Trabajo Fin de Grado

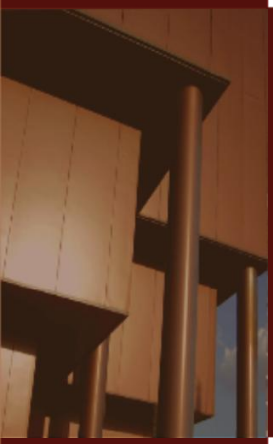
Grado en Ingeniería de las Tecnologías de Telecomunicación

Diseño y desarrollo de un sistema ITS basado en comunicaciones tipo ISO CALM

Autor: Alberto Rodríguez Blázquez

Tutor: Federico Barrero García

Cotutor: Jesús Sánchez García



Dep. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2015

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Diseño y desarrollo de un sistema ITS basado en comunicaciones tipo ISO CALM

Autor:

Alberto Rodríguez Blázquez

Tutor:

Federico Barrero García

Profesor titular

Cotutor:

Jesús Sánchez García

Dep. Ingeniería Electrónica

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2015

Trabajo Fin de Grado: Diseño y desarrollo de un sistema ITS basado en comunicaciones tipo ISO CALM

Autor: Alberto Rodríguez Blázquez

Tutor: Federico Barrero García

Cotutor Jesús Sánchez García

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2015

El Secretario del Tribunal

Sobre el autor

Alberto Rodríguez Blázquez

albertoroblaz@gmail.com

Grado en Ingeniería de las Tecnologías de Telecomunicación

Universidad de Sevilla

Agradecimientos

Realizar el proyecto y este documento, lejos de ser una tarea individual, no habría sido posible sin muchas personas que, de una forma u otra, me han ayudado.

Para empezar quiero agradecer a mi tutor, Federico, la confianza que me ha dado para afrontar este proyecto así como el poder recurrir a él sabiendo que siempre estaba disponible para resolver los problemas que se se me iban presentando.

También mis más profundo agradecimiento a Jesús, la persona que ha estado dirigiéndome y que me ha dado ánimos continuamente, teniendo siempre un momento (ya fuesen 5 minutos o una hora) para resolver cualquier duda repentina que me asaltara. Por otra parte merece una mención honorífica Ignacio, quien me ayudó a establecerme e iniciar el trabajo.

Mi familia me ha dado apoyo y cariño de forma incondicional durante la carrera, en especial en las etapas en las que estaba más agobiado. Pido perdón a mis padres y a mi hermana Marina (quien se ha quedado durante varias temporadas sin su hermano) por las preocupaciones que les he estado causando. Todo el reconocimiento que pueda darles aquí es poco.

Por último, agradecer a todos mis amigos y compañeros de carrera estos cuatro años juntos, y a todas aquellas personas que de una forma u otra han estado a mi lado. Les deseo toda la suerte del mundo es sus proyectos futuros.

Alberto Rodríguez Blázquez

Sevilla, 2015

Resumen

Los sistemas inteligentes de transporte, o ITS, son un conjunto de soluciones tecnológicas para mejorar la operación, seguridad y eficiencia del transporte terrestre, con aplicación a todo tipo de rutas, autopistas y centros urbanos.

Este trabajo fin de grado estudia qué son y cómo funcionan. En primer lugar se realiza un estudio teórico basado en la normativa de referencia. A continuación, se propone una implementación práctica de la parte de un ITS correspondiente a la comunicación entre dos vehículos.

El sistema propuesto consiste en dos placas router AlixBoard 3D2 sobre las que se ha instalado el sistema operativo OpenWrt. Sobre este sistema se ha desarrollado una aplicación distribuida. Este software consiste en un servicio de alertas de peligros en la carretera e intercambio de datos de recorrido entre diferentes automóviles.

Abstract

Intelligent Transport Systems, or ITS, are a set of technologies solutions which improve the operation, safety and efficiency of land transports, applying to all kinds of roads, highways and towns.

This BSc thesis studies what they are and how they work. In the first place a theorical research is done over the relevant legislation. Then, an ITS vehicle communication practical implementation is proposed.

The suggested system consists of two AlixBoard 3D2 router boards where OpenWrt operating system has been installed. On top of this system, a distributed application has been develop. This software is a road hazards alert and data exchange service between different cars.

Índice

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Notación

CALM	Communications Access for Land Mobiles
CF	C*ompactFlash*
CCH	Canal de Control
DSRC	Dedicated Short Range Communications
ETSI	European Telecommunications Standards Institute
FCC	Federal Communications Commission
GCDC	Grand Cooperative Driving Challenge
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPv6	Internet Protocol version 6
ISO	International Organization for Standardization
ITS	Intelligent Transport System
ITS-S	Intelligent Transport System Station
LLC	Logical Link Control
MAC	Medium Access Control
OBE	OnBoard Equipment
OBU	OnBoard Unit
OSI	Open Systems Interconnect
RFC	Request For Comments
RSU	Road Side Unit
SO	Sistema Operativo
SCH	Canal de Servicios
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UTC	Coordinated Universal Time
V2I	Vehicle-to-Infrastructure
V2V	Vehicle-to-Vehicle
WAVE	Wireless Access in Vehicular Environment
WSA	WAVE Service Advertisement

WSMP	WAVE Short Messaje Protocol

1 Introducción y objetivos

Science is more than a body of knowledge; it's a way of thinking, a way of skeptically interrogating the universe with a fine understanding of human fallibility.

Carl Sagan

La creciente demanda en materia de transporte de personas y mercancías ha obligado a un aumento del número de automóviles, generando problemas como las emisiones de CO2, un mayor número de accidentes de tráfico y un uso intensivo de las infraestructuras. Para dar solución a lo anterior, en la última década se ha desarrollado el concepto de sistemas inteligentes de transporte (ITS o *Intelligent Transport Systems*), cuyos avances se centraron en un principio en el funcionamiento interno del vehículo y la instalación de sensores en el mismo para aumentar su seguridad y reducir su consumo. El progreso actual de las tecnologías inalámbricas sin embargo, permite dar a los vehículos la capacidad de comunicarse entre ellos así como con las infraestructuras de tráfico, mejorando enormemente sus prestaciones. El presente proyecto versa sobre esta temática, aventurando una posible implantación de un sistema de comunicaciones vehiculares sobre las placas electrónicas comerciales AlixBoard 3D2.

1.1 **Comunicaciones en vehículos**

El desarrollo y mejora de los sistemas de transporte terrestres suele ser una prioridad en materia de investigación dentro de los países desarrollados. Esta inversión se fundamenta en el hecho de que la infraestructura existente tiene una cada vez mayor importancia económica y social. Por un lado la red de transporte es fundamental en la sociedad moderna, uniendo a consumidores y productores, contribuyendo a su crecimiento y desarrollo económico. Por otra parte, el transporte no solo es necesario para la circulación de bienes, su uso se extiende a las personas cuyas posibilidades culturales, relaciones sociales y bienestar en general, dependen de factores como el coste del desplazamiento, su duración y la comodidad del mismo.

Los avances en la materia se han centrado hasta hace relativamente poco en la ampliación y mejora de las infraestructuras que dan soporte a los vehículos y el perfeccionamiento de los motores de estos, haciéndolos cada vez más rápidos a la vez que se reducía su consumo, en definitiva, mejorando su rendimiento.

No ha sido hasta las últimas décadas cuando se han tenido en cuenta otros factores como el hacerlos más respetuosos con el medio ambiente (regulando las emisiones que puedan causar) y seguros, usando aplicaciones internas (como pueden ser el ABS o el ESP). El perfeccionamiento de los sistemas electrónicos y de las comunicaciones ha posibilitado avanzar en esta linea, permitiendo dotar a los vehículos de cada vez más sensores y sistemas de ayuda a la conducción; siempre mejorando la carretera o el automóvil de manera aislada sin tener en cuenta el conjunto de vehículos que circulan por esa carretera y las infraestructuras de apoyo.

No ha sido hasta la concepción de las "*Smart Cities*" cuando se ha creado una nueva línea de investigación basada no solo en la mejora de las prestaciones a nivel individual, sino también en un avance en la calidad y sostenibilidad de los transportes, tratando para ello a todos sus participantes como parte de una sistema interconectado.

Las *Smart Cities* o ciudades inteligentes son un modelo de desarrollo urbano, en el cual el capital humano, las comunicaciones y las infraestructuras conviven de forma armónica, estando caracterizadas por su compromiso con el entorno y por el uso de las más avanzadas tecnologías TIC. En este tipo de ciudades se ofrece un entorno inteligente y sostenible donde sus ciudadanos pueden interactuar con los diversos elementos de la ciudad, permitiendo una calidad de vida superior a las cotas actuales. Desligándose de la dirección seguida hasta ahora donde cada elemento se trata de forma "independiente", el concepto de "ciudad inteligente" se basa en la aplicación de la tecnología actual para ligar todos los componentes de una ciudad o zona, permitiendo que interactúen entre sí, gestionando a su vez los recursos naturales de una forma más eficiente .

Figura 1: Ciudad inteligente.

Dentro del ideal de *Smart Cities*, el tránsito de vehículos es gestionado desde una "red inteligente" formada tanto por los automóviles como por las infraestructuras. Este modelo es conocido como Sistemas Inteligentes de Transporte o ITS (*Intelligent Transport System*) por sus siglas en inglés.

En este arquetipo de sistema de transporte todos los vehículos e infraestructuras están conectados a una única red por la que pueden recibir información en tiempo real sobre accidentes, atascos y estado de la carretera entre otros. Esta capacidad de comunicación propia de los elementos pertenecientes al ITS, es capaz de conseguir unas tasas de seguridad, ahorro energético y eficiencia vial nunca vistas hasta la fecha. El hecho de que cualquier vehículo pueda intercambiar información con otros automóviles y unidades de carretera propicia la aparición de nuevos servicios. Prestaciones como el de aviso automático a los servicios de emergencia en caso de accidente, el control del tráfico para evitar atascos y otras relacionadas con el ocio como las llamadas entre automóviles y el acceso a internet desde estos, también serían posibles.

En este sentido, varias organizaciones internacionales como la ISO (*International Organization for Standardization*), la IEEE (*Institute of Electrical and Electronics Engineers***)** y la ETSI (*European Telecommunications Standards Institute*) ya han empezado a trabajar en el diseño y normalización de esta tecnología , de forma que los fabricantes de vehículos puedan tener en un futuro las guías necesarias para implementar el equipamiento que haga a sus automóviles participes de este nuevo modelo de sistema. Como resultado de este trabajo, existen actualmente tres grandes referencias para los ITS:

- Los estándares CALM (*Communications Access for Land Mobiles*) de la ISO: Tratan de forma general todo el sistema ITS estableciendo las bases para futuras normas más específicas.
- El estándar WAVE (*Wireless in Access Vehicular Environment*) de la IEEE: Está formado por las normas 1609, antiguo protocolo 802.11p (Ahora integrado en el 802.11-2012). Este estándar trata la parte del ITS correspondiente a las comunicaciones V2V (*vehicle-to-vehicle*) y V2I (*vehicle-to-insfractructure*). Amplia el estándar CALM pormenorizando qué protocolos y tecnologías son usadas para la transmisión de información en esta zona del sistema.
- Referencias ITS de la ETSI: Son una adaptación de las normas CALM y WAVE al ámbito europeo. En esta normativa también se tiene en cuenta los resultados de diversos proyectos llevados a cabo. De entre estos proyectos destacan COMeSafety, COOPERS, CVIS y SAFESTPORT. Gracias a ellos se pudo comprobar la viabilidad de implantar esta tecnología en el sistema de transportes actual. Debido a que este estándar apenas aporta novedades respecto a WAVE y CALM, no se entrara en él en detalle.

Siguiendo la línea de investigación descrita, en el actual proyecto se pretende conseguir una implementación básica de una parte del sistema ITS correspondiente a las comunicaciones entre vehículos siguiendo, con algunas simplificaciones, la normativa existente.

Este trabajo, se engloba dentro de un proyecto mayor llevado a cabo por el grupo de investigación ACE-TI del departamento de ingeniería electrónica de la Universidad de Sevilla, cuyo objetivo es el diseño y desarrollo de un vehículo eléctrico con un sistema de propulsión multifásico, que posea las tecnologías de comunicación más avanzadas para poderlo integrar en un sistema ITS. Con ello se pretende dar respuesta a los principales problemas del sector automovilístico, creando un vehículo no contaminante y capaz de formar parte de una "red inteligente" que ofrezca transmisión de datos y servicios, llevando a un nuevo nivel las prestaciones de los automóviles actuales.

1.2 **Objetivos**

El estudio de las comunicaciones dentro un ITS es una materia bastante extensa debido a que se deben tener en cuenta multitud de escenarios, como pueden ser los intercambios de información del núcleo de la red, el paso de mensajes en las comunicaciones inalámbricas con las estaciones móviles, la integración con otras tecnologías ya existentes y un largo etcétera que hacen que sea muy difícil abarcar todo de forma minuciosa en un trabajo fin de grado. Incluso para las organizaciones del sector, resulta una tarea ardua debido a la heterogeneidad existente

en la red de transportes. Un buen ejemplo de esto se da en el conjunto de normas CALM, las cuales, debido a que deben definir todos los aspectos relacionados con los ITS, pueden resultar demasiado abstractas, marcando únicamente unas pautas básicas a seguir.

Por ello se ha decidido abordar únicamente el problema de comunicaciones desde el punto de vista de los vehículos, tal como ha hecho la IEEE, de forma que se pueda llegar a conseguir un punto de partida a raíz del cual empezar a desarrollar el resto de ámbitos, centrándose por tanto en las comunicaciones V2V y V2I.

Figura 2: Tecnologías participantes en un ITS. ETSI.

Incluso limitándose al estudio de estas dos interacciones, surge el problema de que el paso de información de un nodo a otro necesita estar respaldado por un plano de control (transparente al usuario) que compruebe y ajuste el funcionamiento de la red, y por un plano de seguridad, que garantice la privacidad e integridad de las comunicaciones cursadas. El estudio de cualquiera de estos dos planos añade complejidad al asunto, pues son materias extremadamente amplias sobre las que, en muchos casos, aún no se ha alcanzado un consenso. Se ha optado por dejar de lado la seguridad y el plano de control proponiéndolos como proyectos de mejora.

Además de lo anterior y como se verá más adelante, la implementación de un subsistema ITS en un vehículo conlleva instalar una serie de módulos para alcanzar las distintas funcionalidades, como pueden ser conectar la tecnología de terceros al subsistema ITS del vehículo, controlar el equipamiento "inteligente" del automóvil o permitir que todos ellos puedan intercambiar información y conectar este subsistema al resto de nodos ITS a través de un router, en definitiva, dotar al vehículo de toda una red de comunicaciones interna.

En vez de intentar estudiar y definir todo lo referente al vehículo, se ha resuelto montar una implementación básica de un par de routers en red V2V usando para ello dos AlixBoard 3D2 (Alix a partir de ahora) y proveer una aplicación que pueda ofrecer un servicio básico sobre

ellas. Consiguiendo esto, se aporta un punto de partida para el diseño e implementación de resto de módulos dentro del automóvil. De manera esquemática, se pretende:

- Estudiar la principal normativa de referencia en el ámbito de las comunicaciones V2V y V2I. Se usarán fundamentalmente los protocolos descritos en WAVE debido a su similitud con los ya existentes en las redes de ordenadores.
- Implantar un prototipo de sistema ITS entre dos placas comerciales AlixBoard 3D2 con un servicio ITS de alerta de peligros en carretera. Explícitamente esta implementación tiene las siguientes características:

  - La aplicación se ejecuta sobre un sistema operativo ya usado anteriormente en el proyecto europeo GCDC (*Grand Cooperative Driving Challenge)*. Éste es el OpenWrt, el cual está diseñado como una distribución Linux embebida para routers.
  - El paso de mensajes se realiza en una red inalámbrica situada en la banda de los 5,9 GHZ, las frecuencias reservadas para comunicaciones DSRC (*Dedicated Short Range Communications*). La red usada está definida como de tipo ad-hoc entre vehículos o entre vehículos y unidades de carreteras, sirviendo estas últimas de punto de acceso a otras redes.
  - Se obtiene una batería de datos (posición, id y velocidad) del supuesto vehículo participante del sistema ITS y se transmite al resto de nodos de la red que estén al alcance.
  - La transmisión de estos datos se realizará mediante un servicio que es idéntico en todos los nodos móviles debido al carácter de “iguales” en ad-hoc.
  - La tarjeta receptora de los datos realiza un procesamiento de los mismos y toma las acciones adecuadas, estableciendo conexión con la entidad emisora en caso necesario.
  - La aplicación creada está pensada para que sea escalable, no estando limitada al funcionamiento entre dos nodos. Debido a las peculiaridades de los escenarios ITS, los nodos pueden llevar a cabo lo anterior sin necesidad de conocerse a priori.

2 Fundamentos Teóricos

To a certain extent, more than one term covers the same concept, and sometimes the same term covers more than one concept.

IETF, RFC 4026

El estudio de la red de transportes, la búsqueda de formas de perfeccionar su arquitectura y hacerla más eficiente es un tema complejo que ha sido abordado por multitud de publicaciones. Se manifiesta de forma cada vez más evidente el hecho de que el próximo paso en la evolución de este campo es el de dotar a los vehículos (y a la propia infraestructura) de un sistema de comunicaciones común. Existe, sin embargo, el riesgo de que cada país o región evolucione hacia un arquetipo propio, con lo que se dificulte a la larga el transporte de personas y mercancías entre los diversos sistemas. Para evitar esto, varias organizaciones internacionales de estandarización redactaron una serie de normas que sirviesen de paradigma a la hora de dar este nuevo paso en las tecnologías de los transportes. Las más importantes en esta materia son las normas CALM de la ISO y las normativas 1609 del estándar WAVE de la IEEE. Estas normas introducen el concepto de ITS *(Intelligent Transport System*), y definen diversos formatos de comunicaciones para estos sistemas.

2.1 Sistemas ITS

Un sistema de transporte se puede definir como las infraestructuras necesarias para conducir personas y cosas de un lugar a otro. La eficiencia de la red que posea un país tiene impacto directo en su economía, ya que marca qué cantidad de productos y a qué coste se pueden desplazar, teniendo un gran efecto en su productividad. Típicamente las mejoras en este sector se basan en aumentar el número de infraestructuras disponibles y la calidad de estas, pasando de carreteras a autovías y de calles a avenidas pero, ¿Qué diferencia hay entre un sistema inteligente de transportes y uno que no lo es?

El concepto de ITS se basa en aumentar las capacidades de los vehículos y carreteras actuales haciendo uso de las tecnologías de información y comunicación (TIC) conocidas. Como ya se hiciera con la creación de las redes de ordenadores, si se conectan los diferentes actores del sistema de transporte a una misma red, estos pueden cooperar a la hora de resolver los problemas que se presenten. Establecer una red de vehículos permite entre otras cosas, mejorar la seguridad de sus usuarios, evitar congestiones en las carreteras, disminuir la contaminación producida y, en general, mejorar la productividad agilizando el tránsito de los vehículos. La posibilidad de que el sistema se "auto regule" en tiempo real en función de las condiciones presentes es lo que le ha otorgado el apelativo de "inteligente".

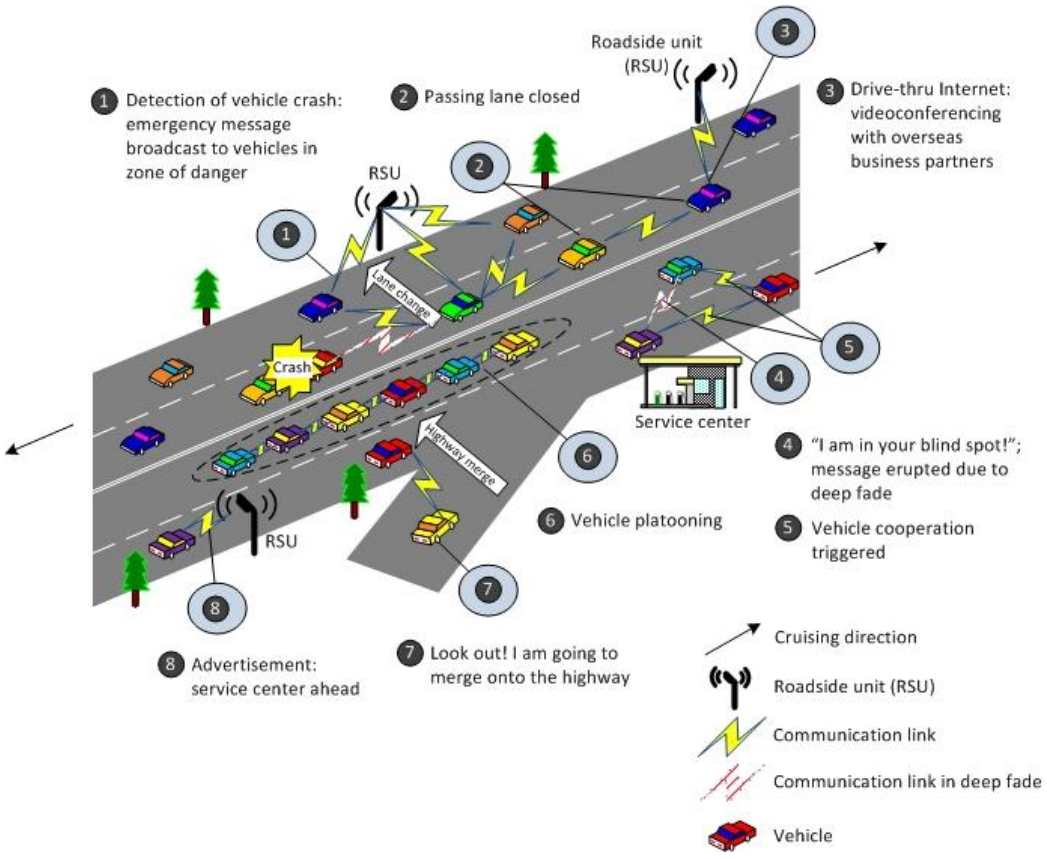


Figura 3: Circulación por carretera en un ITS.

Al contrario de lo que sucede con las redes de ordenadores, el dominio de un ITS no está pensado para interactuar directamente con los usuarios, permaneciendo transparente a estos y siendo en este sentido más parecido a una red que ofrece unos determinados servicios de los que se puede hacer uso. Aun así, los ITS están pensados para que no sean un sistema aislado, pudiendo estar conectados a otras redes con diferentes tecnologías (dominios genéricos) como puede ser Internet, lo que posibilita el poder ofrecer prestaciones de estos dominios a través de esta red.

Un sistema o dominio ITS no es global, y está enfocado a la aplicación de un ITS por zona o región (una ciudad, una carretera, una provincia…). Las entidades usuarias de un ITS son denominadas subsistemas ITS. Estos subsistemas poseen un conjunto de módulos llamado ITS-S (*Intelligent Transport System Station*), que permite a la entidad (vehículo o infraestructura) conectarse a la red inteligente, comunicarse con otras entidades de la misma red y usar los servicios que ésta ofrece.

En general, se hace distinción entre 4 clases de estaciones ITS. La notación cambia dependiendo de si se consulta las normas CALM o el estándar WAVE, aunque ambas coinciden en sus funciones.

2.1.1 **Subsistemas ITS**

Una red inteligente de transportes está formada por diferentes subsistemas ITS interconectados entre sí. Para que un vehículo o infraestructura pueda ser considerado uno de estos subsistemas, es decir, un usuario de la red; éste debe tener instalado una serie de módulos que le permitan acceder y hacer uso de los servicios que el sistema ofrece. Estos módulos se denominan estaciones ITS (ITS-S), debido a que son los que implementan las funciones y la conectividad con el resto de la red inteligente. El paso de información dentro de una ITS-S se realiza a través de una red interna. Esta "red local" está oculta al resto de usuarios de la red, de forma que el conjunto de la ITS-S se considera un único nodo.

Los módulos que conforman la ITS-S varían dependiendo en gran medida de las características de la entidad donde se implemente y de las funcionalidades buscadas. Dentro del estándar CALM se han definido los siguientes tipos de subsistemas ITS, integrando cada uno una ITS-S distinta:

- Personales o viajeros.
- Vehiculares.
- Centrales.
- De carretera o campo.

Estos cuatro tipos englobarían todos los sistemas encontrados en una red ITS. Sin embargo, solo dos de los aquí mostrados (los vehiculares y los de carretera) pertenecen a la zona del ITS

donde ocurren las comunicaciones *vehicle-to-vehicle* (V2V) y las *vehicle-to-infrasctrucure* (V2I). El resto de entidades se comunican con las demás a través de cableado o bien usando otras formas de comunicación inalámbrica como los infrarrojos (IR), Bluetooth o las redes móviles (2G y 3G). Por ello, se tratarán las estaciones de los subsistemas vehiculares y los de carretera con más profundidad, dando tan solo unas pinceladas de las otras.

Para mayor comodidad, a partir de ahora se referenciarán los conceptos ITS-S y subsistema ITS de forma indiscriminada, aludiendo a la entidad usuaria del ITS o a los módulos integrados en ella dependiendo del contexto. Esta simplificación es factible, pues la ITS-S se muestra como un único nodo al resto de la red y, a todos los efectos, es representante del subsistema ITS donde está integrada.

2.1.1.1 **Subsistemas ITS personales**

Los subsistemas ITS personales son equipos de diversas funcionalidades (como puede ser un teléfono móvil o una PDA) que implementan la conectividad necesaria para acceder a los servicios de un ITS. En este apartado se consideran tanto los equipos portátiles que tengan implementados el equipamiento necesario para poder acceder a la red por sí mismos como los que lo hacen a través de una ITS-S que sirva como punto de acceso. Estas estaciones a su vez pueden servir de punto de acceso a otras ITS-S o a otros dispositivos que no dispongan de la interfaz adecuada.

2.1.1.2 **Subsistemas ITS vehiculares**

Son los vehículos de la red de transporte que implementan conectividad CALM. La ITS-S de un subsistema ITS vehicular tienen que tener en cuenta no solo la movilidad a la que se ve sometida la estación (lo que dificulta establecer y mantener una conexión con otros nodos) también debe poder convivir con otros sistemas ya existentes en el automóvil como pueden ser los integrados por los fabricantes para la comunicación entre las diferentes partes del vehículo. El conjunto de módulos de esta ITS-S también es llamado OBE (*On Board Equipment)*.

- **Pasarela ITS del vehículo:** La red interna del vehículo consistiría en la tecnología propietaria que conforman las comunicaciones internas del coche, como puede ser la unidad de control electrónico (ECU) o el ordenador de a bordo. Este módulo permite el intercambio de información entre estos módulos y la red propia de la estación ITS montada en el coche, aportando una interfaz que permita que datos de funcionamiento interno (motor) o de los sensores del vehículo (sensores de lluvia, control de estabilidad…) puedan ser compartidos con dispositivos de otros fabricantes, con otros coches, unidades de carretera o con dispositivos personales.

- **ITS host:** Permite conectar ITS-S personales a la ITS-S del vehículo. La forma de conexión no está definida, pudiendo ser por ejemplo Bluetooth, WiFi, HDMI, etc. Este módulo junto con el router ITS está pensado para que el vehículo pueda ser usado por sus usuarios como punto de acceso a redes externas al mismo y a servicios ofertados (por ejemplo, asistencia en carretera).
- **Router ITS:** También llamado OBU (On Board Unit), permite conectividad entre los dispositivos de red local de la ITS-S y estaciones externas como pueden ser las montadas en otros vehículos o situadas en la carretera. A nivel práctico, proporciona a la estación conexión con otras ITS-S. La implementación práctica de este trabajo se centra en este módulo, cuyo papel esta desempeñado por una placa AlixBoard 3D2. El programa desarrollado comunica un router ITS con otro igual que se supone situado en otro vehículo o en una unidad de carretera.

En la Figura 4 se muestran los módulos que componen la ITS-S de un vehículo.

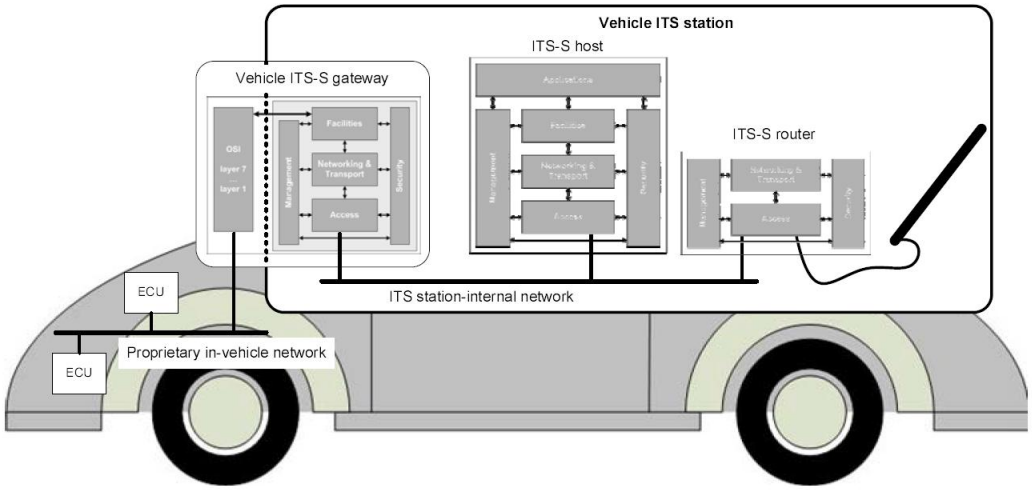


Figura 4: ITS-S vehicular. ISO 21217.

2.1.1.3 **Sistemas ITS de carretera**

Esta clase de subsistema está pensado para ser implementado en unidades fijas de carretera y urbanas (también llamadas *RoadSide Units* o RSU) como pueden ser un poste, un arco o un semáforo. Su principal función consiste en dar conectividad entre las ITS-S móviles montadas en los automóviles y la infraestructura ITS fija. Estos subsistemas también ofrecen compatibilidad para ser implementados con otros ya existentes en la vía, como radares, sensores de lluvia y hielo, controles de gálibo o control de aparcamientos, integrándolos en el ITS. En estas estaciones se pueden diferenciar los siguientes módulos:

- **Pasarela ITS:** Permite conectar un sistema propietario integrado en la carretera a la ITS-S. Tiene la misma función que la pasarela presente en los vehículos pero

enfocada a la conexión de la infraestructura que no implementa el protocolo CALM.

- **ITS-S host:** Conecta entidades ITS con la ITS-S. Los sistemas conectados aquí difieren de los de la pasarela ITS en que estos sí usan CALM.
- **Router ITS:** Soporta conectividad entre la ITS-S de la unidad de carretera y las ITS-Ss de los vehículos. También permite que una RSU se conecte a otras o a ITS-S centrales. En principio es igual a sus homólogos instalados en las ITS-S vehiculares.
- **Router ITS frontera:** Router ITS con funcionalidades adicionales que permite conectar la ITS-S con la red de un sistema del dominio genérico, sirviendo de interfaz entre CALM y otras tecnologías. Las estaciones que presentan este módulo serán denominadas ITS-S frontera. Este módulo debe implementar dos torres de protocolos, una por la parte CALM y otra con el usado por las tecnologías a conectar. En caso de uso de TCP/UDP en la red CALM**.**, implementa típicamente hasta el nivel IP en las torres.

Los módulos de este subsistema se encuentran representados en la Figura 5.

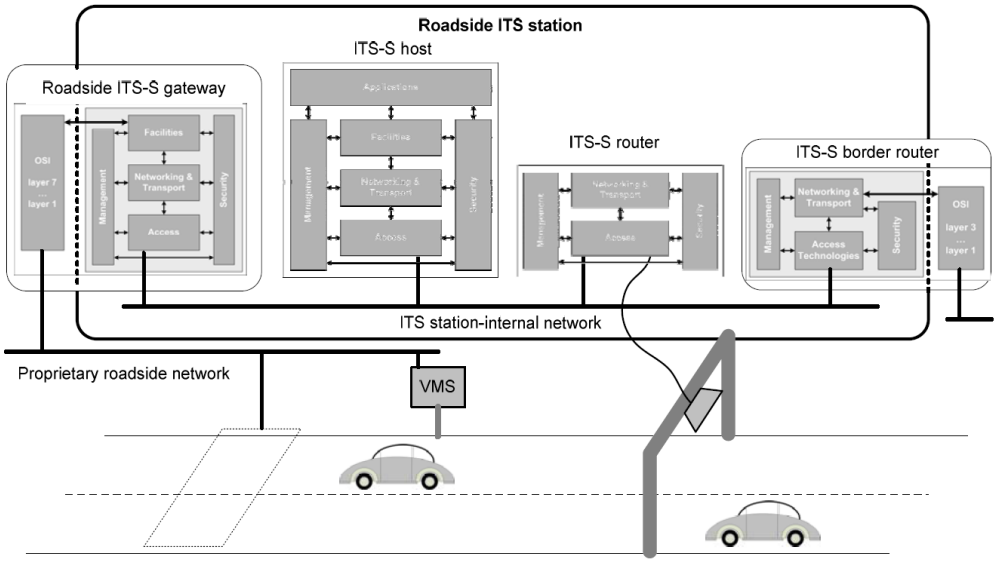


Figura 5: ITS-S de carretera. ISO 21217.

2.1.1.4 **Subsistemas ITS centrales**

Estos subsistemas no tienen comunicación con los vehículos y llevan más funcionalidades que las vistas aquí. En este caso se definirán como ITS-S pertenecientes al núcleo del ITS. Entre sus funciones se encuentran monitorizar y gestionar la red, sirviendo de centros de procesamiento y actuando como proveedores de los servicios principales.

Para el ámbito de este proyecto cumplirán la función de conectar varias ITS-S de carretera o ITS-S centrales entre sí. También es pasarela para acceder al dominio genérico.

Según esta explicación, los módulos encontrados son los siguientes:

- **Pasarela IT-S:** Para la conexión con sistemas de otras tecnologías.
- **ITS host:** Soporte para subsistemas ITS conectados a una ITS-S central.
- **Router ITS:** Permite la conexión con ITS-S de carretera y otras ITS centrales.
- **Router ITS frontera:** Soporta la conectividad con redes del dominio genérico.

En la Figura 6 se representa este subsistema con conectividad al dominio genérico.

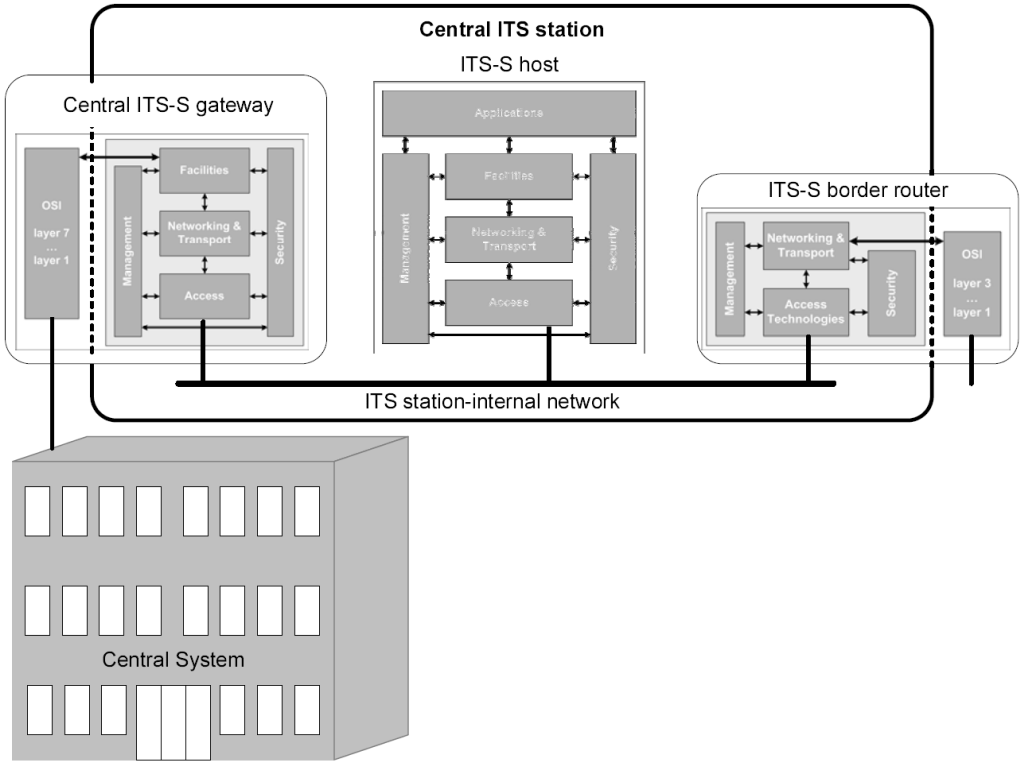


Figura 6: ITS-S central. ISO 21217.

2.2 El estándar CALM.

CALM (*Communications Access for Land Mobiles*) es una familia de normas desarrollada por la ISO (*International Organization for Standardizatio),* una organización ampliamente distinguida por la labor que realiza en la confección de estándares reconocidos internacionalmente. La ISO está dividida en distintas comisiones nacionales y sus normas son el resultado de la tarea llevada a cabo por comités técnicos. Trabaja además en constante colaboración con otras organizaciones, tanto gubernamentales como no gubernamentales, como

pueden ser la *lnternational Electrotechnical Commission* (IEC), la *Internet Engineering Taks Force* (EITF) o la IEEE.

Las normas ISO-CALM, fueron creadas para definir las tecnologías e interfaces usadas en las comunicaciones llevadas a cabo dentro de un sistema inteligente de transportes. Su objetivo es el introducir la posibilidad de que sean los propios vehículos e infraestructuras los que puedan comunicarse a través de un sistema propio, dando a la red de transportes gran flexibilidad y unas funcionalidades muy superiores a las actuales. Las normas CALM, además de definir como se deben comunicar los usuarios de este sistema, delimitan cómo es la estructura básica de esta red inteligente y qué actores la forman. Estas comunicaciones requieren, entre otras características, de la capacidad de transmitir información a largas distancias de forma segura y de soportar grandes flujos de tráfico, garantizando calidad de servicio en algunos de ellos.

