
**Universidad de Valladolid**

E.T.S. Ingeniería Informática

**TRABAJO FIN DE GRADO**

Grado en Ingeniería Informática

# Propuesta de adaptación de un *driver* Linux para comunicaciones 802.11p

**Autor:**
Hernán Maximiliano González Calderón

**Universidad de Valladolid**

E.T.S. Ingeniería Informática

**TRABAJO FIN DE GRADO**

Grado en Ingeniería Informática

# Propuesta de adaptación de un *driver* Linux para comunicaciones 802.11p

**Autor:**
Hernán Maximiliano González Calderón

**Tutor:**
Dr. Diego R. Llanos

# Propuesta de adaptación de un *driver* Linux para comunicaciones 802.11p

Hernán Maximiliano González Calderón

**Palabras clave**: Comunicación intervehicular, WAVE, 802.11p, *driver*, ath5k

## Resumen

Se entiende por comunicación intervehicular a toda transmisión de datos entre dos vehículos o entre un vehículo y elementos infraestructurales. Su ámbito está sobre todo enfocado a la seguridad vial. Posibles aplicaciones son mecanismos de comunicación que adviertan a conductores de peligros que puedan pasar inadvertidos como colisiones o replanificaciones de rutas en caso de posibilidad de atascos. Se toma como estándares de desarrollo el protocolo IEEE 802.11p orientado a al acceso inalámbrico en entornos vehiculares, que junto el protocolo IEEE 1609 conforman lo que se conoce como WAVE (*Wireless Access in Vehicular Environments*). WAVE es un modo de operación dentro de las bandas de 5,9 GHz DSRCC (*Dedicated Short Range Communications*) reservadas para las comunicaciones de sistema de transporte inteligente. El proyecto propuesto consiste en analizar el *driver* `ath5k` para comprender su funcionamiento y elaborar una propuesta de modificación sobre él que permita realizar una implementación del protocolo IEEE 802.11p para tarjetas Atheros de la serie 5xxx y para contribuir a la pila de protocolos necesaria para una comunicación WAVE.

# Agradecimientos

En primer lugar quiero agradecer a mis padres y hermano por el apoyo que me han prestado ne todo momento. Haciendo piña ahora y siempre vayamos donde vayamos.

Agradezco por supuesto a mis compañeros y amigos del Cuarteto por la paciencia que han tenido conmigo a lo largo de estos años y por todo lo que me han enseñado en este tiempo; a todos y cada uno de los miembros de *Kiwicore* por el cariño que me han brindado desde que llegase a España; a mi familia allende los mares, en especial a mi padrino, por haber confiado siempre en mí; y, por supuesto, a Vicky por haberme aguantado en mis peores días de encierro.

Por último, quería dedicar este agradecimiento a mi tutor, Diego, por ayudarme a llevar con más ligereza este trabajo y por su inmensa dedicación, tanto en su papel como tutor en este TFG, como en su papel de docente a lo largo mi paso por la escuela.

Sencillamente, gracias a todos los nombrados y los que haya podido olvidar por haberme ayudado a llegar hasta aquí.

# Índice general

# Índice de figuras

# Índice de tablas

# Índice de fragmentos de Códigos

# Capítulo 1

# Introducción

## 1.1. Descripción del problema

Se entiende por comunicaciones vehículo-a-vehículo a toda transmisión de datos entre dos vehículos y su uso está sobre todo enfocado al ámbito de la seguridad vial. Entre las aplicaciones de este mecanismo de comunicación se incluye advertir a los conductores de peligros que puedan pasar inadvertidos, lo que incluye un vehículo detenido en un carril, una frenada inesperada, la aparición de una ambulancia o una colisión. Para ello el sistema está compuesto generalmente por un microprocesador específico encargado de procesar la información y codificarla para el conductor, un módulo *WiFi* inalámbrico de onda corta, sensores ubicados en lugares estratégicos y un sistema GPS que se comunica con el microprocesador y el dispositivo *WiFi* a través del cual el conductor recibe señales visuales y auditivas. Debido a que la comunicación es vehículo a vehículo la información sobre el estado del tráfico puede propagarse fácilmente entre cada uno de ellos pudiendo ayudar a planificar rutas alternativas.

Este proyecto toma como estándar para su desarrollo el protocolo IEEE 802.11p, la enmienda aprobada en el 2010 del estándar IEEE 802.11, una propuesta para añadir el acceso inalámbrico a los entornos vehiculares. Dicha propuesta fue incorporada finalmente al estándar IEEE 802.11 en la revisión del 2012. Así, WAVE (*Wireless Access in Vehicular Environments*) es un modo de operación utilizado por los dispositivos que siguen el estándar IEEE 802.11 dentro de las bandas DSRC (*Dedicated Short Range Communications*), que es el nombre que reciben las bandas de los 5,9 GHz reservadas para las comunicaciones de sistemas de transporte inteligente o ITS (*Intelligent Transport System*).

El estándar IEEE 802.11p se encarga de las dos primeras capas de la pila de protocolo, la capa física y la capa de enlace, cubriendo el entorno DSRC. Del resto de capas se encarga el estándar IEEE 1609, el cual define la administración de recursos, servicios de seguridad para aplicaciones y administración de mensajes, los servicios de red y operaciones multicanal.

A pesar de que existen tarjetas que en teoría implementan estos estándares o, al menos, lo soportan, los dispositivos que se encuentran en el mercado

constan de tarjetas cerradas sin posibilidad de modificar ni acceder al código fuente de los *drivers*. Su elevado precio y la complicación que supone estudiarlas en detalle debido a sus características hacen necesario adoptar otro enfoque.

## 1.2. Aportaciones del proyecto

El proyecto propuesto consiste en utilizar técnicas de ingeniería inversa para analizar el funcionamiento del *driver* `ath5k`, con el fin de elaborar una propuesta de modificación de las capas de red de Linux y adecuarlas a las características del estándar IEEE 802.11p. Para ello se utilizarán dos tarjetas de bajo coste *WiFi* Mini PCI MikroTik R52H con *chipset* Atheros AR5413/AR5414 y adaptadores PCI, cuyo driver, `ath5k`, debe ser modificado para emitir con las características que indica el estándar 802.11p. El objetivo es el desarrollo de una documentación detallada que permita en un futuro próximo (tan pronto como los fabricantes del *hardware* liberen la información necesaria) desarrollar una implementación libre sobre Linux del protocolo IEEE 802.11p

## 1.3. Estructura de la memoria

La estructura que sigue esta memoria es la siguiente. En **Introducción** se define el problema del que parte este Trabajo de Fin de Grado, la situación actual de la solución, sus carencias y las aportaciones de la solución propuesta. En **Estado del arte** se describen los avances alcanzados hasta la fecha en lo relativo a la implementación del protocolo. En **Descripción de la pila WAVE** se describen los protocolos que interactúan en la pila WAVE, sus propiedades y necesidades. En **Estudio del *driver*** se desarrolla la información obtenida sobre el funcionamiento del *driver* `ath5k` para tarjetas Atheros de la serie 5000. En **Especificación de requisitos** se listan los cambios necesarios a hacer sobre el *driver* `ath5k` para lograr la implementación del protocolo. En **Modelo de análisis** se especifica el modelo de desarrollo utilizado dadas las características del proyecto y se detalla la planificación temporal seguida. En **Modificaciones Propuestas** se listan y detallan las modificaciones que se plantearon realizar tras el estudio del *driver* para lograr una implementación completa del protocolo. En **Pruebas y Validación** se detalla la batería de pruebas propuesta para cada una de las modificaciones propuestas en el capítulo anterior. En **Instalación del entorno de desarrollo** se detallan los pasos a seguir para instalar el entorno de desarrollo así como el código modificado usado para el análisis del *driver* `ath5k` y las herramientas que se estimaron oportunas para conseguir una implementación del protocolo IEEE 802.11p. Por último, en **Conclusiones** se explica qué modificaciones propuestas han podido llevarse a cabo y con qué grado de éxito, qué complicaciones se han encontrado y qué nueva información se ha podido obtener de ellas, así como se hace una breve explicación sobre qué trabajo futuro puede venir ligado a este TFG.

