void
gateway_announce_recv(struct trickle_conn *c)
{
  struct gateway_msg *msg;
  msg = rimebuf_dataptr();
  PRINTF("%d.%d: gateway message: %d.%d\n",
         rimeaddr_node_addr.u8[0], rimeaddr_node_addr.u8[1],
         msg->gateway.u8[0], msg->gateway.u8[1]);

  if(!is_gateway) {
    uip_over_mesh_set_gateway(&msg->gateway);
  }

}
/*---------------------------------------------------------------------------*/
void
uip_over_mesh_make_announced_gateway(void)
{
  struct gateway_msg msg;
  /* Make this node the gateway node, unless it already is the
     gateway. */
  if(!is_gateway) {
    PRINTF("%d.%d: making myself the gateway\n",
           rimeaddr_node_addr.u8[0], rimeaddr_node_addr.u8[1]);
    uip_over_mesh_set_gateway(&rimeaddr_node_addr);
    rimeaddr_copy(&(msg.gateway), &rimeaddr_node_addr);
    rimebuf_copyfrom(&msg, sizeof(struct gateway_msg));
```

```
    trickle_send(&gateway_announce_conn);
    is_gateway = 1;
  }
}
const static struct trickle_callbacks trickle_call = {gateway_announce_recv};
/*---------------------------------------------------------------------------*/
void
uip_over_mesh_init(u16_t channels)
{

  PRINTF("Our address is %d.%d (%d.%d.%d.%d)\n",
         rimeaddr_node_addr.u8[0], rimeaddr_node_addr.u8[1],
         uip_hostaddr.u8[0], uip_hostaddr.u8[1],
         uip_hostaddr.u8[2], uip_hostaddr.u8[3]);

  unicast_open(&dataconn, channels, &data_callbacks);
  route_discovery_open(&route_discovery, CLOCK_SECOND / 4,
                       channels + 1, &rdc);
  trickle_open(&gateway_announce_conn, CLOCK_SECOND * 4, channels + 3,
               &trickle_call);

        unicast_open(&ackconn,channels+10,&ack_callbacks);

        memset(buffer_tss,0,110);

        if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_41)!=0) /*NODO 41*/
        {
                nodo_sig_der.u8[0] = 42;
                nodo_sig_der.u8[1] = 0;
                nodo_oculto_der.u8[0] = 50;
                nodo_oculto_der.u8[1] = 0;
                nodo_sig_izq.u8[0] = 40;
                nodo_sig_izq.u8[1] = 0;
        }

        if
((rimeaddr_cmp(&rimeaddr_node_addr,&nodo_40)==0)&&(rimeaddr_cmp(&rimeaddr_node_addr,&nod
o_41)==0)&&(rimeaddr_cmp(&rimeaddr_node_addr,&nodo_45)==0)&&(rimeaddr_cmp(&rimeaddr_node
_addr,&nodo_50)==0)) /*NODOS 42,43,44*/
        {
                nodo_sig_der.u8[0] = rimeaddr_node_addr.u8[0] + 1;
                nodo_sig_der.u8[1] = 0;
                nodo_oculto_der.u8[0] = 50;
                nodo_oculto_der.u8[1] = 0;
                nodo_sig_izq.u8[0] = rimeaddr_node_addr.u8[0] - 1;
                nodo_sig_izq.u8[1] = 0;
                nodo_oculto_izq.u8[0] = 40;
                nodo_oculto_izq.u8[1] = 0;
        }

        if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_45)!=0) /*NODO 45*/
        {
                nodo_sig_izq.u8[0] = 44;
                nodo_sig_izq.u8[1] = 0;
                nodo_oculto_izq.u8[0] = 40;
                nodo_oculto_izq.u8[1] = 0;
                nodo_sig_der.u8[0] = 50;
                nodo_sig_der.u8[1] = 0;
        }

        if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_50)!=0) /*NODO 50*/
        {
                nodo_sig_izq.u8[0] = 45;
                nodo_sig_izq.u8[1] = 0;
        }

         if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_40)!=0) /*NODO 40*/
        {
                nodo_sig_der.u8[0] = 41;
                nodo_sig_der.u8[1] = 0;
        }
        ret_ll = 0;
        ret_rto = 0;
         ret_ack = 0;
        /*Tomamos una muestra de tiempo para inicializar el bloque de aleatoriedad.*/
        /*Guardamos esta muestra en la estructura RTT1 que mas tarde sera recalculada.*/
        rtt1 = timesynch_time();
```

```
       srand(rtt1);

/* FIN */
  /*  tcpip_set_forwarding(1);*/

}
/*---------------------------------------------------------------------------*/
u8_t
uip_over_mesh_send(void)
{
  rimeaddr_t receiver;
  struct route_entry *rt;
  static rimeaddr_t invento;
  int i=0;

  if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_40)!=0) /*NODO 40*/
  {
       invento.u8[0] = 41;
       invento.u8[1] = 0;
       rimeaddr_copy(&gateway,&invento);
       route_add(&receiver,&gateway,3,0);
       route_add(&gateway,&gateway,1,0);
       i = 1;
  }

  if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_50)!=0) /*NODO 50*/
  {
       invento.u8[0] = 45;
       invento.u8[1] = 0;
       rimeaddr_copy(&gateway,&invento);
       route_add(&receiver,&gateway,3,0);
       route_add(&gateway,&gateway,1,0);
       i = 1;
  }

  if ((uip_ipaddr_maskcmp(&BUF->destipaddr, &netaddr, &netmask))&&(i==0)) {
    receiver.u8[0] = BUF->destipaddr.u8[2];
    receiver.u8[1] = BUF->destipaddr.u8[3];
  } else {
    if(rimeaddr_cmp(&gateway, &rimeaddr_node_addr)) {
      PRINTF("uip_over_mesh_send: I am gateway, packet to %d.%d.%d.%d to local
interface\n",
             uip_ipaddr_to_quad(&BUF->destipaddr));
      if(gw_netif != NULL) {
       return gw_netif->output();
      }
      return UIP_FW_DROPPED;
    } else if(rimeaddr_cmp(&gateway, &rimeaddr_null)) {
      PRINTF("uip_over_mesh_send: No gateway setup, dropping packet\n");
      return UIP_FW_OK;
    } else {
      PRINTF("uip_over_mesh_send: forwarding packet to %d.%d.%d.%d towards gateway
%d.%d\n",
             uip_ipaddr_to_quad(&BUF->destipaddr),
             gateway.u8[0], gateway.u8[1]);
      rimeaddr_copy(&receiver, &gateway);
    }
  }

  PRINTF("uIP over mesh send to %d.%d with len %d\n",
         receiver.u8[0], receiver.u8[1],
         uip_len);

  /*  uip_len = hc_compress(&uip_buf[UIP_LLH_LEN], uip_len);*/

  rimebuf_copyfrom(&uip_buf[UIP_LLH_LEN], uip_len);

  rt = route_lookup(&receiver);
  if(rt == NULL) {
    PRINTF("uIP over mesh no route to %d.%d\n", receiver.u8[0], receiver.u8[1]);
    if(queued_packet == NULL) {
      queued_packet = queuebuf_new_from_rimebuf();
      rimeaddr_copy(&queued_receiver, &receiver);
      route_discovery_discover(&route_discovery, &receiver, ROUTE_TIMEOUT);
    } else if(!rimeaddr_cmp(&queued_receiver, &receiver)) {
      route_discovery_discover(&route_discovery, &receiver, ROUTE_TIMEOUT);
    }
```

```
  } else {

        temps = calcular_tiempo();
        PRINTF("\n*******************************************\n");
        PRINTF("MENSAJE ENVIADO A %d.%d A LOS %u.%u segundos.\n",receiver.u8[0],
receiver.u8[1], temps.segundos, temps.milesimas);
        PRINTF("Se han copiado %d bytes de informacion\n",uip_len);
        PRINTF("*******************************************\n");
        PRINTF("NUMERO DE SECUENCIA = %d\n",calcular_seq(&uip_buf[24]));
        PRINTF("NUMERO DE ACK = %d\n",calcular_seq(&uip_buf[28]));
        PRINTF("FLAGS del mensaje enviado: %02x\n",uip_buf[33]);


        if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_40)!=0) /*ERES EL CLIENTE (40)*/
        {
                if (BUF->flags==24)
                {
                                unicast_send(&dataconn,&gateway);
                                n_paq++;
                }
                else
                {
                        unicast_send(&dataconn,&gateway);
                }
        }

        if (rimeaddr_cmp(&rimeaddr_node_addr,&nodo_50)!=0) /*ERES EL SERVIDOR (50)*/
        {
                if (BUF->flags==16)
                {
                                unicast_send(&dataconn,&gateway);
                                n_ack++;
                }
                else
                {
                        unicast_send(&dataconn,&gateway);
                }

        }

  }
  return UIP_FW_OK;
}
/*---------------------------------------------------------------------------*/
void
uip_over_mesh_set_gateway_netif(struct uip_fw_netif *n)
{
  gw_netif = n;
}
/*---------------------------------------------------------------------------*/
void
uip_over_mesh_set_gateway(rimeaddr_t *gw)
{
  rimeaddr_copy(&gateway, gw);
}
/*---------------------------------------------------------------------------*/
void
uip_over_mesh_set_net(uip_ipaddr_t *addr, uip_ipaddr_t *mask)
{
  uip_ipaddr_copy(&netaddr, addr);
  uip_ipaddr_copy(&netmask, mask);
}
/*---------------------------------------------------------------------------*/
```

Archivo uip.c (NODO 40 – CLIENTE)

```
#define DEBUG_PRINTF(...) /*printf(__VA_ARGS__)*/


/**
 * \file
 * The uIP TCP/IP stack code.
 */


#include "net/uip.h"
#include "net/uipopt.h"
#include "net/uip_arp.h"
#include "net/uip_arch.h"
#include "net/rime/rimebuf.h"

#if !UIP_CONF_IPV6 /* If UIP_CONF_IPV6 is defined, we compile the
                      uip6.c file instead of this one. Therefore
                      this #ifndef removes the entire compilation
                      output of the uip.c file */


#if UIP_CONF_IPV6
#include "net/uip-neighbor.h"
#endif /* UIP_CONF_IPV6 */

#include <string.h>

/*---------------------------------------------------------------------------*/
/* Variable definitions. */