CALM no se limita solamente a los vehículos y relativos, sino que pretende ser un sistema común para la comunicación entre estaciones móviles y las infraestructuras que conforman la ciudad, dando acceso a las otras redes o a nuevos servicios integrando tecnologías actualmente implementadas (como el GPS). Es debido a la heterogeneidad a la que se ve sometida el ITS por el que CALM no entra a definir en profundidad cada una de las partes que conforman el sistema o lo que es lo mismo, cómo se comunican cada una de las clases de ITS-S con el resto. Esto será definido en normativa más específica.

Desde el punto de vista de CALM, las redes se clasifican en dos dominios: El dominio ITS, formado exclusivamente por ITS-Ss que se comunican entre sí; y el dominio genérico, formado por todas aquellas redes fuera del ámbito de CALM. Ambos dominios se comunicarían a través de nodos con routers ITS frontera, los cuales soportan tanto CALM como los protocolos necesarios para conectar a la red externa. De esta forma se presentan dos principales tipos de comunicación en un sistema ITS, aquellas inter-dominio entre ITS-Ss y las extra-dominio entre un ITS-S y un nodo del dominio genérico.

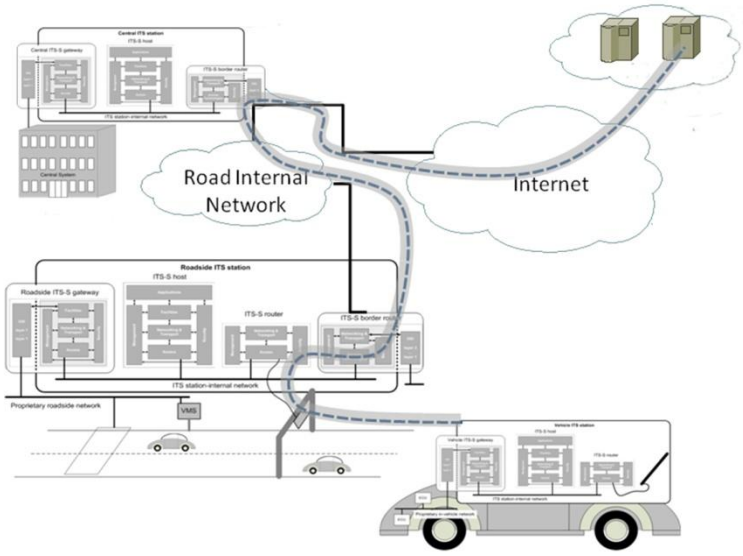


Figura 7: Ejemplo de comunicación extra-dominio usando la arquitectura CALM.

Un punto fundamental buscado en los sistemas CALM es que las aplicaciones sean independientes de las tecnologías de acceso al medio usadas, no limitándose a un único protocolo de enlace o de red[1]. Esto posibilita que una estación transmisora use un protocolo a nivel de red distinto a la estación receptora. Los estándares CALM dejan a fabricantes y desarrolladores el uso y manejo de la información a alto nivel (nivel de aplicación), centrándose en definir las funciones de las capas del modelo a seguir y las interfaces que conectan unas capas con otras. En la Figura 8 puede verse la división en capas con ejemplos de los posibles protocolos y funcionalidades que pueden ir en cada una de ellas.

2.2.1 **Comunicaciones CALM**

El estándar CALM está enfocado a especificar las comunicaciones existentes en un ITS, las cuales pueden darse tanto usando una de interfaz radio como a través de Ethernet u otras tecnologías. Debido a esto, únicamente se define una torre de protocolos general y unas interfaces de paso entre una capa y otra. Un estándar basado en este esquema y enfocado en las comunicaciones V2V y V2I (WAVE) fue desarrollado por el IEEE. Este estándar, que se verá más adelante, define en mayor profundidad la torre para el caso de unidades móviles, donde se aporta una solución definiendo un formato de comunicaciones a usar.

La arquitectura usada por CALM parte de la torre general de protocolos especificada por el modelo OSI (*Open Systems Interconnection*) para Internet, la cual se ha simplificado agrupándose capas con funcionalidades parecidas. Cada una estas capas puede a su vez ser dividida haciendo uso de distintos protocolos o repartiendo su funcionalidad en diferentes dispositivos.

[1] Al hablar de protocolos de enlace, de red y también de capas se hace referencia al modelo OSI (*Open System Interconnection Model*)

Se han añadido dos capas transversales no existentes a priori en la modelo OSI (control y seguridad) que tienen interfaz directa con el resto de capas al estar colocadas de forma transversal a éstas. Como ya se ha comentado, ésta es una torre general que sirve de referencia, por lo que dependiendo del equipo estas pueden presentarse de forma diferente o no estar todas. De nivel superior al inferior serían:

- **Capa de aplicación**

En esta capa se situarían el software y programas que puedan interactuar directamente con CALM, llamados *CALM-Aware*. También es donde se sitúan los servicios ofrecidos en un sistema ITS. Además del uso de las capas inferiores, tiene comunicación directa con las capas de seguridad y control. El uso de la interfaz con la capa de mantenimiento permite que la aplicación CALM reciba mensajes desde otro equipo por medio de protocolos diferentes a los usados en la capa de transporte, de una forma externa al plano de usuario. Esto permite poder negociar y cambiar el protocolo usado antes y durante el intercambio de información en función de los disponibles en cada momento. La interfaz de seguridad proporciona acceso a directivas de encriptación de información sin tener que depender de las capas inferiores.

Existe la posibilidad de que una aplicación no use directamente la interfaz de seguridad y mantenimiento. En este caso, la aplicación solo podría usar los servicios propios de CALM a través de la capa de servicios que en este caso haría las funciones de una API (*Application Programming Interface)*. Sin acceso a la capa de control, el protocolo usado para comunicación tiene que estar pre-seleccionado (típicamente UDP o TCP).

- **Capa de servicios**

Esta capa corresponde a los niveles 5 y 6 del modelo OSI, correspondientes a las funciones de establecimiento de sesión y presentación de la información. Su comportamiento es el de una API para el caso de aplicaciones que no sean *CALM-Aware*, sirviendo de adaptador para las llamadas a las capas de seguridad y control. En caso de que el nivel de aplicación tenga interfaz directa con las capas transversales, la capa de servicios hace las funciones de capa de sesión y presentación especificadas en la torre de protocolos OSI. Esta capa está actualmente poco definida. La capa de servicios es la superior de las pasarelas ITS situadas en los OBE y RSE.

- **Capa de encaminamiento y transporte**

Corresponde a las capas de red y de transporte de OSI (capas 3 y 4). Sus funciones básicas consisten en encaminar información desde un origen hacia su destino dentro de la red. En este nivel, se hace distinción en el uso de protocolos basados en IP y no IP. Como ejemplo de lo primero se dan TCP y UDP, mientras que el segundo caso está pensado para protocolos de tipo geo-routing o basados en saltos como AOVD (*Ad hoc*

On-Demand Distance Vector). En el caso de IP lo recomendado es el uso de IPv6, mientras que en el otro caso se ha definido la norma ISO 29281 para el uso de aplicaciones no IP. Los routers de la red ITS llegan hasta esta capa.

- **Capa de acceso**

Corresponde a los niveles más bajos de la torre de protocolos, y aquí CALM apenas está definido, dejando prácticamente libre la elección de la interfaz acceso al medio. Se puede destacar las tecnologías de acceso 2G, 3G, CALM M5, CALM IR y un largo etcétera. La elección de uno u otro dependerá sobre todo del medio usado para la transmisión. Esta capa se detalla en los estándares más específicos dedicados a cada una de las posibles interacciones dentro de CALM.

Además de las capas mencionadas, CALM añade dos capas más, una de seguridad y otra de control de la información. Estas dos capas son transversales a la torre de protocolos, pudiendo ser accedidas desde cualquiera de las capas anteriores, sus características están definidas en la ISO 24102.

- **Capa de mantenimiento**

Entre sus funciones están la de regular las comunicaciones del equipo en función de la legislación, el control de la congestión, enviar y recibir información de control y ofrecer cambios de protocolo de comunicación en tiempo real.

- **Capa de seguridad**

Proporciona las operaciones necesarias para una comunicación segura extremo a extremo. Puesto que dependiendo de la comunicación esta puede necesitar ser cifrada desde distintos niveles de la torre, esta capa proporciona directivas que pueden ser utilizadas desde todos ellos.

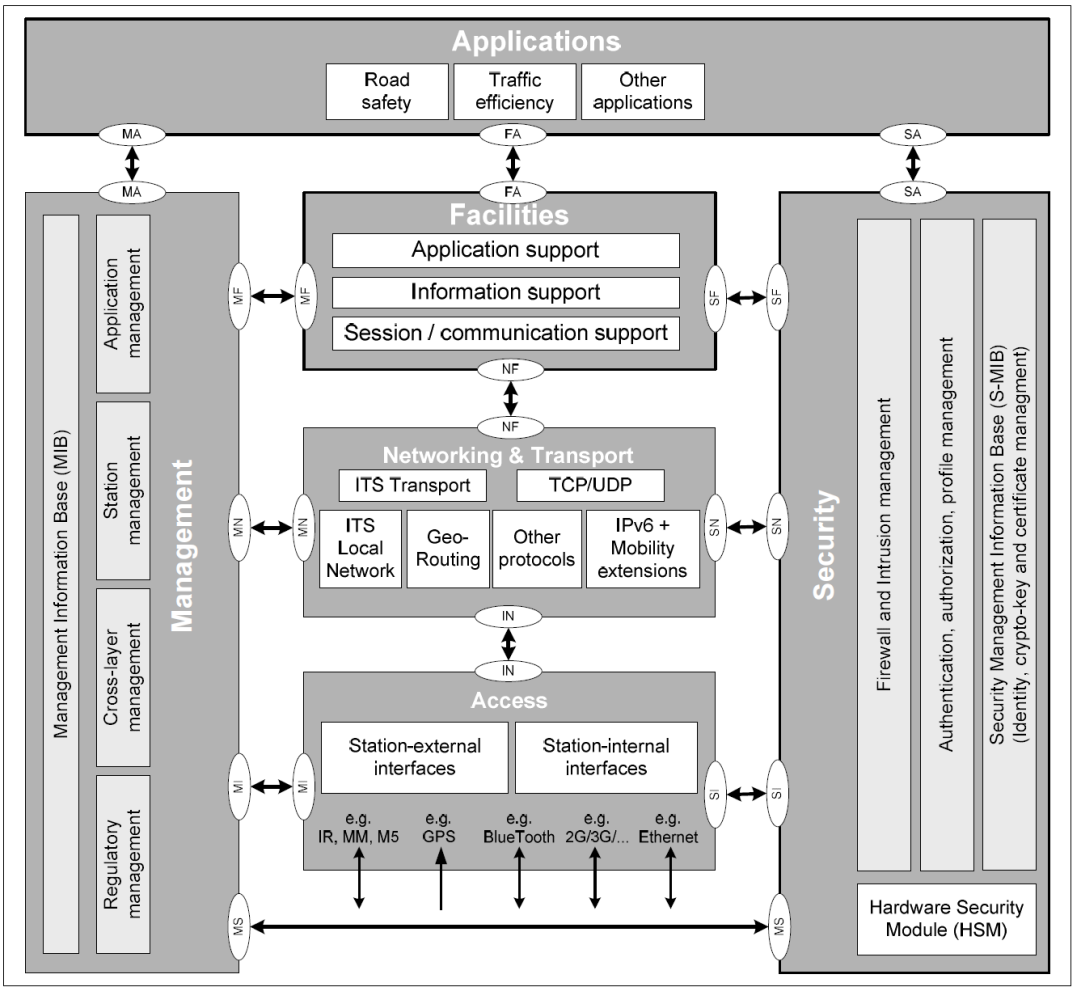


Figura 8: Capas de la arquitectura descrita por CALM. ISO 21217.

2.3 **Redes vehiculares**

La implantación de una red que enlace una estación móvil (automóvil) a las infraestructuras (puntos de acceso) y a otros vehículos mediante una conexión inalámbrica, es un tema de estudio relativamente reciente. Pese a ello, las VANET (*Vehicular Ad-Hoc Networks*) han sido objeto de numerosos estudios con distintos protocolos y pruebas de rendimiento. No es materia por tanto de este documento el hacer simulaciones y análisis de rendimiento de las distintas soluciones sugeridas, sino la implementación de una opción viable de los resultados ya publicados siguiendo lo ya normalizado.

La división entre V2V y V2I anteriormente mencionada no es solo a nivel teórico, pues a pesar de tener grandes similitudes, difieren en algunas premisas que hacen que resulte complicado abordar ambos tipos de comunicaciones con el mismo sistema. Casi toda la documentación coincide en caracterizar a las ITS-S vehiculares como nodos de una red, presentando un movimiento pseudo-arbitrario (limitado a las vías aptas para su conducción) y

con velocidad variable, comúnmente considerada entre 0 y unos 140 Km/h[2]. Haciendo una comparativa de los escritos consultados, se pueden diferenciar dos formas de aproximarse al problema:

a) Conexión de una ITS-S móvil a otra.
b) Conexión de una ITS-S móvil a una fija situada en una RSU.

La distinción entre ambos puntos se basa principalmente en que mientras en el caso a) ningún ITS-S tiene una posición fija (los coches se van moviendo a placer de sus conductores); las ITS-S implantadas en las RSUs del caso b) están fijas en el medio, permitiendo un fácil acceso a servicios orientados a conexión desde estas (a través de sus routers ITS frontera). Esta diferencia se hace patente en las diferentes soluciones aportadas en el encaminamiento de CALM, pues afecta en gran medida al funcionamiento de los protocolos que se pueden usar en cada caso. El propio estándar CALM propone para este nivel dos opciones, una basada en IP (ISO 21210) y otra en una tecnología alternativa (ISO 29281). Para esta última, la normativa más específica ha ido diseñando sus propios protocolos, siendo el caso de WSMP (*Wave Short Message Protocol*) para WAVE o los protocolos de GeoNetworking de la EN 302 636 de la ETSI.

Fundamentando las aplicaciones sobre IP, a nivel de la capa de protocolos de transporte, hay un acuerdo generalizado en el uso de los protocolos TCP/UDP debido a su utilización como protocolos de transporte en las redes de ordenadores, lo que facilita el adaptar los sistemas ya existentes a la red ITS a la vez que permite de una manera sencilla conexiones con redes fuera del ámbito de los ITS, concretamente con Internet.

2.3.1 **Comunicaciones entre vehículos**

En las redes ITS, una de sus principales características es la capacidad que tienen las estaciones móviles de comunicarse entre ellas tanto para mejorar las condiciones del tráfico como para evitar accidentes. Las V2V se basan en una comunicación entre pares, donde no existe una estación que se ofrezca como punto de acceso a las demás, es decir, no se tiene una infraestructura con privilegios sobre el resto. El alcance de este tipo de interacciones es usualmente corto (<1Km), conteniendo servicios únicamente de interés para ITS-S cercanas. Esto supone que el dominio de difusión y de intercambio de información debe estar acotado a 1 ó 2 saltos dependiendo del protocolo usado. Con esta acotación lo que se pretende es no inundar la red de información indiferente al resto de usuarios.

La tecnología usada en este tipo de redes debería de ser capaz de transmitir información de forma fiable entre dos ITS-S en movimiento. Esto exige que el tiempo de establecimiento de la conexión y su latencia sean extremadamente bajos, pues dos vehículos en sentido contrario en una autovía disponen tan solo de unos cuantos segundos para el intercambio de datos antes de

[2] Basado en el rango de velocidades usuales en las simulaciones consultadas.

salir de alcance. Las tablas de encaminamiento del nivel 3 usadas deberán soportar cambios topológicos continuos en la red, lo que supone un duro trago para protocolos basados en el modo infraestructura como IP. Además de lo anterior, este establecimiento debe poder hacerse sin conocer a priori la dirección de la estación a enlazar. Por todo ello, CALM deja espacio al uso de un protocolo no IP que pueda solucionar estas carencias. La Figura 9 representa una topología complicada para IP por el número de nodos y los cambios continuos que se producen debido al movimiento de estos.

Figura 9: Posible topología en comunicaciones V2V

Como alternativas, se han estudiado (a través de numerosas publicaciones y simulaciones) multitud de protocolos de encaminamiento para las VANET, tanto con establecimiento de rutas bajo demanda o reactivos (AODV, AOMDV, DSR o DYMO) como protocolos de búsqueda continua de rutas o proactivos (DSDV, TORA, FSR o LAR), habiéndose llegado a implementar algunas de estas soluciones sobre prototipos funcionales.

Abandonando CALM e introduciéndonos en las normas 1609 del IEEE, éstas establecen un protocolo propio WSMP (*Wave Short Message Protocol*) para comunicaciones no IP en este tipo de casos. Al ser propuesto específicamente por esta norma (la referencia en las VANET) se prevé que sea el protocolo que finalmente se imponga en las conexiones no IP.

2.3.2 **Comunicaciones entre vehículos e infraestructura**

Al contrario que en el apartado anterior, la jerarquía en estas conexiones no es plana, sirviendo las ITS-S fijas como punto de acceso para las móviles y proveedoras de los servicios del núcleo del ITS. Si cada RSU a la que se conecta la ITS-S móvil es quien asigna la dirección de red, esto se traduce en una topología más estable desde el punto de vista del núcleo de la red, lo que permite el uso de algoritmos de encaminamiento más eficientes, como los protocolos basados en direccionamiento geográfico (*geogrid*), reduciendo la latencia y los cortes en las conexiones. Las estaciones situadas en los ITS-S vehiculares del sistema siguen manteniendo su movilidad, por lo que se debería seguir manteniendo un sistema interno de itinerancia similar al ya empleado en las redes de telefonía móvil.

En esta clase de conexiones se siguen ofreciendo dos soluciones, una IP y una basada en un sistema no IP. Nuevamente se toma el estándar WAVE y su protocolo WSMP como referencia de comunicación no IP. Aquí, los protocolos basados en tablas de direccionamiento como IP gana fuerza al disponer de una red jerarquizable con la que sacar partido a sus características. Sigue, sin embargo, siendo necesaria una solución alternativa que ocupe un menor ancho de banda para comunicaciones puntuales y resolver el desconocimiento a priori de las direcciones de los nodos destino en el caso de que estas no sean asignadas por las RSU..

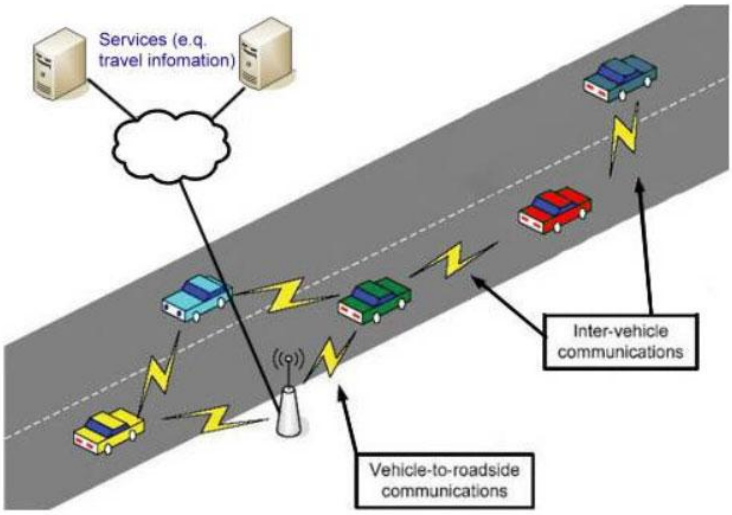


Figura 10: Acceso a servicios en redes externas a través de una RSU.

2.4 **Comunicaciones WAVE**

Debido a que la ISO define únicamente la arquitectura de un ITS de forma general, varias organizaciones han decidido complementar la información aportada por CALM en protocolos centrados en áreas más específicas, de tal forma que puedan ser usados como referencia real a la hora de diseñar un sistema.

La ISO, el IEEE y la IETF (*Internet Engineering Task Force)* realizaron un esfuerzo de coordinación para que sus respectivos estándares fueran compatibles entre ellos. De esta forma CALM define el ITS y su estructura a nivel general, mientras que WAVE encaja y define con exactitud la parte de comunicaciones de carretera V2V y V2I. Tanto WAVE como CALM están adaptados para usar IP y sus protocolos superiores (TCP y UDP) pudiendo operar a través de redes de ordenadores o Internet.

Dentro del dominio ITS, la parte referente a la comunicación entre las diferentes ITS-S vehiculares y entre estas y las RSU fue abordada por la IEEE como una ampliación de sus protocolos de conexiones inalámbricas 802.11 con el nombre de 802.11p y conocida como WAVE (*Wireless Access in Vehicular Environments*). Este estándar, definido en el conjunto de

normas 1609, se centra en el sistema ITS desde el punto de vista de la ITS-S de un vehículo, concretamente su OBU.

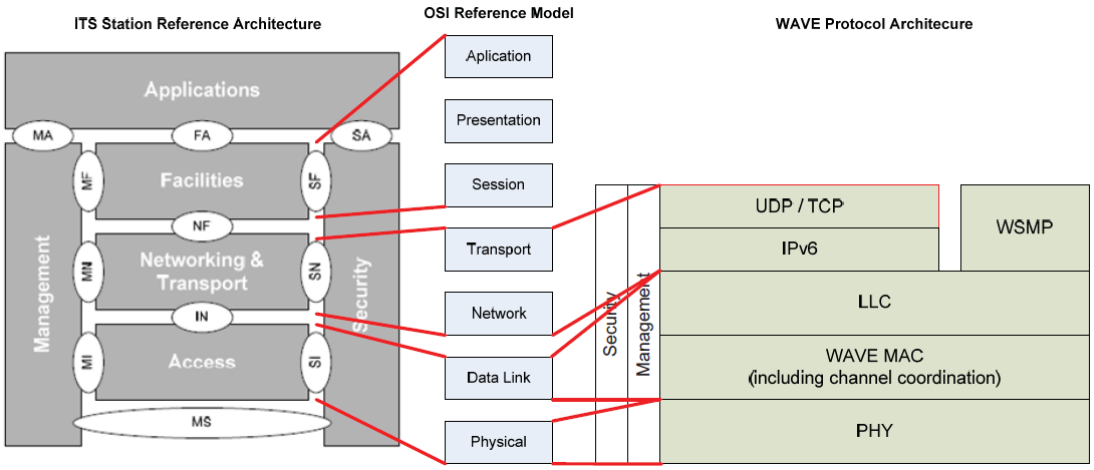


Figura 11: Equivalencia entre la torres de protocolos propuestas por CALM y WAVE.

El Instituto de Ingeniería Eléctrica y Electrónica, IEEE por sus siglas en inglés (*Institute of Electrical and Electronics Engineers*) es la mayor asociación profesional de ingenieros y técnicos del mundo, con más de 425.000 miembros afiliados. Su objetivo como organización es impulsar el avance tecnológico, poniendo éste al servicio de la humanidad.

De esta organización salió el proyecto 802, en donde se han normalizado las capas inferiores de los modelos usados en las comunicaciones entre equipos informáticos. Dentro de este set de normas, se define 802.11 para las comunicaciones inalámbricas entre ordenadores (comúnmente llamado WiFi) siendo el estándar 802.11p una ampliación de éste para las comunicaciones en un ITS. A raíz de esto último, muchas de las características que se definen para los routers ITS provienen directamente de los routers WiFi con el objetivo de maximizar su compatibilidad y ayudar a integrar los nuevos sistemas con los ya existentes. Varias implementaciones experimentales han partido de las normas 802.11 para conseguir implementaciones CALM. Las diferencias más notable entre los routers usados en redes inalámbricas de ordenadores y las OBU están en las bandas de frecuencias usadas y en la introducción de un nuevo protocolo para mensajes de aplicaciones vitales en materia de seguridad y control del propio ITS.

2.4.1 Estructura WAVE

El estándar WAVE se aporta más como una extensión de los protocolos WiFi usados en redes más que como un estándar independiente. Esto consigue en primer lugar que su funcionamiento sea relativamente fácil de entender, al ser una variación de uno muy usado[3]. Por

[3] Las explicaciones dadas aquí toman de base el funcionamiento de WiFi.

otra parte, facilita que los fabricantes puedan empezar al emplear el estándar a partir de modificaciones de hardware y/o firmware de productos ya desarrollados.

Las comunicaciones V2V y V2I sin embargo no se ven sometidas exactamente a las mismas condiciones que las existentes en una red de ordenadores, teniendo que distinguir entre diferentes tipos de tráfico o servicios pudiendo depender la vida de personas del buen funcionamiento de alguno de ellos. Las condiciones de movilidad son más extremas y el número de nodos cercanos (con los que se puede tener comunicación directa) puede variar enormemente de un instante a otro. El sistema debe poder soportan gran cantidad de tráfico manteniendo las comunicaciones esenciales con el mínimo retraso. Las principales diferencias entre WAVE y WiFi vienen recogidas en la Tabla 1.

	WAVE	**Wi-Fi**
Tasa de transmisión	3-27 Mb/s	6-54 Mb/s
Alcance	< 1000 m	< 100 m
Potencia trasmitida	760 mW (US) 2 W EIRP (EU)	100 mW
Ancho de banda del canal	10 MHz 20 MHz	1-40 MHz
Ancho de banda total	75 MHz (US) 30 MHz (EU)	50 MHz para 2.5 GHz 300 MHz para 5 GHz
Mobilidad	Alta	Baja
Banda de frecuencia	5.86-5.92 GHz	2.4 GHz, 5.2 GHz
Estandares	IEEE, ISO, ETSI	IEEE

Tabla 1: Comparativa entre WiFi y WAVE.

2.4.2 **Espectro de los 5,9 GHz**

La Comisión General de Comunicaciones del gobierno de los Estados Unidos estableció la banda de los 5,9GHz para todas las transmisiones DSRC (*Dedicated Short Range Comunication*), incluyendo en este grupo las comunicaciones radios de los ITS. El mismo término DSRC es usado en muchos ámbitos para referirse a las comunicaciones WAVE, esperándose que estas mismas frecuencias sean adoptadas por el resto de países. Las frecuencias habilitadas van desde los 5,85 GHz hasta los 5,929 GHz, estando definidos en WAVE los canales en el rango desde el 172 al 184.

Estos canales se dividen y clasifican en la norma para dos tipos de uso: Un canal para información de control (CCH) y varios canales para servicios de la red ITS (SCH). Los canales y los usos que tienen habilitados se pueden consultar con detalle en la norma 1609.4.

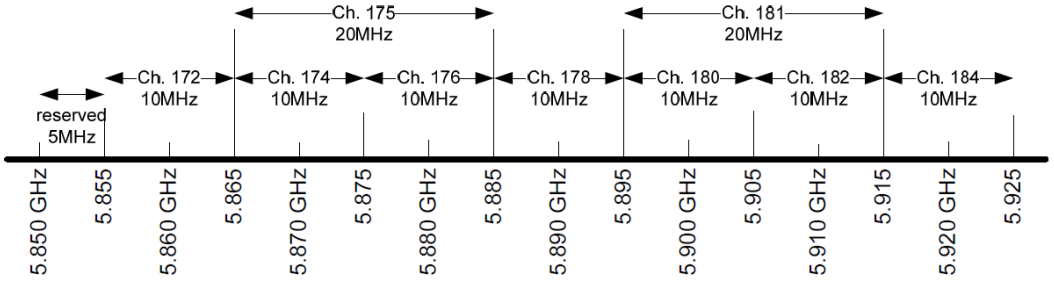


Figura 12: Canales reservados para WAVE. IEEE 1609.0.

2.4.2.1 **Canales de servicio**

Los canales de servicio de WAVE corresponden al 172, 174, 176, 180 y 182 del espectro radioeléctrico. No todos los canales de este tipo pueden ser usados por las aplicaciones de los usuarios, habiéndose designado los canales 172 y 184 para comunicaciones de seguridad. El primero de estos se habilita para comunicaciones de emergencia V2V (como puede ser la elusión de colisiones) mientras que el segundo se reserva para las comunicaciones prioritarias a largas distancias V2I (por ejemplo el aviso de accidente a los servicios de emergencia). El resto de canales, pueden ser usados para aplicaciones "no vitales", no siendo definido todavía qué servicios ofrecerá cada canal.

Tanto los canales 174 y 176 como el 180 y el 182 se pueden unir para crear un canal con el doble de ancho de banda y mayor velocidad.

WAVE admite en estos canales tanto comunicaciones IP (usando IPv6) como comunicación no IP. El soporte de esta última viene dada por el protocolo WSMP (*WAVE Short Message Protocol*) definido específicamente para este estándar.

Los servicios ofertados por estos canales pueden estar preconfigurados en los routers ITS, conociendo de antemano qué dirección, protocolos y frecuencias usar (aplicaciones de emergencias) o configurados de forma externa a través del canal CCH mediante mensajes WSA (*WAVE Service Advertisement*) desde donde se obtiene la información necesaria.

2.4.2.2 **Canal de control**

La función del canal de control es la de coordinar el acceso a los servicios ofrecidos en los canales SCH, siendo anunciada la información básica de los servicios disponibles a través de éste. En principio, en este canal únicamente se permite WSMP. La única excepción a esta norma son los mensajes de anuncio de servicios IP, WSA. Este protocolo permite el control directo de la capa física (potencia y frecuencia usadas) desde las capas superiores pudiendo seleccionar las más adecuadas para transmitir. WSMP está orientado al envío de mensajes simples (sin establecimiento de conexión) tanto de forma *multicast* como *unicast* y con el menor consumo de ancho de banda posible. Debido a que CCH es un canal de información general a todos los

vehículos, sus mensajes serán principalmente anuncios enviados por difusión aunque se permiten comunicaciones extremo a extremo puntuales.

2.4.3 Capa de enlace

La capa de nivel 2 definida en WAVE es prácticamente la misma que la usada por el resto del estándar 802.11, teniéndose una dirección de 48 bits única por tarjeta de red y una capa de adaptación LLC. La diferencia radica en que se han extendido las primitivas de servicio usadas por las capas superiores, de forma que ahora puede controlarse directamente desde éstas el canal usado, la velocidad, la potencia y el tiempo de expiración de la transmisión.

Estos nuevos campos son usados por el protocolo introducido, WSMP.

2.4.4 Protocolos de red

Tal como se ha comentado antes, WAVE soporta comunicaciones basadas en IP y en no-IP. Para las primeras el protocolo usado es IPv6, mientras que para las segundas se introduce en este estándar el protocolo WSMP. La estructura del modelo OSI permite desensamblar unas capas de otras, volviendo a las capas inferiores (MAC y física) independientes de la usada en este nivel.

2.4.4.1 Protocolo IPv6

IP (Internet Protocol), fue diseñado como un protocolo para la interconexión de diversas redes de ordenadores, permitiendo el tráfico de paquetes entre un equipo que pertenezca a una red a otro que pertenezca a una segunda. Desde su aparición, la versión 4 de este protocolo se ha extendido hasta convertirse en el principal protocolo de encaminamiento usado a nivel mundial.

La versión 6 está actualmente en implantación, aportando soluciones y nuevas funcionalidades sobre la anterior:

- Capacidades extendidas de direccionamiento: Se hacen uso de más bits para el campo de direcciones (64), lo que eleva el número de direcciones posibles al cuadrado de las que ya había.
- Se simplifican las cabeceras para reducir el tráfico no útil de las conexiones.
- Se añaden extensiones nuevas de la cabecera, permitiendo un mejor encaminamiento y control de los flujos de información.
- El reparto de direcciones y el uso de los prefijos se mejora para hacer un uso más eficiente de estas y agilizar la conmutación de los paquetes.

El protocolo IP da soporte a los protocolos TCP y UDP, orientado y no orientado a conexión respectivamente. El uso de TCP/UDP sobre IP dentro de WAVE y del ITS en general

permite un encaminamiento y uso de los servicios de Internet en la red de vehículos. Los servicios que usan este protocolo se identifican mediante el par IP/Puerto.

2.4.4.2 **Protocolo WSMP**

WSMP es producto de la colaboración entre la IEEE y SAE (*Society of Automotive Engineers*). Con su creación se buscó un protocolo para comunicaciones de baja carga y latencia dentro de una red V2V/V2I. Está indicado para comunicaciones de un solo mensaje (este lleva todos los datos) y para comunicaciones intermitentes, con un gran intervalo de tiempo entre un contacto y el siguiente. WSMP hace uso de las directivas extendidas en el nivel de la capa de enlace, pudiendo seleccionar directamente la frecuencia y potencia de transmisión más adecuadas para cada instante. El bajo consumo de capacidad del canal del que hace uso, permite que pueda ser usado tanto en los canales SCH como en el CCH.

Los mensajes WSM (*WAVE Short Message*) usualmente van dirigidos a una dirección MAC *broadcast* con un identificador de servicio PSID de forma que la ITS-S receptora pueda comprobar si el mensaje es de su interés e ignorarlo en caso contrario. Una vez recibido uno de estos mensajes, se dispone de la dirección y nodo origen pudiendo establecer comunicación con éste mediante *unicast* (usado la MAC del OBU destino) o *multicast* según proceda.

2.4.5 **Capas superiores**

Las capas superiores a la de transporte TCP/UDP (en el caso del stack IP) y por encima de WSMP (para el caso no IP) están en proceso de ser definidas, previéndose su publicación en la norma 1609.11.

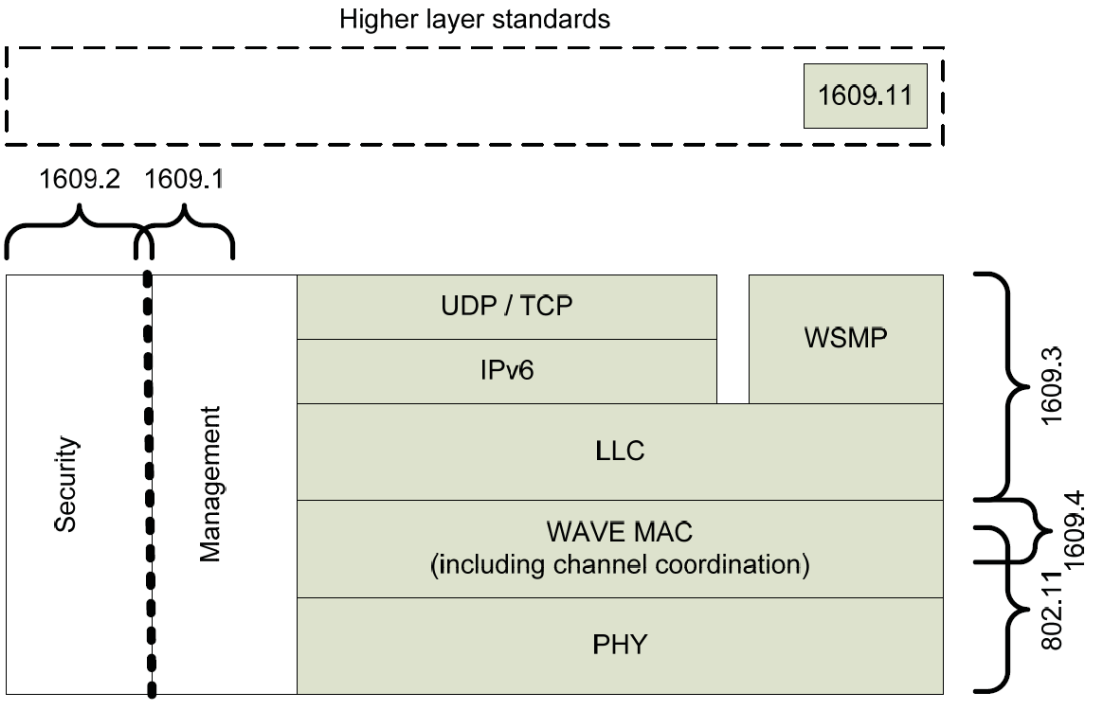


Figura 13: Capas de WAVE y normas donde están definidas. IEEE 1609.0.

2.4.6 **Servicios ofertados**

Dentro de los servicios ofertados por una RSU o por un nodo móvil, estos pueden ser (como ya se ha comentado) preconfigurados o configurados a través del canal CCH u otro medio fuera de este estándar (IR, 2G, 3G…). Para el segundo caso, se han definido los mensajes WSA (*WAVE Service Advertisement*). Este mensaje, bastante parecido al usado en el protocolo FSAP (*Fast Service Advertisement Protocol*) descrito por la ISO para los sistemas ITS, es enviado por el proveedor del servicio (la RSU en muchos casos) a *multicast* y contiene información sobre los servicios que se ofrecen, identificados estos por su PSID, y en qué canal SCH están disponibles. Los usuarios del servicio pueden entonces cambiar al canal SCH definido en el WSA para, a través los protocolos de propios del servicio, hacer uso de este o bien pedir a través de un WSA hacia el proveedor más información sobre un servicio ofertado. Esta clase de mensaje puede ser transmitido tanto a través de IP como de WSMP.

Cuando el servicio es ofertado por la infraestructura (normalmente una RSU) ésta toma el papel de proveedor, mientras que los nodos móviles toman el papel de clientes. El proveedor transmite los mensajes WSA de un servicio ofertado tanto por el canal CCH como por el canal SCH donde se ofrece el servicio. Los protocolos usados en cada servicio deben ser conocidos por los usuarios para acceder a ellos. En el caso de que el mensaje WSA contenga una dirección IP, se supone que el servicio se ofrece sobre este protocolo, en caso contrario, se supondrá que es a través de WSMP.

Una vez solicitado acceder al servicio, será la capa de control quien se encargue de negociar con el proveedor las condiciones de uso.