## 1.4. Abreviaturas

- **ACK:** Del inglés *acknowledgement*, en español acuse de recibo o asentimiento. En el caso de comunicaciones entre computadores, es un mensaje que el destino de la comunicación envía al origen de ésta para confirmar la recepción de un mensaje
- **AHB:** *Advanced High-performance Bus*
- **ANI:** *Adaptative Noise Inmunity*
- **AP:** *Access Point*
- **ART:** *Atheros Radio Test*
- **BSD:** *Berkeley Software Distribution*
- **BSS:** *Infrastructural Basic Service Set*. No hay que confundirlo con *Independet Basic Service Set* o IBSS
- **BSSID:** *Basic Service Set Identification*
- **CPS-CVD:** *Cyber-Physical Systems - Cooperative Vehicle Demonstration*
- **CRDA:** *Central Regulatory Domain Agent*
- **CTS:** *Clear To Send*
- **DMA:** *Direct Memory Access*
- **DS:** *Distribution Service*
- **DSRC:** *Dedicated Short Range Communications*
- **EDCA:** *Enhanced Distributed Channel Access*
- **EEPROM:** *Electrically Erasable Programmable Read-Only Memory*
- **EPC:** *Enhanced Packet Core*
- **ESS:** *Extended Service Set*
- **FCS:** *Frame Check Sequence*
- **FFT:** *Fast Fourier Transformation*
- **FTP:** *File Transfer Protocol*
- **GCDC:** *Grand Cooperation Driving Challenge*
- **GNU:** Acrónomio recursivo que significa *GNU is Not Unix*
- **GNU GPL:** *GNU General Public License*

- **GPL:** *General Public License*
- **GPS:** *Global Positioning System*
- **IBSS:** *Independet Basic Service Set*. No hay que confundirlo con *Infrastructural Basic Service Set* o BSS
- **I+D:** Investigación + Desarrollo
- **IEEE:** *Institute of Electrical and Electronics Engineers*
- **ITS:** *Intelligent Transport Systems*
- **LED:** *Light-Emitting Diode*
- **LTE:** *Long Term Evolution*
- **MAC:** *Media Access Controller*
- capa **MAC:** Se refiere a la capa de enlace
- **OAI:** *OpenAirInterface*
- **OBU:** *Onboard Unit*
- **OFDM:** *Orthogonal Frequency Division Multiplexing*
- **PCI:** *Peripheral Component Interconnect*
- **PCU:** *Protocol Controller Unit*
- capa **PHY**: Se refiere a la capa física
- **PLCP:** *Physical Layer Convergence Protocol*
- **QCU:** *Queue Control Unit*
- **QoS:** *Quality of Service*
- **RF:** *Radio Frequency*
- **RSU:** *Road Side Unit*
- **SBC:** *Single Board Computer*
- **SSID:** *Service Set Identification*
- **TFG:** Trabajo de fin de grado
- **TSF:** *Timing Synchronization Function*
- **US DOT:** *United States Department of Transportation*
- **VANET:** *Vehicular Ad-Hoc Network*

- **VIVAGr:** *Visualization tool of Vanet Graphs in real-time*
- **WAVE:** *Wireless Access in Vehicular Environment*
- **WBSS:** *WAVE BSS*
- **WME:** *WAVE Management Entity*
- **WSMP:** *WAVE Short Message Protocol*

# Capítulo 2

# Estado del arte

La adaptación de las implementaciones existentes al estándar IEEE 802.11p han suscitado un gran interés en los últimos años, debido a sus implicaciones económicas. En este capítulo haremos un repaso a los principales estudios en esta línea.

## 2.1. *University of Moratuwa*

El trabajo realizado en la Universidad de Moratuwa por R. S. A. De Silva, D. P. G. S. R. Fernando, J. R. Kodagoda y I. U. Liyanage, con la supervisión del Dr. Ajith Pasqual fue el primer caso de estudio y la principal razón del planteamiento de este proyecto. Se trata de un proyecto de fin de curso de un alumno de la Universidad de Moratuwa en Sri Lanka en el cual se desarrolló un *driver* para un módulo *WiFi* embebido basado en el protocolo IEEE WAVE (IEEE 801.11p/IEEE 1609.x).

Aunque aseguran que obtuvieron resultados favorables la información sobre sus progresos reales es escasa y, de hecho, la mayor parte de ella proviene de las diapositivas usadas en la presentación del proyecto. No obstante, los datos obtenidos de ellas sirvieron de base para el inicio del análisis del problema.

Según la información obtenida, las tareas realizadas por su equipo fueron la construcción de una plataforma de desarrollo para el módulo de conexión inalámbrica, el desarrollo de las capas de comunicación bajo el protocolo IEEE 802.11p, el desarrollo de la capa de red bajo el protocolo IEEE 1609.3 y el desarrollo de una aplicación de aviso de colisiones.

La plataforma de desarrollo constaba de una tarjeta *WiFi* Mini PCI con un *chipset* Atheros de alta potencia de calidad industrial con soporte para el protocolo WAVE; una placa computadora o *SBC* (*Single Board Computer*) ALIX3D3 de bajo consumo, tamaño compacto y con soporte para Linux; una antena Swivel MikroTik de 2,4 GHz a 5,8 GHz y 3 dBi con cable y conector MMCX; y un módulo de antena GPS Finland Fastrax UP501.

Con sus estudios consiguieron adaptar el *driver* para conseguir una implementación del protocolo IEEE 802.11p partiendo de la base de la implemantación del protocolo IEEE 802.11a, modificando el rango de frecuencia, la amplitud del canal, el ratio de transmisión de datos y otros parámetros a los

especificados por los protocolos WAVE. Para las pruebas y validación utilizaron un analizador de espectro que les permitió comprobar que efectivamente las tarjetas eran capaz de emitir en las frecuencias requeridas y con el ancho de banda adecuado.

Una vez comprobado que los cambios sobre la capa física habían sido realizados correctamente, es decir, que la señal respetaba el protocolo, se centraron en la pila de protocolos y la estructura de los mensajes, los cuales también fueron sometidos a verificación satisfactoriamente, según sus afirmaciones.

Las conclusiones finales demostraron que habían conseguido una implementación y verificación satisfactoria del *driver* IEEE 802.11p y del protocolo de red IEEE 1609.3; una demostración de aplicación de detección de colisiones combinando la tecnología WAVE con la tecnología GPS; la consecución exitosa de comunicación a velocidades relativas cercanas a los 40 Km/h y a distancias mayores a 100 m; y el desarrollo de un producto comercializable de tamaño compacto.

## 2.2. *Universidade de Aveiro*

En la presentación hecha en julio del 2011 del proyecto comenzado por André Cardote, Filipe Neves, André Braga Reis y Susana Sargento pueden verse resultados más palpables. Su estudio sobre las VANET ( *Vehicular Ad-Hoc Network*) nació con la misma motivación que la del equipo anterior, desarrollar una red *ad-hoc* entre vehículos móviles y entre vehículos e infraestructura que permitiera alta movilidad a altas velocidades relativas para la implementación de aplicaciones para la seguridad vial.

Para ello desarrollaron un banco de pruebas a pequeña escala con una placa ALIX3D3 con una distribución Debian Linux *Squeeze*, una tarjeta *WiFi* bajo el estándar IEEE 802.11a/b/g, una tarjeta bajo el estándar IEEE 802.11p, un dispositivo GPS y el *driver* `ath5k` modificado. Una plataforma similar en características a la que presentaron los miembros del equipo de la *University of Moratuwa*.