/* The IP address of this host. If it is defined to be fixed (by
   setting UIP_FIXEDADDR to 1 in uipopt.h), the address is set
   here. Otherwise, the address */
#if UIP_FIXEDADDR > 0
const uip_ipaddr_t uip_hostaddr =
  { UIP_IPADDR0, UIP_IPADDR1, UIP_IPADDR2, UIP_IPADDR3 };
const uip_ipaddr_t uip_draddr =
  { UIP_DRIPADDR0, UIP_DRIPADDR1, UIP_DRIPADDR2, UIP_DRIPADDR3 };
const uip_ipaddr_t uip_netmask =
  { UIP_NETMASK0, UIP_NETMASK1, UIP_NETMASK2, UIP_NETMASK3 };
#else
uip_ipaddr_t uip_hostaddr, uip_draddr, uip_netmask;
#endif /* UIP_FIXEDADDR */

const uip_ipaddr_t uip_broadcast_addr =
#if UIP_CONF_IPV6
  { { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff,
      0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff } };
#else /* UIP_CONF_IPV6 */
  { { 0xff, 0xff, 0xff, 0xff } };
#endif /* UIP_CONF_IPV6 */
const uip_ipaddr_t uip_all_zeroes_addr = { { 0x0, /* rest is 0 */ } };

#if UIP_FIXEDETHADDR
const struct uip_eth_addr uip_ethaddr = {{UIP_ETHADDR0,
                                          UIP_ETHADDR1,
                                          UIP_ETHADDR2,
                                          UIP_ETHADDR3,
                                          UIP_ETHADDR4,
                                          UIP_ETHADDR5}};
#else
struct uip_eth_addr uip_ethaddr = {{0,0,0,0,0,0}};
#endif

#ifndef UIP_CONF_EXTERNAL_BUFFER
u8_t uip_buf[UIP_BUFSIZE + 2];   /* The packet buffer that contains
                                    incoming packets. */
#endif /* UIP_CONF_EXTERNAL_BUFFER */

void *uip_appdata;               /* The uip_appdata pointer points to
```

```
                                    application data. */
void *uip_sappdata;                 /* The uip_appdata pointer points to
                                    the application data which is to
                                    be sent. */
#if UIP_URGDATA > 0
void *uip_urgdata;                  /* The uip_urgdata pointer points to
                                    urgent data (out-of-band data), if
                                    present. */
u16_t uip_urglen, uip_surglen;
#endif /* UIP_URGDATA > 0 */

u16_t uip_len, uip_slen;
                                /* The uip_len is either 8 or 16 bits,
                                depending on the maximum packet
                                size. */

u8_t uip_flags;     /* The uip_flags variable is used for
                                communication between the TCP/IP stack
                                and the application program. */
struct uip_conn *uip_conn;      /* uip_conn always points to the current
                                connection. */

struct uip_conn uip_conns[UIP_CONNS];
                                /* The uip_conns array holds all TCP
                                connections. */
u16_t uip_listenports[UIP_LISTENPORTS];
                                /* The uip_listenports list all currently
                                listning ports. */
#if UIP_UDP
struct uip_udp_conn *uip_udp_conn;
struct uip_udp_conn uip_udp_conns[UIP_UDP_CONNS];
#endif /* UIP_UDP */

static u16_t ipid;              /* Ths ipid variable is an increasing
                                number that is used for the IP ID
                                field. */

void uip_setipid(u16_t id) { ipid = id; }

static u8_t iss[4];             /* The iss variable is used for the TCP
                                initial sequence number. */

#if UIP_ACTIVE_OPEN
static u16_t lastport;          /* Keeps track of the last port used for
                                a new connection. */
#endif /* UIP_ACTIVE_OPEN */

/* Temporary variables. */
u8_t uip_acc32[4];
static u8_t c, opt;
static u16_t tmp16;
int z=0;
int i,j;
int enviando=0;
u16_t k=0;
u8_t num_seq[4]={0,0,0,1};
static u8_t inicial[4]={0,0,0,1};
static int ret=0;
static u8_t cache[MODULO][200];
static int n_ret=0;
#define RAFAGA                  1

/* Structures and definitions. */
#define TCP_FIN 0x01
#define TCP_SYN 0x02
#define TCP_RST 0x04
#define TCP_PSH 0x08
#define TCP_ACK 0x10
#define TCP_URG 0x20
#define TCP_CTL 0x3f

#define TCP_OPT_END     0   /* End of TCP options list */
#define TCP_OPT_NOOP    1   /* "No-operation" TCP option */
#define TCP_OPT_MSS     2   /* Maximum segment size TCP option */

#define TCP_OPT_MSS_LEN 4   /* Length of TCP MSS option. */
```

```
#define ICMP_ECHO_REPLY 0
#define ICMP_ECHO       8

#define ICMP_DEST_UNREACHABLE        3
#define ICMP_PORT_UNREACHABLE        3

#define ICMP6_ECHO_REPLY             129
#define ICMP6_ECHO                   128
#define ICMP6_NEIGHBOR_SOLICITATION  135
#define ICMP6_NEIGHBOR_ADVERTISEMENT 136

#define ICMP6_FLAG_S (1 << 6)

#define ICMP6_OPTION_SOURCE_LINK_ADDRESS 1
#define ICMP6_OPTION_TARGET_LINK_ADDRESS 2


/* Macros. */
#define BUF ((struct uip_tcpip_hdr *)&uip_buf[UIP_LLH_LEN])
#define FBUF ((struct uip_tcpip_hdr *)&uip_reassbuf[0])
#define ICMPBUF ((struct uip_icmpip_hdr *)&uip_buf[UIP_LLH_LEN])
#define UDPBUF ((struct uip_udpip_hdr *)&uip_buf[UIP_LLH_LEN])


#if UIP_STATISTICS == 1
struct uip_stats uip_stat;
#define UIP_STAT(s) s
#else
#define UIP_STAT(s)
#endif /* UIP_STATISTICS == 1 */

#if UIP_LOGGING == 1
#include <stdio.h>
void uip_log(char *msg);
#define UIP_LOG(m) uip_log(m)
#else
#define UIP_LOG(m)
#endif /* UIP_LOGGING == 1 */

int comp_seq(u8_t cadena1[], u8_t cadena2[])
{
  int k;

  for (k=0;k<4;k++)
  {
        if (cadena1[k] > cadena2[k])
        {
                return 1;
        }

        if (cadena1[k]< cadena2[k])
        {
                return 2;
        }
  }

  return 0;
}

#if ! UIP_ARCH_ADD32
void
uip_add32(u8_t *op32, u16_t op16)
{
  uip_acc32[3] = op32[3] + (op16 & 0xff);
  uip_acc32[2] = op32[2] + (op16 >> 8);
  uip_acc32[1] = op32[1];
  uip_acc32[0] = op32[0];

  if(uip_acc32[2] < (op16 >> 8)) {
    ++uip_acc32[1];
    if(uip_acc32[1] == 0) {
      ++uip_acc32[0];
    }
  }


  if(uip_acc32[3] < (op16 & 0xff)) {
```

```
    ++uip_acc32[2];
    if(uip_acc32[2] == 0) {
      ++uip_acc32[1];
      if(uip_acc32[1] == 0) {
       ++uip_acc32[0];
      }
    }
  }
}

#endif /* UIP_ARCH_ADD32 */

#if ! UIP_ARCH_CHKSUM
/*---------------------------------------------------------------------------*/
static u16_t
chksum(u16_t sum, const u8_t *data, u16_t len)
{
  u16_t t;
  const u8_t *dataptr;
  const u8_t *last_byte;

  dataptr = data;
  last_byte = data + len - 1;

  while(dataptr < last_byte) {       /* At least two more bytes */
    t = (dataptr[0] << 8) + dataptr[1];
    sum += t;
    if(sum < t) {
      sum++;           /* carry */
    }
    dataptr += 2;
  }

  if(dataptr == last_byte) {
    t = (dataptr[0] << 8) + 0;
    sum += t;
    if(sum < t) {
      sum++;           /* carry */
    }
  }

  /* Return sum in host byte order. */
  return sum;
}
/*---------------------------------------------------------------------------*/
u16_t
uip_chksum(u16_t *data, u16_t len)
{
  return htons(chksum(0, (u8_t *)data, len));
}
/*---------------------------------------------------------------------------*/
#ifndef UIP_ARCH_IPCHKSUM
u16_t
uip_ipchksum(void)
{
  u16_t sum;

  sum = chksum(0, &uip_buf[UIP_LLH_LEN], UIP_IPH_LEN);
  DEBUG_PRINTF("uip_ipchksum: sum 0x%04x\n", sum);
  return (sum == 0) ? 0xffff : htons(sum);
}
#endif
/*---------------------------------------------------------------------------*/
static u16_t
upper_layer_chksum(u8_t proto)
{
  u16_t upper_layer_len;
  u16_t sum;

#if UIP_CONF_IPV6
  upper_layer_len = (((u16_t)(BUF->len[0]) << 8) + BUF->len[1]);
#else /* UIP_CONF_IPV6 */
  upper_layer_len = (((u16_t)(BUF->len[0]) << 8) + BUF->len[1]) - UIP_IPH_LEN;
#endif /* UIP_CONF_IPV6 */

  /* First sum pseudoheader. */
```

```
  /* IP protocol and length fields. This addition cannot carry. */
  sum = upper_layer_len + proto;
  /* Sum IP source and destination addresses. */
  sum = chksum(sum, (u8_t *)&BUF->srcipaddr, 2 * sizeof(uip_ipaddr_t));

  /* Sum TCP header and data. */
  sum = chksum(sum, &uip_buf[UIP_IPH_LEN + UIP_LLH_LEN],
               upper_layer_len);

  return (sum == 0) ? 0xffff : htons(sum);
}
/*---------------------------------------------------------------------------*/
#if UIP_CONF_IPV6
u16_t
uip_icmp6chksum(void)
{
  return upper_layer_chksum(UIP_PROTO_ICMP6);

}
#endif /* UIP_CONF_IPV6 */
/*---------------------------------------------------------------------------*/
u16_t
uip_tcpchksum(void)
{
  return upper_layer_chksum(UIP_PROTO_TCP);
}
/*---------------------------------------------------------------------------*/
#if UIP_UDP_CHECKSUMS
u16_t
uip_udpchksum(void)
{
  return upper_layer_chksum(UIP_PROTO_UDP);
}
#endif /* UIP_UDP_CHECKSUMS */
#endif /* UIP_ARCH_CHKSUM */
/*---------------------------------------------------------------------------*/
void
uip_init(void)
{
  for(c = 0; c < UIP_LISTENPORTS; ++c) {
    uip_listenports[c] = 0;
  }
  for(c = 0; c < UIP_CONNS; ++c) {
    uip_conns[c].tcpstateflags = UIP_CLOSED;
  }
#if UIP_ACTIVE_OPEN
  lastport = 1024;
#endif /* UIP_ACTIVE_OPEN */

#if UIP_UDP
  for(c = 0; c < UIP_UDP_CONNS; ++c) {
    uip_udp_conns[c].lport = 0;
  }
#endif /* UIP_UDP */


  /* IPv4 initialization. */
#if UIP_FIXEDADDR == 0
  /*  uip_hostaddr[0] = uip_hostaddr[1] = 0;*/
#endif /* UIP_FIXEDADDR */