En el caso de usar IPv6 para los servicios, es posible no solo acceder a los servicios ofrecidos por la red ITS sino también a los existentes en Internet a través de esta red. Esta opción también está planteada en WSMP, aunque es necesario traducir a IP en uno de los routers ITS.

Para el caso de que no se dispongan de infraestructura, hay determinados servicios (como los descritos para el canal 172) que pueden ser usados en modo P2P (*peer-to-peer*). En estos casos se establece una tabla de vecinos conocidos. Esta tabla se construye a partir de los mensajes WSA enviados a difusión recibidos de los nodos que están al alcance.

2.4.7 **Control de aplicaciones y servicios**

Dentro de WAVE se define un sistema de prioridades a la hora de transmitir un mensaje. Estas prioridades indican qué aplicaciones tienen preferencia a la hora de transmitir en la red ITS, pudiendo ser anunciada esta preferencia en el WSA.

Además de este sistema, existe una segunda cola de prioridades que es marcada a nivel de MAC, estableciendo qué equipo tiene preferencia a la hora de transmitir. También es posible establecer prioridades a nivel IP dependiendo del tipo de tráfico que lleve. En todo caso, usando WSMP las prioridades se marcan siempre a nivel MAC.

Alcanzado este punto, si se recapitula, se puede comprobar que se está trabajando con 4 canales para aplicaciones varias, uno de control y 2 de seguridad en carretera, lo que hace un total de 9 canales. Puesto que un router ITS se supone que escucha en una única frecuencia por antena ¿Cuántas antenas o routers son necesarios por vehículo en un ITS? La repuesta a esta pregunta es uno.

Para usar WAVE en una red se requiere atender al menos a un canal SCH, el de seguridad V2V, y al canal CCH, donde se envía la información de control. Para poder usar routers simples de una sola antena, WAVE divide el tiempo de transmisión de uno y otro tipo de canal en intervalos, de forma que no se transmita información vital simultáneamente en los dos tipos de canales a la vez. Una tarjeta de red puede escuchar durante el intervalo de tiempo establecido el canal CCH para cambiar (durante el de tiempo que se deja entre ambos, establecido en 50 ms) a escuchar el canal SCH.

No todos los servicios deben poder soportan este modo “alternante”, requiriendo alguno que el OBU permanezca en el canal un tiempo mayor a la duración del intervalo (modo “extendido”) o cambiar a una frecuencia concreta en el momento (modo “inmediato”). Este dato se transmite en el mensaje WSA del servicio. Por supuesto, con este método no se puede hacer uso de varios servicios a la vez y aparte se sufre una bajada de rendimiento debido precisamente a esta alternancia.

Se recomienda implementar dos tarjetas o antenas de forma que una de ellas pueda permanecer conectada continuamente a una frecuencia (modo “continuo”) realizando las comunicaciones que no puedan ser soportadas en el modo alternante.

Para realizar la escucha con "multiplexación en el tiempo" de cualquiera de los modos excepto el continuo, es necesario que los nodos del sistema estén coordinados, disponiendo del tiempo UTC o equivalente. Este tiempo se puede conseguir a través de una tarjeta conectada de forma continua a una frecuencia (por ejemplo la de emergencias) o bien a través de medios externos como el GPS.

2.4.8 **Otras consideraciones**

Lo visto hasta ahora de WAVE corresponde únicamente al plano de usuario, correspondiente a uso y diseño de aplicaciones para ser implementadas en ITS-S. Del plano de control y de seguridad no se ha mencionado nada. Ambos planos corresponden a sendas capas trasversales de protocolos. Estas capas están ligadas al manejo de información en todos los niveles, por lo que disponen de primitivas de servicio a disposición del resto de las capas. Debido a su extensión y complejidad (normas 1609.2 y 1609.6 de la IEEE) se ha decidido no profundizar en ellas en este trabajo. Sus principales funciones son las mismas que las descritas en el apartado 2.2.1Comunicaciones CALM.

2.5 **Implantación elegida para el proyecto**

En este trabajo, aparte de estudiar las normas ISO-CALM y WAVE-IEEE, se realiza un ejemplo básico de instalación de unas placas comerciales para su uso en un sistema ITS. La parte emulada del ITS corresponde a la comunicación V2V producida entre dos routers ITS correspondientes a dos ITS-S vehiculares.

Este proyecto tiene como punto de partida el trabajo presentado por el proyecto europeo GCDC (*Grand Cooperative Driving Challenge*) en 2011. Entre las conclusiones del este proyecto estaba la adaptación de un sistema operativo de código libre llamado OpenWrt para el uso de las frecuencias de comunicaciones DSRC (*Dedicated Short-Range Communications*) especificadas por la FFC (*Federal Communications Commission*) y adoptadas por los estándares para ser utilizadas en los ITS, la banda de los 5,9 Ghz.

El proyecto fin de carrera de José Manuel Sampedro Armenta, titulado "Comunicación cooperativa entre vehículos: Estudio e implantación del protocolo ISO CALM en un sistema embebido usando OpenWrt" aborda en profundidad una comparativa entre los principales proyectos en la materia y describe el método de implementar este sistema operativo en una placa AlixBoard 3D2.

2.5.1 **Proyecto GCDC**

El proyecto GCDC está enfocado al desarrollo e implementación de tecnologías de conducción cooperativa, es decir, hacer partícipe a las infraestructuras y a otros vehículos de la

actividad y movimiento del automóvil. El planteamiento está basado en aprovechar la infraestructura actual, dotando a ésta y a los vehículos de comunicación inteligente, permitiendo que los vehículos puedan adaptarse unos a otros para transitar manteniendo menores distancias entre ellos, a la par que se garantiza una circulación más fluida. Con ello se pretende un uso más eficiente del sistema ya existente, lo que se traduciría en un alivio parcial de los problemas de tráfico actuales.

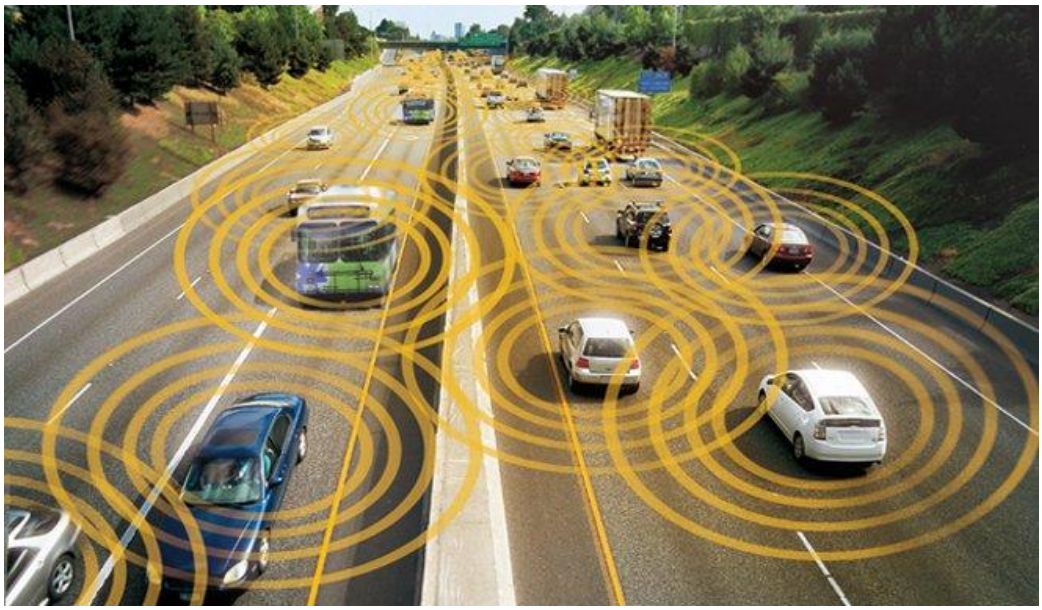


Figura 14: Conducción cooperativa propuesta por el GCDC.

El GCDC está orientado a la competición entre diferentes equipos para conseguir el sistema más eficiente en escenarios predeterminados de tráfico. Anunciado en 2008, el programa no pudo echar a andar hasta 2009. En 2011 se convocó un evento para que los diferentes participantes mostrasen sus avances.

Entre las conclusiones de este evento está un modelo de plataforma inalámbrica 802.11 modificado para cumplir el estándar 802.11p. Este modelo propone el uso de un hardware comercial sobre el que se instala un sistema operativo (SO) Linux cuyo kernel ha sido modificado para que soporte las nuevas frecuencias. Estas modificaciones se realizan sobre el módulo CRDA (*Central Regulatory Domain Agent*).

2.5.1.1 **Dominios reguladores**

El CRDA es un módulo presente en el SO con comunicación inalámbrica o en el driver de la tarjeta en el caso de que el SO no tenga instalado esa parte del kernel. Su función es la de mantener las frecuencias y potencias de transmisión inalámbricas usadas dentro de los límites legales establecidos en la región donde se usa.

Este módulo parte el archivo “db.txt”, el cual está dividido es secciones identificadas por un código de dos letras que representan el país al que se refieren. Dentro de estas secciones se encuentran las bandas de frecuencias que se pueden usar, la potencia máxima de transmisión para cada banda y sus limitaciones en el uso (por ejemplo, solo en modo ad-hoc). El archivo usado por el SO es el “regulatory.bin”, resultante de compilar y firmar digitalmente el “db.txt”. Cada vez que la interfaz radio cambia de configuración, se consulta al “regulatory.bin” comprobando si cumple las condiciones para la región establecida. Esta región en principio es tomada automáticamente por el sistema, pero se deja al usuario pasar a otra mediante comandos.

El GCDC modificó la versión 10.03 del sistema operativo OpenWrt para que el CRDA permitiese el uso de la banda de los 5,9 GHZ en el dominio regulatorio de los países bajos NL (*Netherland)*, lugar donde se celebró el evento de 2011.

2.5.2 **Sistema implantado**

Dentro del contexto descrito, se implementa un sistema de comunicación funcional entre dos routers ITS. Estos routers son dos placas comerciales AlixBoard 3D2 sobre las que se ha instalado la distribución modificada de OpenWrt. Sobre este sistema operativo se ejecuta una aplicación programa en C que simula un posible servicio ITS de seguridad en carretera.

Como conceptos prácticos se inicia a la instalación y puesta en marcha de OpenWrt sobre una AlixBoard 3D2 y se introduce al desarrollo y compilación de aplicaciones para OpenWrt. Se hace hincapié en el diseño de programas enfocados a la implementación de servicios en V2V siguiendo las directrices WAVE, desarrollándose uno de alerta de peligros en carretera.

Debido a las limitaciones de las AlixBoard, se ha tenido que simplificar algunas de las características del sistema implementado respecto al ideal marcado por la normativa de referencia. También existen diversas limitaciones que provienen de la distribución del GCDC. Aun así, se mantiene lo siguiente:

- La frecuencia de comunicación se sitúa en la banda de 5,9 GHz tal como se establece para las DSRC.
- Se usan los protocolos de internet TCP/UDP e IP, emulando el programa un posible servicio ofrecido entre vehículos.
- El router ITS no sabe a priori las direcciones IP o MAC del resto de vehículos, por lo que se ha implementado un sistema de anuncios para descubrir nuevos nodos al alcance. En este caso se han usado mensajes UDP mandados a la dirección de difusión de la red y TCP para el intercambio de datos.
- El sistema ofrece un servicio de aviso en caso de peligro interpuesto en el trayecto actual y de compartición de información propia con otros nodos. Estos servicios hacen uso de uno puertos determinados para escucha y establecimiento de conexión.

- La aplicación se ha diseñado para que sea escalable, pudiendo estar usándose con un número moderado de nodos a la vez. Desgraciadamente debido a que solo se disponen de dos placas AlixBoard no se ha podido comprobar esta escalabilidad de forma cuantitativa.
- El hardware también se ha seleccionado para que se adecue a una posible implantación real. Las AlixBoard son placas base relativamente baratas a las que se le pueden añadir uno o varias tarjetas inalámbricas (Mikrotik en este caso). Se ha instalado un sistema operativo embebido en las placas, OpenWrt, de forma que esta pueda procesar por su cuenta las operaciones propias de un router ITS (manejo de servicios, comunicaciones de control, alertas de seguridad, peticiones de comunicación desde otros módulos…)

Sin embargo, esta implementación se ha visto limitada en ciertos aspectos:

- No se hace uso del protocolo WSMP debido a que OpenWrt en la versión usada no lo soporta. Habría que actualizar el firmware para poder usar aplicaciones de control y seguridad según el formato WAVE. Se considera que el servicio ofrecido esta preconfigurado en el router ITS al ser en este caso una aplicación de seguridad.
- Por limitaciones de software, la tarjeta no puede ponerse en modo alternancia (no puede estar cambiando entre SCH y CHH a intervalos). Aquí se simplifica usando siempre la misma frecuencia de comunicación, luego se supone que se está trabajando en modo continuo.
- La versión usada de IP es la 4 en vez de la 6. Esto es debido nuevamente a que la versión de OpenWrt usada tiene problemas al pasar de una a otra. El programa se podrá adaptar fácilmente al uso de IPv6 cuando éste esté disponible en implementaciones sobre firmware más actual

  .

3 Hardware usado

In theory there is no difference between theory and practice;
in practice there is.

Atribuida a Jan L. A. van de Snepscheut

El próximo capítulo trata sobre los dispositivos que conforman el sistema desarrollado y los usados para llevar a cabo su configuración. En este apartado se detallará qué equipos han sido escogidos para desarrollar este proyecto, sus características, el escenario montado y la forma de uso. La elección de los dispositivos del sistema se basó en buscar un equipo que pudiese llevar a cabo las funcionalidades requeridas en un OBU perteneciente a un ITS y que tuviese un precio lo suficientemente bajo como para poder proponerlo como solución comercial. Partiendo de trabajos previos se han introducido modificaciones sobre el esquema ya propuesto, que facilitan en gran medida el manejo del sistema durante su uso. Con estos cambios se ha buscado conseguir un entorno fácil de usar, montar y guardar; y que además pueda ser transportado con facilidad.

3.1 Dispositivos hardware usados

Dentro del montaje usado en el laboratorio se puede distinguir entre los dispositivos que formarían parte del sistema ITS, y que por tanto irían montados en una OBE o una RSU, y lo usado para configurar el sistema y realizar las pruebas de funcionamiento.

Dentro de la primera categoría encajan las dos placas base AlixBoard 3D2 y sus respectivas tarjetas radio Mikrotik R52H mini-PCI. Tal como se mencionó en el capítulo introductorio, la parte del sistema a desarrollar es la comunicación entre dos routers ITS pertenecientes a dos vehículos de un ITS. Para ello es necesario poder transmitir en la banda de 5,9 GHz, tal como establece WAVE siguiendo la legislación para comunicaciones DSRC. Además, es necesaria una plataforma (un firmware o un sistema operativo) donde se pueda ejecutar el código de los servicios y aplicaciones creadas. Por último, el OBU necesita conectarse al resto de elementos del ITS-S y poder ser controlado durante las pruebas de forma remota. Los conectores de la placa empleada y las tecnologías con las que sea compatible pueden condicionar este aspecto. Tanto las AlixBoard como las Mikrotik se han usado anteriormente para este tipo de proyectos. Puesto que cumplen con lo que expuesto, se justifica continuar con este hardware.

También se recomienda que los componentes usen estándares internacionales. Esto reduce los problemas de compatibilidad al ser sustituidos por otros con iguales o mejores prestaciones.

3.1.1 AlixBoard 3D2

Las placas AlixBoard 3D2 son fabricadas por la compañía *PC Engines*, dedicada a la producción de placas multifunción basadas en microprocesadores AMD Geode x86. La serie Alix.3 reduce a uno el número de puertos LAN de sus predecesoras, aumentando los puertos de expansión miniPCI a 2. Con su procesador de 500 MHz y sus 256 MB de memoria RAM, son una solución de bajo coste en entornos poco exigentes.

Las AlixBoard cumplen la función de placas base de los routers ITS. Poseen un puerto LAN que puede ser usado para controlarlas a través del protocolo SSH y permiten el uso de tarjetas de memoria CompactFlash, para almacenamiento de información, donde se le puede instalar un sistema operativo.

A la hora de usarlas, estas placas requieren alimentación en el rango de 7 a 20 V DC. Su consumo es de 2,5 a 3,5 W con picos de hasta 5 W sin periféricos conectados.

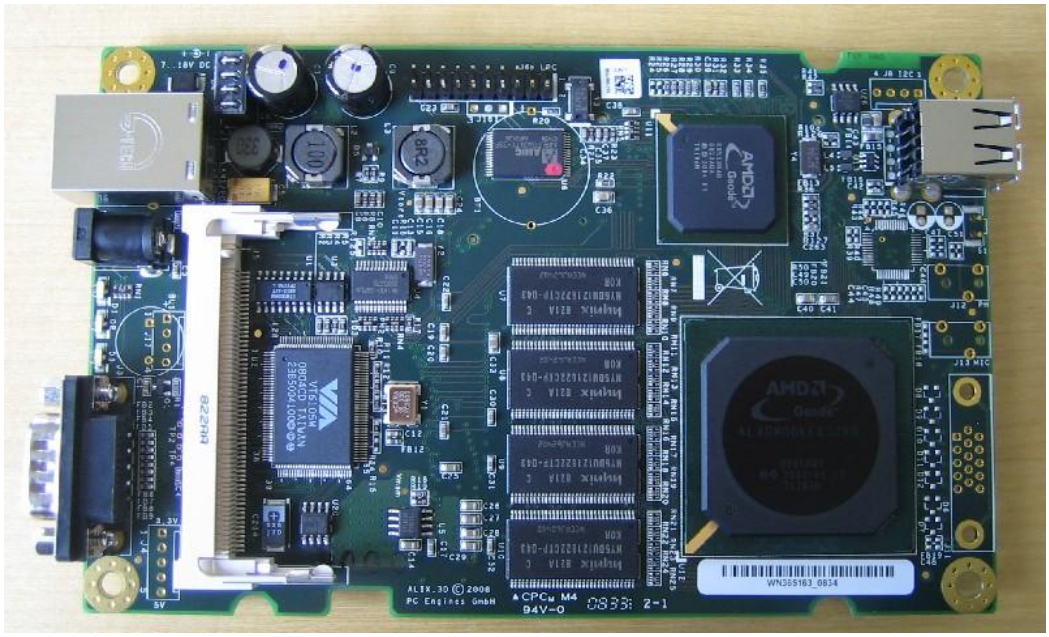


Figura 15: Alix Board 3D2, Parte delantera.

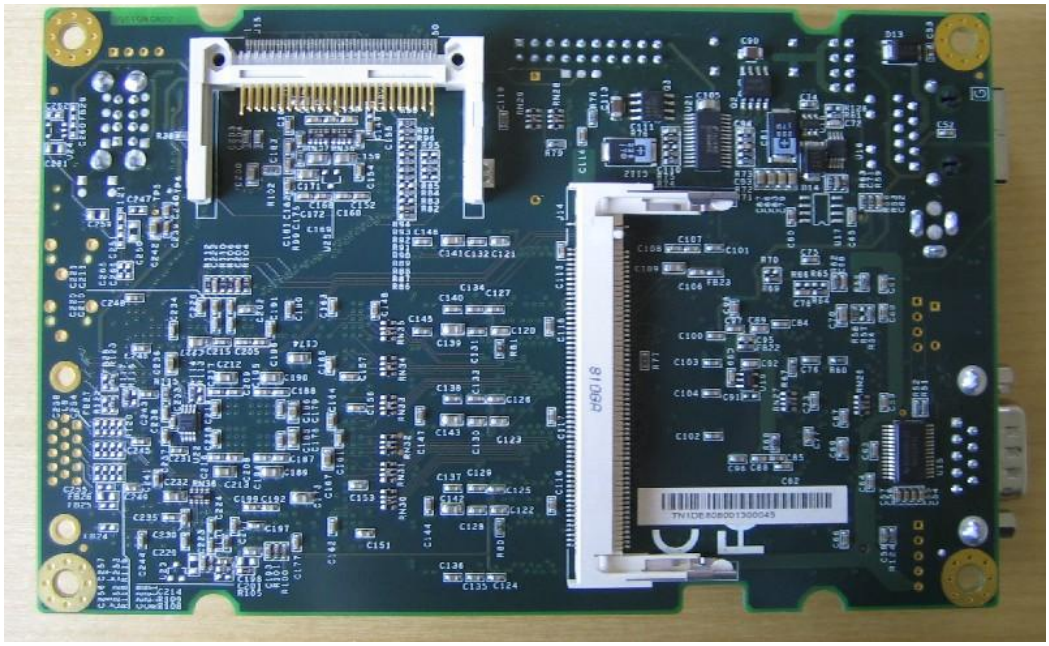


Figura 16: Alix Board 3D2, Parte trasera.

Las principales especificaciones técnicas de la tarjeta usada se muestran en la Tabla 2:

CPU	500 MHz AMD Geode LX800
DRAM	256 MB DDR DRAM
Almacenamiento	Slot CompactFlash
Alimentación	DC jack o POE pasivo, min. 7V to max. 20V
LEDs	Tres
Expansión	2 miniPCI slots, LPC bus
Conectividad	1 Ethernet (Via VT6105M 10/100)

I/O	DB9 puerto serie, dos USB
Dimensiones	100 x 160 mm
Firmware	tinyBIOS

Tabla 2: Especificaciones Técnicas AlixBoard 3D2

3.1.2 **Mikrotik R52H**

La Mikrotik R52H es una tarjeta radio miniPCI para conexiones inalámbricas. Soporta frecuencias en las bandas 2,192-2,539 y 4,920-6,100 GHz radiando con una potencia máxima de 350 mW, características que la hacen adecuada para su uso en una ITS-S, permitiendo a las AlixBoard 3D2 comunicarse en la banda de 5,9 GHz.

La tarjeta cuenta con dos conectores para el uso de antenas externas, permitiendo incrementar su alcance y la sensibilidad en la recepción.

Figura 17: Mikrotik R52H

Las principales características del dispositivo se muestran en la Tabla 3:

Procesador	AR5414
Frecuencias	2.192-2.539MHz / 4920-6100Mhz
Estándares compatibles	IEEE802.11a/IEEE802.11b/IEEE802.11g
Potencia máxima de salida	25dBm
Formato	miniPCI
Dimensiones	6.0cm x 4.5 cm
Conectores	2x uFl
Rango de temperatura de trabajo	-20C to +70C
Alimentación	3.3V +/- 10% DC; 800mA max

Tabla 3: Especificaciones Técnicas Mikrotik R52H

3.1.3 Otros dispositivos

Los dispositivos descritos hasta ahora son los que forman el núcleo del router ITS-S (Placa base más tarjeta radio), los numerados a continuación forman parte del entorno de desarrollo y pueden ser encontrados o no en una implementación real dentro de un vehículo o en una RSU. Para facilitar la continuación de esta línea trabajo en un futuro, se ha procurado usar equipos fácilmente adquiribles:

- **Fuente de tensión compacta:** La serie Alix.3 acepta tensiones de 7 a 20 V en su alimentación siendo 18 V lo recomendado. La potencia pico sin el uso de miniPCI es de 5 W, siendo 15 W lo recomendado por tarjeta. En el montaje se ha usado una fuente de tensión compacta a 18 V con una capacidad que excede los 30 W.
- **Ordenador:** Para la instalación del sistema operativo y su control remoto durante la configuración, es necesario disponer de un equipo (un portátil es más cómodo) que tenga instalado un sistema operativo Linux. Los requisitos y el software usados se verán en el capítulo siguiente.
- **2 Compact Flash (4GB) y un lector de tarjetas externo** Las CF son empleadas para almacenar el SO usado, además de cómo disco duro de la placa. Es necesario un lector CF/USB para poder acceder a su contenido desde el ordenador. Se ha utilizado el lector appCRDNI de la marca aqprox.
- **Cables Ethernet con conectores RJ-45:** Es el cable usado para conexiones LAN en redes de ordenadores. Es necesario al menos uno para poder conectarse al sistema en funcionamiento por medio del protocolo SSH y poder ejecutar comandos de forma remota, editar los ficheros de configuración y, en general, realizar pruebas.
- **Switch:** Este dispositivo no es imprescindible y es un añadido al escenario que ya existía. Permite estar conectado simultáneamente por SSH a las dos AlixBoard durante su funcionamiento, lo que es muy cómodo para la configuración y la ejecución de pruebas.

3.2 Montaje de los dispositivos

El entorno de desarrollo consiste en dos AlixBoard 3D2 equipadas cada una con una tarjeta CompactFlash de 4 GB y una Mikrotik R52H. Ambas placas se alimenta mediante una fuente de tensión compacta regulada a 18 V y que soporta los 30 W recomendados por el fabricante (15 W cada una).

Para dar robustez al conjunto y facilitar su manejo, tanto las placas como la fuente de tensión han sido fijadas en un tablero de metacrilato.

Una vez enchufada la fuente de tensión y conectadas las AlixBoard a ésta, las placas inician automáticamente el sistema operativo montado en las CF. Las conexiones con las placas en funcionamiento se realizan mediante los cables Ethernet, usando los puertos LAN del PC y las placas. El ordenador usado tiene un único conector LAN, por lo que para estar conectado a la vez a ambas AlixBoard se ha usado un switch, conectando el ordenador y cada placa a un puerto distinto. Una vez hecho esto, se debería tener conectividad con las placas siempre y cuando las direcciones IP configuradas en éstas y en el ordenador pertenezcan a la misma subred.

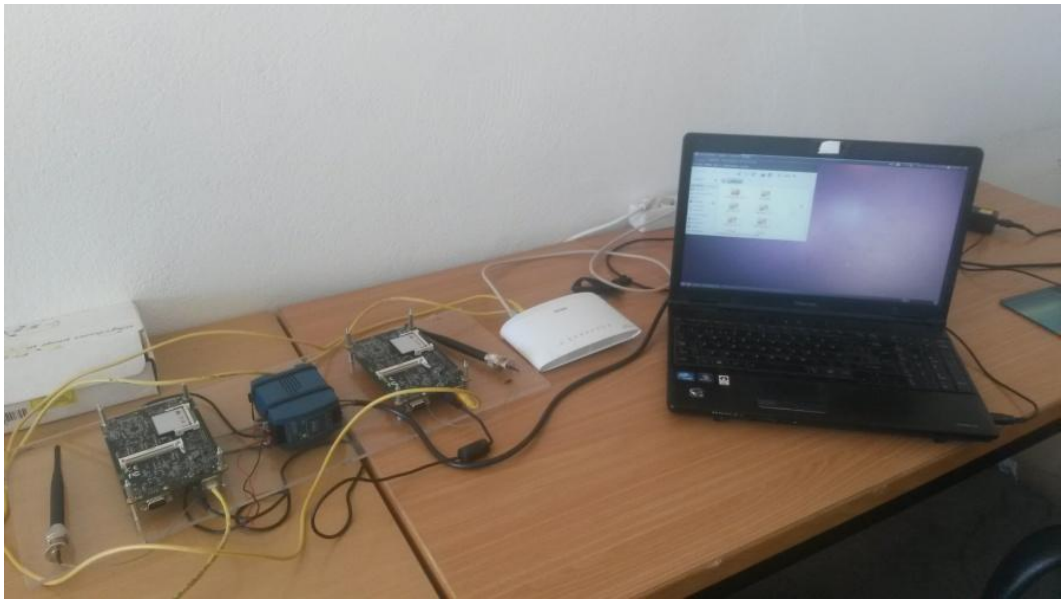


Figura 18: Montaje realizado en el taller.

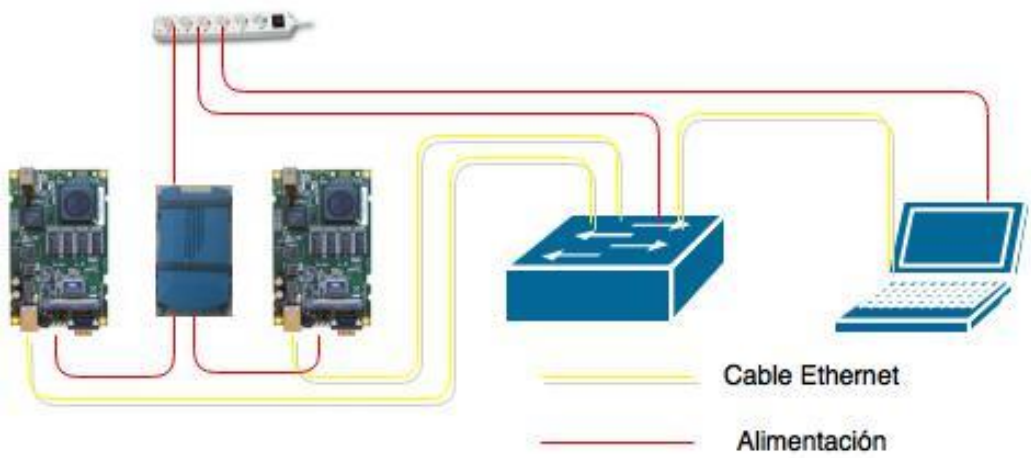


Figura 19: Esquema del montaje realizado.

A la hora de instalar el SO en la CF o de editar archivos de configuración del OpenWrt (porque no se quiera trabajar por SSH o porque sea algo que no se pueda modificar con el sistema en funcionamiento) puede usarse el lector de tarjetas CF. Conectándolo al puerto USB del ordenador, es reconocido como un dispositivo de almacenamiento externo.

3.3 Consideraciones sobre el hardware usado

El trabajo con una placa como la AlixBoard conlleva una serie de limitaciones derivadas de estar diseñada para aplicaciones específicas (router, firewall, proxy…) en redes de ordenadores. Estas tarjetas caen dentro de lo que se denominan sistemas empotrados o embebidos cuya definición sería la de equipos informáticos con prestaciones reducidas, diseñados para aplicaciones en un campo concreto. Este tipo de equipos comportan una dependencia del hardware bastante mayor que la que tienen equipos de labores más genéricas como los ordenadores personales.

¿Por qué usar entonces una tarjeta como la elegida o similar en vez de un PC? Principalmente por dos razones. La primera es monetaria, mientras que un PC "barato" cuesta varios cientos de euros, una AlixBoard puede encontrarse por unos 90 euros, o incluso menos si se compran en grandes cantidades. La segunda razón es la eficiencia, un ordenador personal es un sistema denominado *multitask* o multitarea, está diseñado para ejecutar todo tipo de programas con multitud de propósitos diferentes. Este tipo de plataformas sacrifican velocidad por compatibilidad con todo tipo de módulos. Mientras, los sistemas embebidos están optimizados para que ejecutar un pequeño abanico de tareas de forma muy eficiente. Las AlixBoard 3D2 son denominadas placas routers, por estar especializadas en la conmutación de paquetes.

4 Software empleado

If you don't know anything about computers, just remember that they are machines that do exactly what you tell them but often surprise you in the result.

Richard Dawkins

Para la instalación y el uso de una aplicación o servicio ITS, el hardware descrito en el capítulo anterior resulta insuficiente por sí solo. Se necesita de un soporte lógico que sirva de punto intermediario con el firmware nativo de la AlixBoard y que permita la ejecución de código desarrollado. El presente capítulo se centra en la parte "virtual" o software del proyecto. Para el uso de la aplicación en la placa elegida, es necesario un sistema operativo compatible con la AlixBoard y que además sea instalable en una Compact Flash. El sistema escogido ha sido OpenWrt, una distribución Linux bajo licencia GPL optimizada para ser usada como firmware en routers. Puesto que su puesta en marcha requiere de conocimientos avanzados en el manejo de los detalles de este sistema operativo y de las tarjetas, en este capítulo se describe el trabajo realizado para llegar a su correcta instalación y configuración, pretendiendo llegar a ser una guía de instalación para trabajos futuros. La intención no es solo limitarse a la descarga, traspaso a la memoria CF y uso del propio OpenWrt, también se entrará en detalle en la compilación e instalación de aplicaciones propias sobre este sistema operativo. Por último, se hablará sobre el servicio creado, argumentando el diseño en función de las peculiaridades de las VANET y su implementación en forma de código en lenguaje C.

4.1 Sistemas embebidos

En el capítulo anterior se ha visto que el hardware usado (AlixBoard 3D2), es considerado un sistema embebido. La instalación de un sistema operativo que permita cierto nivel de abstracción se complica en este tipo de aparatos, requiriendo, en el caso que nos ocupa, de un equipo externo (un portátil o un ordenador fijo) desde donde pasarle el sistema operativo a usar.

Los sistemas operativos embebidos, son distribuciones de software creadas para este tipo de hardware. Trabajan con menos recursos que sus hermanos mayores de PC a costa de estar más limitados en el número de tareas que pueden realizar. La distribución a instalar también tiene que estar adaptado a la arquitectura de la tarjeta, no valiendo cualquier SO embebido. Al ser SSOO especializados, su compatibilidad depende mucho de la estructura que tenga el procesador y placa usados. Los sistemas más populares ofrecen distintas versiones en función del tipo de hardware a usar, por lo que hay que tener especial cuidado a la hora de elegir cuál y qué versión instalar. El SO embebido elegido está diseñado para ser usado como firmware en routers, estando ya comprobado que es compatible con la tarjeta usada.

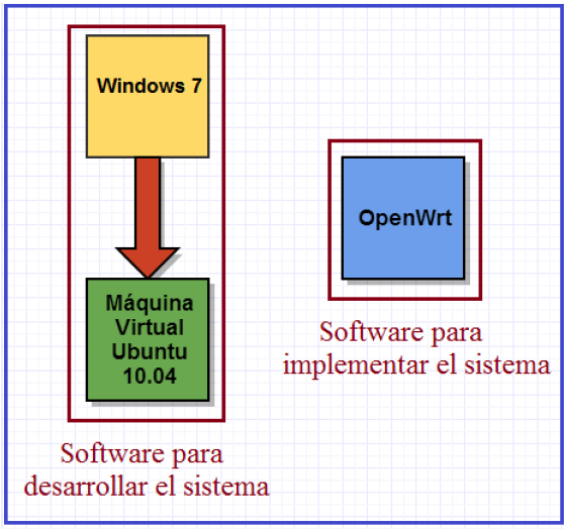


Figura 20: Sistemas operativos usados durante el proyecto.

4.1.1 Plataforma externa

Para la descarga y posterior compilación de lo que después será la distribución que corra sobre las placas AlixBoard, es necesario disponer de un ordenador que cumpla una serie de requisitos que permitan trabajar con el sistema embebido elegido: OpenWrt.

Lo primero que hay que tener en cuenta con el ordenador es el SO del equipo usado. Aunque es posible compilar el código fuente de OpenWrt con una plataforma propietaria (el caso de Windows) se recomienda usar un sistema Linux como puede ser Ubuntu o similar. La razón: estos sistemas ofrecen terminales con mayor cantidad y variedad de comandos

que permiten un control más preciso por parte del usuario de las órdenes a ejecutar. Además, la herramienta usada para personalizar la compilación del sistema, ha sido desarrollada para Linux y el soporte dado tanto en la wiki como en el foro de la página oficial de OpenWrt se centra en este tipo de SSOO. Por otra parte, no es necesario tener un Linux instalado directamente sobre el hardware del equipo, siendo la creación de una máquina virtual, una opción válida y muy cómoda si no se quiere abandonar Windows o recurrir a arranques duales.

4.1.1.1 **Arquitectura usada**

En este caso el equipo de trabajo (con el que se está escribiendo esta memoria) es un Toshiba Intel Celeron P4600 con 3 GB de memoria RAM y un Windows 7 de 64 bits instalado como SO de arranque. Para trabajar en este proyecto se ha usado el software de virtualización VMware player con el que se ha creado una máquina virtual con Ubuntu 10.04 LTS.

4.1.1.1.1 Virtualización del software

VMware player es un programa desarrollado por *VMware Inc*, una empresa fundada en 1998 y que está especializada en software de virtualización. VMware player es usado tanto a nivel particular como de empresas y tiene una licencia gratuita para uso no comercial. Con 7 años de desarrollo, soporta la mayoría de los sistemas operativos actuales instalándolos a través de sus imágenes ISO. El hardware a emular puede ser personalizado de una forma sencilla, pudiéndose especificar qué características técnicas y dispositivos periféricos tiene la maquina emulada.

La virtualización por software permite simular uno o varios ordenadores físicos usando un mismo hardware, lo que permite ahorrar en coste, espacio y energía al poder ejecutar diferentes SSOO a la vez en un mismo equipo, muy común en el ámbito de aplicaciones y servicios web. También permite la portabilidad y el copiado del SO debido a que este se guarda en el sistema anfitrión (sobre el que se virtualiza y donde está instalado el software) como uno o varios archivos manipulables. Por otra parte, emular un sistema suele conllevar un peor rendimiento frente a su instalación directa sobre el disco duro debido a que parte de los recursos existentes en la máquina son usados por el sistema anfitrión tanto para su funcionamiento normal como para mantener la virtualización. Esto se ha ido mejorando a lo largo de los últimos años hasta el punto de que este problema únicamente se hace patente en ejecución de tareas muy pesadas por parte del sistema virtualizado. En el caso actual, puesto que únicamente se usará el SO emulado para la compilación, esto no supondrá una traba importante.

Otro obstáculo que sí que afecta al escenario planteado es el hecho de que la conexiones de red de la máquina virtual no tienen por qué ser “directas” al exterior. Cuando se ejecuta un SO huésped desde otro, las conexiones de red pueden emularse de varias formas. Una de ellas es

usar los mismo puertos UDP/TCP e IP que el SO anfitrión. Este modo de funcionamiento se denomina *"bridge"* y aunque ofrece ventajas como el poder acceder al SO huésped desde una red externa, únicamente permite tener en la máquina virtual los mismos dispositivos de red de los que dispone la máquina física.