Con el *driver* se consiguió coordinación entre canales, enrutado de canales, sincronización multicanal y acceso alternado y continuo a los canales. A diferencia de la *University of Moratuwa*, en este caso podemos encontrar numerosos artículos escritos por André Cardote dando cuenta de sus éxitos y su trabajo, así como documentos visuales que dan fe de los resultados. Dicha información se refleja en el apartado de referencias de este documento.

Dentro de los documentos usados como referencia se encuentra un resumen de las demostraciones realizadas en la Conferencia de Redes Vehiculares de 2011 realizada por la IEEE donde se deja claro que el protocolo IEEE 802.11p/WAVE obliga a que el dispositivo de red sea capaz de cambiar periódicamente (en espacios de 50 ms) entre el canal de control, que sólo puede ser usado para enviar mensajes de control y de seguridad críticos, y el canal de servicios, usado para todos los demás propósitos.

## 2.3. *The Ohio State University*

Aunque no es una aplicación directa del proyecto en cuestión, cabe hacer mención al trabajo de Umit Ozguner y Keith Redmill de la *Ohio State Univesity*, quienes diseñaron un banco de pruebas para una implementación de protocolo WAVE orientada a testear aspectos de conducción cooperativa, interacción autónoma y manual entre vehículos y entre vehículos e infraestructura.

Dicho banco de pruebas recibió el nombre de *Cyber-Physical Systems Cooperative Vehicle Demonstration* (CPS-CVD) y contenía:

- Un vehículo con DSRC totalmente autónomo con control sobre velocidad y dirección
- Un vehículo con DSRC longitudinalmente autónomo con control de velocidad
- Dos vehículos con DSRC conducidos por humanos
- Un semáforo verde-rojo con DSRC emitiendo el tiempo de ciclos, la geometría de las intersecciones y las posiciones de parada
- Una señal de *stop*
- Una base de datos del mapa de carretera completo para el escenario del problema

En la primera fase, única recogida en el documento del que se ha obtenido esta información, el plan de pruebas consistía en que los dos vehículos autónomos serían guiados por el vehículo conducido por humanos en el ciclo continuo que se muestra en la imagen, obedeciendo siempre los semáforos, las señales de *stop* y las reglas de precedencia en las intersecciones, acelerando y decelerando simultáneamente para mantener la distancia intervehicular y permitir que se despejen las intersecciones de la forma más suave y rápida posible.

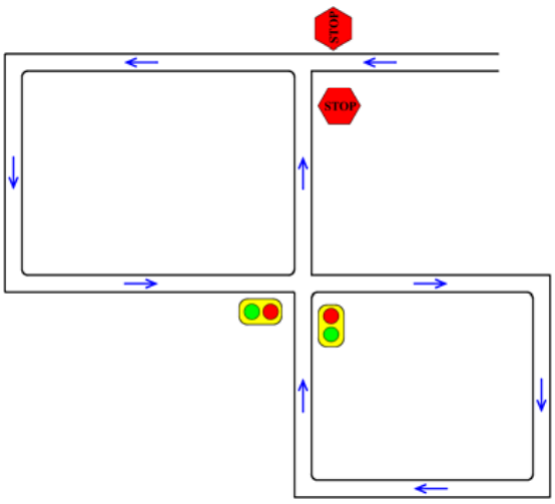


Figura 2.1: Circuito del CPS-CVD

Para este caso de pruebas se han establecido además un modelo de laboratorio con robots móviles llamado *SimVille* utilizado para verificar el algoritmo de toma de decisiones de los vehículos autónomos. Para esta implementación no se utilizó DSRC sino una red *WiFi ad-hoc*.

Con el CPS-CVD se pretendía establecer un sistema de testeo de los distintas soluciones del GCDC para la conducción cooperativa, testear los protocolos para las comunicaciones V2V y V2I, sentar las bases para futuros desarrollos de GCDC y las pruebas para usos en situaciones de tráfico denso.

## 2.4. *University of Cyprus*

Dentro del conjunto de estudios que no corresponden directamente al proyecto propuesto pero que son base para el mismo se puede incluir el realizado por la *University of Ciprus*, VIVAGr una herramienta de visualización tiempo real orientada a gráficos para la conectividad de redes vehiculares *ad-hoc*. Esta herramienta permite realizar una síntesis estructural, topológica y de las características dinámicas de una VANET.

La solución propuesta por esta universidad es una herramienta multiplataforma, portable y modular que incluye diversos módulos independientes que pueden ser usados de forma conjunta o como funciones aisladas. Se ha puesto énfasis en tres puntos fundamentales para la visualización de grafos, i.e., la interactividad, la codificación gráfica y la representación en tiempo real de los patrones de movimiento.

Esta herramienta importa ficheros de seguimiento de los patrones de movilidad a lo largo del tiempo. Cada vez que se instancia una conexión, un grafo indirecto es creado. Los nodos se representan con vértices y los enlaces con arcos entre vértices. Cada vez que la posición de un nodo cambia, una animación muestra la transición al nuevo grafo creado. Además es capaz de confeccionar y exportar estadísticas específicas para análisis futuros y monitorizar las actividades de un vehículo dentro del entorno de la red.

Esta herramienta permite obtener información sobre cómo el sistema de telecomunicaciones subyacente afecta a la topología de la red vehicular, cómo afecta el radio de penetración a la forma de la VANET, cuáles son los vehículos con mayor calidad en términos de conectividad, qué leyes gobiernan la evolución temporal de las propiedades de un grafo VANET, cómo pueden identificarse comunidades dentro de una VANET, cómo afecta la topología del mapa de carreteras a las propiedades del grafo de una VANET o cuál es la mejor estrategia de despliegue para una RSU (*Road Side Unit*) si se pretende maximizar la diseminación de la información.

## 2.5. GCDC

Un gran impulsor del estudio de esta tecnología es el *Grand Cooperative Driving Challenge* (GCDC), una propuesta de innovación competitiva de alcance mundial con el fin de potenciar el I+D sobre los sistemas de conducción

cooperativos o sistemas de conducción asistidos. Participan en él equipos privados y universidades de todo el mundo que se reúnen en competiciones anuales o bianuales a lo largo del mundo. La mayoría de equipos incluidos en este documento han participado de alguna forma en el GCDC.

En el pasado ofrecían en la web del desafío un código base que implementaba el protocolo 802.11p, sin embargo, el portal ha cambiado y ni el código, ni la documentación son accesibles ahora mismo. Dicha implementación del protocolo IEEE 802.11p nace no como el objetivo principal, sino como una herramienta necesaria de la que todos los participantes de la competición pudiesen partir a la hora de establecer un método de comunicación entre los vehículos usados por los equipos de desarrollo. Con todo, pueden encontrarse aún suficientes referencias sobre los progresos de diferentes grupos de investigación que, aunque no aportan conocimiento sobre cómo abordar el problema, al menos dan pruebas de que es posible llegar a una solución partiendo de la base de un *driver* Linux que implemente el protocolo IEEE 802.11a.

## 2.6. *OpenAirITS*

*OpenAirInterace.org* es una plataforma de código abierto con licencia GNU GLP v3 para experimentación en sistemas inalámbricos que toma como objetivo principal las tecnologías celulares (LTE/LTE-Advanced) y las redes de malla y *ad-hoc* de despliegue rápido. La plataforma consta de componentes *hardware* y *software* y puede ser usada para simulación y emulación tanto como para experimentación en tiempo real en equipamiento de radio frecuencia a medida y en equipamiento de *National Instruments* o Ettus sin problema. Consta de la pila de protocolo entera desde la capa física hasta la de red empezando por LTE (*Long-Term Evolution*) *Release* 8, un estándar para comunicaciones inalámbricas de alta velocidad para móviles y terminales de datos. El *software* actual puede interoperar con terminales comerciales LTE y puede ser interconectados con EPC de código cerrado de terceros. El objetivo de la plataforma es proveer de métodos para la validación de protocoloes, evaluaciones de rendimiento y sistemas de testeo predespliegue.