}
/*---------------------------------------------------------------------------*/
#if UIP_ACTIVE_OPEN
struct uip_conn *
uip_connect(uip_ipaddr_t *ripaddr, u16_t rport)
{
  register struct uip_conn *conn, *cconn;

  /* Find an unused local port. */
 again:
  ++lastport;

  if(lastport >= 32000) {
    lastport = 4096;
  }
```

```
  /* Check if this port is already in use, and if so try to find
     another one. */
  for(c = 0; c < UIP_CONNS; ++c) {
    conn = &uip_conns[c];
    if(conn->tcpstateflags != UIP_CLOSED &&
       conn->lport == htons(lastport)) {
      goto again;
    }
  }

  conn = 0;
  for(c = 0; c < UIP_CONNS; ++c) {
    cconn = &uip_conns[c];
    if(cconn->tcpstateflags == UIP_CLOSED) {
      conn = cconn;
      break;
    }
    if(cconn->tcpstateflags == UIP_TIME_WAIT) {
      if(conn == 0 ||
         cconn->timer > conn->timer) {
        conn = cconn;
      }
    }
  }

  if(conn == 0) {
    return 0;
  }

  conn->tcpstateflags = UIP_SYN_SENT;

  conn->snd_nxt[0] = iss[0];
  conn->snd_nxt[1] = iss[1];
  conn->snd_nxt[2] = iss[2];
  conn->snd_nxt[3] = iss[3];

  conn->initialmss = conn->mss = UIP_TCP_MSS;

  conn->len = 1;   /* TCP length of the SYN is one. */
  conn->nrtx = 0;
  conn->timer = 1; /* Send the SYN next time around. */
  conn->rto = UIP_RTO;
  conn->sa = 0;
  conn->sv = 16;   /* Initial value of the RTT variance. */
  conn->lport = htons(lastport);
  conn->rport = rport;
  uip_ipaddr_copy(&conn->ripaddr, ripaddr);

  return conn;
}
#endif /* UIP_ACTIVE_OPEN */
/*---------------------------------------------------------------------------*/
#if UIP_UDP
struct uip_udp_conn *
uip_udp_new(const uip_ipaddr_t *ripaddr, u16_t rport)
{
  register struct uip_udp_conn *conn;

  /* Find an unused local port. */
 again:
  ++lastport;

  if(lastport >= 32000) {
    lastport = 4096;
  }

  for(c = 0; c < UIP_UDP_CONNS; ++c) {
    if(uip_udp_conns[c].lport == htons(lastport)) {
      goto again;
    }
  }


  conn = 0;
  for(c = 0; c < UIP_UDP_CONNS; ++c) {
    if(uip_udp_conns[c].lport == 0) {
      conn = &uip_udp_conns[c];
```

```
      break;
    }
  }

  if(conn == 0) {
    return 0;
  }

  conn->lport = HTONS(lastport);
  conn->rport = rport;
  if(ripaddr == NULL) {
    memset(&conn->ripaddr, 0, sizeof(uip_ipaddr_t));
  } else {
    uip_ipaddr_copy(&conn->ripaddr, ripaddr);
  }
  conn->ttl = UIP_TTL;

  return conn;
}
#endif /* UIP_UDP */
/*---------------------------------------------------------------------------*/
void
uip_unlisten(u16_t port)
{
  for(c = 0; c < UIP_LISTENPORTS; ++c) {
    if(uip_listenports[c] == port) {
      uip_listenports[c] = 0;
      return;
    }
  }
}
/*---------------------------------------------------------------------------*/
void
uip_listen(u16_t port)
{
  for(c = 0; c < UIP_LISTENPORTS; ++c) {
    if(uip_listenports[c] == 0) {
      uip_listenports[c] = port;
      return;
    }
  }
}
/*---------------------------------------------------------------------------*/
/* XXX: IP fragment reassembly: not well-tested. */

#if UIP_REASSEMBLY && !UIP_CONF_IPV6
#define UIP_REASS_BUFSIZE (UIP_BUFSIZE - UIP_LLH_LEN)
static u8_t uip_reassbuf[UIP_REASS_BUFSIZE];
static u8_t uip_reassbitmap[UIP_REASS_BUFSIZE / (8 * 8)];
static const u8_t bitmap_bits[8] = {0xff, 0x7f, 0x3f, 0x1f,
                                    0x0f, 0x07, 0x03, 0x01};
static u16_t uip_reasslen;
static u8_t uip_reassflags;
#define UIP_REASS_FLAG_LASTFRAG 0x01
static u8_t uip_reasstmr;

#define IP_MF   0x20

static u8_t
uip_reass(void)
{
  u16_t offset, len;
  u16_t i;

  /* If ip_reasstmr is zero, no packet is present in the buffer, so we
     write the IP header of the fragment into the reassembly
     buffer. The timer is updated with the maximum age. */
  if(uip_reasstmr == 0) {
    memcpy(uip_reassbuf, &BUF->vhl, UIP_IPH_LEN);
    uip_reasstmr = UIP_REASS_MAXAGE;
    uip_reassflags = 0;
    /* Clear the bitmap. */
    memset(uip_reassbitmap, 0, sizeof(uip_reassbitmap));
  }

  /* Check if the incoming fragment matches the one currently present
     in the reasembly buffer. If so, we proceed with copying the
```

```
   fragment into the buffer. */
if(BUF->srcipaddr[0] == FBUF->srcipaddr[0] &&
   BUF->srcipaddr[1] == FBUF->srcipaddr[1] &&
   BUF->destipaddr[0] == FBUF->destipaddr[0] &&
   BUF->destipaddr[1] == FBUF->destipaddr[1] &&
   BUF->ipid[0] == FBUF->ipid[0] &&
   BUF->ipid[1] == FBUF->ipid[1]) {

  len = (BUF->len[0] << 8) + BUF->len[1] - (BUF->vhl & 0x0f) * 4;
  offset = (((BUF->ipoffset[0] & 0x3f) << 8) + BUF->ipoffset[1]) * 8;

  /* If the offset or the offset + fragment length overflows the
     reassembly buffer, we discard the entire packet. */
  if(offset > UIP_REASS_BUFSIZE ||
     offset + len > UIP_REASS_BUFSIZE) {
    uip_reasstmr = 0;
    goto nullreturn;
  }

  /* Copy the fragment into the reassembly buffer, at the right
     offset. */
  memcpy(&uip_reassbuf[UIP_IPH_LEN + offset],
         (char *)BUF + (int)((BUF->vhl & 0x0f) * 4),
         len);

  /* Update the bitmap. */
  if(offset / (8 * 8) == (offset + len) / (8 * 8)) {
    /* If the two endpoints are in the same byte, we only update
       that byte. */

    uip_reassbitmap[offset / (8 * 8)] |=
          bitmap_bits[(offset / 8 ) & 7] &
          ~bitmap_bits[((offset + len) / 8 ) & 7];
  } else {
    /* If the two endpoints are in different bytes, we update the
       bytes in the endpoints and fill the stuff inbetween with
       0xff. */
    uip_reassbitmap[offset / (8 * 8)] |=
      bitmap_bits[(offset / 8 ) & 7];
    for(i = 1 + offset / (8 * 8); i < (offset + len) / (8 * 8); ++i) {
      uip_reassbitmap[i] = 0xff;
    }
    uip_reassbitmap[(offset + len) / (8 * 8)] |=
      ~bitmap_bits[((offset + len) / 8 ) & 7];
  }

  /* If this fragment has the More Fragments flag set to zero, we
     know that this is the last fragment, so we can calculate the
     size of the entire packet. We also set the
     IP_REASS_FLAG_LASTFRAG flag to indicate that we have received
     the final fragment. */

  if((BUF->ipoffset[0] & IP_MF) == 0) {
    uip_reassflags |= UIP_REASS_FLAG_LASTFRAG;
    uip_reasslen = offset + len;
  }

  /* Finally, we check if we have a full packet in the buffer. We do
     this by checking if we have the last fragment and if all bits
     in the bitmap are set. */
  if(uip_reassflags & UIP_REASS_FLAG_LASTFRAG) {
    /* Check all bytes up to and including all but the last byte in
       the bitmap. */
    for(i = 0; i < uip_reasslen / (8 * 8) - 1; ++i) {
      if(uip_reassbitmap[i] != 0xff) {
        goto nullreturn;
      }
    }
    /* Check the last byte in the bitmap. It should contain just the
       right amount of bits. */
    if(uip_reassbitmap[uip_reasslen / (8 * 8)] !=
       (u8_t)~bitmap_bits[uip_reasslen / 8 & 7]) {
      goto nullreturn;
    }

    /* If we have come this far, we have a full packet in the
       buffer, so we allocate a pbuf and copy the packet into it. We
```

```
        also reset the timer. */
      uip_reasstmr = 0;
      memcpy(BUF, FBUF, uip_reasslen);

      /* Pretend to be a "normal" (i.e., not fragmented) IP packet
        from now on. */
      BUF->ipoffset[0] = BUF->ipoffset[1] = 0;
      BUF->len[0] = uip_reasslen >> 8;
      BUF->len[1] = uip_reasslen & 0xff;
      BUF->ipchksum = 0;
      BUF->ipchksum = ~(uip_ipchksum());

      return uip_reasslen;
    }
  }

 nullreturn:
  return 0;
}
#endif /* UIP_REASSEMBLY */
/*---------------------------------------------------------------------------*/
static void
uip_add_rcv_nxt(u16_t n)
{
  uip_add32(uip_conn->rcv_nxt, n);
  uip_conn->rcv_nxt[0] = uip_acc32[0];
  uip_conn->rcv_nxt[1] = uip_acc32[1];
  uip_conn->rcv_nxt[2] = uip_acc32[2];
  uip_conn->rcv_nxt[3] = uip_acc32[3];
}
/*---------------------------------------------------------------------------*/
void definir_seq(u8_t *op32, u16_t op16)
{
  num_seq[3] = op32[3] + (op16 & 0xff);
  num_seq[2] = op32[2] + (op16 >> 8);
  num_seq[1] = op32[1];
  num_seq[0] = op32[0];

  if(num_seq[2] < (op16 >> 8)) {
    ++num_seq[1];
    if(num_seq[1] == 0) {
      ++num_seq[0];
    }
  }

  if(num_seq[3] < (op16 & 0xff)) {
    ++num_seq[2];
    if(num_seq[2] == 0) {
      ++num_seq[1];
      if(num_seq[1] == 0) {
       ++num_seq[0];
      }
    }
  }
}
/*---------------------------------------------------------------------------*/
void
uip_process(u8_t flag)
{
  register struct uip_conn *uip_connr = uip_conn;

z++;

if (enviando==1)
{
       if (k<MODULO+1)
       {
              uip_connr->len=0;
              uip_flags = UIP_ACKDATA;

              for(j=0;j<4096 * RAFAGA;j++)
              {
                     if ((j%1000)==0)
                     {
                            printf("%d...",j);
                     }
              }
```