El otro modo de funcionamiento es mediante el uso del NAT. Este modo (normalmente es el que suele estar definido por defecto) trata internamente a la máquina virtual como un equipo independiente del sistema anfitrión y a éste como un router en una red local interna. Esta configuración permite, por ejemplo, el uso de una misma conexión (misma IP) para la navegación por internet (usando el WiFi del equipo) y la conexión a las AlixBoard por medio del dispositivo LAN. Esto tiene el inconveniente de que la máquina virtual debe ser la que inicie todas las conexiones con el exterior para ser accesible desde fuera del ordenador. En este modo hay que tener cuidado pues la IP usada por el ordenador es la del SO anfitrión, no teniendo ningún efecto de cara a la conectividad del equipo cambiar la IP de la máquina virtual. Tener esto en cuenta en el apartado 4.2.3 de este mismo capítulo.

4.1.1.1.2 Sistema operativo de trabajo

Terminando con el tema de la virtualización, la elección de Ubuntu como máquina virtual usada se ha debido a varios factores. Ubuntu es un sistema Linux bajo licencia de software libre GNU. Este sistema operativo ha ido evolucionando desde 2004 hasta convertirse en la principal plataforma de software libre tanto para usuarios noveles como promedio debido a que posee una interfaz gráfica amigable y proporciona al mismo tiempo herramientas avanzadas para los desarrolladores. Su gran comunidad, permite que el SO reciba actualizaciones diarias de la versión actual y que salga una nueva cada 6 meses. Cada dos años se lanza una versión LTS (*Long Time Support*) que será mantenida hasta la salida de la siguiente.

La versión actual de Ubuntu es la 14.04 LTS, lo que implica que la usada (10.04 LTS) queda ya bastante desfasada, terminando el soporte LTS de la versión cuando salió la 12.04 LTS. Sin embargo, se ha escogido esta versión debido a que la distribución OpenWrt usada procede del año 2010. El usar un SO del mismo año que la versión del OpenWrt permite reducir al mínimo los problemas de compatibilidad que se puedan dar en la compilación del sistema embebido.

El uso y manejo de Ubuntu quedan fuera del ámbito de este documento, aunque existe una gran comunidad y multitud de guías que se pueden encontrar con una rápida búsqueda por Internet. Mencionar que la arquitectura del sistema de ficheros en casi todos los sistemas Linux es muy parecida (en algunos casos idéntica) incluyéndose en esta afirmación al propio OpenWrt, por lo que los comandos e instrucciones dadas más adelante pueden ser adaptados de forma no excesivamente complicada al uso en otras versiones o distribuciones.

Integrado en casi todas las versiones de Ubuntu (y de Linux en general) está el compilador gcc. El gcc una herramienta bastante potente (lleva la mayoría de las librerías que se usan para la compilación de programas comunes) y su curva de aprendizaje es bastante asequible, permitiendo a usuarios noveles estar compilando programas a las pocas horas de formación. Esta herramienta es la que se ha usado para compilar el OpenWrt, por lo que es recomendable asegurarse de que esté instalado en el sistema. En el caso de que por algún motivo no esté presente en el SO, su instalación (aquí siempre refiriéndose a Ubuntu) se lleva a cabo por medio de:

```
sudo apt-get install gcc
```

La librería de llamadas al sistema usada por defecto con el gcc es la GNU C Library (glibc). Es posible (e incluso recomendable por el ahorro de tiempo que supone), ir depurando en el propio Ubuntu las aplicaciones que se estén desarrollado en lenguaje C para OpenWrt, compilándolas a través del gcc. Sin embargo, hay que tener en cuenta que la librería usada para la compilación en OpenWrt es uClibc, una versión reducida de glicb. Se debe tener en mente durante el desarrollo y la depuración del programa que algunas de las funciones más específicas fuera del estándar POSIX pueden no funcionar al compilar para el sistema embebido.

Teniendo el SO y el compilador preparado, queda elegir y descargar la versión de OpenWrt necesaria.

4.2 **Distribución OpenWrt**

OpenWrt es una distribución Linux para sistemas embebidos que lleva muchos años en desarrollo. El objetivo del proyecto fue el crear un sistema operativo para routers que pudiese ser extendido por los usuarios hasta alcanzar cualquiera de las funcionalidades de un sistema de ordenador. El fin de esto es facilitar un kernel básico sobre el que se puedan instalar las aplicaciones y paquetes que se vayan a usar. Originalmente fue pensada como sustituto al firmware del router Linksys WRT54G, pero con los años ha evolucionado a un SO embebido optimizado para routers en general. Posee una página oficial con una comunidad, que si bien no aparenta ser tan activa como años atrás, mantiene un gran directorio con documentación del sistema y un foro de dudas muy usado.

La numeración de las versiones lleva una nomenclatura similar a las de Ubuntu, siendo las dos primeras cifras las del año de desarrollo, cambiando el mes de publicación por un numero de sub-versión y dándosele un nombre en clave para cada versión. La versión estable más actual, sacada en octubre del año pasado es la 14.07 Barrier Breaker.

La versión usada en este trabajo es la 10.03 Backfire, modificada por el GCDC según se

comentó en la sección 2.5. El usar esta y no una posterior tiene la ventaja de que está muy probada y de que es cubierta aceptablemente por la documentación existente en la wiki, la cual está algo desfasada en las versiones más modernas.

En caso de no disponer del código fuente proporcionado por el GCDC, la distribución OpenWrt 10.03 (o Backfire a partir de ahora) puede ser descargada desde los repositorios de la página oficial usando descarga directa, GitHub o Subversion. Estos repositorios ofrecen multitud de versiones precompiladas en función del hardware sobre el que vaya a funcionar. No obstante, es recomendable descargarse los ficheros fuentes y compilarlos "al gusto" incluyendo en el proceso los módulos que se vayan a necesitar. En la siguiente sección se instruye la descarga y compilación de una versión genérica del OpenWrt.

Se hace notar que las versiones del repositorio oficial no traen el CRDA modificado, no permitiendo el cambio a la banda de los 5,9 GHz. Para tener acceso a las frecuencias CALM, es necesario conseguir el CRDA con el archivo "regulatory.bin" modificado (Por ejemplo si se dispone del módulo con "db.txt" en texto plano éste se consigue compilándolo con "dbparse.py" mediante Python). Sustituyendo un "regulatory.bin" modificado por el existente en el árbol de directorios del sistema debería ser suficiente para que al reiniciar, puedan usarse las nuevas frecuencias introducidas.

4.2.1 **Descarga y compilación del SO**

La tarea de compilar un programa o un SO para un hardware diferente al usado tiene un nivel más de dificultad que en el desarrollo de programas de PC.

Un compilador normal como puede ser el gcc, ya mencionado anteriormente, usa los ficheros de código fuente para conseguir un ejecutable preparado para la máquina donde se ha realizado la compilación pero, ¿cómo conseguir compilar un SO para una plataforma distinta a la usada?

La solución escogida aquí, es la llamada compilación cruzada o *cross-compiling.* Este concepto se basa en usar un kit de herramientas de compilación o *toolchain*, las cuales no son más que usa serie de programas que están preparados para poder seleccionar el tipo de plataforma objetivo. Este método es bastante usado debido a que se puede estar compilando código para multitud de plataformas desde un único hardware.

Por fortuna, las distribuciones de OpenWrt usan una herramienta de compilación cruzada ya preparada llamada *Buildroot*. Esta utilidad es una serie de ficheros *Makefiles* y *scripts* que permiten mediante un menú visual, elegir la plataforma destino del SO y las opciones y paquetes que llevará incorporado. Los programas incluidos en el *toolchain* están situados en la carpeta del mismo nombre.

Los requisitos que son necesarios para el uso de Buildroot con OpenWrt se pueden encontrar en la siguiente página web:

http://wiki.openwrt.org/es/doc/howto/buildroot.exigence

Para un uso más avanzado del que se va a explicar aquí de esta herramienta, es recomendable visitar la guía de su página oficial:

http://buildroot.uclibc.org/downloads/manual/manual.html#_getting_started

Usando los repositorios de descarga directa de la página oficial, se pueden obtener imágenes ya precompiladas para multitud de dispositivos o, si se prefiere, los archivos fuentes con la configuración ya preparada para un tipo de plataforma determinado.

Por otra parte, si se descargan los archivos mediante un controlador de versiones o se quiere adaptar la compilación a unas necesidades específicas, es necesario saber a qué tipo de plataforma corresponde el dispositivo o router a usar y si éste es compatible con el SO. Esta información se consigue buceando tanto en la Wiki de la página oficial como en los foros de OpenWrt.

Un listado del hardware soportado por OpenWrt se puede encontrar en la siguiente página web:

http://wiki.openwrt.org/toh/start

Si no se está seguro, o no hay documentación del hardware usado, se recomienda escoger los archivos de *Linux x86 generic*.

En el siguiente ejemplo se ha optado por descargar mediante GitHub la versión Backfire del OpenWrt. El tipo de plataforma escogido es Linux x86 generic, que es compatible con la arquitectura de la AlixBoard 3D2. Se descargará el código fuente para después configurar el compilador con el tipo de plataforma buscado.

En primer lugar es necesario tener instalado el programa controlador de versiones. En el caso de un Ubuntu con conexión a Internet, basta con ejecutar:

```
sudo apt-get install git-core
```

Para tener Subversion en vez de GitHub, únicamente habría que sustituir “git-core” por “subversion” en el comando.

Teniendo instalado uno de los controladores de versiones es buena idea crear una carpeta donde guardar el código fuente (es recomendable ser ordenado). Esta carpeta se puede crear, por ejemplo, en el directorio de trabajo:

```
mkdir ~/descargaOpenWrt
```

La descarga del código (en este caso la versión 10.03) se podría realizar mediante la ejecución de uno de los dos comandos dentro de la carpeta creada, dependiendo del software instalado:

```
git clone git://git.openwrt.org/10.03/openwrt.git
svn co svn://svn.openwrt.org/openwrt/branches/backfire
```

Para obtener cualquiera de las otras versiones existentes únicamente haría falta cambiar el número o nombre de versión en el comando, según se use una forma u otra. Cualquiera de las dos opciones descarga el código fuente dentro de una carpeta llamada "openwrt".

El código dentro de la carpeta "openwrt" viene dividido en distintos archivos y carpetas. Los más importantes serian:

- **bin:** La herramienta Buildroot además de compilar el sistema, también puede generar paquetes de aplicaciones para ser instalados en éste. En esta carpeta se almacenarán los paquetes creados siguiendo el método del apartado 4.3.
- **build_dir:** Aquí se guardará la imagen del sistema creado al compilar. El formato de la imagen creada es seleccionado en el menú de configuración de Buildroot.
- **docs:** Directorio con información técnica sobre la compilación de OpenWrt.
- **package:** Contiene los programas, herramientas y módulos que va a llevar inicialmente el SO, siendo instalados durante la compilación. Cada uno de ellos va en un subdirectorio con el nombre del programa. Inicialmente la descarga desde el repositorio suele traer únicamente un conjunto básico de herramientas, con idea de descargar durante la compilación los adicionales que se hayan seleccionado. A la hora de compilar, se buscará si los módulos solicitados están y se descargan los que falten. También es posible no incluir en la instalación carpetas que estén aquí, o añadir el software necesario directamente sobre el OpenWrt ya instalado para que no haga falta descargarlo durante su compilación. Más adelante se verá que es posible obtener paquetes de instalación de los programas existentes en esta carpeta, convirtiéndolos en archivos .ipkg sin necesidad de compilar el SO de nuevo.
- **scripts:** Son ficheros y rutinas usados durante la compilación del SO.
- **staging _dir:** Contiene elementos necesarios para realizar la compilación cruzada del sistema operativo y sus paquetes.
- **toolchain:** Al igual que el directorio staging_dir, contiene las herramientas que se usarán para compilar los módulos de OpenWrt.

- **target:** Aquí está la información de las distintas plataformas para las que se puede compilar.
- **tools:** Otras herramientas: Uso de Makefiles, generación de imágenes….

De entre los archivos contenidos en el directorio descargado principalmente destacan tres:

- **Makefile:** Será el archivo que se ejecutará desde el terminal. Permite lanzar la herramienta para personalizar la construcción del sistema, ejecutar el compilador y comprobar que todas las dependencias se cumplen, entre otras funciones.
- **feeds.conf.default:** Este archivo contiene enlaces a las fuentes de paquetes que serán usados durante la instalación. Será el usado por defecto si no se ha configurado el uso de la carpeta feeds. Modificando este fichero es posible indicarle al compilador que instale determinados módulos desde una ruta dada. Normalmente se tiene una referencia a la carpeta donde se guarda el código fuente de los paquetes descargados (packets) y otra a un directorio desde donde se puedan descargar los módulos que falten.
- **Config.in**: Este archivo contiene la configuración que será usada a la hora de compilar. Este archivo es automáticamente generado a partir de las opciones dadas a la herramienta Buildroot.

Antes de la ejecución del Buildroot, es conveniente primero comprobar que todas las dependencias se cumplen. Para ello, situando el entorno de trabajo del terminal dentro de la carpeta "openwrt", se puede ejecutar:

```
make defconfig
make prereq
make menuconfig
```

En el caso de que falten paquetes (como puede ser libncurses5-dev o build-essential en el equipo usado), es necesario instalarlos mediante el comando `apt-get`. Cuando estén todos instalados, es con el último comando con el que se lanza la herramienta.

La interfaz de Buildroot puede parecer un tanto complicada, pero, afortunadamente, únicamente es necesario usar unas pocas opciones. Lo que es más, no es necesario conocer qué módulos son compatibles con cuáles o qué paquetes son incluidos con qué opción debido a que la mayoría pueden ser descargados desde el repositorio (en formato .ipkg) e instalados directamente desde el sistema en funcionamiento. Las opciones a las que se tiene que prestar especial cuidado son las siguientes:

1. **Target system:** Tipo de plataforma usada. En el caso de la AlixBoard 3D2 será "x86".
2. **Subtarget:** Familia del dispositivo. En este caso "PCEngines Alix2", que es compatible con las Alix3D2.
3. **Target Profile:** En caso de existir perfiles de configuración para la familia seleccionada, están aquí. Dejar en "default".
4. **Target Images:** Tipo de fichero del sistema compilado. Pueden ser de diversos tipos como jff2, tar.gz, squash o ext2. Aunque aquí se usará ext2, el proceso no difiere mucho con el uso del resto de formatos. Es posible generar en la misma compilación la imagen del sistema en dos formatos distintos.

A la hora de salir del menú de Buildroot se pedirá una confirmación para guardar los cambios efectuados, sobrescribiendo el fichero .Config.in. Existen ficheros de configuración para algunas plataformas listos para descargar y usar. Con sustituir el fichero por el existente (o seleccionarlo desde la opción del menú preparada para ello) es suficiente.

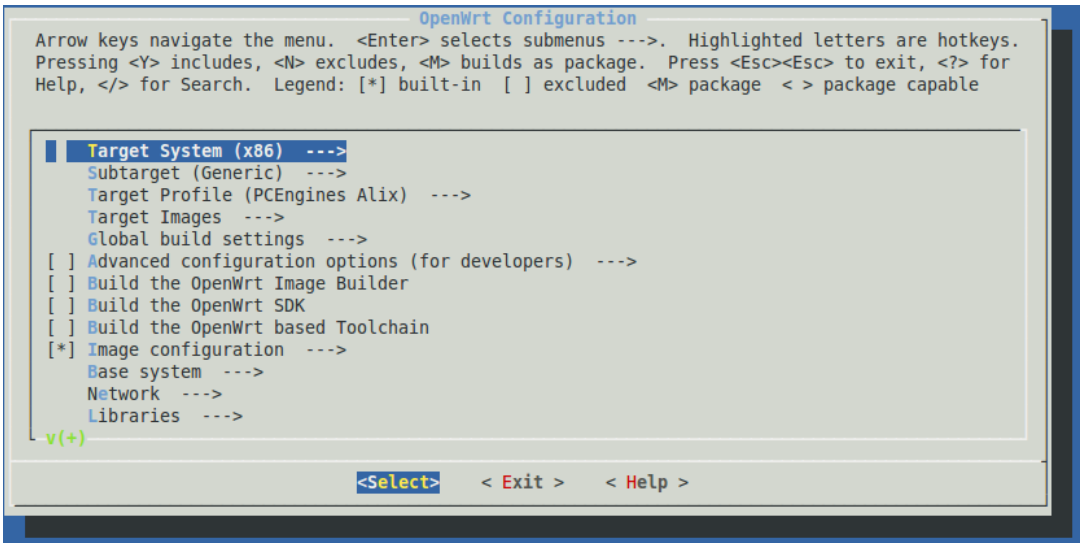


Figura 21: Menú principal de Buildroot.

Una vez seleccionadas las opciones, la compilación se lanzará con el comando `make`. Si es la primera vez que se lanza, tardará un poco más de tiempo que en ejecuciones sucesivas.

Ante posibles errores, recordar que puede ser necesario tener una versión del sistema operativo temporalmente cercana al SO a compilar, habiéndose detectado fallos en la compilación del OpenWrt 10.03 desde el Ubuntu 14.04[4].

Con todo correcto, la imagen creada puede encontrarse dentro de la carpeta bin, quedando la instalación de la misma en el soporte físico.

[4] En caso de error durante la ejecución del *make*, este comando acepta como parámetro V=99. Esta opción muestra toda la información de salida y suele permitir ver con más detalle dónde se ha producido el fallo.

4.2.2 **Instalación del SO**

Una vez que la compilación haya terminado se tendrá un fichero con la imagen en el formato que se haya seleccionado con el Buildroot.

El soporte donde se va a instalar el OpenWrt será una tarjeta CompactFlash (CF), usándose un lector de CF que se pueda conectar a uno de los USB del ordenador de trabajo. También es necesario conocer el identificador del dispositivo de este lector, o bien usando el comando *dmesg* a través de un terminal, o con algún gestor de particiones como Gparted. Un ejemplo del identificador que puede tener es "*dev/sdc".*

Finalmente, es buena práctica realizar un backup de la imagen creada y formatear la CF en FAT 32 antes de realizar el traspaso. Además, en caso de que la imagen se haya creado en un formato de fichero comprimido, será necesario descomprimirla por medio de:

```
gunzip nombre_imagen.img.gz
```

Dentro de la CF se necesitará una partición que sea usada por la AlixBoard como disco de arranque y otra donde estará almacenado el OpenWrt. Ambas son realizadas automáticamente con el siguiente comando:

```
dd if=bin/x86/nombre_imagen.ext2 of=/dev/sdb
```

Siendo "*bin/x86/nombre_imagen.ext2"* la ruta de la imagen creada, mientras que "*dev/sdb"* és el identificador del dispositivo.

Tras este último paso, el sistema está listo para ser ejecutado en la AlixBoard.

4.2.3 **Primeros 5 minutos con OpenWrt**

Una vez cargado el OpenWrt, montada la CF en la AlixBoard y encendida siguiendo el montaje de la sección 3.2, surge la pregunta ¿Y ahora cómo se usa?

La forma más directa de entrar al sistema en funcionamiento para configurarlo sería usar un cable que conecte el puerto serie de la AlixBoard con el del ordenador de trabajo. Para abrir y trabajar con esta conexión es necesario un programa o hyperterminal preparado para la transmisión en serie. En Windows esto es fácil de conseguir usando *Putty*, mientras en Ubuntu, una buena solución es usar *Minicom*.

Al carecer el portátil usado de entrada serie, se ha configurado OpenWrt para un primer uso mediante la lectura de la CF usando el lector de CompactFlash en un puerto USB. Éste será el método explicado a continuación.

Al ser la CF conectada, el ordenador debería reconocer dos nuevos dispositivos legibles, con un comportamiento similar al de un pendrive. Uno de ellos será la partición de arranque,

mientras que el otro es el sistema de ficheros del SO. Estos directorios mantienen una estructura simplificada de la distribución típica de los sistemas Linux, donde se pueden apreciar las siguientes carpetas:

- bin: Script comunes a todos los usuarios del sistema. En el caso de que se creen y usen nuevos usuarios, los archivos de esta carpeta pueden ser ejecutados por todos ellos.
- dev: Referencia a los dispositivos periféricos de sistema.
- etc: Carpeta donde se guarda la configuración general del sistema y sus programas. Dentro de esta carpeta existe un subdirectorio llamado *config*, donde están definidos los archivos de configuración de las conexiones de red.

  El directorio *config* está enfocado a contener los archivos de la configuración de los adaptadores de red que use el sistema. El fichero que interesa en primer lugar es *network*, que define la configuración básica a usar en cada adaptador. Cualquier procesador de textos como puede ser *nano* o *gedit* permite abrir estos archivos. Un ejemplo de cómo se estructura este fichero sería el siguiente:

```
config interface loopback
   option ifname lo
   option proto static
   option ipaddr 127.0.0.1
   option netmask 255.0.0.0

config interface lan
   option ifname eth0
   option proto static
   option ipaddr 192.168.4.4
   option netmask 255.255.255.0
```

Por defecto, OpenWrt trae definida la interface *loopback*, usada para hacer referencia a la propia máquina, y la LAN, que es la que va a ser usada para administrar el sistema.

Cada interfaz definida lleva una serie de opciones:

- **ifname**: Nombre por el cual el sistema identifica a esa interfaz. Sobre todo es utilizado en la ejecución de comandos en el terminal, aunque también sirve para hacer referencia a una de las interfaces desde otros ficheros de configuración.

- **proto**: Protocolo usado para obtener la dirección IP de la interfaz. Los más comunes son *static*, especificando la dirección a usar con las opciones ipaddr y netmask, y *dhcp*, usado para pedir una dirección a otro equipo de la red.
- **ipaddr**: En caso de estar usando *static* en la opción anterior, esta opción especifica cuál es la dirección IP que usara la interfaz.
- **netmask**: Opción también solo disponible si la dirección a usar es de tipo estático. Establece qué parte de la dirección es común a la red local del equipo.

El otro fichero que hay que tener en cuenta dentro de *config* es "*wireless*". En este fichero se estipula la configuración usada por la tarjeta de red inalámbrica, en este caso la Mikrotik. El fichero tiene una apariencia similar a esta:

```
config wifi-device  radio0
   option type     mac80211
   option channel  11
   option hwmode   11g

   # REMOVE THIS LINE TO ENABLE WIFI:
#  option disabled 1

config wifi-iface
   option device   radio0
   option network  wan
   option mode     ap
   option ssid     OpenWrt
   option encryption none
```

Las opciones que se presentan en este ejemplo son las siguientes:

- **type:** Identifica en driver usado por la tarjeta gráfica. Este parámetro es detectado en el arranque y por tanto no suele ser necesaria su modificación.
- **channel:** Especifica en qué frecuencia se va a usar la interfaz radio. Para usar la tarjeta como punto de acceso o como parte de una red ad-hoc, se debe especificar un canal, mientras que para su uso como una estación de una red infraestructura este valor puede ser auto.
- **hwmode:** Selecciona qué estándar se usará para transmitir (11b, 11g o 11a). En caso de no estar esta opción, se calcula de forma automática.

- **device:** Identificador del dispositivo al que se le va a aplicar la configuración. Debe corresponderse con el nombre proporcionado en el apartado *wifi-device*.
- **network:** Nombre de la interfaz radio descrita en el fichero *network* a la que asociar esta configuración.
- **mode:** Indica el modo en el cual va a funcionar la tarjeta. De los valores posibles destacan:
  - sta: Estación dentro de una red infraestructura.
  - ap: Punto de acceso en una red infraestructura.
  - adhoc: Parte de una red entre iguales.
- **ssid:** Nombre asignado a la red inalámbrica. Identificador de la red creada en el modo *ap* o de una ya existente en los modos *adhoc* y *sta*.
- **encryption:** Tipo de encriptación usada en la red. Si el valor es distinto de *none* se necesita añadir el campo *key*, cuyo valor será la clave de la red.

No es obligatorio definir todas las opciones de los ejemplos expuestos de *network* y *wireless*. Las opciones que se pueden usar y sus posibles valores se pueden consultar en la documentación de la página de OpenWrt.

http://wiki.openwrt.org/doc/uci/network
http://wiki.openwrt.org/doc/uci/wireless

- root: Carpeta personal del usuario root del sistema.
- sbin: Misma función que bin, pero los scripts de esta carpeta únicamente pueden ser usados por el usuario root.
- usr: Compuesta por los subdirectorios de los programas actualmente instalados en el SO.

Sin modificar nada, la interfaz Ethernet de la tarjeta usará la dirección 192.168.1.1, con la máscara de red 255.255.255.0. Para el acceso al sistema mediante SSH es necesario configurar el ordenador con el que se vaya a conectar con una dirección IP perteneciente a la subred 192.168.1.X, como puede ser por ejemplo 192.168.1.2. En caso de estar usando una máquina virtual, conviene recordar el posible uso del NAT descrito al final del apartado 4.1.1.1.1.

Otra posibilidad es modificar (leyendo la CF desde el ordenador) las opciones *ipaddr* y *netmask* del fichero *config* para que el sistema inicie con una IP determinada. Los ficheros del sistema operativo están protegidos contra la modificación por parte de usuarios normales, por lo

que hay que modificar sus permisos (totalmente desaconsejado) o cambiar a superusuario o *root* y ejecutar desde ahí el procesador de textos[5].

Una vez que el SO de la AlixBoard y el de la tarjeta LAN del PC tengan direcciones en el mismo rango, se puede proceder con el montaje especificado en el capítulo anterior, tras el cual se realizará la primera conexión con el OpenWrt en funcionamiento ejecutando el comando telnet en un terminal.

```
telnet 192.168.1.1
```

Completados los pasos anteriores, el terminal debería mostrar la pantalla de bienvenida de OpenWrt. Si esto es así, enhorabuena, se ha superado la parte más difícil.

Lo primero que se debería hacer una vez dentro del sistema es cambiar la clave del usuario root por una propia mediante el uso del comando *passwd.* Habiendo hecho esto, de ahora en adelante se puede acceder al sistema mediante el protocolo SSH.

```
ssh root@192.168.1.1
```

El motivo de usar SSH y no telnet se debe a que este último usa una conexión sin cifrar. SSH se creó para sustituir a telnet y desde su puesta en funcionamiento, telnet ha quedado obsoleto, habiendo programas que no admiten su uso por medio de este protocolo.

Si, llegado a este punto, el sistema funciona tal como se comenta, lo recomendable es realizar una copia de seguridad (*backup*) por lo que pueda pasar. La realización de esta copia se puede hacer conectando la CF al PC por el método ya visto y ejecutando en un terminal como superusuario:

```
mkdir ~/openwrt_backup
cp –r –f /media/”dispositivo”/”particion_del_sistema_de_archivos/*
~/openwrt_backup/
```

Aunque rudimentario, lo anterior permite tener un punto de restauración en el caso de que el sistema deje de funcionar por una mala configuración.

4.2.3.1 **Activando la interfaz inalámbrica**

La distribución ofrecida por OpenWrt no está preparada para usar la interfaz inalámbrica de forma inmediata. Es necesaria añadir una serie de paquetes que no vienen incorporados por

[5] Si se tienen conocimientos de seguridad en sistemas Linux, es posible aprovechar aquí para realizar el cambio de contraseña descrito a continuación modificando el archivo “shadow”. **Importante**: Solo usar este método si realmente se sabe lo que se hace y tras haber realizado un backup.

defecto. Estos paquetes pueden ser incluidos al compilar la distribución (incluyéndolos al seleccionarlos con Buildroot) o instalados con OpenWrt ya en funcionamiento.

En este segundo caso es necesario descargarse los ficheros .ipkg faltantes del repositorio de la página oficial:

http://downloads.openwrt.org/backfire/10.03.1/x86_generic/packages/

Los paquetes necesarios y las dependencias que tienen se muestran en la Figura 22.

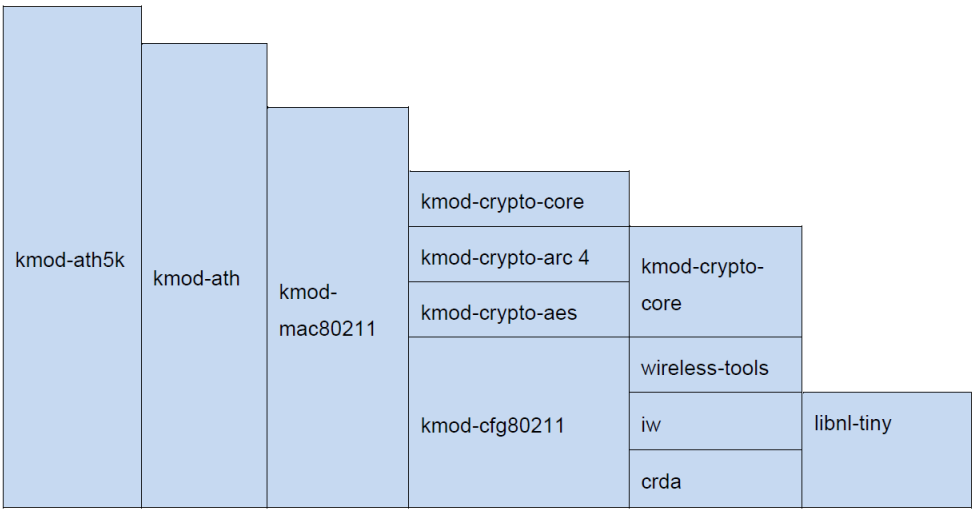


Figura 22: Paquetes para comunicación inalámbrica no incluidos en OpenWrt.

Además de los paquetes de la figura anterior, tanto para la distribución de OpenWrt como para la modificada por el GCDC, es necesario uno de los siguientes paquetes.

Figura 23: Paquetes no incluidos en la distribución del GCDC

El método de instalación de estos paquetes esta descrito en el apartado 4.3.3.

Al entrar por primera vez al sistema se puede comprobar que el archivo wireless no existe. La razón es que en el archivo network vienen por defecto definidas la interfaz loopback y LAN pero no la inalámbrica. En primer lugar es necesario introducir esta interfaz en el fichero añadiendo por ejemplo:

```
config interface wan
   option ifname wlan0
```

```
option proto static
option ipaddr 192.168.2.3
option netmask 255.255.255.0
```

La forma de añadir esta nueva interfaz puede ser modificando el archivo de la CF leyéndola desde el ordenador o haber instalado un procesador de textos como nano mediante un paquete .ipkg.

La nueva configuración será recogida por el sistema al reiniciar o ejecutar el comando:

```
/etc/init.d/network restart
```

Una vez hecho esto se puede generar el fichero wireless mediante el comando:

```
wifi detect > /etc/config/wireless
```

Con esto se consigue detectar el driver a usar para la tarjeta inalámbrica (mac80211) generándose el archivo de configuración en la ruta indicada.

A partir de aquí es cuestión de familiarizarse con los distintos comandos y ficheros de configuración del sistema, que en la mayoría de los casos, guardan muchas similitudes con los encontrados en las versiones de Linux para PC.

4.3 Compilar una aplicación para OpenWrt

Una vez visto OpenWrt desde el punto de vista de un usuario, este apartado se dedica a la instalación, de una aplicación desarrollada, en este sistema operativo.

Al igual que la compilación del sistema, el proceso de generación del ejecutable debe realizarse a través de compilación cruzada. La diferencia radica en que en este caso lo generado no va a ser una imagen lista para descomprimir e instalar en la CF, sino un fichero .ipkg. IPKG (*Itsy Package Management System*) es un gestor de paquetes, es decir, un conjunto de herramientas software enfocadas a automatizar la instalación, actualización, mantenimiento y desinstalación de aplicaciones a partir de su código fuente. Está basado en el dpkg de *Debian* adaptado a sistemas embebidos.

Antes de empezar a desarrollar software para OpenWrt, se debe tener en cuenta que se puede dar el caso de que un programa que funcione correctamente durante las pruebas en el ordenador, no funcione o lo haga mal al pasarlo al sistema embebido. Aunque uClibc (la librería usada para generar el ejecutable a partir del código fuente) soporta la mayoría de las aplicaciones que se compilan con GCC (usando glicb), puede pasar que directivas poco usuales

o demasiado específicas del lenguaje C no estén presentes o funcionen de forma diferente, dando problemas a la hora de ejecutar el software en OpenWrt. Por poner un ejemplo, en la práctica se han descubierto problemas al intentar usar semáforos de la librería *semaphore.h* debido a que uClibc no da soporte a esta librería.

4.3.1 **Herramienta de generación de paquetes**

OpenWrt pone a disposición de los desarrolladores un kit de herramientas de desarrollo (SDK) para poder generar paquetes de instalación de software propio sin necesidad de tener que compilar cada vez todo el sistema operativo. El SDK puede conseguirse compilándolo desde el Buildroot mediante las opciones del comando "`make menuconfig`" o descargar una versión precompilada del mismo para la versión del SO usado desde el repositorio de OpenWrt.

El SDK para la versión Backfire usada (x86 generic Linux) se puede conseguir a través del siguiente enlace:

https://downloads.openwrt.org/barrier_breaker/14.07/x86/generic/OpenWrt-SDK-x86-for-linux-x86_64-gcc-4.8-linaro_uClibc-0.9.33.2.tar.bz2

Una vez descargado el SDK en un archivo comprimido, es conveniente pasarlo a la carpeta "openwrt" creada en el directorio de usuario y descomprimir allí el fichero. La forma de ponerlo en marcha y de resolver las dependencias está explicada en el apartado 4.2.1.

Los archivos del interior del directorio son prácticamente los mismos que los de Buildroot, con la diferencia de que faltan todos los ficheros pertenecientes a la construcción de OpenWrt.

4.3.2 **Generar un fichero IPGK**

Para generar el archivo instalador para OpenWrt se debe acceder a la carpeta "*Packages"* y ahí crear una carpeta con el nombre del ejecutable. Dentro de esta carpeta se guardará todo lo relativo a ese programa, teniendo una carpeta por aplicación a compilar. Tras su creación, el código fuente del programa debe ir metido dentro de un subdirectorio llamado "*src"*. Para que el SDK pueda generar el .ipkg a partir del código fuente son necesarios dos archivos "*Makefile"*.

El primero de ellos debe estar situado dentro de la carpeta "*src"* y tiene la función de generar el ejecutable, un ejemplo de este fichero sería el siguiente:

```
# Makefile del ejecutable de Calm.
todo: compilacion
   rm *.o

compilacion: inicio.o cliente.o servidor.o comun.o
```

```
	$(CC) $(LDFLAGS) -o calm inicio.o cliente.o servidor.o comun.o

inicio.o:	inicio.c libreria.h
	$(CC) $(LDFLAGS) -c inicio.c libreria.h

cliente.o:	cliente.c libreria.h
	$(CC) $(LDFLAGS) -c cliente.c libreria.h

servidor.o:	servidor.c libreria.h
	$(CC) $(LDFLAGS) -c servidor.c libreria.h

comun.o: comun.c libreria.h
	$(CC) $(LDFLAGS) -c comun.c libreria.h
```

Las variables $(CC) y $(LDFLAGS) son dadas por el propio SDK a la hora de generar el paquete IPKG y corresponden al compilador llamado y a las opciones pasadas a ese. Es importante usar estas variables y no hacer la llamada directamente al gcc con las opciones. El .ipkg generado en este último caso se podrá instalar en OpenWrt pero el programa no será reconocido como ejecutable (al estar destinado a otra arquitectura).

El segundo fichero es más complicado, siendo el que define todas las opciones del archivo ipgk creado. Las partes del fichero están explicadas en los comentarios del código siguiente:

```
include $(TOPDIR)/rules.mk

# Nombre del paquete y su versión, usado a la hora de comprobar actualizaciones.
PKG_NAME:=calm
PKG_RELEASE:=1


# Directorio donde se construirá el paquete, por defecto es el build_dir.
PKG_BUILD_DIR := $(BUILD_DIR)/$(PKG_NAME)


include $(INCLUDE_DIR)/package.mk



# Informacion del paquete. Poco que explicar.
define Package/calm
	SECTION:=utils
	CATEGORY:=Utilities
	TITLE:=Calm -- prototipo de transmision calm
endef
```

```
   #Si se le quiere dar una descripcion al programa, este es el sitio.
   define Package/description
   endef

   # Se copian los ficheros fuentes del directorio donde están a donde seran
#empaquetados.
   define Build/Prepare
      mkdir -p $(PKG_BUILD_DIR)
      $(CP) ./src/* $(PKG_BUILD_DIR)/
   endef

   # Este bloque indica donde se generará el paquete de la aplicación.
   define Package/calm/install
      $(INSTALL_DIR) $(1)/bin
      $(INSTALL_BIN) $(PKG_BUILD_DIR)/calm $(1)/bin/
   endef


   # Esta linea lanza la creacion del paquete.
   $(eval $(call BuildPackage,calm))
```

Con los dos ficheros, se puede lanzar la creación del paquete desde la carpeta superior (donde se encuentra *package*) mediante el comando "`make`". Este comando intentará generar un paquete .ipkg por cada carpeta situada en "*package"*.

El paquete generado se encuentra (con los parámetros dados en este makefile) dentro de "*bin/x86/packages"*.

4.3.3 **Instalación del paquete generado**

Una forma de pasar el paquete al sistema sería introducirlo directamente en la CF usando el lector mediante el comando *cp*, por ejemplo, dentro del directorio "root". Recordar que para escribir en la CF es necesario ser superusuario si no se han cambiado los permisos.

Iniciando de nuevo el OpenWrt en la AlixBoard y conectándose por SSH, la instalación se realiza mediante el siguiente comando (siendo "/root" la ruta sugerida en el ejemplo):

```
   opkg install /root/calm.ikpg
```

Al haber configurado el "*makefile"* para que sitúe el ejecutable del programa en la carpeta "/bin", éste puede ser ejecutado escribiendo el nombre del ejecutable en la terminal, como si fuese un comando más del sistema. El nombre usado en este caso, elegido en el *makefile* del directorio "*src"* mediante la opción –o, es calm.