Dentro de este proyecto existía un pequeño apartado dedicado a conseguir una implementación del protocolo IEEE 802.11p llamado OpenAirITS, sin embargo esa línea de desarrollo se ha abandonado y no se mantiene. De hecho, en la última actualización del código previenen de la incertidumbre sobre el correcto funcionamiento de ese código. Es un código realizado para el kernel 3.4 y, aunque las pruebas realizadas con él no han permitido cambiar la frecuencia de emisión de la tarjeta para poder tomarlo como punto de partida en la modificación del driver `ath5k`, sí ha servido en gran medida para identificar dónde se encuentran las partes críticas que necesitan modificación.

## 2.7. Resumen

Los estudios analizados sugieren que se han conseguido avances en la adaptación de las implementaciones existentes al protocolo IEE 802.11p. Sin embargo, las implementaciones generadas no son públicas, y la información necesaria para reproducir sus resultados es escasa. En este Trabajo de Fin de Grado analizaremos el problema más en detalle, con el fin de enumerar los cambios necesarios en la implementación Linux del *driver* correspondiente. Como se describirá más adelante, para llevar a cabo alguno de estos cambios se requiere la colaboración del fabricante del *hardware* de comunicaciones.

# Capítulo 3

# Especificación de requisitos

En este proyecto se pretende realizar un análisis del *driver* `ath5k` con el objetivo de comprender su funcionamiento e identificar los parámetros que necesitan ser modificados para conseguir implementación del protocolo IEEE 802.11p desde una implementación funcional del estándar IEEE 802.11a. Se ha decidido tomar este enfoque dada la gran similitud existente entre ambos protocolos. A continuación se listan los cambios que deben realizarse para conseguirlo.

1. El estudio de las modificaciones necesarias para una implementación del protocolo de comunicaciones IEEE 802.11p deberá utilizar como punto de partida la implementación *software* disponible para el protocolo IEEE 802.11a `ath5k`

2. Se deberá proponer una modificación que permita que la tarjeta emita en las frecuencias 5,85 GHz a 5,925 GHz

3. Se deberá proponer una modificación que permita que la tarjeta emita con un ancho de canal de 10 MHz, la mitad que en el estándar IEEE 802.11a

4. Se deberá proponer una modificación que permita que la tarjeta emita con los *bitrates* en Mbits siguientes: 3; 4,5; 6; 9; 12; 18; 24; 27; la mitad que los *bitrates* del estándar IEEE 802.11a

5. Se deberá proponer una modificación que permita que la tarjeta emita con una duración de símbolo de 8 $\mu$s, el doble que en el estándar IEEE 802.11a

6. Se deberá proponer una modificación que permita que la tarjeta emita con un intervalo de guarda de 1,6 $\mu$s, el doble que en el estándar IEEE 802.11a

7. Se deberá proponer una modificación que permita que la tarjeta emita con una duración de preámbulo de 32 $\mu$s, el doble que en el estándar IEEE 802.11a

8. Se deberá proponer una modificación que permita que la tarjeta emita con una duración de señal de campo de 8 $\mu$s, el doble que en el estándar IEEE 802.11a

9. Se deberá proponer una modificación que permita que la tarjeta emita con un espaciado de subportadoras de 0,15625 MHz, la mitad que en el estándar IEEE 802.11a

10. Se deberá proponer una modificación que permita que la tarjeta emita con una duración SIFS de 32 $\mu$s, el doble que en el estándar IEEE 802.11a

# Capítulo 4

# Modelo de análisis

## 4.1. Análisis

El trabajo de fin de grado consiste en el estudio de la pila de protocolos usada en la comunicación inalámbrica vehicular. Su objetivo final es dar una propuesta para el futuro desarrollo de un *driver* que responda a las restricciones impuestas por el estándar IEEE 802.11p y resto de protocolos relacionados con las comunicaciones WAVE.

En este TFG sólo se trabajará sobre las modificaciones necesarias para conseguir una implementación del protocolo IEEE 802.11p en lo relativo a la capa física.

*Grosso modo* los cambios necesarios se reducen a ajustar parámetros físicos del *driver* y construir una base de datos de regulación propia. Por ello se tomará como modelo de desarrollo el modelo iterativo, en el que cada iteración se afrontará un requisito.

## 4.2. Planifiación

La dedicación al TFG fue de seis (6) horas diarias desde el inicio del curso el 22 de septiembre de 2014 de lunes a viernes y dos (2) horas durante sábados, domingos y festivos. La suma total de tiempo invertidos es de 1209 horas. A continuación veremos un desglose de la planificación. En algunos casos las horas efectivamente dedicadas no coincidieron con las planificadas: tras el desglose se explican los motivos.

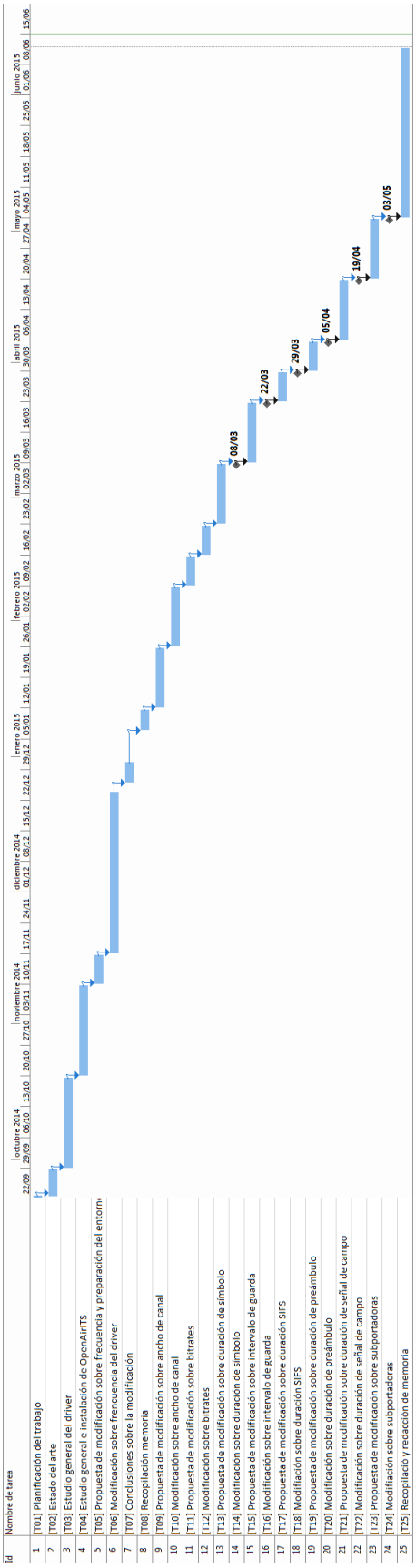


Figura 4.1: Diagrama de Gantt sobre dedicación del proyecto

T1 **Planificación del trabajo**

- **Tiempo planificado:** Tres horas
- **Tiempo dedicado:** Tres horas
- **Fecha inicio:** 22/09/2014
- **Fecha fin:** 22/09/2014
- **Entregable:** Documento de planificación

T2 **Estado del arte**

- **Tiempo planificado:** 31 horas
- **Tiempo dedicado:** 31 horas
- **Fecha inicio:** 22/09/2014
- **Fecha fin:** 28/09/2014