```
                printf("\n");
        }
        else
        {
                enviando=0;
        }

        goto reconocemos;
}

if (ret==1)
{
        if (k<MODULO+1)
        {
                for(j=0;j<2048 * RAFAGA;j++)
                {
                        if ((j%1000)==0)
                        {
                                printf("%d...",j);
                        }
                }
                printf("\n");
                goto reconocemos;
        }
        else
        {
                ret=0;
                goto reconocemos;
        }
}


#if UIP_UDP
  if(flag == UIP_UDP_SEND_CONN) {
    goto udp_send;
  }
#endif /* UIP_UDP */

  uip_sappdata = uip_appdata = &uip_buf[UIP_IPTCPH_LEN + UIP_LLH_LEN];

  /* Check if we were invoked because of a poll request for a
     particular connection. */
  if(flag == UIP_POLL_REQUEST) {
    if((uip_connr->tcpstateflags & UIP_TS_MASK) == UIP_ESTABLISHED &&
       !uip_outstanding(uip_connr)) {
       uip_flags = UIP_POLL;
       UIP_APPCALL();
       goto appsend;
    }
    goto drop;

    /* Check if we were invoked because of the perodic timer fireing. */
  } else if(flag == UIP_TIMER) {
#if UIP_REASSEMBLY
    if(uip_reasstmr != 0) {
      --uip_reasstmr;
    }
#endif /* UIP_REASSEMBLY */
    /* Increase the initial sequence number. */
    if(++iss[3] == 0) {
      if(++iss[2] == 0) {
       if(++iss[1] == 0) {
         ++iss[0];
       }
      }
    }

    /* Reset the length variables. */
    uip_len = 0;
    uip_slen = 0;

    /* Check if the connection is in a state in which we simply wait
       for the connection to time out. If so, we increase the
       connection's timer and remove the connection if it times
       out. */
    if(uip_connr->tcpstateflags == UIP_TIME_WAIT ||
       uip_connr->tcpstateflags == UIP_FIN_WAIT_2) {
```

```
      ++(uip_connr->timer);
      if(uip_connr->timer == UIP_TIME_WAIT_TIMEOUT) {
       uip_connr->tcpstateflags = UIP_CLOSED;
      }
    } else if(uip_connr->tcpstateflags != UIP_CLOSED) {
      /* If the connection has outstanding data, we increase the
         connection's timer and see if it has reached the RTO value
         in which case we retransmit. */
      if(uip_outstanding(uip_connr)) {
       if(uip_connr->timer-- == 0) {
         if(uip_connr->nrtx == UIP_MAXRTX ||
            ((uip_connr->tcpstateflags == UIP_SYN_SENT ||
              uip_connr->tcpstateflags == UIP_SYN_RCVD) &&
             uip_connr->nrtx == UIP_MAXSYNRTX)) {
           uip_connr->tcpstateflags = UIP_CLOSED;

           /* We call UIP_APPCALL() with uip_flags set to
              UIP_TIMEDOUT to inform the application that the
              connection has timed out. */
           uip_flags = UIP_TIMEDOUT;
           UIP_APPCALL();

           /* We also send a reset packet to the remote host. */
           BUF->flags = TCP_RST | TCP_ACK;
           goto tcp_send_nodata;
         }

         /* Exponential backoff. */
         uip_connr->timer = UIP_RTO << (uip_connr->nrtx > 4?
                                        4:
                                        uip_connr->nrtx);
         ++(uip_connr->nrtx);

         /* Ok, so we need to retransmit. We do this differently
            depending on which state we are in. In ESTABLISHED, we
            call upon the application so that it may prepare the
            data for the retransmit. In SYN_RCVD, we resend the
            SYNACK that we sent earlier and in LAST_ACK we have to
            retransmit our FINACK. */
         UIP_STAT(++uip_stat.tcp.rexmit);
         switch(uip_connr->tcpstateflags & UIP_TS_MASK) {
         case UIP_SYN_RCVD:
           /* In the SYN_RCVD state, we should retransmit our
              SYNACK. */
           goto tcp_send_synack;

#if UIP_ACTIVE_OPEN
         case UIP_SYN_SENT:
           /* In the SYN_SENT state, we retransmit out SYN. */
           BUF->flags = 0;
           goto tcp_send_syn;
#endif /* UIP_ACTIVE_OPEN */

         case UIP_ESTABLISHED:
           /* In the ESTABLISHED state, we call upon the application
              to do the actual retransmit after which we jump into
              the code for sending out the packet (the apprexmit
              label). */
              /*ATENCION, Ponemos entre comentarios las 2 siguientes lineas*/
           /*uip_flags = UIP_REXMIT;
           UIP_APPCALL();*/
              ret=1;
              n_ret++;
              k=1;
           goto apprexmit;

         case UIP_FIN_WAIT_1:
         case UIP_CLOSING:
         case UIP_LAST_ACK:
           /* In all these states we should retransmit a FINACK. */
           goto tcp_send_finack;

         }
       }
      } else if((uip_connr->tcpstateflags & UIP_TS_MASK) == UIP_ESTABLISHED) {
       /* If there was no need for a retransmission, we poll the
          application for new data. */
```

```
        uip_flags = UIP_POLL;
        UIP_APPCALL();
        goto appsend;
       }
     }
     goto drop;
   }
#if UIP_UDP
   if(flag == UIP_UDP_TIMER) {
     if(uip_udp_conn->lport != 0) {
       uip_conn = NULL;
       uip_sappdata = uip_appdata = &uip_buf[UIP_LLH_LEN + UIP_IPUDPH_LEN];
       uip_len = uip_slen = 0;
       uip_flags = UIP_POLL;
       UIP_UDP_APPCALL();
       goto udp_send;
     } else {
       goto drop;
     }
   }
#endif

   /* This is where the input processing starts. */
   UIP_STAT(++uip_stat.ip.recv);

   /* Start of IP input header processing code. */

#if UIP_CONF_IPV6
   /* Check validity of the IP header. */
   if((BUF->vtc & 0xf0) != 0x60)  { /* IP version and header length. */
     UIP_STAT(++uip_stat.ip.drop);
     UIP_STAT(++uip_stat.ip.vhlerr);
     UIP_LOG("ipv6: invalid version.");
     goto drop;
   }
#else /* UIP_CONF_IPV6 */
   /* Check validity of the IP header. */
   if(BUF->vhl != 0x45)  { /* IP version and header length. */
     UIP_STAT(++uip_stat.ip.drop);
     UIP_STAT(++uip_stat.ip.vhlerr);
     UIP_LOG("ip: invalid version or header length.");
     goto drop;
   }
#endif /* UIP_CONF_IPV6 */

   /* Check the size of the packet. If the size reported to us in
      uip_len is smaller the size reported in the IP header, we assume
      that the packet has been corrupted in transit. If the size of
      uip_len is larger than the size reported in the IP packet header,
      the packet has been padded and we set uip_len to the correct
      value.. */

   if((BUF->len[0] << 8) + BUF->len[1] <= uip_len) {
     uip_len = (BUF->len[0] << 8) + BUF->len[1];
#if UIP_CONF_IPV6
     uip_len += 40; /* The length reported in the IPv6 header is the
                       length of the payload that follows the
                       header. However, uIP uses the uip_len variable
                       for holding the size of the entire packet,
                       including the IP header. For IPv4 this is not a
                       problem as the length field in the IPv4 header
                       contains the length of the entire packet. But
                       for IPv6 we need to add the size of the IPv6
                       header (40 bytes). */
#endif /* UIP_CONF_IPV6 */
   } else {
     UIP_LOG("ip: packet shorter than reported in IP header.");
     goto drop;
   }

#if !UIP_CONF_IPV6
   /* Check the fragment flag. */
   if((BUF->ipoffset[0] & 0x3f) != 0 ||
      BUF->ipoffset[1] != 0) {
#if UIP_REASSEMBLY
     uip_len = uip_reass();
     if(uip_len == 0) {
```

```
      goto drop;
    }
#else /* UIP_REASSEMBLY */
    UIP_STAT(++uip_stat.ip.drop);
    UIP_STAT(++uip_stat.ip.fragerr);
    UIP_LOG("ip: fragment dropped.");
    goto drop;
#endif /* UIP_REASSEMBLY */
  }
#endif /* UIP_CONF_IPV6 */

  if(uip_ipaddr_cmp(&uip_hostaddr, &uip_all_zeroes_addr)) {
    /* If we are configured to use ping IP address configuration and
       hasn't been assigned an IP address yet, we accept all ICMP
       packets. */
#if UIP_PINGADDRCONF && !UIP_CONF_IPV6
    if(BUF->proto == UIP_PROTO_ICMP) {
      UIP_LOG("ip: possible ping config packet received.");
      goto icmp_input;
    } else {
      UIP_LOG("ip: packet dropped since no address assigned.");
      goto drop;
    }
#endif /* UIP_PINGADDRCONF */

  } else {
    /* If IP broadcast support is configured, we check for a broadcast
       UDP packet, which may be destined to us. */
#if UIP_BROADCAST
    DEBUG_PRINTF("UDP IP checksum 0x%04x\n", uip_ipchksum());
    if(BUF->proto == UIP_PROTO_UDP &&
       uip_ipaddr_cmp(&BUF->destipaddr, &uip_broadcast_addr)
       /*&&
        uip_ipchksum() == 0xffff*/) {
      goto udp_input;
    }
#endif /* UIP_BROADCAST */

    /* Check if the packet is destined for our IP address. */
#if !UIP_CONF_IPV6
    if(!uip_ipaddr_cmp(&BUF->destipaddr, &uip_hostaddr)) {
      UIP_STAT(++uip_stat.ip.drop);
      goto drop;
    }
#else /* UIP_CONF_IPV6 */
    /* For IPv6, packet reception is a little trickier as we need to
       make sure that we listen to certain multicast addresses (all
       hosts multicast address, and the solicited-node multicast
       address) as well. However, we will cheat here and accept all
       multicast packets that are sent to the ff02::/16 addresses. */
    if(!uip_ipaddr_cmp(&BUF->destipaddr, &uip_hostaddr) &&
       BUF->destipaddr.u16[0] != HTONS(0xff02)) {
      UIP_STAT(++uip_stat.ip.drop);
      goto drop;
    }
#endif /* UIP_CONF_IPV6 */
  }