4.4 Aplicación ITS

Una de los principales avances que tienen los vehículos pertenecientes a un sistema ITS es la ejecución de aplicaciones distribuidas. Gracias a ello es posible crear nuevos servicios basados en la división y réplica de la información, usando cada ITS-S de la red como servidor y cliente de cara al resto de nodos. En este ámbito destaca la posibilidad de implementar nuevos servicios de seguridad vial no centralizados, de forma que sean más resistentes ante fallos. El software creado cae dentro de este entorno, permitiendo al vehículo compartir con otros la información almacenada sobre peligros en la vía y recibir a su vez, nuevos obstáculos no conocidos. Alertar al conductor sobre un obstáculo próximo permite que éste extreme las precauciones (reduciendo velocidad, aumentando distancia de seguridad…) disminuyendo las probabilidades de un accidente. El código creado es un ejemplo práctico de las aplicaciones que podrían desarrollarse al implantar CALM en la red de transportes.

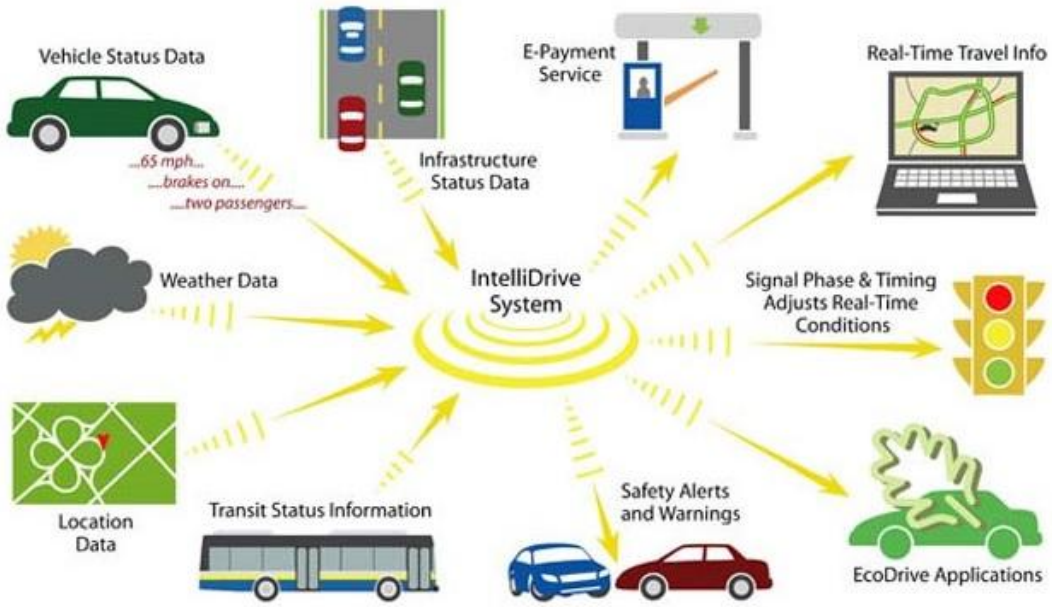


Figura 24: Posibles servicios disponibles en un ITS.

4.4.1 Creando un servicio ITS

El servicio creado tiene su marco de funcionamiento en las comunicaciones inalámbricas que tienen lugar entre dos ITS-S vehiculares. Tal como dicta WAVE, estas comunicaciones se desarrollan en un entorno V2V.

La forma más fácil de implementar un servicio de estas características es hacerlo directamente sobre los routers ITS omitiendo de esta forma las comunicaciones internas de las ITS-Ss. En este software se ha seguido el modelo cliente-servidor para las comunicaciones, anunciándose los servidores en la red de manera que estos puedan ser contactados por los clientes. Como se pretende que la aplicación pueda usarse en una red de jerarquía plana, cada

ITS-S estará compuesta por un cliente y un servidor, de forma que todos los nodos del servicio sean idénticos en funcionalidad, añadiendo las ventajas de la arquitectura P2P.

El lenguaje de programación escogido para la codificación ha sido C. Este lenguaje está muy extendido y es especialmente compatible con los sistemas Unix y sus evoluciones Linux. El principal motivo para usar éste y no uno de mayor nivel (JAVA o Python) es el poder ejecutar directamente el código sobre el kernel del OpenWrt sin necesidad de un módulo o máquina virtual adicional, que por otra parte podrían no estar soportados por el SO embebido. Además, este lenguaje permite programar a bajo nivel y es compilado (ejecutando el software directamente en instrucciones de la maquina), haciéndolo muy eficiente, lo que es adecuado para un servicio de seguridad en el que la vida de personas puede depender de la velocidad de reacción del programa.

La estructura general del servicio se divide en dos procesos (un cliente y un servidor) que corren de forma paralela sobre el OpenWrt instalado en cada uno de los routers ITS. Este servicio consiste en una comunicación básica en la que se trasmite información del estado del vehículo a otros nodos móviles o RSU recibiendo una alerta en caso de que se esté dirigiendo hacia un peligro conocido por éstos. En la Figura 25 se muestra un ejemplo del uso de este servicio. En este ejemplo el coche B conoce que existe un obstáculo en la carretera a la altura del coche A. (No se considera si es el vehículo accidentado quien ha lanzado la alerta o ha sido una tercera parte). El router ITS del coche B transmitirá el peligro a todos los coches al alcance cuya trayectoria les lleve hacia el peligro (C), que a su vez irán propagando la alerta (bajo las mismas condiciones) de unos nodos a otros del sistema. Esta alerta no será comunicada a los nodos que se estén alejando del peligro (D), ahorrando transmisiones innecesarias. El coche E será alertado por B cuando B cuando se encuentre al alcance de éste.

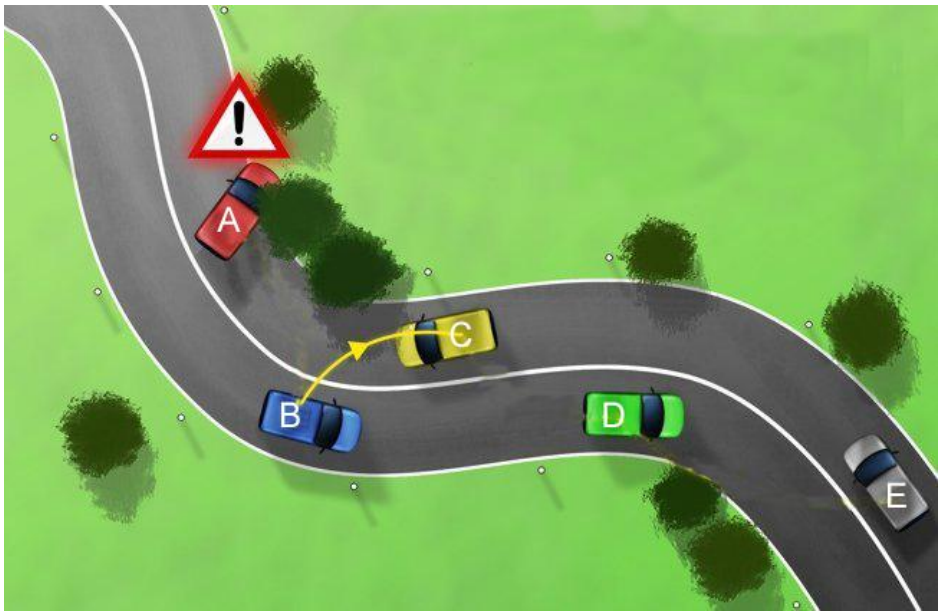


Figura 25: Uso práctico del servicio creado.

El software ha sido desarrollado siguiendo el estándar CALM, y de forma más específica la estructura de WAVE, conforme a lo especificado en el punto 2.4.

4.4.2 **Escenario**

El escenario en torno al cual se ha diseñado la aplicación consiste en una carretera de un carril en cada sentido similar a la mostrada en la Figura 25. Para simplificar el proceso de localización de los vehículos y el trazado de sus trayectorias en los cálculos se ha considerado que la carretera es recta pasándose conceptualmente de las 3 dimensiones reales a 1. La posición y velocidad de los vehículos estará determinada por números decimales. Considerando que la carretera empieza en 0 (por ejemplo posición del coche E), la posición de un vehículo estará dada por un número decimal positivo que marca la distancia al punto 0. Su velocidad será también un numero decimal, negativo en el caso de que vaya con sentido al punto 0 y positivo cuando se esté alejando de este.

4.4.3 **Estructura general de la aplicación**

La aplicación está basada en dos procesos ejecutándose de forma simultánea, llevando respectivamente las tareas de cliente y servidor. Diseñar la aplicación separada en dos procesos en vez de tener cliente y servidor en uno solo permite mayor tolerancia a fallos. Ambos procesos pueden ser independientes uno del otro, permitiendo conservar parte de la funcionalidad incluso cuando uno de los dos muere. Para facilitar su uso dentro del OBU como un servicio, el software se compila como un único programa. En la función inicial se usa el método *fork*, el cual divide el programa en los dos procesos dándoles un identificador distinto a cada uno. Con esta forma de ejecución se puede controlar la ejecución del programa a través de parámetros de entrada, pudiendo decidir lanzar únicamente uno de los dos procesos.

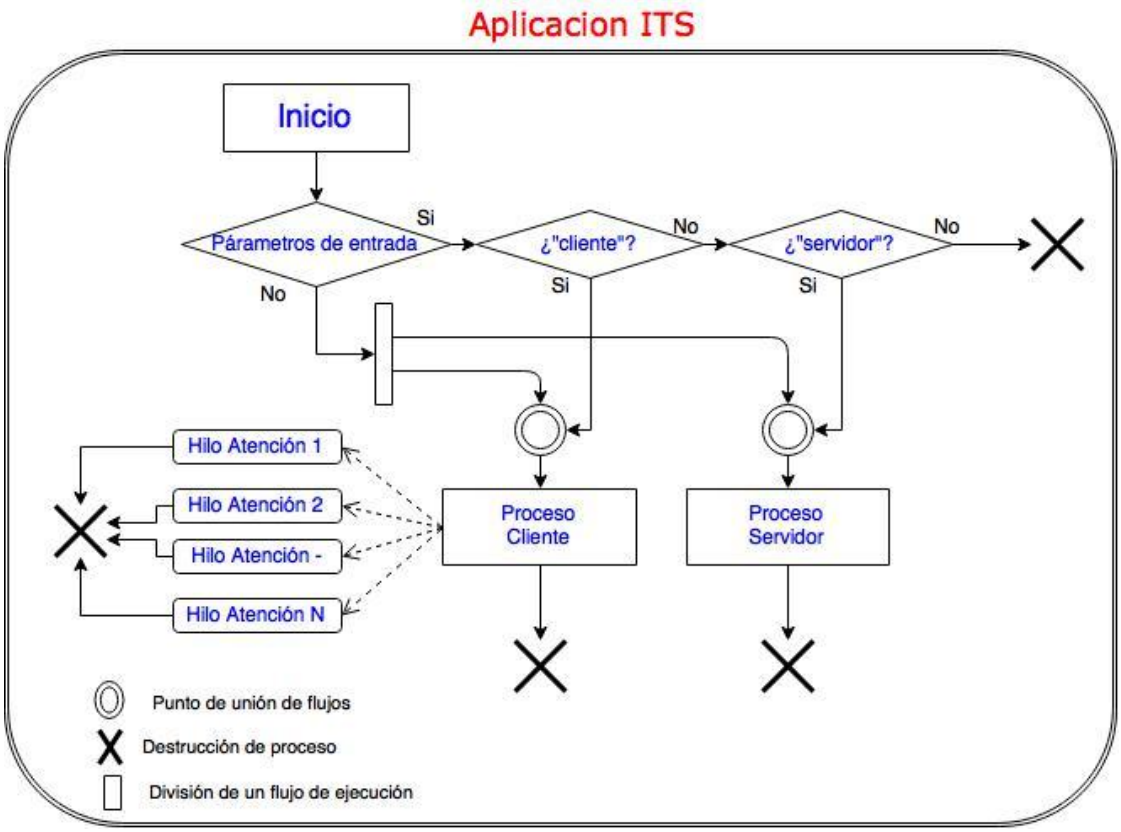


Figura 26: Estructura general de la aplicación.

Por sencillez, el software desarrollado pertenece a la categoría de servicios preconfigurados, llevando ya el propio programa qué puertos y protocolos usar. Cada servidor en ejecución envía de forma periódica un mensaje anuncio de servicio similar al WSA definido en WAVE, que permite al resto de nodos conocer la existencia del servicio en ejecución y del vehículo. Este mensaje de descubrimiento (que ha sido denominado como HI), contiene la posición, velocidad y carga de la batería del vehículo. Con estos datos, el resto de nodos que ejecutan la aplicación pueden ya de entrada conocer y prever la trayectoria que va a seguir, información que puede usarse para evitar colisiones.

El servidor del programa tiene la función de aceptar las conexiones TCP de clientes situados en otras ITS-S que quieran intercambiar mensajes. En función de los mensajes recibidos a través de estas conexiones el servidor realizará la acción adecuada. Con lo definido en el trabajo actual, el servidor tiene la opción de reenviar al cliente interno (el proceso que está corriendo en la misma OBU) datos que le han sido transmitidos desde un cliente externo (procesos situados en otros routers ITS) o bien responder con datos del trayecto del vehículo a una petición realizada desde otro vehículo.

En el caso del cliente, este permanece a la escucha de posibles mensajes de saludo que envíen los procesos servidor situados en otros nodos. El cliente cuenta con un listado de nodos conocidos (que se supone están siendo atendidos), ignorando aquellos mensajes HI de nodos existentes en este listado. La identificación de los nodos se realiza mediante un numero único (ID). Al recibir un saludo de un nodo no conocido, el cliente lee la dirección origen de este

mensaje y crea un hilo de comunicación con este nuevo vecino. Dentro de este hilo (específicamente creado para comunicarse con una ITS-S en concreto) se repasa la lista de peligros conocidos en la carretera y se calcula si alguno de ellos puede afectar de forma inmediata al nodo emisor. En caso de ser así, se avisará al otro extremo enviándole un mensaje con la posición del peligro y requiriendo un ACK (a nivel de aplicación) de éste. Cuando no se encuentra un peligro que pueda afectar próximamente al otro nodo, se realiza una petición de estado avanzado, donde se pregunta por la distancia y tiempo que lleva recorrido en el trayecto actual y el nivel de batería que quedó tras la última recarga.

El almacenamiento de los datos obtenidos se realiza en el proceso cliente, mientras que su procesamiento es propio de cada conexión, llevándose ésta a cabo mediante el uso de hilos. Los hilos permiten descargar al cliente de carga lógica, aliviando problemas de sobrecarga cuando se tienen multitud de conexiones simultaneas.

En general, los intercambios de información en el servicio se realizaran siempre entre un cliente y un servidor. Un cliente situado en un vehículo A únicamente se comunica con el servidor de un vehículo B y nunca con su cliente. A la inversa, el servidor del vehículo A tan solo se comunicará con el cliente del vehículo B. Como los datos son almacenados y procesados en los procesos clientes, debe existir una comunicación para que los datos recibidos por la parte del servidor lleguen al cliente local.

Durante el arranque, el cliente inicia una conexión TCP con el servidor propio. Esta conexión será usada para pasar información del servidor al cliente. Concretamente, la información pasada serán los peligros que reciba el servidor desde clientes de nodos externos de manera que se puedan almacenar los nuevos peligros en la lista de peligros conocidos.

Aunque pueda parecer una estructura enrevesada, esta forma de conexión simplifica enormemente la codificación de ambos procesos, al estar claramente diferenciados en cuanto a funciones. Como ya se ha comentado, el que sean dos procesos independientes permite que un fallo en uno de ellos no tire abajo todo el servicio. En caso de que el servidor del nodo A falle, éste aún podrá seguir recibiendo los mensajes HI en su proceso cliente y enviar alertas con los peligros que conoce. Sin embargo, no podrá recibir los nuevos peligros que se comuniquen por la red, y no informará de su posición al resto de nodos. Por otra parte, si lo que falla es el servidor, la ITS-S mantendrá informados de su posición y velocidad al resto de nodos, y podrá recibir nuevas alertas de peligro, pero éstas no podrán ser almacenadas ni comunicadas a otras ITS-S.

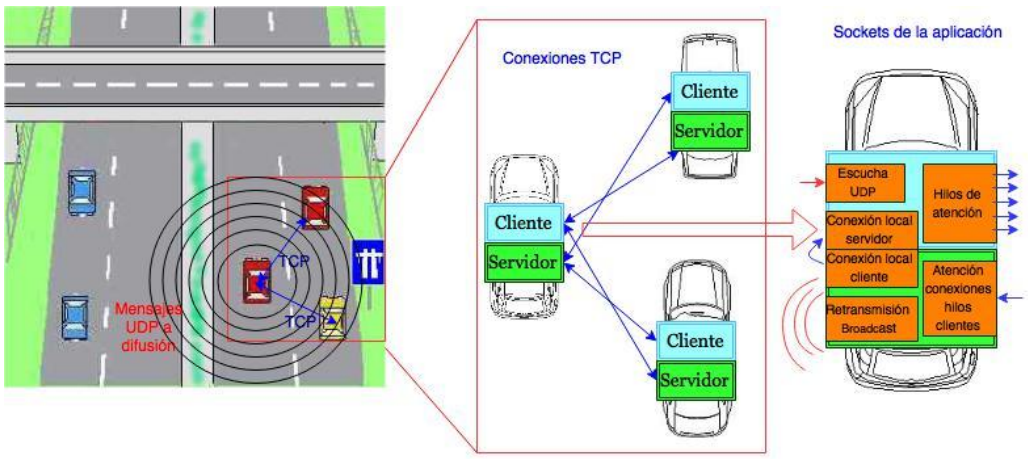


Figura 27: Esquema de comunicaciones entre ITS-Ss.

El motivo por el cual el paso de información entre los dos procesos se realiza a tan alto nivel (puerto TCP en capa de transporte) se debe a que no ha sido posible usar zonas de memoria compartida. Estas zonas de memoria requieren estar bajo un semáforo para evitar posibles condiciones de carrera, sin embargo uClibc (que recordemos que la librería usada al generar aplicaciones en OpenWrt) no da soporte a las funciones de semaphore.h, no reconociéndose entonces las funciones de apertura de semáforos durante la ejecución del código en las AlixBoard.

4.4.4 **Conexiones existentes en la aplicación**

La aplicación usa diferentes conexiones y puertos en función de los datos a transmitir en cada momento. Estas conexiones están divididas por el tipo de conexión buscado (TCP o UDP), o dependiendo de dónde esté situado el proceso al que se le envía la información (local o externo).

Los mensajes HI, que realizan la función de anuncio del servicio, son enviados a la dirección de difusión de la red mediante mensajes UDP, debido a que es información que se transmite de forma periódica sin esperar respuesta. El puerto al que se envían los mensajes HI es el 21215, por lo que los clientes deben tener un socket de escucha en ese puerto.

Por otra parte, para el intercambio de información entre dos ITS-S, ya sea de un peligro o del histórico de un trayecto, se establece una conexión TCP desde el proceso cliente de la placa receptora del mensaje HI al proceso servidor de la placa emisora. Los servidores tienen la escucha de esta conexión en el puerto TCP 21210.

La tercera conexión es la usada para comunicaciones “internas” entre el proceso cliente y el servidor. Esta conexión es fundamentalmente usada para pasar al servidor los mensajes de peligro recibidos por el cliente. La conexión la establece el cliente con una petición al puerto 21221.

4.4.5 **Listas de estructuras de información**

Para el almacenamiento y procesado de los datos del servicio, en el cliente se crean dos listas simples de datos tipo estructuras (*struct*). La primera de ellas, denominada nodos conocidos, contiene la información de las ITS-S con las que actualmente el cliente está manteniendo contacto. Esta información está guardada en unas estructuras llamadas DIRIP que contienen la información de un nodo. Esta información consiste en su dirección IP y el identificador único del nodo. La información de cada nodo es almacenada hasta que se termina la comunicación con éste, entonces se procede a borrarla.

Esta lista sirve para controlar los nodos ya conocidos, de forma que no se pueda establecer más de una conexión por ITS-S. Puesto que una ITS-S puede tener varias OBU y por tanto varias IP, en el manejo de esta lista se considera el campo ID, el cual es único por ITS-S. Al llegar un mensaje HI al cliente éste comprueba la existencia de ese nodo en la lista, lanzando un hilo para conectar a él en el caso de que no se encuentre.

La segunda lista creada y usada por la aplicación es la de peligros conocidos. Aquí se crea una estructura por cada peligro conocido del que se tenga constancia en la carretera. Esta información es adquirida típicamente a través del servidor local, el cual la recibe de otras ITS-S. Sería posible adaptar la aplicación para que reciba peligros existentes por otros medios, por ejemplo mediante la lectura de un fichero o un mensaje desde otro módulo de la ITS-S.

Un peligro está identificado por su posición considerándose por simplicidad que no puede haber dos peligros exactamente en la misma posición. Actualmente el listado de peligros conocidos se borra únicamente al finalizar el cliente de manera ordenada, aunque lo aconsejable sería que los peligros expirasen o al cabo de un intervalo (usando una marca de tiempo) o al alejarse de ellos una determinada distancia (cuando estén tan lejos que no sean de interés para ninguno de los nodos que puedan estar alrededor).

4.4.6 **Tipos de mensajes creados**

El servicio desarrollado hace uso de 6 tipos de mensajes, identificados por un código de dos letras que se incluye como el primer campo del mensaje. Cada mensaje consiste en el envío de una estructura del mismo nombre que el código identificador.

- **HI:** Mensaje de saludo o descubrimiento, es enviado por el servidor de cada OBU a la dirección de difusión de la red. Su función es la de anunciar en la red la existencia de la ITS-S, junto con su ID, su velocidad, posición y carga de la batería actuales. Es un mensaje UDP enviado al puerto 21215.
- **DA:** El mensaje DA (*Danger Advertisement*) contiene la posición y tipo de un peligro que afecte al nodo receptor en su trayectoria actual. Este mensaje se manda a través de una conexión TCP establecida desde un cliente de una ITS-S que ha recibido un mensaje HI hacia el servidor emisor del mensaje. Se lanza tras procesar el HI recibido y haber encontrado en el listado de peligros uno que cumpla las condiciones de alerta en

función de los datos transmitidos por el nodo emisor del HI. Cuando varios nodos reciben un mismo HI, puede darse el caso de que estos envíen sendos mensajes DA del mismo peligro. Para evitar información duplicada en la lista, se supondrá que solo existen un peligro en una posición determinada, no añadiendo el nuevo peligro a la lista si ya existe uno en esa posición. Debido a que la información enviada es de suma importancia para la seguridad de los ocupantes del vehículo servidor, el cliente emisor de este mensaje espera recibir un mensaje DR como confirmación de su recepción.

Al ser recibido un mensaje DA en el servidor, este reenvía ese mismo mensaje a través de la conexión local creada en el dispositivo. El cliente local almacena el peligro en el listado de peligros conocidos en caso de que sea nuevo.

- **DR**: Este mensaje únicamente está conformado por su código DR (*Danger Receive*), sirviendo de confirmación de que el otro extremo ha recibido correctamente un mensaje DA. Si tras un tiempo prudencial este mensaje no se ha recibido, el cliente reintenta el envió del mensaje DA hasta cinco veces. Al expirar todos los intentos sin recibir el mensaje DR, se considera que el otro extremo ha salido del alcance de comunicación radio o que hay demasiadas interferencias. En este punto se finaliza la conexión TCP y se marca al nodo en la lista de conocidos para borrar.
- **SR**: Este mensaje es únicamente enviado cuando no existe un peligro inmediato para el otro extremo de la comunicación. SR (*Status Request*) es un mensaje consistente únicamente en el código del mensaje. Este mensaje es enviado a través de la conexión TCP cliente-servidor externo establecida tras el HI. Con ello se espera recibir como respuesta un mensaje ST con datos del recorrido que lleva realizado el otro vehículo.
- **ST:** El mensaje ST (Status) es la respuesta por parte del servidor externo al envió de un mensaje SR. Este mensaje contiene la distancia, tiempo y nivel de carga de la última recarga del trayecto que se está realizando actualmente. La obtención de los valores a enviar se ha aislado en una función aparte para permitir que sobre ésta se pueda programar desde donde se obtienen los datos a enviar.
- **SF**: El servidor que recibe este mensaje comienza su rutina de apagado. Se ha usado durante las pruebas con el sistema ITS y se ha dejado en el programa para que este se cierre trascurrida una cierta actividad. Con este mensaje en un futuro se puede implementar un cierre ordenado de la aplicación en caso de error grave.

En la Figura 28 se muestra la salida por pantalla producida durante la ejecución por el envío y recepción de los diferentes mensajes.

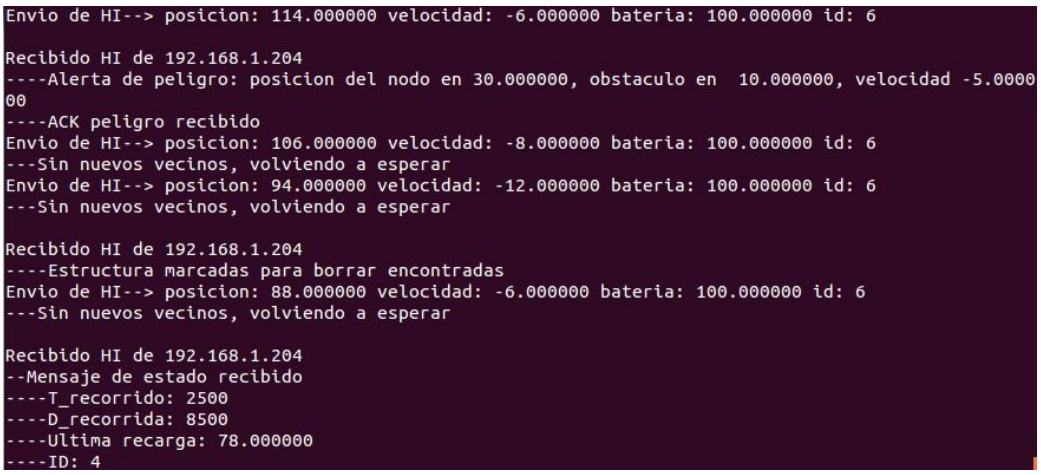


Figura 28: Envió y recepción de mensajes.

Los mensajes especificados dotan al servicio de sus funcionalidades básicas. Pensando en futuras ampliaciones de la aplicación, el servidor se ha codificado de forma que resulte relativamente sencillo el añadir nuevas clases de mensajes. Para ello, el código del mensaje es leído e identificado para llamar a la función de procesamiento correspondiente a ese mensaje.

Únicamente hace falta crear una nueva estructura en "librería.h" (cuyo primer campo sea el código de identificación) y añadir una sentencia if en la zona de discriminación de mensajes que llame a la función de procesamiento creada.

4.4.7 **Descripción del cliente**

La parte correspondiente al cliente de la aplicación está compuesta por un proceso principal y los distintos hilos de atención creados por este.

El proceso en su inicio crea un socket de escucha en el puerto 21215, al que irán dirigidos los mensajes HI del servicio. También intenta establecer conexión al puerto TCP 21221 del servidor interno. Para este paso es necesario que el servidor esté en funcionamiento. Si no se establece la conexión o ésta se corta, el cliente supone que el servidor no se está ejecutando. En caso de estar caído, el cliente continúa su ejecución pero no recibirá nueva información procedente de otras estaciones a excepción del contenido de los HI.

Una vez que se están escuchando los mensajes HI y establecida (o no) una conexión con el servidor local, el cliente entra en un bucle donde se va comprobando cada cierto tiempo determinado si se ha recibido algún mensaje por el socket de escucha UDP o alguna comunicación por la conexión local.

En el caso de recibir un mensaje de saludo éste se lee obteniendo de la cabecera la dirección IP de la OBU que lo ha enviado y su posición, velocidad e ID de los datos contenidos en él. Debido a que el mensaje HI se lanza a difusión cada cierto tiempo para anunciar la existencia de la ITS-S, no es raro estar recibiendo anuncios desde una ITS-S mientras se está en contacto con ella. Por ello, antes de realizar una conexión con el nodo transmisor de un HI, se comprueba si su ID coincide con alguno de los nodos guardados en la lista de nodos conocidos. Si se

encuentra en esta lista se procede a actualizar su dirección IP, abarcando el caso de que un router ITS haya tenido que cambiar su dirección (por ejemplo porque estas sean reasignadas en cada RSU a la que se conecta, en la red ad-hoc haya otro vehículo con la misma IP, o porque una de las OBU haya fallado y la ITS-S haya cambiado a otra para continuar usando el servicio).

En caso de no encontrar el nodo en esta lista, se reserva memoria para una nueva estructura DIRIP enganchándola al final de la lista, añadiendo los datos del nuevo nodo. Tras cada proceso de búsqueda se realiza automáticamente otro en el que se eliminan las estructuras marcadas para borrar. De esta forma se libera espacio y se permite que se pueda realizar una nueva conexión con los nodos ya atendidos.

Para las ITS-S nuevas, el cliente lanza un nuevo hilo para intercambiar información con esa ITS-S. La conexión se realiza mediante TCP, protocolo adecuado para intercambios de información que no pueda sufrir pérdidas.

En primer lugar, el hilo al establecer la conexión comprueba, en función de la posición y velocidad anunciadas por el otro extremo, si hay algún peligro u obstáculo interpuesto en la trayectoria. Los criterios seguidos para considerar el envío de una alerta son:

- El peligro debe encontrarse a menos de una distancia determinada (se ha dejado 20[6] de ejemplo) de la posición actual de la ITS-S.
- La ITS-S debe estar dirigiéndose hacia el peligro.

Ante un resultado positivo de la búsqueda en la lista de peligros, se genera una estructura DA conteniendo la posición del peligro, su tipo, y la ID de la ITS-S de donde procede la información. Estos criterios son fácilmente modificables en la función de cálculo de las alertas.

El mensaje creado es enviado a través del túnel TCP al servidor de la otra estación, esperándose un mensaje DR como confirmación de que se ha recibido la información. Si no se recibe en un intervalo de un número determinado de segundos, se reenvía el mismo mensaje repitiendo el proceso hasta en 5 ocasiones. Tras el quinto intento se abandona, considerándose que la otra ITS-S ha salido de alcance.

En caso de que se considere que no haya ningún peligro inmediato para la ITS-S emisora del HI, se le pide que comparta sus datos mediante el mensaje SR. Estos datos son devueltos por el servidor interno mediante una estructura ST siendo los datos la distancia recorrida y tiempo desde el inicio del trayecto actual y el nivel al que quedo la batería en su última carga.

Otros tipos de mensajes y funciones pueden ser implementados. El proceso para implementarlas esta descrito más adelante.

[6] Los valores aquí expuestos no corresponden a ninguna unidad de medida en particular.

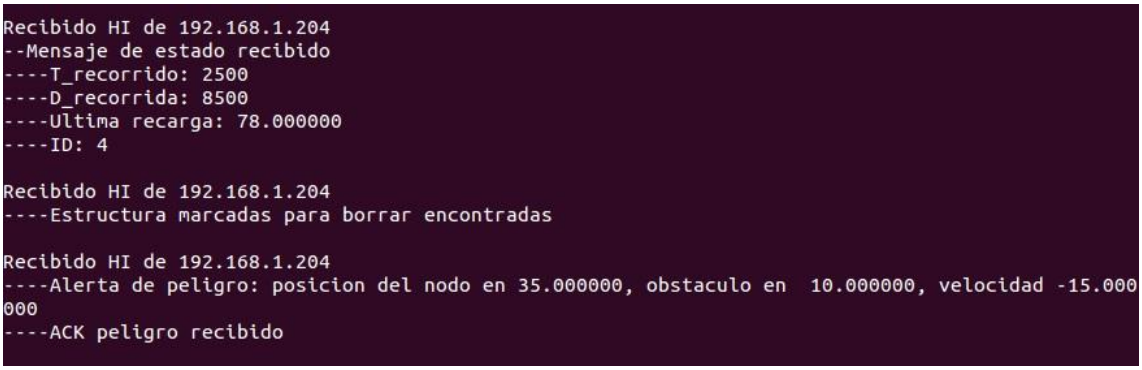


Figura 29: Posibles salidas por pantalla del cliente en función de los mensajes recibidos.

Por último, tras cortar la comunicación TCP y justo antes de finalizar la ejecución del hilo, se marca para borrar la estructura que identifica al nodo externo de forma que más adelante pueda iniciarse otro intercambio de datos al recibir un HI. Para ayudar a entender el funcionamiento aquí explicado, se proporciona la Figura 30.

Actualmente el proceso cliente contiene un contador de iteraciones del bucle de lectura de mensajes HI, iniciando la rutina de salida al llegar a 25 ejecuciones. Esto se realiza para permitir un fin controlado del programa durante las pruebas. En una implementación real, este contador debería ser sustituido por una señal que se lance al apagar el vehículo u ocurrir una excepción grave. En la rutina de salida se envía el código SF al servidor interno (iniciando este su propia salida) y se borran las estructuras existentes en las listas de nodos y peligros conocidos.

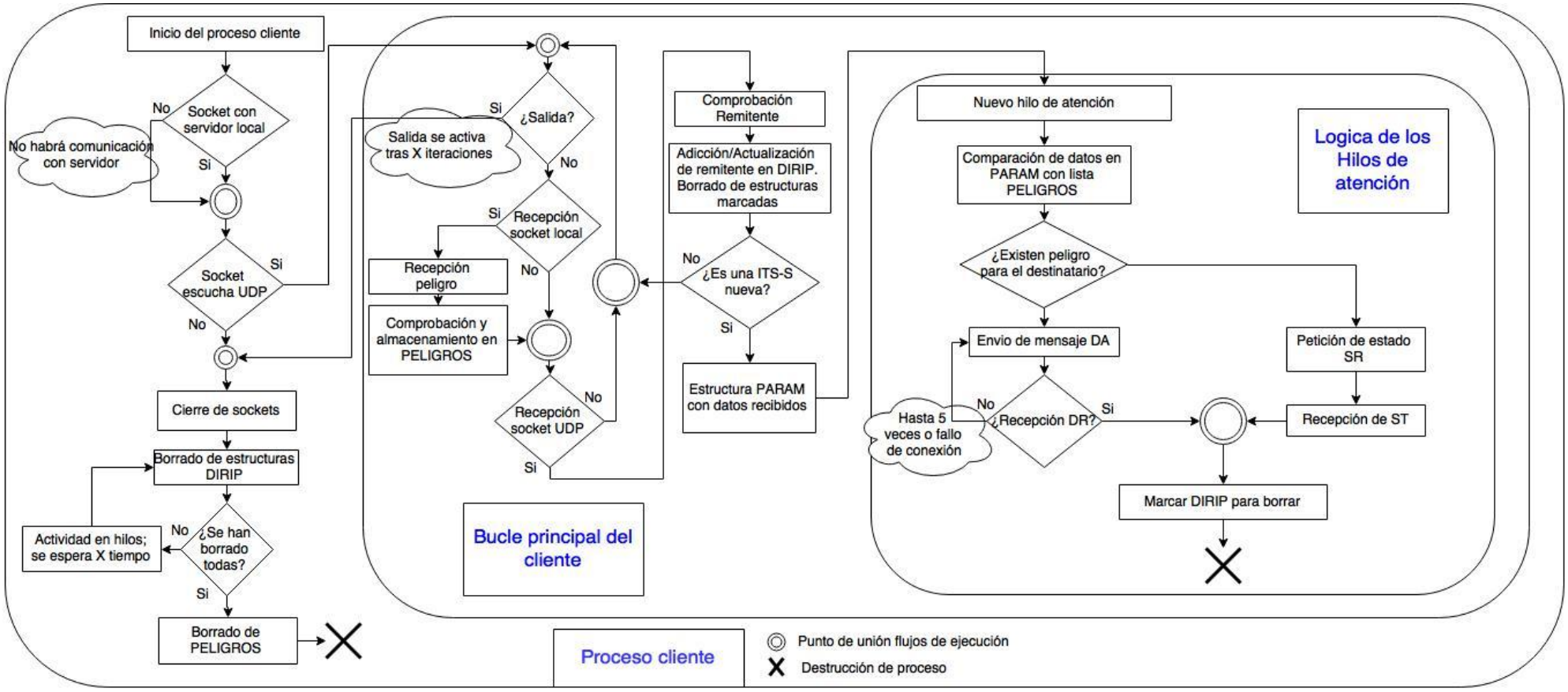


Figura 30: Diagrama de flujo del proceso cliente.

.

4.4.8 **Descripción del servidor**

El servidor se compone de un único proceso que tiene dos funciones principales:

- El envío de mensajes HI comunicando la existencia de la ITS-S y sus datos básicos.
- La atención a las conexiones que se quieran realizar desde otras ITS-S para intercambiar información.

El diagrama de este proceso se inicia con el establecimiento de dos sockets de escucha TCP, uno en el puerto 21210 para aceptar conexiones desde clientes situados en ITS-S externas y otro en el puerto 21221 para establecer una conexión con el cliente local. Además de estos dos sockets, se prepara otro más, este UDP, para el envío a difusión de los mensajes HI.

Una vez terminada la preparación de los sockets, el servidor entra en el bucle de procesamiento. En cada pasada de este bucle se envía un mensaje HI al resto de nodos que puedan existir en la red y se espera durante cierto tiempo posibles peticiones de conexión. Por defecto, el contenido de este mensaje se crear a partir de los valores leídos de un fichero "posición" (por defecto buscado en "*/bin*") con una estructura parecida al formato CSV (*comma-separated values)* con valores (posición,velocidad,batería,id:….).

Esta petición puede proceder del cliente interno (que siempre se inicia un par de segundos más tarde que el servidor) estableciendo la conexión local entre las dos partes de la aplicación. Si no se ha recibido un intento de conexión por parte del cliente, se considera que ésta caído, y la información que se reciba no se intentará comunicar a éste. Es este caso, se ha escogido descartar la información al no poder pasársela al cliente, aunque aquí podrían implementarse otros criterios de actuación como:

a. Almacenar, en el propio servidor, la información y procesarla, abrir los puertos del cliente y tomar las funcionalidades de éste.
b. Mostrar la información directamente a los usuarios del vehículo antes de descartarla.