T3 **Estudio general del *driver***

- **Tiempo planificado:** 68 horas
- **Tiempo dedicado:** 102 horas
- **Fecha inicio:** 29/09/2014
- **Fecha fin:** 19/10/2014

T4 **Estudio general e instalación de OpenAirITS**

- **Tiempo planificado:** 136 horas
- **Tiempo dedicado:** 102 horas
- **Fecha inicio:** 20/10/2014
- **Fecha fin:** 09/11/2014

T5 **Propuesta de modificación sobre frecuencia y preparación del entorno**

- **Tiempo planificado:** 34 horas
- **Tiempo dedicado:** 34 horas
- **Fecha inicio:** 10/11/2014
- **Fecha fin:** 16/11/2014

T6 **Modificación sobre frecuencia del *driver***

- **Tiempo planificado:** 102 horas
- **Tiempo dedicado:** 182 horas
- **Fecha inicio:** 17/11/2014
- **Fecha fin:** 23/12/2014

T7 **Conclusiones sobre la modificación**

- **Tiempo planificado:** Diez horas
- **Tiempo dedicado:** Diez horas
- **Fecha inicio:** 26/12/2014
- **Fecha fin:** 30/12/2014

T8 **Recopilación memoria**

- **Tiempo planificado:** Diez horas
- **Tiempo dedicado:** Diez horas
- **Fecha inicio:** 07/01/2015
- **Fecha fin:** 11/01/2015

T9 **Propuesta de modificación sobre ancho de canal**

- **Tiempo planificado:** 34 horas
- **Tiempo dedicado:** 68 horas
- **Fecha inicio:** 12/01/2015
- **Fecha fin:** 25/01/2015

T10 **Modificación sobre ancho de canal**

- **Tiempo planificado:** 68 horas
- **Tiempo dedicado:** 68 horas
- **Fecha inicio:** 26/01/2015
- **Fecha fin:** 08/02/2015

T11 **Propuesta de modificación sobre *bitrates***

- **Tiempo planificado:** 34 horas
- **Tiempo dedicado:** 34 horas
- **Fecha inicio:** 09/02/2015
- **Fecha fin:** 15/02/2015

T12 **Modificación sobre *bitrates***

- **Tiempo planificado:** 34 horas
- **Tiempo dedicado:** 34 horas
- **Fecha inicio:** 16/02/2015
- **Fecha fin:** 22/02/2015

T13 **Propuesta de modificación sobre duración de símbolo**

- **Tiempo planificado:** 34 horas

- **Tiempo dedicado:** 68 horas
- **Fecha inicio:** 23/02/2015
- **Fecha fin:** 08/03/2015

T14 **Modificación sobre duración de símbolo**

- **Tiempo planificado:** 34 horas
- **Tiempo dedicado:** Cero horas
- **Fecha inicio:** 08/03/2015
- **Fecha fin:** 08/03/2015

T15 **Propuesta de modificación sobre intervalo de guarda**

- **Tiempo planificado:** 34 horas
- **Tiempo dedicado:** 68 horas
- **Fecha inicio:** 09/03/2015
- **Fecha fin:** 22/03/2015

T16 **Modificación sobre intervalo de guarda**

- **Tiempo planificado:** 34 horas
- **Tiempo dedicado:** Cero horas
- **Fecha inicio:** 22/03/2015
- **Fecha fin:** 22/03/2015

T17 **Propuesta de modificación sobre duración SIFS**

- **Tiempo planificado:** 34 horas
- **Tiempo dedicado:** 34 horas
- **Fecha inicio:** 23/03/2015
- **Fecha fin:** 29/03/2015

T18 **Modificación sobre duración SIFS**

- **Tiempo planificado:** 34 horas
- **Tiempo dedicado:** Cero horas
- **Fecha inicio:** 29/03/2015
- **Fecha fin:** 29/03/2015

T19 **Propuesta de modificación sobre duración de preámbulo**

- **Tiempo planificado:** 34 horas
- **Tiempo dedicado:** 34 horas
- **Fecha inicio:** 30/03/2015

- **Fecha fin:** 05/04/2015

T20 **Modificación sobre duración de preámbulo**

- **Tiempo planificado:** 34 horas
- **Tiempo dedicado:** Cero horas
- **Fecha inicio:** 06/04/2015
- **Fecha fin:** 06/04/2015

T21 **Propuesta de modificación sobre duración de señal de campo**

- **Tiempo planificado:** 34 horas
- **Tiempo dedicado:** 68 horas
- **Fecha inicio:** 06/04/2015
- **Fecha fin:** 19/04/2015

T22 **Modificación sobre duración de señal de campo**

- **Tiempo planificado:** 34 horas
- **Tiempo dedicado:** Cero horas
- **Fecha inicio:** 20/04/2015
- **Fecha fin:** 20/04/2015

T23 **Propuesta de modificación sobre subportadoras**

- **Tiempo planificado:** 34 horas
- **Tiempo dedicado:** 68 horas
- **Fecha inicio:** 20/04/2015
- **Fecha fin:** 03/05/2015

T24 **Modificación sobre subportadoras**

- **Tiempo planificado:** 34 horas
- **Tiempo dedicado:** Cero horas
- **Fecha inicio:** 03/05/2015
- **Fecha fin:** 03/05/2015

T25 **Recopilación y redacción de memoria**

- **Tiempo planificado:** 191 horas
- **Tiempo dedicado:** 191 horas
- **Fecha inicio:** 04/05/2015
- **Fecha fin:** 11/06/2015

Como puede observarse en el listado anterior existen discrepancias entre la dedicación planificada y la realizada realmente el proyecto. En el caso de las tareas T3 y T4 se debe a que el estudio estático de *driver* `ath5k` requirió más esfuerzo del pensado en un primer momento, pero, en consecuencia, el conocimiento adquirido en esa etapa agilizó la siguiente tarea de instalación de OAI en gran medida, lográndose que en conjunto ambas tareas se consiguiesen realizar en el mismo tiempo.

Aunque se tratará en capítulos posteriores, la tarea T6 es una tarea crítica del proyecto y, por ello, se convirtió un cuello de botella dentro del mismo. Por esa razón se decidió añadir una dedicación de ochenta (80) horas adicionales, aproximadamente dos semanas y media según la dedicación semanal definida. Por la misma razón se amplió el tiempo de dedicación de treinta y cuatro (34) horas a sesenta y ocho (68) en el caso de la tarea T9 encargada del dar una propuesta para la modificación del ancho de canal, parámetro íntimamente relacionado con la frecuencia de emisión.

En el caso de las tareas T13 (Propuesta de modificación sobre duración de símbolo), T14 (Modificación sobre duración de símbolo), T17 (Propuesta de modificación sobre duración SIFS), T18 (Modificación sobre intervalo de guarda), T19 (Propuesta de modificación sobre duración SIFS) y T20 (Modificación sobre intervalo de guarda), aunque se prolongó el estudio sobre la modificación de los parámetros, se descubrió que dichos parámetros no necesitan modificación, por tanto las tareas de modificación tuvieron dedicación cero.

Distinto es el caso de las tareas T15 (Propuesta de modificación sobre intervalo de guarda), T16 (Modificación sobre intervalo de guarda), T21 (Propuesta de modificación sobre duración de señal de campo), T22 (Modificación sobre duración de señal de campo), T23 (Propuesta de modificación sobre subportadoras) y T24 (Modificación sobre subportadoras), donde dado que no se encontró en el código referencias a esos parámetros para modificarlos se decidió ampliar la dedicación de las tareas de Propuesta para obtener la mayor cantidad de información posible al respecto para trabajos futuros.

## 4.3. Presupuesto

Para este proyecto los gastos totales ascenderían a 18875,54 €, desglosados de la siguiente manera.