#if !UIP_CONF_IPV6
  if(uip_ipchksum() != 0xffff) { /* Compute and check the IP header
                                    checksum. */
    UIP_STAT(++uip_stat.ip.drop);
    UIP_STAT(++uip_stat.ip.chkerr);
    UIP_LOG("ip: bad checksum.");
    goto drop;
  }
#endif /* UIP_CONF_IPV6 */

  if(BUF->proto == UIP_PROTO_TCP) { /* Check for TCP packet. If so,
                                       proceed with TCP input
                                       processing. */
    goto tcp_input;
  }

#if UIP_UDP
  if(BUF->proto == UIP_PROTO_UDP) {
    goto udp_input;
```

```
  }
#endif /* UIP_UDP */

#if !UIP_CONF_IPV6
  /* ICMPv4 processing code follows. */
  if(BUF->proto != UIP_PROTO_ICMP) { /* We only allow ICMP packets from
                                        here. */
    UIP_STAT(++uip_stat.ip.drop);
    UIP_STAT(++uip_stat.ip.protoerr);
    UIP_LOG("ip: neither tcp nor icmp.");
    goto drop;
  }

#if UIP_PINGADDRCONF
 icmp_input:
#endif /* UIP_PINGADDRCONF */
  UIP_STAT(++uip_stat.icmp.recv);

  /* ICMP echo (i.e., ping) processing. This is simple, we only change
     the ICMP type from ECHO to ECHO_REPLY and adjust the ICMP
     checksum before we return the packet. */
  if(ICMPBUF->type != ICMP_ECHO) {
    UIP_STAT(++uip_stat.icmp.drop);
    UIP_STAT(++uip_stat.icmp.typeerr);
    UIP_LOG("icmp: not icmp echo.");
    goto drop;
  }

  /* If we are configured to use ping IP address assignment, we use
     the destination IP address of this ping packet and assign it to
     ourself. */
#if UIP_PINGADDRCONF
  if(uip_ipaddr_cmp(&uip_hostaddr, &uip_all_zeroes_addr)) {
    uip_hostaddr = BUF->destipaddr;
  }
#endif /* UIP_PINGADDRCONF */

  ICMPBUF->type = ICMP_ECHO_REPLY;

  if(ICMPBUF->icmpchksum >= HTONS(0xffff - (ICMP_ECHO << 8))) {
    ICMPBUF->icmpchksum += HTONS(ICMP_ECHO << 8) + 1;
  } else {
    ICMPBUF->icmpchksum += HTONS(ICMP_ECHO << 8);
  }

  /* Swap IP addresses. */
  uip_ipaddr_copy(&BUF->destipaddr, &BUF->srcipaddr);
  uip_ipaddr_copy(&BUF->srcipaddr, &uip_hostaddr);

  UIP_STAT(++uip_stat.icmp.sent);
  BUF->ttl = UIP_TTL;
  goto ip_send_nolen;

  /* End of IPv4 input header processing code. */
#else /* !UIP_CONF_IPV6 */

  /* This is IPv6 ICMPv6 processing code. */
  DEBUG_PRINTF("icmp6_input: length %d\n", uip_len);

  if(BUF->proto != UIP_PROTO_ICMP6) { /* We only allow ICMPv6 packets from
                                         here. */
    UIP_STAT(++uip_stat.ip.drop);
    UIP_STAT(++uip_stat.ip.protoerr);
    UIP_LOG("ip: neither tcp nor icmp6.");
    goto drop;
  }

  UIP_STAT(++uip_stat.icmp.recv);

  /* If we get a neighbor solicitation for our address we should send
     a neighbor advertisement message back. */
  if(ICMPBUF->type == ICMP6_NEIGHBOR_SOLICITATION) {
    if(uip_ipaddr_cmp(&ICMPBUF->icmp6data, &uip_hostaddr)) {

      if(ICMPBUF->options[0] == ICMP6_OPTION_SOURCE_LINK_ADDRESS) {
        /* Save the sender's address in our neighbor list. */
        uip_neighbor_add(&ICMPBUF->srcipaddr, &(ICMPBUF->options[2]));
```

```
      }

      /* We should now send a neighbor advertisement back to where the
        neighbor solicication came from. */
      ICMPBUF->type = ICMP6_NEIGHBOR_ADVERTISEMENT;
      ICMPBUF->flags = ICMP6_FLAG_S; /* Solicited flag. */

      ICMPBUF->reserved1 = ICMPBUF->reserved2 = ICMPBUF->reserved3 = 0;

      uip_ipaddr_copy(&ICMPBUF->destipaddr, &ICMPBUF->srcipaddr);
      uip_ipaddr_copy(&ICMPBUF->srcipaddr, &uip_hostaddr);
      ICMPBUF->options[0] = ICMP6_OPTION_TARGET_LINK_ADDRESS;
      ICMPBUF->options[1] = 1;  /* Options length, 1 = 8 bytes. */
      memcpy(&(ICMPBUF->options[2]), &uip_ethaddr, sizeof(uip_ethaddr));
      ICMPBUF->icmpchksum = 0;
      ICMPBUF->icmpchksum = ~uip_icmp6chksum();

      goto send;

    }
    goto drop;
  } else if(ICMPBUF->type == ICMP6_ECHO) {
    /* ICMP echo (i.e., ping) processing. This is simple, we only
       change the ICMP type from ECHO to ECHO_REPLY and update the
       ICMP checksum before we return the packet. */

    ICMPBUF->type = ICMP6_ECHO_REPLY;

    uip_ipaddr_copy(&BUF->destipaddr, &BUF->srcipaddr);
    uip_ipaddr_copy(&BUF->srcipaddr, &uip_hostaddr);
    ICMPBUF->icmpchksum = 0;
    ICMPBUF->icmpchksum = ~uip_icmp6chksum();

    UIP_STAT(++uip_stat.icmp.sent);
    goto send;
  } else {
    DEBUG_PRINTF("Unknown icmp6 message type %d\n", ICMPBUF->type);
    UIP_STAT(++uip_stat.icmp.drop);
    UIP_STAT(++uip_stat.icmp.typeerr);
    UIP_LOG("icmp: unknown ICMP message.");
    goto drop;
  }

  /* End of IPv6 ICMP processing. */

#endif /* !UIP_CONF_IPV6 */

#if UIP_UDP
  /* UDP input processing. */
 udp_input:
  /* UDP processing is really just a hack. We don't do anything to the
     UDP/IP headers, but let the UDP application do all the hard
     work. If the application sets uip_slen, it has a packet to
     send. */
#if UIP_UDP_CHECKSUMS
  uip_len = uip_len - UIP_IPUDPH_LEN;
  uip_appdata = &uip_buf[UIP_LLH_LEN + UIP_IPUDPH_LEN];
  if(UDPBUF->udpchksum != 0 && uip_udpchksum() != 0xffff) {
    UIP_STAT(++uip_stat.udp.drop);
    UIP_STAT(++uip_stat.udp.chkerr);
    UIP_LOG("udp: bad checksum.");
    goto drop;
  }
#else /* UIP_UDP_CHECKSUMS */
  uip_len = uip_len - UIP_IPUDPH_LEN;
#endif /* UIP_UDP_CHECKSUMS */

  /* Demultiplex this UDP packet between the UDP "connections". */
  for(uip_udp_conn = &uip_udp_conns[0];
      uip_udp_conn < &uip_udp_conns[UIP_UDP_CONNS];
      ++uip_udp_conn) {
    /* If the local UDP port is non-zero, the connection is considered
       to be used. If so, the local port number is checked against the
       destination port number in the received packet. If the two port
       numbers match, the remote port number is checked if the
       connection is bound to a remote port. Finally, if the
       connection is bound to a remote IP address, the source IP
```

```
       address of the packet is checked. */
    if(uip_udp_conn->lport != 0 &&
       UDPBUF->destport == uip_udp_conn->lport &&
       (uip_udp_conn->rport == 0 ||
        UDPBUF->srcport == uip_udp_conn->rport) &&
       (uip_ipaddr_cmp(&uip_udp_conn->ripaddr, &uip_all_zeroes_addr) ||
       uip_ipaddr_cmp(&uip_udp_conn->ripaddr, &uip_broadcast_addr) ||
       uip_ipaddr_cmp(&BUF->srcipaddr, &uip_udp_conn->ripaddr))) {
      goto udp_found;
    }
  }
  UIP_LOG("udp: no matching connection found");
#if UIP_CONF_ICMP_DEST_UNREACH && !UIP_CONF_IPV6
  /* Copy fields from packet header into payload of this ICMP packet. */
  memcpy(&(ICMPBUF->payload[0]), ICMPBUF, UIP_IPH_LEN + 8);

  /* Set the ICMP type and code. */
  ICMPBUF->type = ICMP_DEST_UNREACHABLE;
  ICMPBUF->icode = ICMP_PORT_UNREACHABLE;

  /* Calculate the ICMP checksum. */
  ICMPBUF->icmpchksum = 0;
  ICMPBUF->icmpchksum = ~uip_chksum((u16_t *)&(ICMPBUF->type), 36);

  /* Set the IP destination address to be the source address of the
     original packet. */
  uip_ipaddr_copy(&BUF->destipaddr, &BUF->srcipaddr);

  /* Set our IP address as the source address. */
  uip_ipaddr_copy(&BUF->srcipaddr, &uip_hostaddr);

  /* The size of the ICMP destination unreachable packet is 36 + the
     size of the IP header (20) = 56. */
  uip_len = 36 + UIP_IPH_LEN;
  ICMPBUF->len[0] = 0;
  ICMPBUF->len[1] = (u8_t)uip_len;
  ICMPBUF->ttl = UIP_TTL;
  ICMPBUF->proto = UIP_PROTO_ICMP;

  goto ip_send_nolen;
#else /* UIP_CONF_ICMP_DEST_UNREACH */
  goto drop;
#endif /* UIP_CONF_ICMP_DEST_UNREACH */

 udp_found:
  uip_conn = NULL;
  uip_flags = UIP_NEWDATA;
  uip_sappdata = uip_appdata = &uip_buf[UIP_LLH_LEN + UIP_IPUDPH_LEN];
  uip_slen = 0;
  UIP_UDP_APPCALL();

 udp_send:
  if(uip_slen == 0) {
    goto drop;
  }
  uip_len = uip_slen + UIP_IPUDPH_LEN;

#if UIP_CONF_IPV6
  /* For IPv6, the IP length field does not include the IPv6 IP header
     length. */
  BUF->len[0] = ((uip_len - UIP_IPH_LEN) >> 8);
  BUF->len[1] = ((uip_len - UIP_IPH_LEN) & 0xff);
#else /* UIP_CONF_IPV6 */
  BUF->len[0] = (uip_len >> 8);
  BUF->len[1] = (uip_len & 0xff);
#endif /* UIP_CONF_IPV6 */

  BUF->ttl = uip_udp_conn->ttl;
  BUF->proto = UIP_PROTO_UDP;