En caso de petición de un servidor externo se crea un socket para atenderlo y se establece la conexión TCP esperando durante cierto tiempo la recepción de algún mensaje por parte de éste. Pasado el intervalo sin recibir ningún mensaje se corta la conexión para evitar ocupar ancho de banda en la VANET.

Tanto si se recibe un mensaje a través de la conexión local como a través de la externa, su código de identificación es leído, usándolo para clasificar el mensaje y ejecutar la función correspondiente al procesamiento de ese mensaje.

Una de las ventajas que tiene el código creado es que las conexiones locales y las externas son recibidas y leídas en zonas diferentes del mismo. Esto permite que en un futuro sea posible establecer actuaciones diferentes para un mismo mensaje dependiendo de si éste es local o externo.

La recepción de un mensaje DA implica que un cliente externo está avisando de que hay algún peligro próximo en la dirección en la que se mueve la ITS. El peligro recibido es inmediatamente enviado al cliente local para su catalogación en el listado, informándo a su vez al nodo que lo ha enviado de su correcta recepción mediante un mensaje DR. Es posible que el cliente externo por algún motivo no reciba este mensaje DR, volviendo a enviar un mensaje DA avisando del peligro. Este mensaje se trata como uno independiente al anterior, pues el problema puede haber estado en que el primero no fue recibido.

En caso de que no haya ningún peligro inminente, el cliente externo preguntará por el trayecto actual mediante un mensaje SR. La respuesta dada por el servidor será un mensaje ST cuya creación se modulariza en una función independiente con idea de que en ella se defina de qué forma se consigue esa información. Para el desarrollo actual, los valores se leen de un fichero llamado "estado" escrito con un formato concreto (recorrido,tiempo,recarga:…) y situado en el directorio "*/bin*", lugar de instalación del programa (si se usa el makefile proporcionado). En caso de no existir este fichero, son transmitidos unos valores por defecto.

```
Codigo DA recibido: Recibiendo peligro....
Peligro en posicion 61.000000
Tipo: 1
ID: 2
Cliente-->AVISO, obstaculo en la carretera
Envio de HI--> posicion: 54.000000 velocidad: -8.000000 bateria: 85.000000 id: 6
---Sin nuevos vecinos, volviendo a esperar
Envio de HI--> posicion: 42.000000 velocidad: -12.000000 bateria: 80.000000 id: 6
Codigo SR recibido: Recibida peticion de estado.
Envio de HI--> posicion: 36.000000 velocidad: -6.000000 bateria: 75.000000 id: 6
---Sin nuevos vecinos, volviendo a esperar
Envio de HI--> posicion: 28.000000 velocidad: -8.000000 bateria: 70.000000 id: 6
Codigo SR recibido: Recibida peticion de estado.
Envio de HI--> posicion: 16.000000 velocidad: -12.000000 bateria: 65.000000 id: 6
---Sin nuevos vecinos, volviendo a esperar
Envio de HI--> posicion: 10.000000 velocidad: -6.000000 bateria: 60.000000 id: 6
Codigo DA recibido: Recibiendo peligro....
Peligro en posicion 10.000000
Tipo: 1
ID: 5
Cliente-->Peligro ya conocido, no se vuelve a lanzar aviso
Envio de HI--> posicion: 2.000000 velocidad: -8.000000 bateria: 55.000000 id: 6
---Sin nuevos vecinos, volviendo a esperar
```

Figura 31: Salida por pantalla del proceso servidor.

El servidor empezará una nueva interacción del bucle (enviando un mensaje HI) tanto si se ha procesado alguna conexión como si se ha agotado el tiempo de espera sin recibir ninguna. La única excepción se da ante la recepción de un mensaje SF, lanzando el fin del bucle cierra los sockets y conexiones existentes iniciando la rutina de fin de proceso. La Figura 32 representa la lógica del servidor.

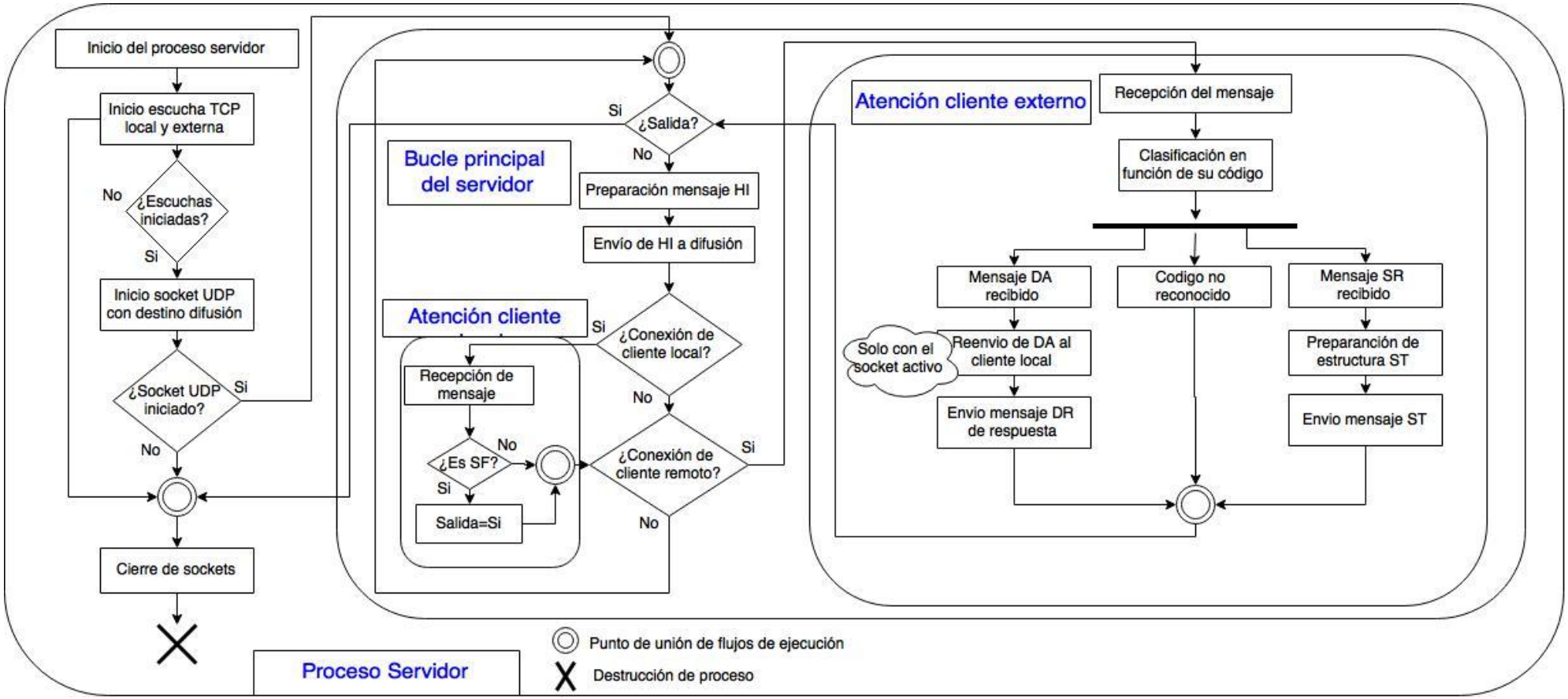


Figura 32: Diagrama de flujo proceso servidor.

.

4.4.9 **Consideraciones finales del código**

Durante la fase de diseño de la aplicación se ha prestado especial atención al sistema donde va a ser implantada. La forma de codificación está enfocada a su uso en la parte WAVE de una red ITS. Un servicio distribuido en un sistema de transportes se ejecuta en un escenario donde en algunos casos puede existir una alta densidad de vehículos, lo que se traduce en una red que puede estar saturada y la necesidad de que el servicio sea escalable, de forma que se pueda estar usando en multitud de vehículos de forma simultánea. Como aplicación de seguridad debe ser especialmente resistente a los fallos que se produzcan, debiendo poder hacer frente a situaciones imprevistas en la medida de lo posible. Por último, la arquitectura de dos procesos también ofrece algunas desventajas, como la posibilidad de que se produzcan condiciones de carrera (ambos procesos intentando acceder a la misma variable a la vez). Estas situaciones deben tenerse en cuenta a la hora de planificar el funcionamiento del programa, de manera que se eviten zonas de acceso comunes o que éstas estén bien controladas durante la ejecución, para evitar conflictos.

4.4.9.1 **Escalabilidad del servicio**

La escalabilidad es una propiedad fundamental de los programas distribuidos. La cantidad de nodos que hagan uso de un servicio puede influir de forma significativa, ocasionando que aplicaciones que en principio funcionan correctamente en una red de pocos nodos, fallen al someterlas a las redes más pobladas. Por supuesto, este concepto tiene que ir ligado al uso previsto, siendo igual de absurdo enfocar el diseño de un servicio público en Internet a una docena de ordenadores, que considerar el uso de miles de equipos en un programa circunscrito a una red doméstica.

En el estudio actual se considera que el número de ITS-S usuarias con las que se tiene contacto directo en la red ad-hoc, variará entre menos de una decena para carreteras poco transitadas y más de una centena en una retención localizada. Al no realizar pruebas prácticas (no se dispone ni de tiempo ni de material) ni simulaciones (que escapan al carácter práctico del proyecto) no se puede afirmar que el prototipo cumpla estas exigencias. Aún así, su diseño se ha orientado a estos números, viéndose reflejado en la estructura del código en muchas cosas ya comentadas.

La división cliente-servidor dentro de la aplicación permite dividir la carga de trabajo de las conexiones, lo que aumenta el rendimiento en el caso de microprocesadores con dos núcleos. El dejar la conexión a los servidores externos en hilos independientes permite que el proceso del cliente se centre en recibir mensajes HI y catalogarlos, previéndose que esto mejore su rendimiento en redes en las que existan decenas de servidores enviando esta clase de mensajes.

El uso de hilos no se ha implementado en el servidor para evitar condiciones de carreras que requieren del uso de semáforos (recordemos que estos no están soportados por limitaciones de software impuestas por la librería de compilación usada en OpenWrt). Para minimizar este hecho en el servidor, la mayor parte del procesamiento de datos se ha pasado al proceso cliente, reduciendo las funciones del servidor a atender las conexiones que le lleguen de forma ordenada, con lo que se cree que será capaz de soportar una red de decenas de nodos. Se contempla el caso de que debido a la cantidad de peticiones de conexión al servidor, alguna de ellas pueda expirar antes de ser atendidas. En ese caso el cliente volvería a lanzar la petición con el siguiente HI del servidor, según se ha descrito anteriormente.

4.4.9.2 **Condiciones de carrera**

El uso de varios procesos aligera la carga del programa, volviéndolo más eficiente y permitiendo funcionar con los dos roles de forma simultánea. Sin embargo, la división del trabajo en varios hilos o procesos siempre conlleva el riesgo de que haya accesos simultáneos a una misma zona de memoria, pudiendo ocasionar lecturas o escrituras de valores inesperados si no se trata de forma correcta. Este suceso se denomina condiciones de carrera. El obstáculo se vuelve mayor cuando no se disponen del uso de cerrojos o semáforos que permitan que las operaciones en la zona de riesgo se realicen de forma atómica.

La primera medida que se ha tomado para minimizar este problema ha sido independizar la información a la que acceden las dos zonas del software. En caso de que el servidor requiera algo del cliente y viceversa se ha hecho uso de una conexión TCP. Las conexiones TCP pueden verse como tuberías en donde se introduce información por un extremo y esta sale por el otro. Tienen la característica de que son bidireccionales, de forma que la información transmitida en una dirección es independiente de la que se envía en la opuesta. Reducir el intercambio de información a este formato permite eliminar las colisiones en memoria.

Dentro del cliente se tiene un proceso principal y unos hilos de atención. Debido a su funcionamiento tanto los hilos como el cliente deben acceder a los dos listados especificados anteriormente: nodos y peligros conocidos. La solución aquí no pasa por conexiones TCP, el número de hilos puede ser alto y, establecer una conexión con cada uno de ellos desde el cliente, ralentizaría el funcionamiento de éste. En vez de discriminar la entrada a las listas, se ha optado por minimizar y controlar las posibles colisiones que se produzcan en ellas.

Los hilos requieren del acceso a las estructuras PELIGROS y DIRIP en dos ocasiones. La primera de ellas es en el cálculo de una posible alerta del peligro, en la que el hilo debe leer toda la lista de peligros conocidos para compararlos con la posición y velocidad recibida. El acceso realizado aquí es únicamente de lectura. El segundo acceso se realiza a la estructura DIRIP del servidor con el que el hilo debe contactar. Aquí es necesario leer los datos (Dirección IP del hilo) para poder abrir una conexión TCP con éste.

Además de lo anterior. Las estructuras DIRIP de un nodo deben ser borradas cuando se corta la comunicación con este nodo. Realizar este paso es complicado, tan solo los hilos que crean los sockets con esa ITS-S conocen cuando se cierra la comunicación, mientras que el único que puede añadir una nueva estructura a la lista es quién decide si se lanza un nuevo hilo o no, es decir, el proceso cliente.

La solución de esta encrucijada pasa por asegurarse de que un hilo únicamente va a acceder a la estructura DIRIP del nodo al que va a contactar y que ésta no será modificada mientras se estén realizando acceso sobre ella. Por ello se ha definido un nuevo campo en estas estructuras llamado *flag*, cuyo cometido es el de marcar si la estructura se ha dejado de usar. El proceso cliente leerá todos los nodos al recibir un HI pero únicamente le pasará la dirección de la estructura a usar al hilo de atención. Este hilo realiza accesos a la estructura hasta que termine la conexión con el nodo, momento en el que la marca para borrar, usando *flag*, justo antes de morir, como se detalla en la **¡Error! No se encuentra el origen de la referencia.** .

Tras comprobar si el remitente del HI está siendo atendido, el cliente realiza una segunda comprobación en la que elimina aquellas estructuras marcadas para borrar. Al acceder cada hilo a su estructura mediante un puntero a ésta, en la lista se pueden añadir, eliminar y cambiar de orden siempre y cuando se mantengan las referencias en memoria.

Un caso particular de condiciones de carrera se podría dar al acceder simultáneamente el cliente (para leer) y un hilo (para modificar) al campo *flag*. Tanto la lectura como el establecimiento del valor en el campo está gestionado por el SO y pueden ser consideradas operaciones atómicas. Si el cliente lee en su caso un 1 la estructura se borrará. En el caso de que el cliente lea un 0 (valor por defecto con la que es creada la estructura DIRIP), tan solo hay que esperar a la recepción del siguiente HI para volver a realizar la comprobación.

4.5 Instalación del programa en OpenWrt

Antes de instalar el software, es necesario compilarlo mediante el proceso ya visto en la sección 4.3. El código del programa está repartido en 8 ficheros y un archivo .h donde se importan las librerías necesarias y se definen las variables y las estructuras de los mensajes usados. Estos ficheros deben ser introducidos en la carpeta "*src"* dentro de una carpeta de nombre "*calm"* en el directorio "*packages"* del SDK. Añadiendo los "*makefiles"* pertinentes y lanzando la compilación con *make*, se genera un paquete .ipkg. Este paquete será el instalador de la aplicación. Una vez que se tiene, es necesario pasarlo a la CF. Usar para ello el lector o el comando *scp*.

Con el .ipkg en el árbol de directorios de OpenWrt y conectado a la placa en funcionamiento, la instalación se realiza mediante el comando:

```
opkg install /ruta/nombre_paquete.ipkg
```

Donde "ruta" corresponde a la ruta correspondiente al directorio donde se ha guardado el .ipkg y "nombre_paquete" al nombre de este fichero. El nombre del paquete es seleccionado en el "*Makefile*"situado al mismo nivel que "*src"* en el directorio de compilación.

El software no tiene dependencia más allá de los paquetes que vienen incluidos por defecto en OpenWrt. En caso de que salga un mensaje de error por la falta de algún módulo, hay que descargarlo del repositorio de OpenWrt e instalar este primero. Si se usa el "*makefile"* la instalación crea un ejecutable "*calm*" en la carpeta "*bin"* del SO. Los ejecutables de esta carpeta son reconocidos como comandos del propio sistema.

4.6 **Puesta en marcha**

Antes de poner en funcionamiento la aplicación, hay que configurar el SO para que use una red ad-hoc a frecuencias CALM. Este proceso se realiza manualmente mediante comandos.

En primer lugar se debe cambiar la región del dominio regulador. El evento del GCDC fue celebrado en los Países Bajos y consecuentemente la región modificada en el CRDA es la correspondiente a este país. Este cambio se realiza con el siguiente comando donde "NL" corresponde al código de país de Netherland.

```
iw reg set NL
```

Usando la opción *get* del comando se puede comprobar en que dominio regulador se está y que frecuencias permite. Esto se muestra en la Figura 33.

```
root@OpenWrt:~# iw reg get
country 00:
        (2402 - 2472 @ 40), (3, 20)
        (2457 - 2482 @ 20), (3, 20), PASSIVE-SCAN, NO-IBSS
        (2474 - 2494 @ 20), (3, 20), NO-OFDM, PASSIVE-SCAN, NO-IBSS
        (5170 - 5250 @ 40), (3, 20), PASSIVE-SCAN, NO-IBSS
        (5735 - 5835 @ 40), (3, 20), PASSIVE-SCAN, NO-IBSS
root@OpenWrt:~# iw reg set NL
root@OpenWrt:~# iw reg get
country NL:
        (2402 - 2482 @ 40), (N/A, 20)
        (5170 - 5250 @ 40), (N/A, 20)
        (5250 - 5330 @ 40), (N/A, 20), DFS
        (5490 - 5710 @ 40), (N/A, 27), DFS
        (5842 - 5932 @ 40), (N/A, 27), DFS
```

Figura 33: Cambio de región y comprobación de frecuencias disponibles.

Una vez situado en el nuevo dominio se puede proceder a pasar la interfaz radio a modo ad-hoc, en caso de que no lo esté ya.

```
ifconfig wlan0 down
iwconfig wlan0 mode ad-hoc
ifconfig wlan0 up
```

A continuación se debe indicar al sistema que se una a la red ad-hoc proporcionándole la frecuencia y el nombre de la red. En este ejemplo se ha usado una red llamada ITS_PRUEBAS situada en la frecuencia 5,9 GHz.

```
iw wlan0 ibss ITS_PRUEBAS join 5900 fixed-freq
```

Este proceso se tiene que realizar en cada AlixBoard antes de ejecutar el servicio. Se puede comprobar la correcta ejecución de este proceso usando el comando *iwconfig*[7].

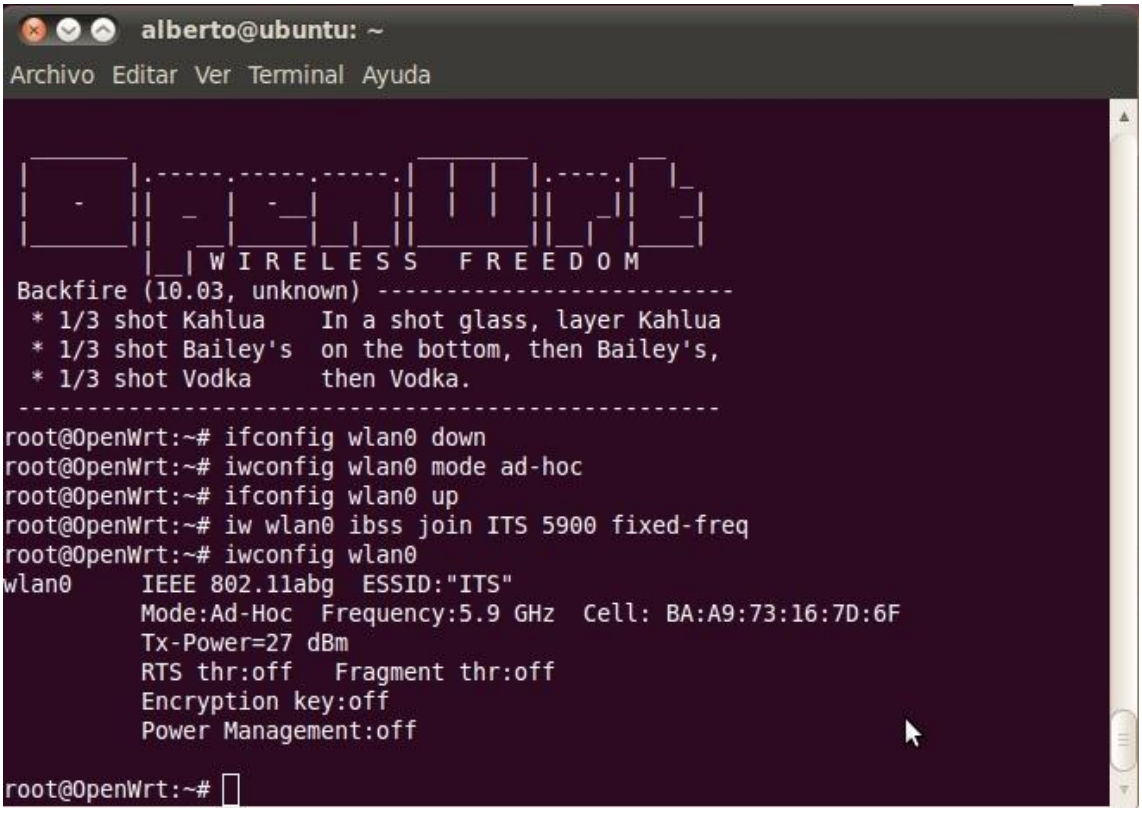


Figura 34: Cambio de frecuencia de transmisión en la AlixBoard 3D2.

Una vez puestas las dos tarjetas en la frecuencia CALM, es recomendable realizar un ping para comprobar que hay conectividad. Se recomienda usar la opción –I del comando ping si ambas Alix están conectadas al *switch*. Esta opción asegura que se use una interfaz concreta para realizar el ping y no dé falsos positivos.

La ejecución del servicio se realiza mediante:

[7] El campo Cell solo tomará un valor cuando haya al menos dos equipos en la red ad-hoc.

```
calm [opción]
```

El programa acepta el paso de un parámetro que puede tener el valor “cliente” o “servidor”. Si se le pasa uno de los dos parámetros, la aplicación únicamente iniciará el proceso cliente o el servidor respectivamente. Esta opción está pensada para facilitar las pruebas individuales de cada parte de la aplicación. En caso de no pasarle ninguna opción, se iniciarán ambos procesos.

Antes de terminar la sección queda un último apunte sobre el uso de ficheros con la aplicación. Como ya se ha comentado, el servidor intenta adquirir los datos de los mensajes ST y HI de ficheros con nombres concretos, usándose valores arbitrarios en caso de no encontrar los archivos. Tal como está implementado, los archivos deben estar situados en el directorio “/bin”, no siendo encontrados por la aplicación si se ubican en otro lugar.

5 Conclusiones finales

Tout ce qu'un homme est capable d'imaginer, d'autres hommes sont capables de le réaliser.

Atribuida a Julio Verne

A modo de desenlace, en este último capítulo se sintetizan los puntos principales que se han ido tratando a lo largo del proyecto. Asimismo se aprovecha para describir posibles mejoras sobre el trabajo realizado, muchas de las cuales se han ido ya exponiendo. Para finalizar se realiza una pequeña valoración personal sobre la posible evolución de los ITS en los próximos años.

5.1 Resumen general

Con el trabajo realizado se ha conseguido obtener un primer avance en la implementación de un sistema de vehículos inteligentes. En primer lugar se ha realizado un acercamiento teórico al concepto de sistemas inteligentes de transportes y a la normativa existente en este campo. Durante esta aproximación, además de ver los arquetipos de los estándares CALM y WAVE creados por la ISO y la IEEE respectivamente, también se ha intentado, en la sección 2.3, abordar la disciplina desde los resultados y soluciones propuestas en los diferentes estudios que versan sobre la materia. Todo esto con el objetivo de reunir información para el sistema propuesto en los capítulos 3 y 4.

A su vez, en la parte práctica se ha visto que es posible conseguir una plataforma factible para la implementación de aplicaciones ITS sobre placas comerciales AlixBoard 3D2 e instalando un sistema operativo embebido (OpenWrt) sobre ellas.

Una vez con soporte para la ejecución de software propio, se ha presentado un servicio ITS para comunicaciones V2V consistente en la detección y aviso de peligros existentes en la calzada, además de la adquisición de una batería de datos de otros nodos usuarios del servicio.

5.2 Propuesta de mejora

Como ya se expuso en la sección Objetivos, debido a la imposibilidad de abarcar todo lo referente a un sistema inteligente de transportes en este trabajo, se ha dejado de lado multitud de conceptos. El sistema aquí presentado es un punto de partida para desarrollos mayores, siendo un pequeño esbozo de lo que realmente serán los ITS usados dentro de algunos años.

Entre lo excluido destaca todo lo referente el estudio e implementación práctica de las capas de control y seguridad propuestas por CALM (y de forma más especifíca por WAVE). La adición de estas capas y del protocolo WSMP a OpenWrt, permitiría estar más cerca de una implementación 100% funcional de un OBU. En este sentido también es necesario conseguir que las tarjetas inalámbricas soporten los modos de alternancia entre frecuencias definidos por WAVE.

En lo referente a las posibles aplicaciones en un ITS, queda el desarrollo de servicios mayores en los que las comunicaciones puedan extenderse a V2I e incluso hagan partícipe a los diferentes elementos del núcleo de la red ITS.

5.3 Futuro de los ITS

El campo de los transportes es uno de los que más atención recibe por parte de la comunidad científica y las empresas. Esto ha contribuido a que en los últimos años haya sufrido

un desarrollo extremadamente rápido, de manera que muchas de las aplicaciones que hace meramente una década se concebían como un concepto se encuentran ya en fase de pruebas para salir al mercado en los próximos años.

La razón: La tendencia actual se centra en aumentar las virtudes de los elementos cotidianos mediante el uso de las TIC, dándoles cada vez mayor inteligencia y capacidad de decisión. En este aspecto los automóviles se perfilan como un sector en el que no es difícil, ni especialmente caro, llevar a la práctica los avances conseguidos con estas tecnologías.

La velocidad de este proceso de mejora también conlleva algunas desventajas. Esta rapidez repercute en la normativa vigente, la cual se ve en la obligación de ser revisada, ampliada e incluso modificada cada poco tiempo, lo que provoca una cierta falta de consenso sobre qué es exactamente un ITS y qué funciones son las que convierten a la infraestructura en inteligente.

Lo que sí está claro, ya sea mediante tecnologías que automaticen la conducción o que simplemente supongan una ayuda al conductor, es que las redes vehiculares han llegado para quedarse y que van a ganar relevancia en la sociedad con el paso del tiempo. Hoy en día no es descabellado pensar en un futuro en el que los coches conversen entre ellos. Los avances que ocurran después de esto, únicamente pueden (de momento) ser imaginados.

Bibliografía

1. **Analistas Económicos de Andalucía**. (2001). *Las infraestructuras de transporte del Eje Mediterráneo andaluz: efectos socioeconómicos (N-340).*
2. **Bilstrup, K., Uhlemann, E., G. Ström, E., & Urban, B**. (Septiembre de 2008). *Evaluation of the IEEE 802.11p MAC method for Vehicle-to-Vehicle Communication.*
3. **Bishop, E**. (8 de Agosto de 2007). *Writing and Compiling A Simple Program For OpenWrt*. (Último acceso en Junio de 2015), http://www.gargoyle-router.com/wiki/doku.php?id=openwrt_coding
4. **Doumenc, H**. (Junio de 2008). *Estudio comparativo de protocolos de encaminamiento en redes VANET.*
5. **Eichler, S**. (Octubre de 2007). *Performance Evaluation of the IEEE 802.11p WAVE Communication Standard.*
6. **ETSI**. (2010-09). *ETSI EN 302 665. Intelligent Transport Systems (ITS); Communications Architecture.*
7. **ETSI**. (2012-14). *ETSI EN 302 363-3. Intelligent Transport Systems (ITS); Vehicular Communications; GeoNetworking; Part 3: Network Architecture.*
8. **IEEE**. (s.f.). *IEEE Smart Cities*. (Último acceso en Junio de 2015), http://smartcities.ieee.org/about.html
9. **IEEE Vehicular Technology Society**. (2012-13). *1609.0-2013 - IEEE Guide for Wireless Access in Vehicular Environments (WAVE) - Architecture.*
10. **IEEE Vehicular Technology Society**. (2010). *1609.3-2010 - IEEE Standard for Wireless Access in Vehicular Environments (WAVE) - Networking Services.*
11. **IEEE Vehicular Technology Society**. (2009). *IEEE 1609.4 DSRC multi-channel operations and its implications on vehicle safety communications .*
12. **IETF**. (1998). *RFC 2460. Internet Protocol, Version 6 (IPv6). Specification.*
13. **IETF**. (1999). *RFC 2663. IP Network Address Translator (NAT). Terminology and Considerations.*
14. **ISO**. (2012). *ISO 21210:2012. Intelligent transport systems -- Communications access for land mobiles (CALM) -- IPv6 Networking.*
15. **ISO**. (2010). *ISO 21215:2010. Intelligent transport systems -- Communications access for land mobiles (CALM) -- M5.*
16. **ISO**. (2010). *ISO 21217:2010. Intelligent transport systems- Communications access for land mobiles (CALM)- Architecture.*
17. **Jiménez Pinto, G., López, D., & F. Pedraza, L**. (28 de Noviembre de 2011). *Simulación y análisis de desempeño de protocolos unicast para Redes VANET.*

18. **Karagiannis, G., Altintas, O., Ekici, E., Heijenk, G., Jarupan, B., Lin, K., y otros**. (3 de Noviembre de 2011). *Vehicular networking: A survey and tutorial on requirements, architectures, challenges, standards and solutions.*
19. **OpenWrt Wireless Freedom**. (s.f.). *Pagina oficial de OpenWrt*. (Último acceso en Junio de 2015), https://openwrt.org/
20. **PC Engines**. (2010). *Alix.2/ Alix.3/ Alix.6 series system boards.*
21. **PC Engines**. (s.f.). *Web del fabricante de las AlixBoard*. (Último acceso en Junio de 2015), http://www.pcengines.ch/alix2.htm
22. **Reutner-Fischer, B**. (1999). *Pagina web de uClibc*. (Último acceso en Junio de 2015), http://www.uclibc.org
23. **Routerboard**. (s.f.). *Especificaciones de la Mikrotik R52H*. (Último acceso en Junio de 2015), http://routerboard.com/R52H
24. **Sampedro Armenta, J. M**. (2013). *PFC: Comunicación cooperativa entre vehículos: Estudio e implantación del protocolo ISO CALM en un sistema embebido usando OpenWrt.* Sevilla: Universidad de Sevilla.
25. **Tecnovía S.A.** (s.f.). *Página web de Tecnovía*. (Último acceso en Junio de 2015), http://www.tecnovia.info/productos/its.php
26. **Vandenberghe, W., Moerman, I., & Demeester, P**. (Agosto de 2011). Approximation of the IEEE 8 02.11p standard using commercial off-the-shelf IEEE 802.11a hardware.
27. **VMware Inc**. (s.f.). *Portal web de vmware*. (Último acceso en Junio de 2015), de http://www.vmware.com/es

Anexo A

Este anexo muestra el código desarrollado para el servicio ITS descrito en la memoria. El código fuente ha sido dividido en 9 ficheros (8 archivos de funciones y 1 que define las librerías, funciones y estructuras usadas por el programa). La división facilita el manejo y la mejora del programa agrupando el código por funcionalidad. Al principio de cada fichero se pueden encontrar comentarios que indican las funcionalidades recogidas dentro.

Para facilitar su uso en ordenadores y sistemas operativos que no usen la codificación UTF-8, se han omitido las tildes y eñes del código fuente.

LIBRERÍA.H

```
#ifndef LIBRERIA_H
#define LIBRERIA_H

/*
 * libreria.h
 *
 *  Created on: 05/05/2015
 *      Author: Alberto Rodriguez Blazquez
 *      E-mail: albertoroblaz@hotmail.com
 *      TFG: Diseño y desarrollo de un sistema ITS basado en comunicaciones tipo ISO CALM.
 *      Descripcion: Estructuras, funciones, valores y librerias usadas por la aplicacion.
 *      Version: 0.9 Final
 *
 */

/***Librerias del sistema***/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <signal.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <pthread.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <ifaddrs.h>


//Constantes usadas en las funciones
#define DST_PORT    "21210"     //Puerto escucha servidores. (Conexiones tcp).
#define SC_PORT     "21221"     //Puerto escucha servidores. (Conexion tcp con cliente propio).
#define CBROAD_PORT "21215"     //Puerto escuchar clientes. (Destino HI periodico).

#define HELLO_GROUP     "192.168.1.255"     //Direccion IP de difusion de la red.
                                //IMPORTANTE: Debe pertenecer a la subred de la IP de la interfaz usada.
```

```
#define MAX_BUFF    1024        //Tamano de la tabla de lectura en el envio/recepcion de ficheros.
#define IP_LENGTH   16          //Tamano maximo de una IP version 4
#define ERROR       -1          //Valor usado para indicar que se ha producido alguna clase de error.

//Valores usados en la invocacion de traduce_a_direccion.
#define SOCKET_UDP 0
#define SOCKET_TCP 1
#define SOCKET_TCP_PASIVO 2
#define SOCKET_UDP_PASIVO 3

//Valores de ejecucion del servicio.
#define T_ESPERA        1       // Tiempo de espera activa en la comprobacion de los sockets. (Afecta a la
velocidad del programa).
#define T_HELLO         4       //Tiempo minimo entre un hello y el siguiente. (Afecta a la carga de datos em la
red)

#define NUM_ITER    1250            //Numero de iteraciones del bucle del cliente antes de iniciar salida.
                        //(Poner a -1 si se quiere desactivar)
#define DIST_ALERTA 20          //Distancia al peligro a partir de la cual se emite una alerta.
#define IDDISPOSITIVO 6


/*Este programa usa usa seria de tipo de mensajes propios. Cada mensaje lleva al
inicio un codigo de 3 char identificando la clase de mensaje que es*/

/* LISTADO DE CODIGOS USADOS:

HI--> Mensaje de hola
AR--> Recibir fichero del cliente.
SF--> Cerrar servidor.
DA--> Mensaje alerta.
ST--> Mensaje de status.
DR--> Peligro recibido.
SR--> Peticion de estado.

*/

/*A continuación la estructura de cada tipo de mensaje:*/
//Los mensajes SF y DR llevan unicamente su codigo de identificacion.
//El mensaje AR lleva un identificador y una cadena char de tamano maximo MAX_BUFF.


/*Estructura de los mensajes complejos*/
//Estructura enviada en los mensajes HI (descubrimiento).
struct HI
{
    char codigo[3];
    float pos;
    float vel;
    float bateria;
    int id;
};

//Estructura mensaje DA (Alerta de peligro).
struct DA{
    char codigo[3];
    float pos;
    int tipo;
    int id;
};

//Estructura mensaje ST (Estado del vehiculo).
```

```
struct ST
{
   char codigo[3];
   long t_recorrido;
   int d_recorrido;
   float recarga;
   int id;
};


/*Ademas de los mensajes, se han usado 3 clases de estructuras auxiliares**/

//El listado de estructuras DIRIP guarda la informacion sobre los nodos descubiertos.
struct DIRIP
{
   char direccion [IP_LENGTH];
   int id;
   int flag;
   struct DIRIP * next;
};

//Cada estructura PELIGROS almacena la posicion de un peligro notificado.
struct PELIGROS{
   float pos;
   int tipo;
   int id;
   int flag;
   struct PELIGROS * next;
};


//PARAM es usado durante el calculo de los avisos de peligro.
struct PARAM{
   char direccion [IP_LENGTH];
   struct DIRIP * nodo_ip;
   struct PELIGROS ** inicio_alertas;
   float pos;
   float vel;
   float bateria;
   int tipo;
   int id;
   sem_t * s_nodos;
   sem_t * s_peligros;
   struct DA peligro;
};

/*LISTADO DE FUNCIONES DE LA APLICACION*/

//Funciones en cliente.c (Logica principal de los clientes).
int cliente();
void *atencion_mensaje(void *argc);

//Funciones en servidor.c (Logica principal de los servidores).
int servidor();

//Funciones en comun.c (Funciones usadas por cliente y servidor).
int traduce_a_direccion(const char *maquina, const char *puerto, int tipo, struct addrinfo *info);
char * obtenerIPPropia(char* interfaz, char *host);
void *get_in_addr(struct sockaddr *sa);

//Funciones en sockets.c (Funciones para la creacion y puesta en marcha de los sockets).
int inicia_socket_cliente(const char *maquina, const char *puerto);
int iniciaEscuchaUDP(char *puerto);
```

```
int iniciaEscuchaTCP(char *puerto);
int iniciaHolaUDP(struct sockaddr_in * sock_int);
int comprueba_recepcion(int descriptor, int segundos);

//Funciones en peligros.c (Administracion de la informacion almacenada sobre avisos).
int anadirPeligroConocido(struct PELIGROS ** inicio_lista, struct DA peligro);
int buscarPeligro(struct PELIGROS * inicio_lista, float posicion);
int borrarPeligro(struct PELIGROS ** inicio_lista, float posicion);
int borrarPeligros(struct PELIGROS ** inicio_lista);

//Funciones en nodos.c (Manejan las estructuras que guardan los estremos conocidos).
struct DIRIP* anadirNodoConocido(struct DIRIP ** inicio_lista, struct HI saludo, char *direccion_ip );
struct DIRIP* buscarNodo(struct DIRIP * inicio_lista, char* direccion_ip);
int borrarNodo(struct DIRIP ** inicio_lista, int id);
int borrarNodos(struct DIRIP ** inicio_lista);
int borrarNodosMarcados(struct DIRIP ** inicio_lista);

//Funciones en mensajes.c (Procesamiento, recepcion y envio de los mensajes).
struct DA calculos_alertas(struct PARAM parametros);
void adquirirDatos(struct HI * datos);
int procesarPeligro(int socket_interno, int socket_externo);
int devolverEstado(int socket_externo);
int recibirAlerta(int sock, struct DA* alerta);
void datos_default(float * pos, float * vel, float* bat);
int envia_fichero(int socket);
int recibir_fichero(int socket);

    #ENDIF
```