- **Horas de trabajo (15 €/hora):** 18135,00 €
- **Dos ordenadores de sobremesa con Linux:** 400,00 €
- **Dos juegos de teclado y ratón:** 48,00 €
- **Dos pantallas:** 200,00 €
- **Dos tarjetas *WiFi* Mini PCI MikroTik R52H:** 92,54 €

# Capítulo 5

# Descripción de la pila WAVE

Las tecnologías WAVE están sustentadas por un grupo de estándares relacionados que cubren todas las capas de protocolos DSRC. Estos protocolos son el IEEE 802.11p, encargado de la capa física y de enlace, y una serie de protocolos IEEE 1609, encargados de la capa de enlace, en colaboración con IEEE 802.11p, y las capas superiores hasta la de aplicación.

Es objetivo del protocolo IEEE 802.11p definir las condiciones físicas de emisión y los requisitos de enlace. En la capa física se define la banda de emisión utilizada, los *bitrates* usados, tiempos de préambulo y duración de símbolo necesarios, entre otros parámetros. En la capa de enlace se define cómo han de llevarse a cabo las comunicaciones de principio a fin para lograr transmisiones de datos efectivas a velocidades relativas altas, lo que implica reducir al mínimo los retardos derivados de los establecimientos de comunicación.

El objetivo de los protocolos definidos en el estándar IEEE 1609 es variado. En la capa de enlace, junto a IEEE 802.11p, IEEE 1609.4 coordina los cambios de canal entre el canal de control y el canal de servicio, las colas de servicios,los paquetes IPv6 y WSMP (*Wave Short Message Protocol*), encolan los paquetes teniendo en cuenta la prioridad. El estándar IEEE 1609.3 se encarga de los servicios de transporte de red, del direccionado y enrutado, de las tareas de configuración y administración de conexiones. El estándar IEEE 1609.1 se encarga de la capa de aplicación. Y, por último, transversal a todas las capas, la seguridad es maneja con el estándar IEEE 1609.2.

Dada las similitudes que existe entre IEEE 802.11a y IEEE 802.11p en este capítulo se especificarán cuáles son las necesidades concretas de cambio que existe para conseguir una implementación del estándar en la capa física y de enlace.

## 5.1. WAVE e IEEE 802.11p

El proceso de estandarización nace al reservarse en los Estados Unidos un espectro de banda para las DSRC y del esfuerzo para definir tecnologías que usen dichas bandas.

En 1999, la *U.S. Federal Communication Commission* reservó una banda de 75 MHz de DSRC en el espectro de los 5,9 GHz para ser usados exclusivamente

en comunicaciones vehículo a vehículo e infraestructura a vehículo.

En 2004 el esfuerzo por llegar a esta estandarización pasó a manos del grupo de estandarización de IEEE 802.11, dado que las comunicaciones DSRC son esencialmente comunicaciones IEEE 802.11a modificadas para reducir la carga en las operaciones dentro del espectro DSRC. Dentro del estándar IEEE 802.11, DSRC es conocido como IEEE 802.11p WAVE.

Pero este estándar no es autónomo, sino que es parte de un grupo de estándares relacionados con todas las capas de protocolos para las operaciones DSRC. El estándar IEEE 802.11p está limitado al alcance que tiene IEEE 802.11, i.e. está limitado a las capas MAC (capa de enlace) y PHY (capa física).

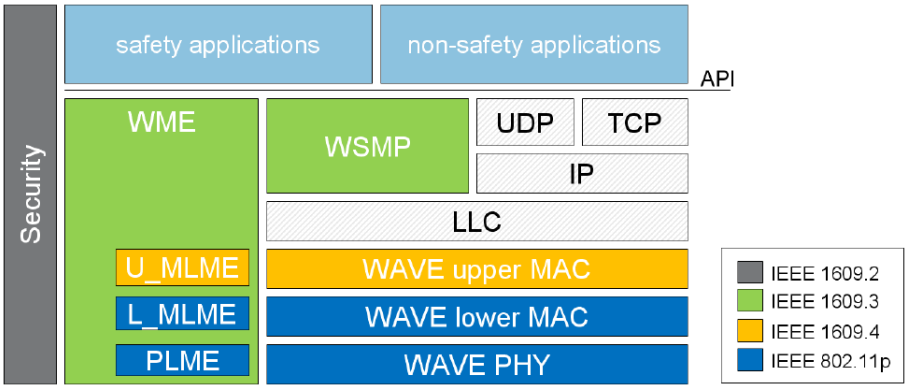


Figura 5.1: Pila de protocolos WAVE

Las funciones de este estándar son:

- Describir las funciones y servicios requeridos por estaciones WAVE para operar en un entorno en constante cambio e intercambiar mensajes sin necesidad de unirse a una BSS, como ocurre en el caso de uso de IEEE 802.11 tradicional.
- Definir las técnicas de señal y funciones de interfaz para WAVE que son controladas por la capa MAC de IEEE 802.11.

Toda la complejidad y conocimiento relacionado con la planificación de canales y conceptos operacionales están cargo de las capas superiores manejadas por el estándar IEEE 1609.

En la capa de enlace, conviviendo con IEEE 802.11p, se encuentra el estándar IEEE 1609.4 que habilita las operaciones de las capas superiores a través de los múltiples canales sin necesidad de conocer los parámetros PHY. Entre los dos coordinan los cambios de canal entre el canal de control y el canal de servicio, coordinan la cola de servicios, manejan los paquetes IPv6 y WSMP (*Wave Short Message Protocol*), encolan los paquetes para la trasmisión por el canal correcto y administran la prioridad de los mensajes usando el mecanismo EDCA.

La capa de red es manejada por el estándar IEEE 1609.3 que se encarga de los servicios de transporte de red, del direccionado y enrutado de datos

para el intercambio de datos entre entidades WAVE, así como de las tareas de configuración y administración de las conexiones WAVE. En esta capa donde se encuentra el WME (*WAVE Management Entity*), responsable de la publicación de los servicios. Según este estándar, el plano de datos, la parte de la red que soporta tráfico del usuario, implementa el protocolo IPv6 y WSMP.

La capa de aplicación es manejada por el estándar IEEE 1609.1. Gestiona recursos e interactúa con un procesador de comandos en el OBU. La aplicación de gestión de recursos puede estar en la misma máquina o distribuida a través de la red.

Transversal a todas las capas se encuentra la capa de seguridad construida bajo el estándar IEEE 1609.2. Contiene servicios de seguridad para los mensajes de aplicación y administración.

## 5.2. IEEE 802.11p capa MAC

Bajo una perspectiva simplificada la capa MAC de IEEE 802.11 se encarga de administrar conjuntos de emisiones para establecer y mantener grupos cooperativos. Los miembros de una red inalámbrica pueden comunicarse libremente entre ellos, pero cualquier trasmisión del exterior es ignorada. Una BSS (*Infrastructural Basic Service Set*) es un ejemplo de ello y existen multitud de protocolos destinados a proveer de comunicaciones seguras y robustas dentro de una BSS.

Una BSS es un grupo de estaciones IEEE 802.11 ancladas a un AP (*Access Point*) y configuradas para comunicarse entre ellas por el aire. Los mecanismos de control de la BSS permiten el control de acceso a los recursos y servicios de un AP y permiten a los miembros de la red ignorar las trasmisiones de redes circundantes. Una red primero escucha las *beacons* o balizas de un AP y luego se une a la BSS tras un número de pasos interactivos que incluye autenticación y asociación.

Las redes *ad-hoc* podrían parecer una solución viable para establecer comunicaciones intervehiculares dado que no dependen de una infraestructura pre-existente, como *routers* en redes cableadas o puntos de accesos en redes inalámbricas administradas. En lugar de ello, cada nodo participa en el encaminamiento mediante el reenvío de datos hacia otros nodos, de modo que la determinación de estos nodos hacia la información se hace dinámicamente sobre la base de conectividad de la red. En adición al encaminamiento clásico, las redes *ad-hoc* pueden usar un *flooding* o inundación de red para el reenvío de datos.