  UDPBUF->udplen = HTONS(uip_slen + UIP_UDPH_LEN);
  UDPBUF->udpchksum = 0;

  BUF->srcport  = uip_udp_conn->lport;
  BUF->destport = uip_udp_conn->rport;

  uip_ipaddr_copy(&BUF->srcipaddr, &uip_hostaddr);
```

```
  uip_ipaddr_copy(&BUF->destipaddr, &uip_udp_conn->ripaddr);

  uip_appdata = &uip_buf[UIP_LLH_LEN + UIP_IPTCPH_LEN];

#if UIP_UDP_CHECKSUMS
  /* Calculate UDP checksum. */
  UDPBUF->udpchksum = ~(uip_udpchksum());
  if(UDPBUF->udpchksum == 0) {
    UDPBUF->udpchksum = 0xffff;
  }
#endif /* UIP_UDP_CHECKSUMS */

  goto ip_send_nolen;
#endif /* UIP_UDP */

  /* TCP input processing. */
 tcp_input:
  UIP_STAT(++uip_stat.tcp.recv);

  /* Start of TCP input header processing code. */

  if(uip_tcpchksum() != 0xffff) {   /* Compute and check the TCP
                                       checksum. */
    UIP_STAT(++uip_stat.tcp.drop);
    UIP_STAT(++uip_stat.tcp.chkerr);
    UIP_LOG("tcp: bad checksum.");
    goto drop;
  }

  /* Demultiplex this segment. */
  /* First check any active connections. */
  for(uip_connr = &uip_conns[0]; uip_connr <= &uip_conns[UIP_CONNS - 1];
      ++uip_connr) {
    if(uip_connr->tcpstateflags != UIP_CLOSED &&
       BUF->destport == uip_connr->lport &&
       BUF->srcport == uip_connr->rport &&
       uip_ipaddr_cmp(&BUF->srcipaddr, &uip_connr->ripaddr)) {
      goto found;
    }
  }

  /* If we didn't find and active connection that expected the packet,
     either this packet is an old duplicate, or this is a SYN packet
     destined for a connection in LISTEN. If the SYN flag isn't set,
     it is an old packet and we send a RST. */
  if((BUF->flags & TCP_CTL) != TCP_SYN) {
    goto reset;
  }

  tmp16 = BUF->destport;
  /* Next, check listening connections. */
  for(c = 0; c < UIP_LISTENPORTS; ++c) {
    if(tmp16 == uip_listenports[c]) {
      goto found_listen;
    }
  }

  /* No matching connection found, so we send a RST packet. */
  UIP_STAT(++uip_stat.tcp.synrst);

 reset:
  /* We do not send resets in response to resets. */
  if(BUF->flags & TCP_RST) {
    goto drop;
  }

  UIP_STAT(++uip_stat.tcp.rst);

  BUF->flags = TCP_RST | TCP_ACK;
  uip_len = UIP_IPTCPH_LEN;
  BUF->tcpoffset = 5 << 4;

  /* Flip the seqno and ackno fields in the TCP header. */
  c = BUF->seqno[3];
  BUF->seqno[3] = BUF->ackno[3];
  BUF->ackno[3] = c;
```

```
  c = BUF->seqno[2];
  BUF->seqno[2] = BUF->ackno[2];
  BUF->ackno[2] = c;

  c = BUF->seqno[1];
  BUF->seqno[1] = BUF->ackno[1];
  BUF->ackno[1] = c;

  c = BUF->seqno[0];
  BUF->seqno[0] = BUF->ackno[0];
  BUF->ackno[0] = c;

  /* We also have to increase the sequence number we are
     acknowledging. If the least significant byte overflowed, we need
     to propagate the carry to the other bytes as well. */
  if(++BUF->ackno[3] == 0) {
    if(++BUF->ackno[2] == 0) {
      if(++BUF->ackno[1] == 0) {
        ++BUF->ackno[0];
      }
    }
  }

  /* Swap port numbers. */
  tmp16 = BUF->srcport;
  BUF->srcport = BUF->destport;
  BUF->destport = tmp16;

  /* Swap IP addresses. */
  uip_ipaddr_copy(&BUF->destipaddr, &BUF->srcipaddr);
  uip_ipaddr_copy(&BUF->srcipaddr, &uip_hostaddr);

  /* And send out the RST packet! */
  goto tcp_send_noconn;

  /* This label will be jumped to if we matched the incoming packet
     with a connection in LISTEN. In that case, we should create a new
     connection and send a SYNACK in return. */
 found_listen:
  /* First we check if there are any connections avaliable. Unused
     connections are kept in the same table as used connections, but
     unused ones have the tcpstate set to CLOSED. Also, connections in
     TIME_WAIT are kept track of and we'll use the oldest one if no
     CLOSED connections are found. Thanks to Eddie C. Dost for a very
     nice algorithm for the TIME_WAIT search. */
  uip_connr = 0;
  for(c = 0; c < UIP_CONNS; ++c) {
    if(uip_conns[c].tcpstateflags == UIP_CLOSED) {
      uip_connr = &uip_conns[c];
      break;
    }
    if(uip_conns[c].tcpstateflags == UIP_TIME_WAIT) {
      if(uip_connr == 0 ||
         uip_conns[c].timer > uip_connr->timer) {
        uip_connr = &uip_conns[c];
      }
    }
  }

  if(uip_connr == 0) {
    /* All connections are used already, we drop packet and hope that
       the remote end will retransmit the packet at a time when we
       have more spare connections. */
    UIP_STAT(++uip_stat.tcp.syndrop);
    UIP_LOG("tcp: found no unused connections.");
    goto drop;
  }
  uip_conn = uip_connr;

  /* Fill in the necessary fields for the new connection. */
  uip_connr->rto = uip_connr->timer = UIP_RTO;
  uip_connr->sa = 0;
  uip_connr->sv = 4;
  uip_connr->nrtx = 0;
  uip_connr->lport = BUF->destport;
  uip_connr->rport = BUF->srcport;
  uip_ipaddr_copy(&uip_connr->ripaddr, &BUF->srcipaddr);
```

```
  uip_connr->tcpstateflags = UIP_SYN_RCVD;

  uip_connr->snd_nxt[0] = iss[0];
  uip_connr->snd_nxt[1] = iss[1];
  uip_connr->snd_nxt[2] = iss[2];
  uip_connr->snd_nxt[3] = iss[3];
  uip_connr->len = 1;

  /* rcv_nxt should be the seqno from the incoming packet + 1. */
  uip_connr->rcv_nxt[3] = BUF->seqno[3];
  uip_connr->rcv_nxt[2] = BUF->seqno[2];
  uip_connr->rcv_nxt[1] = BUF->seqno[1];
  uip_connr->rcv_nxt[0] = BUF->seqno[0];
  uip_add_rcv_nxt(1);

  /* Parse the TCP MSS option, if present. */
  if((BUF->tcpoffset & 0xf0) > 0x50) {
    for(c = 0; c < ((BUF->tcpoffset >> 4) - 5) << 2 ;) {
      opt = uip_buf[UIP_TCPIP_HLEN + UIP_LLH_LEN + c];
      if(opt == TCP_OPT_END) {
       /* End of options. */
       break;
      } else if(opt == TCP_OPT_NOOP) {
       ++c;
       /* NOP option. */
      } else if(opt == TCP_OPT_MSS &&
              uip_buf[UIP_TCPIP_HLEN + UIP_LLH_LEN + 1 + c] == TCP_OPT_MSS_LEN) {
       /* An MSS option with the right option length. */
       tmp16 = ((u16_t)uip_buf[UIP_TCPIP_HLEN + UIP_LLH_LEN + 2 + c] << 8) |
         (u16_t)uip_buf[UIP_IPTCPH_LEN + UIP_LLH_LEN + 3 + c];
       uip_connr->initialmss = uip_connr->mss =
         tmp16 > UIP_TCP_MSS? UIP_TCP_MSS: tmp16;

       /* And we are done processing options. */
       break;
      } else {
       /* All other options have a length field, so that we easily
          can skip past them. */
       if(uip_buf[UIP_TCPIP_HLEN + UIP_LLH_LEN + 1 + c] == 0) {
         /* If the length field is zero, the options are malformed
            and we don't process them further. */
         break;
       }
       c += uip_buf[UIP_TCPIP_HLEN + UIP_LLH_LEN + 1 + c];
      }
    }
  }

  /* Our response will be a SYNACK. */
#if UIP_ACTIVE_OPEN
 tcp_send_synack:
  BUF->flags = TCP_ACK;

 tcp_send_syn:
  BUF->flags |= TCP_SYN;
#else /* UIP_ACTIVE_OPEN */
 tcp_send_synack:
  BUF->flags = TCP_SYN | TCP_ACK;
#endif /* UIP_ACTIVE_OPEN */

  /* We send out the TCP Maximum Segment Size option with our
     SYNACK. */
  BUF->optdata[0] = TCP_OPT_MSS;
  BUF->optdata[1] = TCP_OPT_MSS_LEN;
  BUF->optdata[2] = (UIP_TCP_MSS) / 256;
  BUF->optdata[3] = (UIP_TCP_MSS) & 255;
  uip_len = UIP_IPTCPH_LEN + TCP_OPT_MSS_LEN;
  BUF->tcpoffset = ((UIP_TCPH_LEN + TCP_OPT_MSS_LEN) / 4) << 4;
  goto tcp_send;

  /* This label will be jumped to if we found an active connection. */
 found:
  uip_conn = uip_connr;
  uip_flags = 0;
  /* We do a very naive form of TCP reset processing; we just accept
     any RST and kill our connection. We should in fact check if the
     sequence number of this reset is wihtin our advertised window
```

```
   before we accept the reset. */
if(BUF->flags & TCP_RST) {
  uip_connr->tcpstateflags = UIP_CLOSED;
  UIP_LOG("tcp: got reset, aborting connection.");
  uip_flags = UIP_ABORT;
  UIP_APPCALL();
  goto drop;
}
/* Calculate the length of the data, if the application has sent
   any data to us. */
c = (BUF->tcpoffset >> 4) << 2;
/* uip_len will contain the length of the actual TCP data. This is
   calculated by subtracing the length of the TCP header (in
   c) and the length of the IP header (20 bytes). */
uip_len = uip_len - c - UIP_IPH_LEN;

/* First, check if the sequence number of the incoming packet is
   what we're expecting next. If not, we send out an ACK with the
   correct numbers in. */
if(!(((uip_connr->tcpstateflags & UIP_TS_MASK) == UIP_SYN_SENT) &&
     ((BUF->flags & TCP_CTL) == (TCP_SYN | TCP_ACK)))) {
  if((uip_len > 0 || ((BUF->flags & (TCP_SYN | TCP_FIN)) != 0)) &&
     (BUF->seqno[0] != uip_connr->rcv_nxt[0] ||
     BUF->seqno[1] != uip_connr->rcv_nxt[1] ||
     BUF->seqno[2] != uip_connr->rcv_nxt[2] ||
     BUF->seqno[3] != uip_connr->rcv_nxt[3])) {
    goto tcp_send_ack;
  }
}