INICIO.C

```
/*
 * inicio.c
 *
 *  Created on: 05/05/2015
 *      Author: Alberto Rodriguez Blazquez
 *      E-mail: albertoroblaz@hotmail.com
 *      TFG: Diseño y desarrollo de un sistema ITS basado en comunicaciones tipo ISO CALM.
 *      Descripcion: Inicio del programa CALM, lanzamiento del servidor y el cliente en procesos separados.
 *       Version: 0.9 Final
 *
 */

 /*BIBLIOGRAFIA
  *
  *   https://github.com/PingChung/C_socket_file_transfer
  *   http://stackoverflow.com/questions/337422/how-to-udp-broadcast-with-c-in-linux
  *   http://www.tack.ch/multicast/broadcast.shtml
  *   http://www.ccplusplus.com/2011/09/udp-broadcast-client-server-example.html
  *   http://stackoverflow.com/questions/2283494/get-ip-address-of-an-interface-on-linux
  *   http://stackoverflow.com/questions/11135801/how-to-broadcast-message-using-udp-sockets-locally
  */

 #include "libreria.h"

/*
 *  Funcion principal de la aplicación. Lanza el servidor y el cliente en funcion del parametro pasado.
 *  Forma de invocacion: calm [opcion]
 *  opcion== cliente--> Se lanza unicamente el cliente de la aplicación.
 *  opcion== servidor--> Se lanza solamente el servidor de la aplicación.
 *  sin opcion--> Se lanzan cliente y servidor.
```

```
 *
 *  Paramentros de entrada:
 * -opcion --> En formato argc/args.
 *
 *  Parametros de salida:
 * -Ninguno
 */

int main(int argc, char* args[]){

  pid_t pid;                //Identificador de proceso
  int control=0;            //Controla si se ejecuta el cliente, el servidor o ambos.
  char server[]="servidor";   //Posible valor del parametro de entrada.
  char client[]="cliente";    //Posible valor del parametro de entrada.


//Pequeno control para poder lanzar el cliente y servidor por separados.
if(argc>2)
{
  control=-1;
} else {
  if(argc==2){
    if(strcmp(args[1],server)==0)
    {
      control=1;
    } else {
      if(strcmp(args[1],client)==0)
      {
        control=2;
      } else {
        control=-1;
      }
    }
  }
}
    if(control!=-1)
    {

       pid=fork();   //Divide el programa en dos procesos, uno con pid=0 y el otro con pid=Nº de proceso.

       switch(pid)
       {

         case -1:
         perror("fork");
         break;

         case 0:
         if(control==0 || control==1)
         {
           //Aqui todo lo relativo al servidor. (Proceso hijo)
           printf("Inicio del servidor\n");
           printf("--------------------\n");
           servidor();
         }
           break;

         default:
         if(control==0 || control==2)
         {
           //Aqui todo lo relativo al cliente. (Proceso padre)
           sleep(2);     //Se deja tiempo a que el servidor inicie.
           printf("inicio del cliente\n");
```

```
            printf("--------------------\n");
            cliente();
          }
        }

    } else {
      printf("Llamada al programa: calm [servidor/cliente]\n");
    }
    return 0;
       }
```

CLIENTE.C

```
/*
 * cliente.c
 *
 *  Created on: 05/05/2015
 *      Author: Alberto Rodriguez Blazquez
 *      E-mail: albertoroblaz@hotmail.com
 *      TFG: Diseño y desarrollo de un sistema ITS basado en comunicaciones tipo ISO CALM.
 *      Descripcion: Funciones usadas por la parte cliente de la aplicacion.
 *      Version: 0.9 Final
 *
 */

  #include "libreria.h"

/*
 * Funcion principal del cliente. Establece conexion tcp con el servidor propio y abre escucha
 * a los mensajes HI de servidores externos. Determina los nuevos nodos en la red y lanza hilos para
 * conectar con ellos.
 *
 *Parametros de entrada:
 * -Ninguno
 *Parametros de salida:
 * -Ninguno/No importante
 */

 int cliente(){

  struct DA pruebas;          //Estructura PELIGRO pasada introducida en la lista al iniciar al programa.

  //SOCKETS
  int socket_escucha;         //Socket de escucha de mensajes HI.
  int socket_servidor_propio;  //Socket para la comunicacion con el servidor propio.

  //PUERTOS
  char puerto_escucha[]=CBROAD_PORT;      //Puerto usado para la escucha de mensajes HI.
  char puerto_servidor_interno[]=SC_PORT; //Puerto de escucha del servidor propio.

  //DIRECCIONES IP
  char ip_server[IP_LENGTH];    //Almacena la IP origen del mensaje HI. Usado para iniciar conexion con ese
nodo.
  char ip_propia[IP_LENGTH];    //Almacena IP propia.

  //ESTRUCTURAS
  struct sockaddr_storage their_addr;  //Estructura IP de un mensaje. Almacena los campos del mensaje UDP HI
(La IP origen).
  socklen_t addr_len;                 //Tamano de la estructura sockaddr_storage.
  struct DIRIP* inicio_lista=NULL;      //Puntero al inicio de lista de nodos conocidos.
```

```
  struct PELIGROS *inicio_peligros=NULL;//Puntero al inicio de lista de peligros conocidos.
  struct PARAM inicio_hilo;          //Estructura con los parametros pasado a los hilos creados para la conexion
con los servidores.
  struct HI saludo;                 //Estructura de un mensaje HI.
  struct DA alerta_interna;           //Estructura de un mensaje DA.
  struct DIRIP* aux;                //Puntero auxiliar para pasar la direccion de una DIRIP a los hilos.

  //HILOS
  pthread_attr_t attr;              //Estructura con los atributos del hilo a iniciar.
  pthread_t hilo;                  //Id del proceso servidor. No usado.

  //VARIABLES DE CONTROL
  char codigo[3];                   //Almacena codigos de mensajes.
  int salida=0;                    //Codicion de salida del bucle de atencion de conexiones.
  int control1=0;                   //Indica si se ha recibido una comunicacion de un servidor externo.
  int control2=0;                   //Indica si se ha recibido una comunicacion de un servidor propio.
  int num_bytes=0;                    //Numero de bytes recibidos en una lectura de socket.
  int contador=0;                   //Contador de iteraciones del bucle principal.

    signal (SIGPIPE, SIG_IGN);      //Tratamiento de errores para una conexión dentro del mismo equipo.


  //Este primer bloque obtiene la IP propia de la interfaz wlan0 y la almacena en ip_propia.
  if(obtenerIPPropia("wlan0", ip_propia)==NULL){
    perror("No se ha podido determinar IP propia\n");
  } else {
    printf("IP propia es: %s\n", ip_propia);
  }

  /*CONDICIONES INICIALES DE LA LISTA DE PELIGROS*/
  //Usando el bloque de codigo aqui descrito se pueden introducir obstaculos para que el programa no empiece de
0.
  bzero(&pruebas, sizeof(struct DA));
  strcpy(pruebas.codigo,"DA");
  pruebas.pos=(float)10;
  pruebas.tipo=(int)1;
  pruebas.id=(int)9;
  anadirPeligroConocido(&inicio_peligros,pruebas);

  /*************************/

  addr_len = sizeof (their_addr); //Longitud estructura parametros mensajes recibidos.
  salida=0;                   //Control salida bucle espera.
  control1=0;                   //Control recepcion mensaja descubrimiento.
  control2=0;                   //Control recepcion mensaje servidor interno.

  //Establecer comunicacion con el proceso servidor.
  if ( (socket_servidor_propio = inicia_socket_cliente(ip_propia, puerto_servidor_interno)) < 0)
  {
    //Conexion no establecida. Este no se ha iniciado. No habra comunicacion con servidor propio.
    printf("Cliente: No hay conexion con el servidor local.\n");
    socket_servidor_propio=-1;
  }

  //Iniciar escucha de broadcast de servidores.
    if((socket_escucha=iniciaEscuchaUDP(puerto_escucha))<0)
    {
      //Conexion no establecida, no se recibiran mensajees HI. Salida del programa.
      perror("Cliente:Fallo al iniciar socket de escucha. Cierre de la aplicacion");
      close(socket_servidor_propio);
      exit(1);
    } else {
```

```
    //Bucle atencion a los sockets.
    //De momento su salida se realiza cuando hayan fallado la conexion externa e interna o tras un numero de iteraciones.
    while(salida==0){
      //El programa comprobara cada T_ESPERA si hay una nueva conexion en los socket, tras lo cual reiniciara el bucle.
      //En caso de haber una nueva conexion, esta se manda a procesar antes de continuar.
      if((control1=comprueba_recepcion(socket_escucha,T_ESPERA))==1 || (control2=comprueba_recepcion(socket_servidor_propio,T_ESPERA))==1)
      {

        //Comprobacion de que el cliente tenga algo operativo.
        if(socket_servidor_propio<0 && socket_escucha<0){
          printf("No ha conexion con servidor propio o externo. Iniciando salida\n");
          salida=1;
        }

        //Atencion al servidor propio.
        if(control2==1 && socket_servidor_propio>0)
        {
          //Se supone que el cliente solo recibira posiciones de peligro del servidor interno (Mensajes DA).
          //Se prepara la estructura para mensaje DA.
          bzero(&alerta_interna,sizeof(struct DA));
          control2=recibirAlerta(socket_servidor_propio, &alerta_interna); //Esta lee la estructura DA recibida.
          if(control2<=0)
          {
            //Habria que cerrar el socket e informar de que esta parte a quedado inutilizada.
            perror("Fallo en comunicacion con servidor interno");
            close(socket_servidor_propio);
            printf("Reintentando conexion...\n");
            socket_servidor_propio=inicia_socket_cliente(ip_propia, puerto_servidor_interno);
            //En caso de no volver a poder conectar, abandonar esta parte del programa.
            if(socket_servidor_propio<0)
            {
              perror("No ha sido posible volver a conectar. Servidor propio caido.\n");
              socket_servidor_propio=-1;
            }
          }
          if(control2>0)
          {
            //Se anade el peligro recibido desde el servidor a la lista (En caso de que no estubiese).
            if(!anadirPeligroConocido(&inicio_peligros,alerta_interna)==0){
              printf("Cliente-->Peligro ya conocido, no se vuelve a lanzar aviso\n");
            } else {
              printf("Cliente-->AVISO, obstaculo en la carretera\n");
            }
          }
          control2=0;   //Se devuelve la variable de control a su valor original.
        }


        //Recibido un mensaje de un servidor externo.
        if(control1==1)
        {
          //Preparacion estructura parametros origen.
          bzero(&their_addr, addr_len);
          //Preparacion estructura HI.
           bzero(&saludo,sizeof(struct HI));
          if((num_bytes=recvfrom(socket_escucha, &saludo, sizeof(struct HI), 0 ,(struct sockaddr *)&their_addr, &addr_len))==-1)
          {
            perror("Error al recibir HI");
            //No se hace nada, se espera al siguiente. Esto se repite.
```

```
        } else {
          //Se lee el codigo del mensaje.
          memset(codigo, '\0', sizeof(codigo));
          strncpy(codigo, saludo.codigo, 2);

          //Comprobacion del codigo de saludo.
          if(codigo[0]=='H' && codigo[1]=='I')
          {
            //La IP del servidor se saca del propio mensaje.
            memset(ip_server, '\0', IP_LENGTH);
            inet_ntop(their_addr.ss_family, get_in_addr((struct sockaddr *)&their_addr),ip_server, sizeof
(ip_server));

            //Comprobar que no corresponda a un servidor propio o a alguno ya conocido.
            //Comparacion con IP propia.
            if(strncmp(ip_propia,ip_server,IP_LENGTH)!=0)
            {
              printf("\nRecibido HI de %s\n", ip_server);
              //Comparacion con IPs almacenadas.
              if((aux=(anadirNodoConocido(&inicio_lista, saludo, ip_server)))==NULL)
              {
                //En caso de estar conocido (se usa la id para ello), se actualiza su IP.
              } else {

                //Nodo nuevo y almacenado, se procede a contactar con el.

                /*LANZAMIENTO DE HILO PARA EL NODO*/
                bzero(&inicio_hilo, sizeof(struct PARAM));   //Param sera la estructura pasada a este hilo.

                //Se asignan los parametros a la estructura pasada.
                memset(inicio_hilo.direccion,'\0',IP_LENGTH);
                strncpy(inicio_hilo.direccion,ip_server,IP_LENGTH);
                inicio_hilo.nodo_ip=aux;
                inicio_hilo.inicio_alertas=&inicio_peligros;
                inicio_hilo.pos=saludo.pos;
                inicio_hilo.vel=saludo.vel;
                inicio_hilo.id=saludo.id;
                inicio_hilo.bateria=saludo.bateria;
                inicio_hilo.s_nodos=NULL;
                inicio_hilo.s_peligros=NULL;

                //Caracteristicas del hilo a crear: DETACHED (El proceso no espera su fin).
                pthread_attr_init(&attr);                    //Iniciacion de la estructura de con los atributos a
usar.
                pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);  //Un hilo DETACHED
es independiente del funcionamiento del resto del programa.

                //Creacion del hilo de atencion a ese nodo, el proceso principal se desentiende de el.
                pthread_create(&hilo, &attr, atencion_mensaje, &inicio_hilo);
              }
            }
          } else {
            //Este caso de da cuando se recibe un codigo distinto de HI por el puerto de escucha.
            //Se supone que el paquete esta corrupto y se descarta.
            perror("Codigo desconocido en saludo");
          }
        }
        control1=0;
      }
    }
    //Salida del programa por iteraciones.
    contador++;
    if(contador>=NUM_ITER && NUM_ITER>0){
```

```
          salida=1;
          printf("Fin de iteraciones\n");
        }
      }
      //Envio aviso de cierre al servidor.
      if(socket_servidor_propio>0){
        memset(codigo,'\0',sizeof(codigo));
        strncpy(codigo,"SF",sizeof(codigo));
        if(send(socket_servidor_propio,codigo,sizeof(codigo),0)<0){
          perror("Mensaje de cierre no enviado al servidor local");
        }
      }
    }
//Si no estan marcados todos los nodos de la lista DIRIP para borrar es que hay hilos en funcionamiento,
//Se espera su fin durante 2 segundo y se vuelve a intentar.
while ((borrarNodosMarcados(&inicio_lista))==0){
  printf("Esperando a que los hilos terminen\n");
  sleep(2);
}

if(!borrarPeligros(&inicio_peligros)){
  printf("Peligros conocidos borrados\n");
}
return 0;
}

/*
 * Funcion principal de los hilos de comunicacion con los servidores externos. Descarga al proceso cliente de la iteración
 * con otros nodos y de la logica del intercambio de informacion.
 *
 * Parametros de entrada:
 * - void *argc==&struct PARAM parametros.
 *
 * Parametros de salida:
 * -Ninguno.
 */

void *atencion_mensaje(void *argc){

  struct PARAM * p_parametros=(struct PARAM *) argc;//Conversion del parametro pasado a su formato.
  struct PARAM parametros=*p_parametros;          //Se convierte el puntero indefinido pasado a una estructura PARAM.
  struct DA peligro;                              //Estructura para mensaje DA.
  struct DIRIP * p_nodo=parametros.nodo_ip;       //Puntero a la estructura DIRIP del nodo atendido.
  struct ST estado;                               //Estructura para mensaje ST.
  int sock;                                       //Sockect de comunicacion tcp con servidor externo.
  char puerto[]=DST_PORT;                         //Puerto de peticion de comunicacion tcp del servidor externo.
  char codigo[3];                                 //Lectura y escritura de codigo de mensajes.
  int control=0;                                  //Control de ejecucion del bucle while.
  int error=0;                                    //Control de errores del bucle while.
  int contador=0;                                 //Contador del bucle while.

//printf("Iniciada atencion del servidor externo\n");
  bzero(&peligro, sizeof(struct DA));             //Se inicia a 0 una estructura DA.
  memset(peligro.codigo, '\0', sizeof(codigo));
  memset(codigo, '\0', sizeof(codigo));           //Puesta a 0 de codigo.

  peligro=calculos_alertas(parametros);     //Calculo de alertas de peligros. Gereracion de mensaje DA en caso de que alguno afecte al servidor.

  //Se copia el campo de codigo de la estructura DA en codigo. En caso de haberse generado el mensaje, este debe ser enviado al servidor.
```

```
  strncpy(codigo, peligro.codigo, sizeof(codigo));

  //Envio de mensaje de alerta de peligro.
  if (codigo[0]=='D')
  {
    //Salida por pantalla para indicar que se ha enviado un mensaje DA.
    printf("----Alerta de peligro: posicion del nodo en %f, obstaculo en  %f, velocidad %f\n", parametros.pos,
peligro.pos, parametros.vel);
    //Se inicia una comnicacion con el socket que ha enviado el HI.
    if((sock=inicia_socket_cliente(parametros.direccion, puerto)) <0)
    {
      //No ha sido posible la comunicacion, se procede a cerrar el hilo.
      perror("Socket no iniciado en hilo");
      error=1;
    } else {
      //El mensaje se enviara hasta 5 veces esperando un ACK.
      memset(codigo,'\0', sizeof(codigo));
      strcpy(codigo, "DA");
      while(control==0 && error==0 && contador<=5)
      {
        contador++;
        if((send(sock,codigo, sizeof(codigo),0))<0)
        {
          perror("Error al enviar el codigo de mensaje desde un hilo");
        } else {
          //Intento de envio del mensaje DA.
          if((send(sock, &peligro, sizeof(struct DA), 0))<0)
          {
            printf("Error de conexion con el servidor externo, intentando volver a conectar...\n");
            close(sock);
            sock=-1;
            if((sock=inicia_socket_cliente(parametros.direccion, puerto)) <0)
            {
              perror("No se puede conectar con servidor externo, saliendo del hilo\n");
              error=1;
            }
          } else {
            //El mensaje se ha enviado, se esperara 2 segundos el ACK. En caso de no darse se reenviara.
            if(comprueba_recepcion(sock,2)==1)
            {
              memset(codigo,'\0', sizeof(codigo));    //Limpieza de cadena codigo.
              recv(sock, codigo, sizeof(codigo),0);   //Recepcion de mensaje.
              if(codigo[0]=='D' && codigo [1]=='R')
              {
                printf("----ACK peligro recibido\n");
                control=1;
              } else {
                printf("----ACK no recibido\n");
              }
            }
          }
        }
      }
    }
  } else {
    //No existen peligros inmediatos. Se requiere status.
    //Se inicia una comnicacion con el socket que ha enviado el HI.
    if((sock=inicia_socket_cliente(parametros.direccion, puerto)) <0)
    {
      //No ha sido posible la comunicacion, se procede a cerrar el hilo.
      perror("Hilo: Socket no iniciado");
      error=1;
    } else {
```

```
    //El mensaje se enviara hasta 5 veces esperando un ACK.
    memset(codigo,'\0', sizeof(codigo));
    strcpy(codigo, "SR");
    while(control==0 && error==0 && contador<=5)
    {
      contador++;
      if((send(sock,codigo, sizeof(codigo),0))<0)
      {
        perror("Error al enviar el codigo de mensaje");
      } else {
        //El mensaje se ha enviado, se esperara 5 segundos el mensaje de estado. En caso de no darse se reenviara
la peticion.
        if(comprueba_recepcion(sock,2)==1)
        {
          bzero(&estado, sizeof(struct ST));
          memset(estado.codigo,'\0', sizeof(codigo));    //Limpieza de cadena codigo.
          memset(codigo,'\0', sizeof(codigo));
          recv(sock, &estado, sizeof(struct ST),0);      //Recepcion de mensaje.
          strncpy(codigo, estado.codigo, sizeof(codigo));
          if(codigo[0]=='S' && codigo[1]=='T')
          {
            //Se han adquiridos los datos Status del otro vehiculos.
            //Aqui va la funcion que hace algo con los datos de estatus.
            printf("--Mensaje de estado recibido\n");
            printf("----T_recorrido: %li\n", estado.t_recorrido );
            printf("----D_recorrida: %i\n", estado.d_recorrido);
            printf("----Ultima recarga: %f\n", estado.recarga );
            printf("----ID: %d\n",estado.id );
            control=1;
          } else {
            perror("La recepcion de ST ha fallado\n");
          }
        }
      }
    }
  }

  }
  //El socket sigue abierto si no ha habido error.
  if(error!=1){
    close(sock);
    sock=-1;
  }
  //Alerta de que algo a fallado.
  if(error==1 || contador==5)
  {
    perror("El estado no ha llegado\n");
  }

  //Se marca la estructura del nodo para borrar antes de cerrar el hilo.
  p_nodo->flag=1;
  pthread_exit(NULL);
      }
```

SERVIDOR.C

```
/*
 * servidor.c
 *
 *  Created on: 05/05/2015
```

```
 *      Author: Alberto Rodriguez Blazquez
 *      E-mail: albertoroblaz@hotmail.com
 *      TFG: Diseño y desarrollo de un sistema ITS basado en comunicaciones tipo ISO CALM.
 *      Descripcion: Funciones usadas por el servidor de la aplicacion.
 *      Version: 0.9 Final
 *
 */

 #include "libreria.h"

/*
 * Funcion principal del servidor. Establede un socket udp de envio y dos de escucha tcp.
 * Envia HI periodicos a difusion de la red. Acepta y establece conexion con el cliente interno y los externos.
 * Recibe mensajes de clientes externos, los clasifica y actua en consecuencia. En caso de mensaje DA, lo
notifica
 * al cliente interno mediante el socket creado en caso de que se haya establecido una conexion.
 *
 *  Parametros de entrada:
 * -Ninguno
 *  Parametros de salida:
 * -Ninguno/No importante
 */
int servidor(){

  //PUERTOS
    char puerto_escucha[]=DST_PORT;     //Puerto de escucha y atencion a clientes externos.
  char puerto_interno[]=SC_PORT;      //Puerto de escucha y atencion al cliente interno.

  //SOCKETS
    int socket_escucha;               //Socket de escucha en puerto DST_PORT.
    int socket_servidor;              //Socket de atencion a los clientes externos.
  int socket_broadcast;              //Socket para el envio de mensajes HI (descubrimiento).
  int socket_cliente_interno;        //Socket para escucha del cliente interno
  int socket_cliente=-1;             //Socket para la atencion del cliente interno.

  //DIRECCIONES IP
  char ip_propia[IP_LENGTH];         //Almacenamiento de direccion IP propia.

  //ESTRUCTURAS
  struct HI hola_envio;                    //Estructura usada para mensajes HI.
  struct sockaddr_in destino;               // direccion socket broadcast
  struct sockaddr dir_cliente;              // direccion socket del cliente
  socklen_t len_dir = sizeof(dir_cliente);    // len_dir es un parametro de entrada/salida en la funcion accept
  socklen_t len_des = sizeof(destino);        // Idem a len_dri.

  //VARIABLES DE CONTROL
    int salida=0;                     //Control de salida del buble while.
  int control1=0;                     //Control, peticion de clientes externos.
  int control2=0;                     //Control, peticion de cliente interno.
  char codigo[3];                     //Almacenamiento de codigo de mensaje.
  int establecido=0;                  //Marca si existe una conexion activa con el cliente local.

    signal (SIGPIPE, SIG_IGN);      //Tratamiento de errores para una conexión dentro del mismo equipo.


  //Iniciar sockets de escuchar tcp externo e interno.
    if((socket_escucha=iniciaEscuchaTCP(puerto_escucha))<0 ||
(socket_cliente_interno=iniciaEscuchaTCP(puerto_interno))<0)
    {
      //Fin de programa en caso de fallo.
        perror("Error al iniciar escucha TCP");
    } else {
      //Inicio del socket para el envio de HI y obtencion de direccion propia.
```

```
    if((socket_broadcast=iniciaHolaUDP(&destino))<0 || obtenerIPPropia("wlan0", ip_propia)==NULL)
    {
      //Discriminar el error. En caso de no iniciar socket udp, finalizar servidor (No tiene sentido continuar).
      if(socket_broadcast<0){
        perror("Socket broadcast no iniciado");
      } else {
        perror("No se ha podido determinar IP propia\n");
      }
    } else {
      printf("IP propia es: %s\n", ip_propia);

      //Bucle de procesamiento del servidor. Aqui se llevaran a cabo todas las tareas del server.
      while(salida==0){
        sleep(T_HELLO);   //Tiempo minimo de espera entre envio de un mensaje HI y otro.
                          //Evita la saturación de la red producida por mensajes HI continuos.
        //Se envia el HOLA a toda la red, tras lo que se espera respuesta durante X tiempo.
        //En caso de recibir una respuesta esta se procesa antes de continuar.

        //Preparacion del mensaje HOLA. Estructura HI.
        //Direccion/puerto destino.
        memset(destino.sin_zero, '\0', sizeof(destino.sin_zero));
        destino.sin_addr.s_addr = inet_addr(HELLO_GROUP);
        destino.sin_port = htons((unsigned int)atoi(CBROAD_PORT));
        destino.sin_family = AF_INET;

        //Puesta a 0 de la estructura HI.
        memset(hola_envio.codigo, '\0', sizeof(hola_envio.codigo));
        bzero(&(hola_envio.pos), sizeof(hola_envio.pos));
        bzero(&(hola_envio.vel), sizeof(hola_envio.vel));
        bzero(&(hola_envio.id), sizeof(hola_envio.id));
        bzero(&(hola_envio.bateria), sizeof(hola_envio.bateria));

        //Introduccion de datos en la estructura:
        strcpy(hola_envio.codigo, "HI");
        adquirirDatos(&hola_envio);

        //Envio descubrimiento.
        if(sendto(socket_broadcast, &hola_envio, sizeof(struct HI),0, (struct sockaddr *) &destino, len_des)==-1)
        {
          //En caso de fallo, no se hace nada, ya se enviara otro Hi en la siguiente iteracion.
          perror("Error al enviar HOLA");
        } else {
          //Bonito mensaje por pantalla que indica que va a ver los coches que reciban el HI.
          printf("Envio de HI--> posicion: %f velocidad: %f bateria: %f id: %d\n",  hola_envio.pos,
hola_envio.vel, hola_envio.bateria, hola_envio.id);
        }
        //Vuelve a ponerse a 0 la estructra direccion/puerto.
        bzero(&dir_cliente, sizeof(dir_cliente));
        //Se espera T_ESPERA segundos escuchando cada socket antes de enviar otro HOLA.
          if((control1=comprueba_recepcion(socket_escucha,T_ESPERA))==1 ||
(control2=comprueba_recepcion(socket_cliente_interno,T_ESPERA))==1)
        {
          //Llega el cliente interno:
          if(control2==1 && establecido==0)
          {
            //Se atiende al cliente interno y se establece un socket para comunicarse con el.
            socket_cliente=-1;
            socket_cliente=accept(socket_cliente_interno, (struct sockaddr*)&dir_cliente, &len_dir);
            if(socket_cliente<0)
            {
              //Vuelve a esperarse conexion desde el cliente.
              perror("Conexion fallida con cliente interno.");
              control2=0;
```

```
  } else {
    establecido=1;
    control2=0;
  }
}

//Conexion desde el cliente.
//Actualmente el cliente solo acepta por local el codigo SF.
if(establecido==1){
  if(comprueba_recepcion(socket_cliente,T_ESPERA)==1){
    memset(codigo,'\0',sizeof(codigo));
    if(recv(socket_cliente,codigo,sizeof(codigo),0)<0){
      perror("Fallo al recibir mensaje de cliente interno");
    } else {
      if(codigo[0]=='S' && codigo[1]=='F'){
        printf("Iniciando cierre del servidor\n");
        salida=1;
      } else {
        perror("Codigo enviado por cliente interno no reconocido \n");
      }
    }
  }
}

//Llega una nueva conexion de un cliente externo.
if(control1==1)
{
  socket_servidor=-1;
  socket_servidor=accept(socket_escucha, (struct sockaddr*)&dir_cliente, &len_dir);
  if(socket_servidor<0)
  {
    //En caso de fallo se espera a que se realice una nueva peticion desde el cliente.
    perror("Conexion fallida con cliente externo");
    control1=0;
  } else {
      //En caso de recibir mensaje se procede a leer su codigo.
      memset(codigo, '\0', sizeof(codigo));

      if((recv(socket_servidor, codigo, sizeof(codigo), 0))==-1)
      {
        //No se ha recibido bien el mensaje, se corta la comunicacion.
        perror("Fallo al recibir datos, Cierre de socket");
      } else {
        //Si se lee bien el codigo se procede a identificarlo.
        //En funcion del codigo recibido se realizara una un otra accion.
        printf("Codigo %s recibido: ", codigo); //Se muestra por pantalla para mayor comodidad.

        //Introducir en esta seccion la llamada a el procesamiento de nuevos mensajes.
        //Usar un if con el codigo del mensaje anadido.

        //RECIBO DE FICHERO.
        if(codigo[0]=='A' && codigo[1]=='R')
        {
          printf("Recibiendo fichero...\n");
          if(recibir_fichero(socket_servidor)<0)
          {
            perror("Error recibiendo fichero");
          } else {
            printf("Fichero recibido\n");
            fflush(stdout);
          }
        control1=0;
        }
```

```
                //INICIAR SALIDA DEL SERVIDOR.
                //Iniciar salida se ha quitado de aqui para evitar que clientes externos puedan cerrarnos el servidor.
                //No obstante, al tener posibles aplicaciones interesantes en un futuro, se deja comentado.
                /*if (codigo[0]=='S' && codigo[1]=='F')
                {
                  printf("Iniciando salida del servidor \n");
                  salida=1;
                  control1=0;
                }*/

                //MENSAJE DE PELIGRO.
                if (codigo[0]=='D' && codigo[1]=='A')
                {
                  printf("Recibiendo peligro....\n");
                  /*PROCESAR PELIGRO*/
                  if(!procesarPeligro(socket_cliente,socket_servidor)==0)
                  {
                    perror("Error al recibir alerta");
                  }
                  control1=0;
                }

                //PETICION DE ESTADO.
                if (codigo[0]=='S' && codigo[1]=='R')
                {
                  printf("Recibida peticion de estado.\n");
                  /*DEVOLVER ESTADO*/
                  if(!devolverEstado(socket_servidor)>0)
                  {
                    perror("Fallo al devolver estado");
                  }
                  control1=0;
                }

                //En caso de que no se haya reconocido el codigo, aqui se llega con control=1. Este if debe ir al
final.
                if(control1==1)
                {
                  perror("No se a captado bien el codigo \n");
                  control1=0;
                }

               }
              //En todo los casos se cierra el socket y se pone el estado a 0.
              control1=0;
              close(socket_servidor);
              socket_servidor=-1;
            }
          }

            } else {
          //No se ha recibido nada por ningun socket.
          printf("---Sin nuevos vecinos, volviendo a esperar\n");
        }
          }
      //Ha partir de aqui solo se ejecuta si salida es 1.
      close(socket_broadcast);
      }
    close(socket_cliente_interno);
    close(socket_escucha);
    }
```

```
 return 0;
    }
```