Una red *ad-hoc* se refiere típicamente a cualquier conjunto de redes donde todos los nodos tienen el mismo estado dentro de la red y son libres de asociarse con cualquier otro dispositivo de red *ad-hoc* en el rango de enlace. Las redes *ad-hoc* se refieren generalmente a un modo de operación de las redes inalámbricas IEEE 802.11. El modo de operación *ad-hoc* definido para IEEE 802.11 también sigue un proceso de establecimiento interactivo similar al de las BSS llamado IBSS (*Independet Basic Service Set*), pero sigue aquejado de

demasiada complejidad y carga adicional para ser adecuado para las comunicaciones intervehiculares.

Una BSS se da a conocer por su identificación SSID (*Service Set Identification*), un campo de información de entre 0 y 32 *bytes* que corresponde al nombre del punto de acceso *WiFi* que normalmente ve la gente en sus dispositivos y a los que se conecta.

No debe confundirse con BSSID (*Basic Service Set Identification*), que es el nombre por el que se conoce a una BSS entre las redes a nivel MAC. Al igual que la dirección MAC, es un número de 48 *bits*. Cada BSS debe tener un BSSID único que se comparta entre todos sus miembros. Para asegurar ésto se usa la dirección MAC del AP.

En una IBSS se usa una dirección MAC administrada localmente con el bit individual/grupo a 0, el bit universal/local a 1 y los 46 restantes aleatorios.

El filtrado BSSID es la clave del mecanismo para restringir los paquetes entrantes a sólo aquellos recibidos de miembros de la misma BSS. Como se muestra en la siguiente imagen cada paquete IEEE 802.11 incluye hasta cuatro campos de dirección. Dichos campos pueden contener la dirección fuente o SA (*Source Address*), la dirección destino o DA (*Destination Address*), la dirección estación de envío o TA (*Transmiting STA Address*), dirección estación de recepción o RA (*Receiving STA Address*), o el BSSID. El uso dado a cada campo difiere según el valor de los bits "a DS" o "desde DS" en función de la siguiente tabla.

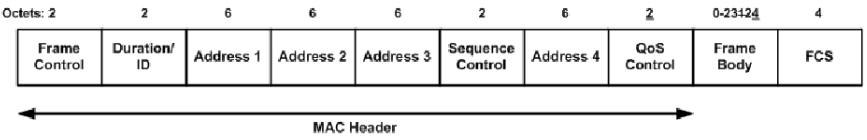


Figura 5.2: Formato de paquete 802.11

| A DS | Desde DS | Dir 1 | Dir 2 | Dir 3 | Dir 4 |
|---|---|---|---|---|---|
| 0 | 0 | RA = DA | TA = SA | BSSID | N/A |
| 0 | 1 | RA = DA | TA = BSSID | SA | N/A |
| 1 | 0 | RA = BSSID | TA = SA | DA | N/A |
| 1 | 1 | RA | TA | DA | SA |

Tabla 5.1: Relación entre el contenido de los campos de dirección y el valor de los bit "a DS" y "desde DS"

El nivel MAC de la estación, al recibir un paquete de la capa PHY, usa el contenido de la dirección 1 para realizar la comprobación de concordancia con las reglas de recepción. Si la dirección 1 contiene una dirección de grupo (e.g., una dirección de difusión o *broadcast*), el BSSID es comparado para asegurar que el *broadcast* o *multicast* se ha originado dentro de la misma BSS.

Cabe hacer especial mención al caso de la BSSID denominada *wildcard* BSSID, la cual está compuesta exclusivamente por unos. El estándar IEEE

802.11 actual limita el uso de este *wildcard* sólo a paquetes de administración del subtipo de peticiones de sondeo.

El problema que presentan estas operaciones dentro de entornos vehiculares es que consumen mucho tiempo. Las comunicaciones vehiculares orientadas a la seguridad vial exigen intercambios de datos casi instantáneos y no pueden permitirse perder tiempo escaneando canales hasta encontrar una baliza de BSS y realizar la serie de pasos necesarios para el establecimiento de una comunicación.

Por ésto es esencial que todos los dispositivos de emisión de redes IEEE 802.11p pertenezcan por defecto al mismo canal y estén configurados con el mismo BSSID para habilitar comunicaciones seguras.

Un cambio clave introducido por IEEE 802.11p WAVE es el término "modo WAVE". Una estación en modo WAVE puede transmitir y recibir paquetes de datos con la *wildcard* BSSID sin necesidad de pertenecer a ninguna BSS de ningún tipo. Ésto permite que dos vehículos puedan comunicarse entre ellos inmediatamente al producirse el encuentro.

El estándar WAVE añade un nuevo tipo de BSS, el WBSS (*WAVE BSS*). Una estación crea una WBSS transmitiendo primero una baliza bajo demanda o *on demand beacon*. Una estación WAVE usa la baliza de demanada, que es un paquete de baliza bien conocido que no necesita ser repetido periódicamente, para anunciar una WBSS. Contiene todo la información necesaria para que la estaciones receptoras entiendan los servicios ofrecidos en la WBSS, es decir, la sola recepción del aviso permite a la estación decidir si unirse o no sin necesidad de más interacción.

Esta aproximación permite reducir enormemente los costes de una configuración WBSS eliminando todo proceso de asociación y autenticación. Se necesitan mecanismos adicionales para administrar el uso de grupos WBSS, así como la seguridad, pero éstos quedan fuera del ámbito de IEEE 802.11 y, por tanto, de IEEE 802.11p.

El DS también está disponible en los dispositivos WAVE. A través del aire ésto significa que es posible transmitir paquetes de datos con los bits "a DS" y "desde DS" a 1. De todas formas, la capacidad de un dispositivo de emisión en una WBSS de enviar y recibir paquetes con la *wildcard* BSSID introduce complicaciones. Es probable que un miembro de la red se vea restringido a enviar paquetes de datos con la *wildcard* BSSID sólo si "a DS" y "desde DS" están a 0. En otras palabras, los dispositivos de emisión que se estén comunicando en el contexto de una WBSS necesitan enviar paquetes usando una BSSID conocida para acceder a una DS.

En resumen los cambios necesarios dentro de la capa MAC son los siguientes:

- Una estación en modo WAVE puede enviar y recibir paquetes de datos con la *wildcard* BSSID si los bits "a DS" y "desde DS" están a 0, sin importar si es miembro o no de una WBSS.

- Una WBSS es un tipo de BSS que consiste en un grupo de estaciones en cooperación funcionando en modo WAVE que se comunican usando una BSSID común. Una WBSS se inicializa cuando un dispositivo de emisión

en modo WAVE envía una baliza WAVE que incluye toda la información para que un emisor se una.

- Un dispositivo de emisión se une a una WBSS cuando está configurado para enviar y recibir paquetes de datos con la BSSID definida por la WBSS en cuestión. Y deja de pertenecer a la WBSS cuando su MAC deja de enviar y recibir paquetes que usan la BSSID de esa WBSS.

- Una estación no puede ser miembro de más de una WBSS al mismo tiempo. Una estación WAVE no puede unirse a una BSS o a una IBSS, ni puede usar escaneo activo o pasivo y, por último, no puede usar procedimientos de autenticación ni asociación MAC.

- Una WBSS deja de existir cuando no tiene miembros. El dispositivo que la inicia no se diferencia de ninguna forma con el resto de miembros. Por eso puede ser que una WBSS seguirá existiendo aún cuando su originador desaparezca.