/* Next, check if the incoming segment acknowledges any outstanding
   data. If so, we update the sequence number, reset the length of
   the outstanding data, calculate RTT estimations, and reset the
   retransmission timer. */
if((BUF->flags & TCP_ACK) && uip_outstanding(uip_connr)) {
  uip_add32(uip_connr->snd_nxt, uip_connr->len);

  if(BUF->ackno[0] == uip_acc32[0] &&
     BUF->ackno[1] == uip_acc32[1] &&
     BUF->ackno[2] == uip_acc32[2] &&
     BUF->ackno[3] == uip_acc32[3]) {
    /* Update sequence number. */
    uip_connr->snd_nxt[0] = uip_acc32[0];
    uip_connr->snd_nxt[1] = uip_acc32[1];
    uip_connr->snd_nxt[2] = uip_acc32[2];
    uip_connr->snd_nxt[3] = uip_acc32[3];

    /* Do RTT estimation, unless we have done retransmissions. */
    if(uip_connr->nrtx == 0) {
      signed char m;
      m = uip_connr->rto - uip_connr->timer;
      /* This is taken directly from VJs original code in his paper */
      m = m - (uip_connr->sa >> 3);
      uip_connr->sa += m;
      if(m < 0) {
        m = -m;
      }
      m = m - (uip_connr->sv >> 2);
      uip_connr->sv += m;
      uip_connr->rto = (uip_connr->sa >> 3) + uip_connr->sv;

    }
    /* Set the acknowledged flag. */
    uip_flags = UIP_ACKDATA;
    /* Reset the retransmission timer. */
    uip_connr->timer = uip_connr->rto;

    /* Reset length of outstanding data. */
    uip_connr->len = 0;
  }

}

/* Do different things depending on in what state the connection is. */
switch(uip_connr->tcpstateflags & UIP_TS_MASK) {
  /* CLOSED and LISTEN are not handled here. CLOSE_WAIT is not
     implemented, since we force the application to close when the
```

```
       peer sends a FIN (hence the application goes directly from
       ESTABLISHED to LAST_ACK). */
  case UIP_SYN_RCVD:
    /* In SYN_RCVD we have sent out a SYNACK in response to a SYN, and
       we are waiting for an ACK that acknowledges the data we sent
       out the last time. Therefore, we want to have the UIP_ACKDATA
       flag set. If so, we enter the ESTABLISHED state. */
    if(uip_flags & UIP_ACKDATA) {
      uip_connr->tcpstateflags = UIP_ESTABLISHED;
      uip_flags = UIP_CONNECTED;
      uip_connr->len = 0;
      if(uip_len > 0) {
        uip_flags |= UIP_NEWDATA;
        uip_add_rcv_nxt(uip_len);
      }
      uip_slen = 0;
      UIP_APPCALL();
      goto appsend;
    }
    goto drop;
#if UIP_ACTIVE_OPEN
  case UIP_SYN_SENT:
    /* In SYN_SENT, we wait for a SYNACK that is sent in response to
       our SYN. The rcv_nxt is set to sequence number in the SYNACK
       plus one, and we send an ACK. We move into the ESTABLISHED
       state. */
    if((uip_flags & UIP_ACKDATA) &&
       (BUF->flags & TCP_CTL) == (TCP_SYN | TCP_ACK)) {
      /* Parse the TCP MSS option, if present. */
      if((BUF->tcpoffset & 0xf0) > 0x50) {
        for(c = 0; c < ((BUF->tcpoffset >> 4) - 5) << 2 ;) {
          opt = uip_buf[UIP_IPTCPH_LEN + UIP_LLH_LEN + c];
          if(opt == TCP_OPT_END) {
            /* End of options. */
            break;
          } else if(opt == TCP_OPT_NOOP) {
            ++c;
            /* NOP option. */
          } else if(opt == TCP_OPT_MSS &&
                    uip_buf[UIP_TCPIP_HLEN + UIP_LLH_LEN + 1 + c] == TCP_OPT_MSS_LEN) {
            /* An MSS option with the right option length. */
            tmp16 = (uip_buf[UIP_TCPIP_HLEN + UIP_LLH_LEN + 2 + c] << 8) |
              uip_buf[UIP_TCPIP_HLEN + UIP_LLH_LEN + 3 + c];
            uip_connr->initialmss =
              uip_connr->mss = tmp16 > UIP_TCP_MSS? UIP_TCP_MSS: tmp16;
            /* And we are done processing options. */
            break;
          } else {
            /* All other options have a length field, so that we easily
               can skip past them. */
            if(uip_buf[UIP_TCPIP_HLEN + UIP_LLH_LEN + 1 + c] == 0) {
              /* If the length field is zero, the options are malformed
                 and we don't process them further. */
              break;
            }
            c += uip_buf[UIP_TCPIP_HLEN + UIP_LLH_LEN + 1 + c];
          }
        }
      }
      uip_connr->tcpstateflags = UIP_ESTABLISHED;
      uip_connr->rcv_nxt[0] = BUF->seqno[0];
      uip_connr->rcv_nxt[1] = BUF->seqno[1];
      uip_connr->rcv_nxt[2] = BUF->seqno[2];
      uip_connr->rcv_nxt[3] = BUF->seqno[3];
      uip_add_rcv_nxt(1);
      uip_flags = UIP_CONNECTED | UIP_NEWDATA;
      uip_connr->len = 0;
      uip_len = 0;
      uip_slen = 0;
      UIP_APPCALL();
      goto appsend;
    }
    /* Inform the application that the connection failed */
    uip_flags = UIP_ABORT;
    UIP_APPCALL();
    /* The connection is closed after we send the RST */
    uip_conn->tcpstateflags = UIP_CLOSED;
```

```
    goto reset;
#endif /* UIP_ACTIVE_OPEN */

  case UIP_ESTABLISHED:
    /* In the ESTABLISHED state, we call upon the application to feed
    data into the uip_buf. If the UIP_ACKDATA flag is set, the
    application should put new data into the buffer, otherwise we are
    retransmitting an old segment, and the application should put that
    data into the buffer.

    If the incoming packet is a FIN, we should close the connection on
    this side as well, and we send out a FIN and enter the LAST_ACK
    state. We require that there is no outstanding data; otherwise the
    sequence numbers will be screwed up. */

    if(BUF->flags & TCP_FIN && !(uip_connr->tcpstateflags & UIP_STOPPED)) {
      if(uip_outstanding(uip_connr)) {
       goto drop;
      }
      uip_add_rcv_nxt(1 + uip_len);
      uip_flags |= UIP_CLOSE;
      if(uip_len > 0) {
       uip_flags |= UIP_NEWDATA;
      }
      UIP_APPCALL();
      uip_connr->len = 1;
      uip_connr->tcpstateflags = UIP_LAST_ACK;
      uip_connr->nrtx = 0;
    tcp_send_finack:
      BUF->flags = TCP_FIN | TCP_ACK;
      goto tcp_send_nodata;
    }

    /* Check the URG flag. If this is set, the segment carries urgent
       data that we must pass to the application. */
    if((BUF->flags & TCP_URG) != 0) {
#if UIP_URGDATA > 0
      uip_urglen = (BUF->urgp[0] << 8) | BUF->urgp[1];
      if(uip_urglen > uip_len) {
       /* There is more urgent data in the next segment to come. */
       uip_urglen = uip_len;
      }
      uip_add_rcv_nxt(uip_urglen);
      uip_len -= uip_urglen;
      uip_urgdata = uip_appdata;
      uip_appdata += uip_urglen;
    } else {
      uip_urglen = 0;
#else /* UIP_URGDATA > 0 */
      uip_appdata = ((char *)uip_appdata) + ((BUF->urgp[0] << 8) | BUF->urgp[1]);
      uip_len -= (BUF->urgp[0] << 8) | BUF->urgp[1];
#endif /* UIP_URGDATA > 0 */
    }

    /* If uip_len > 0 we have TCP data in the packet, and we flag this
       by setting the UIP_NEWDATA flag and update the sequence number
       we acknowledge. If the application has stopped the dataflow
       using uip_stop(), we must not accept any data packets from the
       remote host. */
    if(uip_len > 0 && !(uip_connr->tcpstateflags & UIP_STOPPED)) {
      uip_flags |= UIP_NEWDATA;
      uip_add_rcv_nxt(uip_len);
    }

    /* Check if the available buffer space advertised by the other end
       is smaller than the initial MSS for this connection. If so, we
       set the current MSS to the window size to ensure that the
       application does not send more data than the other end can
       handle.

       If the remote host advertises a zero window, we set the MSS to
       the initial MSS so that the application will send an entire MSS
       of data. This data will not be acknowledged by the receiver,
       and the application will retransmit it. This is called the
       "persistent timer" and uses the retransmission mechanim.
    */
    tmp16 = ((u16_t)BUF->wnd[0] << 8) + (u16_t)BUF->wnd[1];
```

```
if(tmp16 > uip_connr->initialmss ||
   tmp16 == 0) {
  tmp16 = uip_connr->initialmss;
}
uip_connr->mss = tmp16;

/* If this packet constitutes an ACK for outstanding data (flagged
   by the UIP_ACKDATA flag, we should call the application since it
   might want to send more data. If the incoming packet had data
   from the peer (as flagged by the UIP_NEWDATA flag), the
   application must also be notified.

   When the application is called, the global variable uip_len
   contains the length of the incoming data. The application can
   access the incoming data through the global pointer
   uip_appdata, which usually points UIP_IPTCPH_LEN + UIP_LLH_LEN
   bytes into the uip_buf array.