MENSAJES.C

```
#include "libreria.h"

/*
 * mensajes.c
 *
 *  Created on: 17/05/2015
 *      Author: Alberto Rodriguez Blazquez
 *      E-mail: albertoroblaz@hotmail.com
 *      TFG: Diseño y desarrollo de un sistema ITS basado en comunicaciones tipo ISO CALM.
 *      Descripcion: Funciones de procesamiento de los mensajes de la aplicacion.
 *      Version: 0.9 Final
 *
 */


/*
 * Se le pasa una estructura PARAM con los datos de un nodo recibidos en un HI.
 * Esta funcion recorre la lista de peligros y genera un mensaje DA si encuentra un
 * peligro que afecte al modo. En caso de no afectarle ninguno devuelve un DA vacio.
 * Un nodo sera avisado de un peligro si se encuentrar a menos de 20 de este y va en
 * direccion al obtaculo.
 *
 * Parametros de entrada:
 * -parametros --> struct DA con lo datos de nodo a avisar.
 *
 * Parametros de salida:
 * -peligro --> struct DA vacia en caso de no avisar de peligro y
 *              preparada para enviar en caso de haber alguno.
 */

struct DA calculos_alertas(struct PARAM parametros){

  struct DA peligro;           //Estructura para el mensaje de peligro a devolver.
  struct PELIGROS *puntero;    //Puntero para recorrer la lista de estructuras PELIGROS.
  int resultado=0;             //Resultado de calculos de aviso de peligros.
  char codigo[3];              //Lertura/escritura de codigo de mensajes.
  int salida=0;               //Indicador de salida del bucle.

  //En primer lugar se apunte el puntero al inicio de la lista.
  puntero=*(parametros.inicio_alertas);

  //Puesta a 0 de la estructura DA y de codigo.
  bzero(&peligro, sizeof(struct DA));
  memset(codigo, '\0', sizeof(codigo));
  //En el bucle se comprueba una a una las estructuras PELIGRO, saliendo tras encontrar una o llegar al final.
  while(puntero!=NULL && salida==0){
    //Calculo de la distancia a la que se encuentra el nodo del peligro revisado
    //La distancia puede ser en negativo o positivo.
    resultado=(puntero->pos)-(parametros.pos);
    if(resultado<DIST_ALERTA && resultado>(-(DIST_ALERTA))){
      //En caso de estar a menos de 20 del peligro se procede a comprobar si esta en la direccion de
desplazamiento.
      //Para ello se tiene en cuenta la posicion del nodo, del peligro y el signo de la velocidad.
      if( (((*puntero).pos<=(parametros.pos)) && (parametros.vel<=0)) || (((*puntero).pos>=(parametros.pos))
&& (parametros.vel>=0))  )
```

```
      {
        //Al cumplirse la condicion anterior se determinaque hay que avisar al otro nodo del peligro.
        //Se genera entonces el mensaje DA.
        memset(codigo,'\0',sizeof(codigo));
        strcpy(peligro.codigo,"DA");
        peligro.pos=(puntero->pos);
        peligro.id=puntero->id;
        peligro.tipo=puntero->tipo;
        salida=1;       //Tras generar se indica la salida del bucle.
      }
    }
    if(puntero->flag==1 || salida==1){
      puntero=NULL;
    } else {
      puntero=puntero->next;
    }
  }
  return peligro;
}


/*
 * Pequeña funcion que va dando diferentes datos para rellenar el mensaje HI.
 * Es usada cuando no existe un fichero desde donde leer estos datos.
 *
 * Parametros de entrada:
 * -Punteros a cada valor a rellenar.
 *
 * Parametros de salida:
 * -Ninguno
 */
 void datos_default(float * pos, float * vel, float* bat){
  static float posicion=120;
  static float velocidad=1;
  static float bateria=100;
  static int contador=0;

  //Calculo velocidad
  switch ((int)velocidad){
    case 12:
      velocidad=6;
      break;
    case 6:
      velocidad=8;
    break;
    case 8:
      velocidad=12;
    break;

    default:
    velocidad=6;
  }
  //Calculo posicion
  posicion=posicion-velocidad;
  if(posicion<=0){
    posicion=120;
  }

  //Calculo bateria
  if (contador>=5){
    bateria=bateria-5;
    if(bateria<0){
      bateria=100;
```

```
    }
  }
  //Paso de valores.
  contador++;
  *pos=posicion;
  *vel=-(velocidad);
  *bat=bateria;
 }

/*
 * Aqui se adquiriran la posicion y velocidad actual por el medio que sea para enviarlo a los otros coches.
 *
 * Parametros de entrada:
 * -datos --> Puntero a la structura HI que se enviara a difusion.
 *
 * Parametros de salida:
 * -Ninguno
 */

void adquirirDatos(struct HI * datos){

  static FILE *fichero=NULL;
  float posicion=0;
  float velocidad=0;
  float bateria=0;

  if(fichero==NULL)
  {
    fichero=fopen("/bin/posicion","r");
    if(fichero==NULL)
    {
      //No hay fichero, se usa la funcion datos_default.
      datos_default(&posicion,&velocidad,&bateria);
      datos->pos=posicion;
      datos->vel=velocidad;
      datos->bateria=bateria;
      datos->id=IDDISPOSITIVO;
    } else {
      if(EOF!=fscanf(fichero, "%f,%f,%f:", &posicion,&velocidad,&bateria))
      {
        datos->pos=posicion;
        datos->vel=velocidad;
        datos->bateria=bateria;
        datos->id=IDDISPOSITIVO;
      } else {
        //Esta zona en teoria no se ejecuta nunca a no ser que el fichero abierto este vacio.
        fclose(fichero);
        fichero=NULL;
        datos_default(&posicion,&velocidad,&bateria);
        datos->pos=posicion;
        datos->vel=velocidad;
        datos->bateria=bateria;
      }
    }
  } else {
    if(EOF!=fscanf(fichero, "%f,%f,%f:", &posicion,&velocidad,&bateria))
    {
      datos->pos=posicion;
      datos->vel=velocidad;
      datos->bateria=bateria;
      datos->id=IDDISPOSITIVO;
    } else {
      fclose(fichero);
```

```
      fichero=NULL;
      datos_default(&posicion,&velocidad,&bateria);
      datos->pos=posicion;
      datos->vel=velocidad;
      datos->bateria=bateria;
    }
  }
}


/*
 *  Recoge un peligro mandado por otro nodo y lo comunica al cliente interno
 *
 * Parametros de entrada:
 * -socket_interno --> socket de comunicacion con el cliente interno.
 * -socket_externo --> socket de comunicacion con el cliente externo.
 *
 * Parametros de salida:
 * -error --> =1, en caso de error
 *            =0, todo correcto
 */

int procesarPeligro(int socket_interno, int socket_externo){

  int n_bytes;              //Bytes leidos del mensaje.
  struct DA peligros;         //Estructura DA a enviar al cliente interno.
  char codigo[3];           //Codigo identificativo del ACK al mensaje DA.
  int error=0;              //Control de errores producidos.
  int contador=0;           //Contador del bucle while.
  int salida=0;             //Control salida del bucle while.

  //Puestaa cero de DA y recepcion de la posicin del peligro (El codigo ya se ha recibido antes).
  bzero(&peligros, sizeof(struct DA));
  memset(codigo,'\0', sizeof(codigo));
  n_bytes = recv(socket_externo, &(peligros), sizeof(struct DA), 0);
  if(n_bytes<=0){
    //Fallo en la recepcion.
    perror("Conexion interrupida o perdida");
    error=1;
  }
  if(n_bytes>0){
    printf("Peligro en posicion %f \n", peligros.pos );
    printf("Tipo: %d\n" ,peligros.tipo);
    printf("ID: %d\n",peligros.id );
  }
  //Si no ha habido errores de recepcion, se intenta enviar el ACK hasta 5 veces.
  while(error==0 && contador<5 && salida==0)
  {
    memset(codigo,'\0', sizeof(codigo));
    strcpy(codigo, "DR");
    if(send(socket_externo, codigo, sizeof(codigo),0)<0)
    {
      contador++;
    } else {
      salida=1;
    }
  }

  //Se informa al servidor interno del error en caso de tener conexion con el.
  if(socket_interno>0)
  {
    if(send(socket_interno, &peligros, sizeof(struct DA),0)<0)
    {
```

```
      perror("Fallo de comunicacion con el cliente interno");
      error=1;
    }
  } else {
    printf("Cliente interno caido; no se hace nada\n");
  }
  if(contador>=5){
    perror("No se ha podido enviar ACK al servidor externo");
    error=1;
  }
  return error;
}



/*
 * Devuelve el estado del nodo es caso de haber recibido una peticion.
 *
 * Parametros de entrada:
 * -socket_externo --> Conexion con el nodo.
 *
 * -n_bytes --> >0, mensaje transmitido.
 *          <=0, conexion interrumpida o perdida.
 */

int devolverEstado(int socket_externo){

  int n_bytes=0;      //Numero de bytes transmitidos.
  struct ST estado;   //Mensaje ST transmitido al cliente externo.
  static FILE *fichero=NULL;
  long t_recorrido=0;
  int d_recorrido=0;
  float recarga_bateria=0;

//Rellenar estructura de estado....
  bzero(&estado, sizeof(struct ST));
  memset(estado.codigo, '\0', sizeof(estado.codigo));

  if(fichero==NULL)
  {
    //Cambiar /bin/estado para situar el fichero en otra ruta.
    //IMPORTANTE: OpenWrt requiere que la ruta se indique de forma absoluta.
    fichero=fopen("/bin/estado","r");
    if(fichero==NULL)
    {
      //No hay fichero estado, se usan valoras arbitrarios.
      t_recorrido=2500;
      d_recorrido=8500;
      recarga_bateria=78;
    } else {
      if(EOF==fscanf(fichero, "%li,%i,%f:", &t_recorrido,&d_recorrido,&recarga_bateria))
      {
        t_recorrido=2500;
        d_recorrido=8500;
        recarga_bateria=78;
      }
    }
  } else {
    if(EOF==fscanf(fichero, "%li,%i,%f:", &t_recorrido,&d_recorrido,&recarga_bateria))
    {
       perror("Fallo en lectura de estado");
       fclose(fichero);
       fichero=NULL;
```

```
        t_recorrido=2500;
        d_recorrido=8500;
        recarga_bateria=78;
    }
  }

  strcpy(estado.codigo, "ST");
  estado.t_recorrido=t_recorrido;
  estado.d_recorrido=d_recorrido;
  estado.recarga=recarga_bateria;
  estado.id=IDDISPOSITIVO;

//Envio
  n_bytes=send(socket_externo,&estado,sizeof(struct ST),0);

  return n_bytes;
}


/*
 * Recibe una alerta enviada por el servidor interno.
 *
 *Parametros de entrada:
 * -sock --> Conexion con el socket interno.
 * -alerta --> Puntero a una estructura DA donde guardar lo leido.
 *
 * Parametros de salida:
 * -n_bytes --> <=0, conexion perdida o interrumpida.
 *              >0, mensaje recibido.
 */

int recibirAlerta(int sock, struct DA* alerta){

  int n_bytes=0;

  //Recepcion del mensaje.
  n_bytes=recv(sock,alerta, sizeof(struct DA), 0);

  //Zona de control
  if(n_bytes<=0){
    perror("Error al recibir mensaje");
  }
return n_bytes;
}

/*
 * SIN USO. Envia un fichero de nombre "envio" situado en la carpeta de ejecucion a
 * traves del socket tcp pasado.
 *
 * Parametros de entrada:
 * -socket --> socket tcp conectado con el receptor.
 *
 * Parametros de salida:
 * -control --> ==0, El fichero se ha transmitido correctamente.
 *              <0, Ha habido un error durente la transmision.
 */

int envia_fichero(int socket){

char codigo[]="AR";     //Codigo de este tipo de comunicacion (Transmision de fichero).
char buffer[MAX_BUFF];  //Tamano maximo de cada envio.
FILE *fichero;          //Descriptor del fichero a enviar.
int n_bytes=0;          //Numero de bytes enviados por mensaje.
```

```
int control=0;          //Control de errores y de la salida del bucle.

//Puesta a cero del buffer de transmision.
bzero(&buffer, sizeof(buffer));
printf("Abriendo archivo\n");

//Apertura del fichero a enviar.
fichero = fopen("envio", "r");
if (fichero==NULL){
  //No existente/error.
  perror("Apertura del fichero ha fallado");
} else{
  //Apertura correcta, se prodece a iniciar su envio. Se manda el codigo.
  if(send(socket,codigo, sizeof(codigo),0)==-1){
    //Fallo en el envio del codigo, posible problema de conectividad.
    perror("Fallo al enviar datos");
  } else {
    //Envio del codigo ejecutado, se pasa a leer y enviar el fichero.
    control=1;
    //Se sale del bucle al terminar el envio del fichero o ante un error de conexion.
    //En primer lugar se lee un bloque del fichero (fread) almacenandolo en buffer, tras lo cual se envia (send).
    while (control>0 && (n_bytes=fread(buffer,sizeof(char),sizeof(buffer),fichero))>0){
      control=send(socket, buffer,n_bytes,0);
    }
    if(control<0){
      perror("Fallo en el envio");
    } else {
      control=0;
      printf("Terminado de transmitir\n");
      fflush(stdout);
    }
  }
//Pase lo que pase el descriptor del fichero debe cerrarse (Siempre que se halla llegado a abrir).
fclose(fichero);
}
return control;     //Resultado de la operacion.
}


/*
 * SIN USO. Recibe un fichero a traves de un socket tcp y lo guarda en el directorio de ejecucion
 * con el nombre "recibido".
 *
 * Parametros de entrada:
 * -socket --> socket tcp conectado con el emisor.
 *
 * Parametros de salida:
 * -control -->  >0, El fichero se ha transmitido correctamente.
 *             <=0, Ha habido un error durente la transmision.
 */

int recibir_fichero(int socket){

  FILE *fichero;
  int n_bytes=0;
  char buffer[MAX_BUFF];
  int control=0;

  fichero=fopen("recibido","a");
  if(fichero==NULL){
    perror("Error al abrir el fichero. Cierre de socket");
  } else {
    while ((n_bytes = recv(socket, buffer, sizeof(buffer), 0)) > 0 && control==0)
```

```
   {
     fwrite(buffer, sizeof(char), n_bytes, fichero);
     bzero(&buffer,sizeof(buffer));
     if (n_bytes< (int)sizeof(buffer))
     {
       control=1;
       printf("Fin de recepcion\n");
       fflush(stdout);
     }
   }
   if(n_bytes<0){
     perror("Error al recibir fichero");
     control=-1;
   }
  fclose(fichero);
  }
return control;
     }
```

COMUN.C

```
/*
 * comun.c
 *
 *  Created on: 05/05/2015
 *      Author: Alberto Rodriguez Blazquez
 *      E-mail: albertoroblaz@hotmail.com
 *      TFG: Diseño y desarrollo de un sistema ITS basado en comunicaciones tipo ISO CALM.
 *      Descripcion: Funciones de manejo de direcciones IP.
 *      Version: 0.9 Final
 */

#include "libreria.h"

/*
 *
 * Convierte una direccion de Internet y un puerto de servicio
 * (ambos cadena de caracteres) a valores numericos para poder
 * ser utilizados en otras funciones, como bind y connect.
 * Parametros de entrada:
 * - maquina - cadena de caracteres con la direccion de Internet
 * - puerto - cadena de caracteres con el puerto de servicio
 * - tipo - SOCKET_UDP, SOCKET_TCP o SOCKET_TCP_PASIVO (escucha)
 * Parametros de salida:
 * - info - estructura addrinfo con el primer valor encontrado
 * Devuelve:
 * - Verdadero=0, si ha tenido exito.
 * - Falso!=0, si ha habido error.
 */
int traduce_a_direccion(const char *maquina, const char *puerto,
                        int tipo, struct addrinfo *info)
{
  struct addrinfo hints;       /* Estructura utilizada para afinar la
                                busqueda de direcciones*/
  struct addrinfo *result;     /*Puntero a la lista de estructuras addrinfo con
                                los diferentes destinos validos para la maquina/direccion/tipo dados*/

  int error = 0;          /*Variable de control. Indica al hilo superior el resultado de esta funcion*/

  /* Ponemos a 0 la estructura hints */
```

```
memset(&hints, 0, sizeof(struct addrinfo));

/*Inicializamos la estructura hints. Ver "man getaddrinfo para saber porque se hace los siguiente*/

hints.ai_family = AF_INET;    /* AF_UNSPEC Permite IPv4 o IPv6, AF_INET solo IPv4 */
hints.ai_canonname = NULL;    //Existente en la estructura addrinfo pero no usado por getaddrinfo().
hints.ai_addr = NULL;
hints.ai_next = NULL;
hints.ai_protocol = 0;        /* Cualquier protocolo */

/*Tipo de socket pasado por paramentro*/
if (SOCKET_UDP == tipo || SOCKET_UDP_PASIVO==tipo)
  hints.ai_socktype = SOCK_DGRAM;     /* Socket de datagramas */
else
  hints.ai_socktype = SOCK_STREAM;    /* Socket de flujo */

/*Si el socket no es pasivo, ai_flags queda a 0. Los sockets pasivos son siempre TCP.*/
if (SOCKET_TCP_PASIVO == tipo || SOCKET_UDP_PASIVO == tipo)
  hints.ai_flags |= AI_PASSIVE;       /* Cualquier direccion IP */


/*Parte IMPORTANTE*/
/*Llamamos a la funcion de busqueda de nombres */
error = getaddrinfo(maquina, puerto, &hints, &result);

/*Tratamiento de error y de la no existencia de conexiones a la maquina/puerto dados*/
if (error != 0)
  {
    //Mostramos informacion del error usando la funcion gai_strerror.
    perror("traduce_a_direccion: Error obteniendo socket");
    fprintf(stderr, "traduce_a_direccion: getaddrinfo: %s\n", gai_strerror(error));
  }
else
  {

    if (result == NULL)
      {                   /* No se ha devuelto ninguna direccion */
        perror("traduce_a_direccion: No se han encontrado direcciones al destino.\n");
        error = 1;          //Indicar el error a la salida de la funcion.
      }
    else
      {
        //Copiamos solo los campos del primer resultado que interesan.

        /*Aclaracion: getaddrinfo devuelve una lista de estructuras addrinfo enlazadas
        mediante el campo ai_next. Cada una de las estructuras que devuelve seria una
        "ruta" valida al destino buscado. Por norma general, un socket pasivo tiene una
        sola ruta (una lista de una unica estructura) y para sockets "activos", lo recomendable
        es ir probando en el orden.
        En esta primera fase de pruebas, usaremos unicamente la primera, dandonos igual es resto
        de caminos validos. Más adelante quizas si sea intenresante probar con una u otra en funcion
        de si se quiere usar un protocolo o tipo de socket determinado*/

        info->ai_family = result->ai_family;
        info->ai_socktype = result->ai_socktype;
        info->ai_protocol = result->ai_protocol;
        *info->ai_addr = *result->ai_addr;    //Copiamos contenido del puntero
        info->ai_addrlen = result->ai_addrlen;
      }

    freeaddrinfo(result);     /* Liberamos los datos de la estructura devuelta,
                                 getaddrinfo especifica que es necesario liberarlos manualmente */
```

```
    }
  return error;
}

/* Funciones de obtiene la direccion IP del dispositivo de la interfaz del equipo pasada
 * Funciones procedente de http://stackoverflow.com/questions/2283494/get-ip-address-of-an-interface-on-linux.
 *
 * Parametros de entrada:
 * - interfaz: Puntero a una cadena con el nombre de una interfaz activa.
 * - host: Puntero a la cadena cdonde se guardaŕa la IP usada en la interfaz pasada.
 *
 * Parametros de salida:
 * - Puntero a la cadena con la direccion IP en caso de haberse obtenido.
 *
 */

char * obtenerIPPropia(char* interfaz, char *host){

    struct ifaddrs *ifaddr, *ifa;
    int s;

    if (getifaddrs(&ifaddr) == -1)
    {
        perror("Fallo al obtener direccion propia: etifaddrs");
        host=NULL;
    }


    for (ifa = ifaddr; ifa != NULL; ifa = ifa->ifa_next)
    {
        if (ifa->ifa_addr == NULL)
            continue;

        s=getnameinfo(ifa->ifa_addr,sizeof(struct sockaddr_in),host,IP_LENGTH, NULL, 0,
NI_NUMERICHOST);

        if((strcmp(ifa->ifa_name,interfaz)==0)&&(ifa->ifa_addr->sa_family==AF_INET))
        {
            if (s != 0)
            {
                printf("getnameinfo() failed: %s\n", gai_strerror(s));
                host=NULL;
            }
        }
    }
freeifaddrs(ifaddr);
return host;
}

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
      }
```

SOCKETS.C

```c
#include "libreria.h"

/*
 * sockets.c
 *
 *  Created on: 17/05/2015
 *      Author: Alberto Rodriguez Blazquez
 *      E-mail: albertoroblaz@hotmail.com
 *      TFG: Diseño y desarrollo de un sistema ITS basado en comunicaciones tipo ISO CALM.
 *      Descripcion: Creacion de los sockets del cliente y el servidor.
 *      Version: 0.9 Final
 *
 */

/*
  Realiza el proceso de inicializacion estandar de un socket TCP activo.
  En caso de error el valor del socket devuelto es negativo.
  Parametros de entrada:
  - maquina - cadena de texto con el nombre o direccion IP de la maquina.
  - puerto  - cadena de texto con el puerto a usar. Puede ser un numero
            o un nombre de servicio existente en el fichero services
  Devuelve:
  - s   -Nuevo socket. <0 si ha habido error.
*/
int inicia_socket_cliente(const char *maquina, const char *puerto)
{
  int s;                    /* Variable del socket a crear*/
  struct addrinfo infoserv;     /* informacion extremo local */
  struct sockaddr dirserv;      /* direccion internet socket servidor */

  /* Ponemos a 0 la estructura infoserv */
  memset(&infoserv, 0, sizeof(struct addrinfo));
  infoserv.ai_addr = &dirserv;  //aqui se guardara la direccion
  /* Obtenemos datos del extremo al que queremos llamar.
     Si tiene exito, utilizaremos los datos almacenados en infoserv
     para crear el socket. */
    if (traduce_a_direccion(maquina, puerto, SOCKET_TCP, &infoserv)) {
       perror("inicia_socket_cliente: Fallo en busqueda de direcciones");
       s = ERROR;
     } else {

       /* Se ha encontrado el destino, se crea el socket TCP con los datos devueltos. */
       if ((s = socket(infoserv.ai_family, infoserv.ai_socktype, infoserv.ai_protocol)) < 0){
          perror("inicia_socket_cliente: Error en la creacion del socket");
       } else{
          /* Conectamos el socket */
          if (connect(s, &dirserv, sizeof(dirserv)) < 0)
          {
             close(s);
             perror("inicia_socket_cliente: Error al conectar el socket");
             s = ERROR;
          }
       }
    }
    return s;
}


/*
 * Funcion para iniciar el socket de escucha UDP.
 *
 * Parametros de entrada:
 * -puerto -->Cadena char con el numero de puerto a usar.
```

```
 *
 * Parametros de salida:
 * -socket_escucha -->Descriptor del socket creado (-1 en caso de error).
 */

 int iniciaEscuchaUDP(char *puerto){

  int socket_escucha;
  struct addrinfo infoserv;     /* informacion extremo local */
  struct sockaddr dirserv;      /* direccion internet socket servidor */
  int on = 1;

  /* Ponemos a 0 la estructura infoserv */
  memset(&infoserv, 0, sizeof(struct addrinfo));
  infoserv.ai_addr = &dirserv;  //aqui se guardara la direccion
  /* Al indicar SOCKET_TCP_PASIVO, indicamos que queremos escuchar
     en todas las interfaces de la maquina. Si tiene exito, utilizaremos
     los datos almacenados en infoserv para crear el socket. */

  if (traduce_a_direccion(NULL, puerto, SOCKET_UDP_PASIVO, &infoserv)!=0){
    perror("iniciaEscuchaUDP: Fallo en busqueda de direcciones");    //Se da cuando ha devuelto algo distinto de
0.
    socket_escucha=-1;
  }
  else{
       /*Creacion del socket de escucha a partir de la estructura infoserv*/
    if ((socket_escucha = socket(infoserv.ai_family, infoserv.ai_socktype,infoserv.ai_protocol)) < 0)
    perror("iniciaEscuchaUDP: Error al crear socket de escucha");

    else{
      /* Reusar direccion si hay conexiones activas pero no escuchando. */
          /* Reusar direccion si hay conexiones activas pero no escuchando. */
      if (setsockopt(socket_escucha, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0){
/*ANALIZAR*/
        close(socket_escucha);
        socket_escucha=-1;
        perror("iniciaEscuchaUDP: setsockopt");
      } else {
        /* Asigna nombre local al socket */
        /* Esto permite que las peticiones entrantes puedad ser atendidas por el SO.*/
        if (bind(socket_escucha, &dirserv, sizeof(dirserv)) < 0) //Esta funcion da negativo en el caso de que el
puerto escogido este ocupado.
        {
          close(socket_escucha);
          socket_escucha=-1;
          perror("iniciaEscuchaUDP: Error en bind");
        }
      }
    }
  }
  return socket_escucha;
 }


 /*
 *Inicia un puerto de escucha TCP.
 *
 * @param puerto : int, puerto donde se va a realizar la escucha.
 * @return s: Numero de puerto donde se realiza la escucha.
 */
int iniciaEscuchaTCP(char *puerto)
 {
  int on = 1;
```

```
  int s;                    /* Socket TCP de escucha*/
  struct addrinfo infoserv;     /* informacion extremo local */
  struct sockaddr dirserv;      /* direccion internet socket servidor */

  //* Calcula "nombre local" */
  /* Ponemos a 0 la estructura infoserv */
  memset(&infoserv, 0, sizeof(struct addrinfo));
  infoserv.ai_addr = &dirserv;  //aqui se guardara la direccion
  /* Al indicar SOCKET_TCP_PASIVO, indicamos que queremos escuchar
     en todas las interfaces de la maquina. Si tiene exito, utilizaremos
     los datos almacenados en infoserv para crear el socket. */
  if (traduce_a_direccion(NULL, puerto, SOCKET_TCP_PASIVO, &infoserv)!=0){
    perror("iniciaEscuchaTCP: Fallo en busqueda de direcciones");    //Se da cuando ha devuelto algo distinto de
0.
    s=-1;
  }
  else{

    /*Creacion del socket de escucha a partir de la estructura infoserv*/
    if ((s = socket(infoserv.ai_family, infoserv.ai_socktype,infoserv.ai_protocol)) < 0)
    perror("iniciaEscuchaTCP: Error al crear socket de escucha");

      else{
        /* Reusar direccion si hay conexiones activas pero no escuchando. */
        if (setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0)   /*ANALIZAR*/
        {
        close(s);
        s=-1;
        perror("iniciaEscuchaTCP: setsockopt");
        } else{
           /* Asigna nombre local al socket */
           /* Esto permite que las peticiones entrantes puedad ser atendidas por el SO.*/
           if (bind(s, &dirserv, sizeof(dirserv)) < 0) //Esta funcion da negativo en el caso de que el puerto escogido
este ocupado.
           {
           close(s);
           s=-1;
           perror("iniciaEscuchaTCP: Error en bind");
           } else{
              /* Prepara al servidor para aceptar conexiones */
              if (listen(s, 1) < 0)   //Maximo 1 cliente esperando a ser atendido. IMPORTANTE. No son necesarios
mas, de momento.
              {
              close(s);
              s=-1;
              perror("iniciaEscuchaTCP: Error en la preparacion de la escucha");
              }
           }
        }
    }
  }
  return s;   //Se devuelve el socket operativo.
}


/*
*/
int iniciaHolaUDP(struct sockaddr_in * sock_int){

  int sock;
  int sinlen;
  int on = 1;
  struct sockaddr_in sock_in=*sock_int;
```

```
 sinlen = sizeof(struct sockaddr_in);
 if((sock=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP))<0){
   perror("¡nicioHolaUDP: Error al crear socket broadcast");
 } else {

   if (setsockopt(sock,SOL_SOCKET,SO_BROADCAST,&on,sizeof(on))<0){
     close(sock);
     sock=-1;
     perror("iniciaHolaUDP: setsockopt");
   } else {

     memset(&sock_in, 0, sizeof(sinlen));
     sock_in.sin_addr.s_addr = inet_addr(HELLO_GROUP);
     sock_in.sin_port = htons(21216);
     sock_in.sin_family = AF_INET;
     //No es necesario que se fije a un puerto concreto.
     if(bind(sock, (struct sockaddr *) &sock_in, sinlen)<0){
       close(sock);
       sock=-1;
       perror("iniciaHolaUDP: Error en bind");
     }
   }
 }
 return sock;

}


/*
 *
 * Bloquea comprueba durante un tiempo maximo de "segundos" si se ha
 * recibido algo a traves de un socket determinado.
 *
 * Parametros de entrada:
 *
 * - descriptor - descriptor del fichero o socket a comprobar.
 * - segundos - tiempo maximo de espera en segundos.
 * Devuelve:
 * - 1 si se puede leer, 0 si ha vencido el plazo.
 */
int comprueba_recepcion(int descriptor, int segundos)
{

 if(descriptor>0)
 {
   struct timeval plazo = { segundos, 0L };      /* plazo de recepcion */

   fd_set fds;                 //Conjunto de descriptores a monitorizar
   FD_ZERO(&fds);              //Limpiamos fds
   FD_SET(descriptor, &fds);     //Insertamos descriptor en el conjunto
   return (select
         (descriptor + 1, &fds, (fd_set *) NULL, (fd_set *) NULL, &plazo));
 } else {
   return 0;
 }

    }
```

NODOS.C

```c
#include "libreria.h"

/*
 * nodos.c
 *
 *  Created on: 17/05/2015
 *      Author: Alberto Rodriguez Blazquez
 *      E-mail: albertoroblaz@hotmail.com
 *      TFG: Diseño y desarrollo de un sistema ITS basado en comunicaciones tipo ISO CALM.
 *      Descripcion: Majejo de la lista de nodos conocidos.
 *      Version: 0.9 Final
 *
 */


/*
 * Busca un nodo en la lista de nodos conocidos y lo anade si no esta.
 *
 * Parametros de entrada:
 * -inicio_lista --> Puntero al puntero a la primera estructura DIRIP de la lista.
 * -saludo --> Mensaje HI recibido de un nodo.
 * -direccion_ip --> Puntero a donde esta guardada la direccion IP del nodo que envio el HI.
 *
 * Parametros de salida:
 * -puntero -->  =NULL, Error o ya existente.
 *          -->  !=NULL, Estructura creada.
 */

struct DIRIP* anadirNodoConocido(struct DIRIP ** inicio_lista, struct HI saludo, char *direccion_ip ){

 struct DIRIP * puntero=NULL;      //Puntero para moverse por la lista de estructuras DIRIP.
 struct DIRIP * aux=NULL;          //Puntero auxiliar para moverse por la lista de estructuras DIRIP.
 int control=0;                    //Control del bucle while y de errores.

 //Reserva de memoria para una estructura DIRIP.
 if((puntero=(struct DIRIP *) calloc(1, sizeof(struct DIRIP)))==NULL)
 {
   //No se ha podido reservar memoria.
   printf("anadirNodoConocido: Memoria no reservada\n");
   control=-1;
   puntero=NULL;
 } else{
   if(*inicio_lista==NULL){
     //Si la lista el puntero al inicio esta a null, se crea la primera estructura.
     *inicio_lista=puntero;
     memset(puntero->direccion,'\0',IP_LENGTH);
     strncpy(puntero->direccion, direccion_ip, IP_LENGTH);
     puntero->id=saludo.id;
     puntero->flag=0;
     puntero->next=NULL;
     control=0;
   } else {
     //En caso de haber estructura en la lista se recorren buscado una que coincida en id.
     aux=*inicio_lista;
     do{
       if(aux->id==saludo.id)
       {
         //Si se encuentre se actualiza su IP.
         //Tambien se libera la memoria reservada, no es necesaria.
         //Por seguridad, el campo IP se pone a o antes de sobreescribirlo.
         memset((*aux).direccion, '\0', IP_LENGTH);
```

```
        strncpy((*aux).direccion, direccion_ip, IP_LENGTH);
        control=1;
        free(puntero);
        puntero=NULL;
      } else {
        if(aux->next!=NULL)
        {
          aux=aux->next;
        } else {
          //En caso de haber llegado al final de la lista se sale con control=2.
          control=2;
        }
      }
    } while(control<1);

    //Se ha alcanzado el final de la lista, se anade una nueva estructura a ella.
    if(control==2)
    {
      printf("Creacion de un nuevo nodo \n");
      aux->next=puntero;
      memset(puntero->direccion,'\0',IP_LENGTH);
      strncpy(puntero->direccion, direccion_ip, IP_LENGTH);
      puntero->id=saludo.id;
      puntero->next=NULL;
      puntero->flag=0;
      control=0;
    }
  }
 }
 if(borrarNodosMarcados(inicio_lista)==-1){
   printf("anadirNodoConocido: Lista vacia.\n");
 }
 return puntero;
}

/*
 * SIN USO. Busca una estructura DIRIP con una direccion ip determinana en la lista de nodos conocidos.
 *
 * Parametros de entrada:
 * -inicio_lista --> Puntero al inicio de la lista de estructuras DIRIP.
 * -direccion_ip --> Puntero a donde esta guardada la direccion IP a buscar.
 *
 * Parametros de salida:
 * -puntero --> Puntero a la estructura DIRIP buscada si se ha encontrado.
 *          --> NULL, es caso de no haberse encontrado.
 */

struct DIRIP* buscarNodo(struct DIRIP * inicio_lista, char* direccion_ip){

 struct DIRIP * puntero=NULL;
 int control=0;
 int comparacion=0;

 if(inicio_lista==NULL){
   printf("buscarNodo: Lista de nodos vacia\n");
   control=-1;
 } else {
   *puntero=*inicio_lista;
   do{
     comparacion=strncmp((*puntero).direccion,direccion_ip,IP_LENGTH);
     if(comparacion==0){
       control=1;
     } else{
```

```
      puntero=(*puntero).next;
     }
    } while(control==0 && puntero!=NULL);
  }
return puntero;
}

/*
 * Busca y borra una estructura DIRIP que tenga el id de la lista de nodos conocidos.
 *
 * Parametros de entrada:
 * -inicio_lista --> Puntero al inicio de la lista de nodos conocidos.
 * -id --> campo id de la estructura a aliminar.
 *
 * Parametros de salida:
 * -control --> =-1, inicio_lista=NULL.
 *              =0, Estructura no encontrada.
 *              =1, Estructura borrada.
 */

int borrarNodo(struct DIRIP ** inicio_lista, int id){

  struct DIRIP * puntero=NULL;   //Puntero a la estructura siguiente a la borrada.
  struct DIRIP ** aux=NULL;      //Puntero usado para desplazarse por la lista.
  int control=0;                 //Control de resultados.

  if((*inicio_lista)==NULL)
  {
    //No esta la lista creada.
    printf("borrarNodo: Lista nodos vacia\n");
    control=-1;
  } else {
    aux=inicio_lista;
    do{
      if((**aux).id==id)
      {
        //Si el campo id de la estructura coincide se elimina y el puntero de la anterior pasa a apuntar a la siguiente.
        puntero=(**aux).next;
        free(*aux);
        *aux=puntero;
        control=1;
      } else {
        if((**aux).next==NULL)
        {
          //En este caso no se ha encontrado la estructura a borrar.
          control=2;
        } else {
          *aux=(*aux)->next;
        }
      }
    } while (control==0);
    if(control==2){
      printf("borrarNodo: Estructura no encontrada\n");
      control=0;
    }
  }
  return control;
}


/*
 * Borar todas las estructuras DIRIP existentes en la lista de nodos conocidos.
 *
```

```
 * Parametros de entrada:
 * -inicio_lista --> Puntero al inicio de la lista de estructuras.
 *
 * Parametros de salida:
 * control --> =-1, La lista ya esta vacia.
 *         -->  =0, Se ha borrado la lista.
 */

int borrarNodos(struct DIRIP ** inicio_lista){

  struct DIRIP * puntero=NULL;  //Punteo para irse moviendo por la lista.
  struct DIRIP * aux=NULL;      //Puntero auxiliar para el borrado de la lista.
  int control=0;                //Control de salida de la funcion.

  if(*inicio_lista==NULL)
  {
    //La lista ya esta vacia.
    printf("borrarNodos: Lista nodos vacia\n");
    control=-1;
  } else {
    //Se copia el puntero a puntero y se pone a NULL el otro.
    puntero=*inicio_lista;
    *inicio_lista=NULL;
    //Se pasa por todas las listas liberandolas.
    while(puntero!=NULL){
      aux=puntero->next;
      free(puntero);
      puntero=aux;
    }
  }
  return control;
}


/*
 * Busca y borra las estructuras DIRIP que tenga las bandera flag a 1 de la lista de nodos conocidos.
 *
 * Parametros de entrada:
 * -inicio_lista --> Puntero al inicio de la lista de nodos conocidos.
 *
 * Parametros de salida:
 * -control --> =-1, inicio_lista=NULL.
 *              =0, Salida normal
 */

int borrarNodosMarcados(struct DIRIP ** inicio_lista){

  struct DIRIP * puntero=NULL;   //Puntero a la estructura siguiente a la borrada.
  struct DIRIP ** aux=NULL;      //Puntero usado para desplazarse por la lista.
  int control=0;                 //Control de resultados.

  if((*inicio_lista)==NULL)
  {
    //No esta la lista creada.
    printf("borrarNodosMarcados: Lista nodos vacia\n");
    control=-1;
  } else {
    aux=inicio_lista;
    do{
      if((**aux).flag==1)
      {
        //Si el campo id de la estructura coincide se elimina y el puntero de la anterior pasa a apuntar a la siguiente.
        puntero=(**aux).next;
```

```
        free(*aux);
        *aux=puntero;
        if(puntero==NULL){
          control=1;
        }
      } else {
        if((**aux).next==NULL)
        {
          //En este caso no se ha encontrado la estructura a borrar.
          control=2;
        } else {
          *aux=(*aux)->next;
        }
      }
    } while (control==0);
    if(control==2){
      control=0;
    } else {
      control=0;
    }
  }
  return control;

      }
```

PELIGROS.C

```
  #include "libreria.h"

/*
 * peligros.c
 *
 *  Created on: 17/05/2015
 *      Author: Alberto Rodriguez Blazquez
 *      E-mail: albertoroblaz@hotmail.com
 *      TFG: Diseño y desarrollo de un sistema ITS basado en comunicaciones tipo ISO CALM.
 *      Descripcion: Manejo de la lista de peligros conocidos.
 *      Version: 0.9 Final
 */


/*
 * Anade un peligro al listado de peligros conocidos.
 *
 * Parametros de entrada:
 * -inicio_lista -->Puntero al inicio de la lista de peligros.
 * -peligro --> Mensaje DA con el nuevo peligro.
 *
 * Parametros de salida:
 * -control --> =-1, Error.
 *          -->  =0, Peligro anadido.
 *          -->  =1, Peligro ya conocido.
 */

int anadirPeligroConocido(struct PELIGROS ** inicio_lista, struct DA peligro){

  struct PELIGROS * puntero=NULL;   //Puntero a la nueva estructura a anadir a la lista.
  struct PELIGROS * aux=NULL;       //Puntero para desplazarse por la lista.
  int control=0;                    //Control de salida de la funcion.
```

```
  if((puntero=(struct PELIGROS *) calloc(1, sizeof(struct PELIGROS)))==NULL)
  {
    //No se ha podido crear la memoria, se sale.
    perror("anadirPeligroConocido: Memoria no reservada\n");
    control=-1;
  } else{
    if(*inicio_lista==NULL)
    {
      *inicio_lista=puntero;
      puntero->tipo=peligro.tipo;
      puntero->pos=peligro.pos;
      puntero->id=peligro.id;
      puntero->flag=1;
      puntero->next=NULL;
      control=0;
    } else {
      //En caso de haber estructura en la lista se recorren buscado una que coincida la posicion del nuevo peligro.
      aux=*inicio_lista;
      do{
        if(aux->pos==peligro.pos)
        {
          //Si se encuentra el peligro ya guardado no se hace nada.
          //Tambien se libera la memoria reservada, no es necesaria.
          control=1;
          free(puntero);
        } else {
          if(aux->next!=NULL)
          {
            aux=aux->next;
          } else {
            //En caso de haber llegado al final de la lista se sale con control=2.
            control=2;
          }
        }
      } while(control<1);
      //Se ha alcanzado el final de la lista, se anade una nueva estructura a ella.
      if(control==2)
      {
        aux->next=puntero;
        puntero->tipo=peligro.tipo;
        puntero->pos=peligro.pos;
        puntero->id=peligro.id;
        puntero->next=NULL;
        puntero->flag=1;
        aux->flag=0;
        control=0;
      }
    }
  }
  return control;
}


/*
 * Se busca un peligro en la lista de peligros conocidos. Se usa su posicion.
 *
 * Parametros de entrada:
 * -inicio_lista-->Puntero a la lista de peligros conocidos.
 * -posicion -->Campo posicion del peligro buscado.
 *
 * Parametros de salida:
 * -control --> =0, No se ha encontrado un peligro en esa posicion.
 *          --> =1, Peligro conocido encontrado.
```

```
 */

int buscarPeligro(struct PELIGROS * inicio_lista, float posicion){

  struct PELIGROS * puntero=NULL;
  int control=0;

  if(inicio_lista==NULL)
  {
    //La lista de peligros esta vacia.
    printf("buscarPeligro: Lista de peligros vacia\n");
    control=0;
  } else {
    puntero=inicio_lista;
    do{
      if((*puntero).pos==posicion){
        control=1;
      } else {
        puntero=puntero->next;
      }
    } while(control==0 && puntero!=NULL);
  }
return control;
}


/*
 * SIN USO. Borra un peligro de la lista de peligros conocidos dada su posicion.
 *
 * Parametros de entrada:
 * -inicio_lista -->Puntero al puntero inicio de la lista.
 * -posicion -->Posicion que identifica el peligro a borrar.
 *
 * Parametros de salida:
 * -control --> =-1, inicio_lista=NULL.
 *              =0, Peligro no encontrado.
 *              =1, Peligro borrado.
 */

int borrarPeligro(struct PELIGROS ** inicio_lista, float posicion){

  struct PELIGROS * puntero=NULL;
  struct PELIGROS ** aux=NULL;
  int control=0;

  if((*inicio_lista)==NULL)
  {
    //No esta la lista creada.
    printf("borrarPeligro: Memoria no reservada\n");
    control=-1;
  } else {
    aux=inicio_lista;
    do{
      if((**aux).pos==posicion)
      {
        //Si el campo pos de la estructura coincide se elimina y el puntero de la anterior pasa a apuntar a la
siguiente.
        puntero=(**aux).next;
        free(*aux);
        *aux=puntero;
        control=1;
      }  else {
        if((**aux).next==NULL)
```

```
      {
        //En este caso no se ha encontrado la estructura a borrar.
        control=2;
      } else {
        *aux=(*aux)->next;
      }
    }
  } while (control==0);
  if(control==2){
    printf("borrarPeligro: Peligro no encontrado\n");
    control=0;
  }
 }
 return control;
}


/*
 * Borar todas las estructuras PELIGROS existentes en la lista de peligros conocidos.
 *
 * Parametros de entrada:
 * -inicio_lista --> Puntero al inicio de la lista de estructuras.
 *
 * Parametros de salida:
 * control --> =-1, La lista ya esta vacia.
 *         -->  =0, Se ha borrado la lista.
 */

int borrarPeligros(struct PELIGROS ** inicio_lista){

 struct PELIGROS * puntero=NULL;  //Punteo para irse moviendo por la lista.
 struct PELIGROS * aux=NULL;      //Puntero auxiliar para el borrado de la lista.
 int control=0;                   //Control de salida de la funcion.

 if(*inicio_lista==NULL)
 {
  //La lista ya esta vacia.
  perror("borrarPeligros: Lista peligros vacia\n");
  control=-1;
 } else {
  //Se copia el puntero a puntero y se pone a NULL el otro.
  puntero=*inicio_lista;
  *inicio_lista=NULL;
  //Se pasa por todas las listas liberandolas.
  while(puntero!=NULL){
   aux=puntero->next;
   free(puntero);
   puntero=aux;
  }
 }
 return control;
    }
```