   If the application wishes to send any data, this data should be
   put into the uip_appdata and the length of the data should be
   put into uip_len. If the application don't have any data to
   send, uip_len must be set to 0. */

reconocemos:

if(uip_flags & (UIP_NEWDATA | UIP_ACKDATA)) {
   uip_slen = 0;
  UIP_APPCALL();
   enviando=1;

appsend:
  if(uip_flags & UIP_ABORT) {
   uip_slen = 0;
   uip_connr->tcpstateflags = UIP_CLOSED;
   BUF->flags = TCP_RST | TCP_ACK;
   goto tcp_send_nodata;
  }

  if(uip_flags & UIP_CLOSE) {
   uip_slen = 0;
   uip_connr->len = 1;
   uip_connr->tcpstateflags = UIP_FIN_WAIT_1;
   uip_connr->nrtx = 0;
   BUF->flags = TCP_FIN | TCP_ACK;
   goto tcp_send_nodata;
  }

  /* If uip_slen > 0, the application has data to be sent. */
  if(uip_slen > 0) {
   /* If the connection has acknowledged data, the contents of
      the ->len variable should be discarded. */
   if((uip_flags & UIP_ACKDATA) != 0) {
     uip_connr->len = 0;
   }

   /* If the ->len variable is non-zero the connection has
      already data in transit and cannot send anymore right
      now. */
   if(uip_connr->len == 0) {
     /* The application cannot send more than what is allowed by
        the mss (the minumum of the MSS and the available
        window). */
     if(uip_slen > uip_connr->mss) {
       uip_slen = uip_connr->mss;
     }

     /* Remember how much data we send out now so that we know
        when everything has been acknowledged. */
     uip_connr->len = uip_slen;
   } else {
     /* If the application already had unacknowledged data, we
        make sure that the application does not send (i.e.,
        retransmit) out more than it previously sent out. */
     uip_slen = uip_connr->len;
   }
  }
  uip_connr->nrtx = 0;
```

```
  apprexmit:

      if(ret==1)
      {
             goto enviar_ret;
      }

    uip_appdata = uip_sappdata;

    /* If the application has data to be sent, or if the incoming
       packet had new data in it, we must send out a packet. */
    if(uip_slen > 0 && uip_connr->len > 0) {
      /* Add the length of the IP and TCP headers. */
      uip_len = uip_connr->len + UIP_TCPIP_HLEN;
      /* We always set the ACK flag in response packets. */
      BUF->flags = TCP_ACK | TCP_PSH;
      /* Send the packet. */
      goto tcp_send_noopts;
    }
    /* If there is no data to send, just send out a pure ACK if
       there is newdata. */
    if(uip_flags & UIP_NEWDATA) {
      uip_len = UIP_TCPIP_HLEN;
      BUF->flags = TCP_ACK;
      enviando=1;
      goto tcp_send_noopts;
    }
  }
      if (ret==1)
      {
             goto apprexmit;
      }
  goto drop;
case UIP_LAST_ACK:
  /* We can close this connection if the peer has acknowledged our
     FIN. This is indicated by the UIP_ACKDATA flag. */
  if(uip_flags & UIP_ACKDATA) {
    uip_connr->tcpstateflags = UIP_CLOSED;
    uip_flags = UIP_CLOSE;
    UIP_APPCALL();
  }
  break;

case UIP_FIN_WAIT_1:
  /* The application has closed the connection, but the remote host
     hasn't closed its end yet. Thus we do nothing but wait for a
     FIN from the other side. */
  if(uip_len > 0) {
    uip_add_rcv_nxt(uip_len);
  }
  if(BUF->flags & TCP_FIN) {
    if(uip_flags & UIP_ACKDATA) {
      uip_connr->tcpstateflags = UIP_TIME_WAIT;
      uip_connr->timer = 0;
      uip_connr->len = 0;
    } else {
      uip_connr->tcpstateflags = UIP_CLOSING;
    }
    uip_add_rcv_nxt(1);
    uip_flags = UIP_CLOSE;
    UIP_APPCALL();
    goto tcp_send_ack;
  } else if(uip_flags & UIP_ACKDATA) {
    uip_connr->tcpstateflags = UIP_FIN_WAIT_2;
    uip_connr->len = 0;
    goto drop;
  }
  if(uip_len > 0) {
    goto tcp_send_ack;
  }
  goto drop;

case UIP_FIN_WAIT_2:
  if(uip_len > 0) {
    uip_add_rcv_nxt(uip_len);
  }
  if(BUF->flags & TCP_FIN) {
```

```
      uip_connr->tcpstateflags = UIP_TIME_WAIT;
      uip_connr->timer = 0;
      uip_add_rcv_nxt(1);
      uip_flags = UIP_CLOSE;
      UIP_APPCALL();
      goto tcp_send_ack;
    }
    if(uip_len > 0) {
      goto tcp_send_ack;
    }
    goto drop;

  case UIP_TIME_WAIT:
    goto tcp_send_ack;

  case UIP_CLOSING:
    if(uip_flags & UIP_ACKDATA) {
      uip_connr->tcpstateflags = UIP_TIME_WAIT;
      uip_connr->timer = 0;
    }
  }
  goto drop;

  /* We jump here when we are ready to send the packet, and just want
     to set the appropriate TCP sequence numbers in the TCP header. */
 tcp_send_ack:
  BUF->flags = TCP_ACK;

 tcp_send_nodata:
  uip_len = UIP_IPTCPH_LEN;

 tcp_send_noopts:
  BUF->tcpoffset = (UIP_TCPH_LEN / 4) << 4;

  /* We're done with the input processing. We are now ready to send a
     reply. Our job is to fill in all the fields of the TCP and IP
     headers before calculating the checksum and finally send the
     packet. */
 tcp_send:

if ((enviando==1)&&(comp_seq(uip_connr->snd_nxt,inicial)!=0)&&(BUF->flags!=1))
{
       definir_seq(num_seq,70); /*Se entiende 70 como mss*/
       uip_connr->snd_nxt[0] = num_seq[0];
       uip_connr->snd_nxt[1] = num_seq[1];
       uip_connr->snd_nxt[2] = num_seq[2];
       uip_connr->snd_nxt[3] = num_seq[3];
}

if ((k==2) && (comp_seq(uip_connr->snd_nxt,inicial)==0))
{
       definir_seq(num_seq,70); /*Se entiende 70 como mss*/
       uip_connr->snd_nxt[0] = num_seq[0];
       uip_connr->snd_nxt[1] = num_seq[1];
       uip_connr->snd_nxt[2] = num_seq[2];
       uip_connr->snd_nxt[3] = num_seq[3];
}

if ((enviando==1) && (k==1) && (comp_seq(uip_connr->snd_nxt,inicial)==0))
{
       for(j=0;j<4096 * RAFAGA;j++)
       {
              if ((j%1000)==0)
              {
                     printf("%d...",j);
              }
       }
}

  BUF->ackno[0] = uip_connr->rcv_nxt[0];
  BUF->ackno[1] = uip_connr->rcv_nxt[1];
  BUF->ackno[2] = uip_connr->rcv_nxt[2];
  BUF->ackno[3] = uip_connr->rcv_nxt[3];

  BUF->seqno[0] = uip_connr->snd_nxt[0];
  BUF->seqno[1] = uip_connr->snd_nxt[1];
  BUF->seqno[2] = uip_connr->snd_nxt[2];
```

```
  BUF->seqno[3] = uip_connr->snd_nxt[3];

  BUF->proto = UIP_PROTO_TCP;

  BUF->srcport  = uip_connr->lport;
  BUF->destport = uip_connr->rport;

  uip_ipaddr_copy(&BUF->srcipaddr, &uip_hostaddr);
  uip_ipaddr_copy(&BUF->destipaddr, &uip_connr->ripaddr);

  if(uip_connr->tcpstateflags & UIP_STOPPED) {
    /* If the connection has issued uip_stop(), we advertise a zero
       window so that the remote host will stop sending data. */
    BUF->wnd[0] = BUF->wnd[1] = 0;
  } else {
    BUF->wnd[0] = ((UIP_RECEIVE_WINDOW) >> 8);
    BUF->wnd[1] = ((UIP_RECEIVE_WINDOW) & 0xff);
  }

 tcp_send_noconn:
  BUF->ttl = UIP_TTL;
#if UIP_CONF_IPV6
  /* For IPv6, the IP length field does not include the IPv6 IP header
     length. */
  BUF->len[0] = ((uip_len - UIP_IPH_LEN) >> 8);
  BUF->len[1] = ((uip_len - UIP_IPH_LEN) & 0xff);
#else /* UIP_CONF_IPV6 */
  BUF->len[0] = (uip_len >> 8);
  BUF->len[1] = (uip_len & 0xff);
#endif /* UIP_CONF_IPV6 */

  BUF->urgp[0] = BUF->urgp[1] = 0;

  /* Calculate TCP checksum. */
  BUF->tcpchksum = 0;
  BUF->tcpchksum = ~(uip_tcpchksum());

 ip_send_nolen:
#if UIP_CONF_IPV6
  BUF->vtc = 0x60;
  BUF->tcflow = 0x00;
  BUF->flow = 0x00;
#else /* UIP_CONF_IPV6 */
  BUF->vhl = 0x45;
  BUF->tos = 0;
  BUF->ipoffset[0] = BUF->ipoffset[1] = 0;
  ++ipid;
  BUF->ipid[0] = ipid >> 8;
  BUF->ipid[1] = ipid & 0xff;
  /* Calculate IP checksum. */
  BUF->ipchksum = 0;
  BUF->ipchksum = ~(uip_ipchksum());
  DEBUG_PRINTF("uip ip_send_nolen: chkecum 0x%04x\n", uip_ipchksum());
#endif /* UIP_CONF_IPV6 */
enviar_ret:
       if (ret==1)
       {
              uip_flags = 24;
              memcpy(&uip_buf[0],&cache[k-1][0],110);
              uip_len = 110;
              k++;
       }
  UIP_STAT(++uip_stat.tcp.sent);
#if UIP_CONF_IPV6
 send:
#endif /* UIP_CONF_IPV6 */


  DEBUG_PRINTF("Sending packet with length %d (%d)\n", uip_len,
               (BUF->len[0] << 8) | BUF->len[1]);

  UIP_STAT(++uip_stat.ip.sent);
  /* Return and let the caller do the actual transmission. */

  uip_flags = 0;

       if (enviando==1)
```

```
        {
                memcpy(&cache[k-1][0],&uip_buf[0],uip_len);
                k++;
        }
        printf("NUMERO DE RET: %d\n",n_ret);
  return;

 drop:
        if ( (enviando==1) && (k==MODULO+1) && (uip_flags==1))
        {
                k=1;
        }
  uip_len = 0;
  uip_flags = 0;
        printf("NUMERO DE RET: %d\n",n_ret);
  return;
}
/*---------------------------------------------------------------------------*/
u16_t
htons(u16_t val)
{
  return HTONS(val);
}

u32_t
htonl(u32_t val)
{
  return HTONL(val);
}
/*---------------------------------------------------------------------------*/
void
uip_send(const void *data, int len)
{
  int copylen;
#define MIN(a,b) ((a) < (b)? (a): (b))
  copylen = MIN(len, UIP_BUFSIZE - UIP_LLH_LEN - UIP_TCPIP_HLEN -
                (int)((char *)uip_sappdata - (char *)&uip_buf[UIP_LLH_LEN +
UIP_TCPIP_HLEN]));
  if(copylen > 0) {
    uip_slen = copylen;
    if(data != uip_sappdata) {
      memcpy(uip_sappdata, (data), uip_slen);
    }
  }
}
/*---------------------------------------------------------------------------*/
/** @} */
#endif /* UIP_CONF_IPV6 */
```