## 5.3. IEEE 802.11p capa PHY

La filosofía dentro de 802.11p en lo relativo a los cambios en la capa física pasa por realizar el mínimo número de cambios posibles. Esta aproximación es factible desde el momento en que IEEE 802.11a ya opera a 5 GHz y es razonablemente sencillo configurar los dispositivos emisores para operar en la banda de los 5,9 GHz en los Estados Unidos y bandas internacionales similares según lo reflejado en los ejemplos de éxitos encontrados. También es deseable y razonable por el desafío técnico que implicaría realizar un cambio radical en el diseño de la capa PHY inalámbrica. Mientras que las modificaciones en la capa MAC son principalmente *software*, las de la capa PHY necesitan ser limitadas para evitar tener que diseñar tecnologías inalámbricas nuevas.

El estándar IEEE 802.11p está esencialmente basado en el OFDM (*Orthogonal Frequency Division Multiplexing*) definido para 802.11a, con una amplitud de canal de 10 MHz en lugar de la de 20 MHz usada en los dispositivos IEEE 802.11a. El estándar IEEE 802.11 ya define canales de amplitud de 10 MHz y su implementación es directa dado que prácticamente lo único que se necesita hacer es doblar los parámetros temporales OFMD usados en las transmisiones regulares IEEE 802.11a con amplitud de 20 MHz.

La razón principal para este escalado desde IEEE 802.11a es abordar el problema del aumento de la media cuadrática de los retardos de propagación en entornos vehiculares. El intervalo de guarda para 20 MHz no es lo suficientemente largo para compensar el peor caso de la media cuadrática del retardo de propagación, i.e., no es suficientemente largo para evitar interferencias entre símbolos dentro de las propias trasmisiones de un dispositivo emisor en un entorno vehicular.

Mientras que existen dentro de los Estados Unidos e internacionalmente una serie de canales disponibles para el desarrollo y uso de IEEE 802.11p, la

naturaleza de las comunicaciones vehiculares, que implica vehículos distribuidos en la carretera, hace que aumente la preocupación sobre las interferencias entre canales. Este tipo de interferencias son ya bien conocidas y pertenecen a propiedades físicas naturales de las comunicaciones inalámbricas. La solución más efectiva a este problema pasa por la aplicación de políticas de administración de canales que caen fuera del alcance de IEEE 802.11. Aún así IEEE 802.11p introduce algunos requerimientos en el rendimiento de receptores en lo que respecta al rechazo de canales adyacentes.

En el uso de dispositivos de emisión IEEE 802.11p en la banda ITS se definen cuatro máscaras de espectro referidas a las clases de operación de la A a la D.

- Para las operaciones de **clase A** usando un espaciado de canales de 10 MHz, el espectro de transmisión debería tener un ancho de banda de 0 dBr de potencia que no exceda los 9 MHz. Adicionalmente, su potencia no debería exceder los -10 dBr con un *offset* de 5 MHz, ni los -20 dBr con un *offset* de a 5,5 MHz, ni los -28 dBr con un *offset* de 10 MHz, ni los -40 dBr con un *offset* de 15 MHz o superior.

- Para las operaciones de **clase B** usando un espaciado de canales de 10 MHz, el espectro de transmisión debería tener un ancho de banda de 0 dBr de potencia que no exceda los 9 MHz. Adicionalmente, su potencia no debería exceder los -16 dBr con un *offset* de 5 MHz, ni los -20 dBr con un *offset* de 5,5 MHz, ni los -28 dBr con un *offset* de 10 MHz, ni los -40 dBr con un *offset* de 15 MHz o superior.

- Para las operaciones de **clase C** usando un espaciado de canales de 10 MHz, el espectro de transmisión debería tener un ancho de banda de 0 dBr de potencia que no exceda los 9 MHz. Adicionalmente, su potencia no debería exceder los -26 dBr con un *offset* de 5 MHz, ni los -32 dBr con un *offset* de 5,5 MHz, ni los -40 dBr con un *offset* de 10 MHz, ni los -50 dBr con un *offset* de 15 MHz o superior.

- Para las operaciones de **clase D** usando un espaciado de canales de 10 MHz, el espectro de transmisión debería tener un ancho de banda de 0 dBr de potencia que no exceda los 9 MHz. Adicionalmente, su potencia no debería exceder los -35 dBr con un *offset* de 5 MHz, ni los -45 dBr con un *offset* de 5,5 MHz, ni los -55 dBr con un *offset* de 10 MHz, ni los -65 dBr a 15 MHz de compensación de frecuencia o superior.

# Capítulo 6

# Estudio del *driver* `ath5k`

El estudio se ha realizado con una máquina con Slackware 14.1 y el *kernel* linux 3.10.0. Las tarjetas usadas en el TFG son dos tarjetas Mini PCI MikroTik R52H con *chipset* AR5413 (*Eagle lite*)/AR5414 (*Eagle*), por tanto el *driver* a estudiar y modificar de los contenidos en el *kernel* es el `ath5k`.

Dicho *driver* tiene una licencia *Dual BSD/GPL* y fue desarrollado Nick Kossifidis y Jiri Slaby para dar soporte a las tarjetas de red Atheros de la serie 5000. Su funcionamiento depende de los módulos `mac80211`, `cfg80211` y `ath`, módulos que en el futuro deberán ser modificados también para conseguir una implementación completa del protocolo IEEE 802.11p, según lo que puede verse al comparar el código *vanilla* del *kernel* con la versión modificada de OpenAirITS

En el siguiente grafo puede verse más claramente la relación de dependencia entre los módulos.

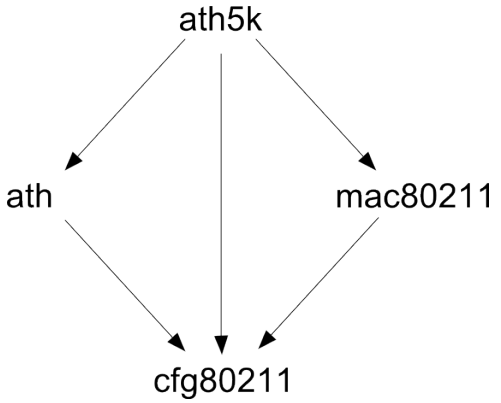


Figura 6.1: Diagrama de dependencias entre los módulos problema

A continuación se listan los ficheros incluidos en la carpeta `ath5k` que conforman el código fuente del módulo.

- `ahb.c`: Código para el control de la comunicación AHB (*Advanced High-performance Bus*).
- `ani.c` y `ani.h`: Código para el control de la inmunidad de ruido adaptativa o ANI (*Adaptative Noise Immunity*). Controla los parámetros de

inmunidad de ruido dependiendo del nivel de interferencias en el ambiente.

- `ath5k.h`: Cabecera del módulo con definiciones de constantes del *driver*, definiciones genéricas del *chipset*, definiciones de transmisión, etc. Mención especial merecen las definiciones de modos de operación o la definición de los modos de antena.
- `attach.c`: Código del módulo para la conexión con la tarjeta.
- `base.c` y `base.h`: Código principal del módulo que será explicado con más detenimiento en apartados posteriores.
- `caps.c`: Código encargado principalmente de completar las estructuras de las capacidades potenciales o *capabilities* de la tarjeta. Ésto es, rangos de emisión para cada una de sus bandas y modos de emisión soportados.
- `debug.c` y `debug.h`: Código encargado de las operaciones de *debugging*.
- `desc.c` y `desc.h`: Código encargado de manejar el procesado de los descriptores *hardware* de bajo nivel que leen y escriben vía DMA (*Direct Memory Access*) para cada intento de transmisión y recepción.
- `dma.c`: Código encargado de configurar los punteros a descriptores de comienzo y parada de las comunicaciones DMA, manejar la configuración de colas para el *chipset* 5210 (el resto se maneja en `qcu.c`), configurar el registro de máscara de interrupciones y leer los diversos registros de estado de interrupción.
- `eeprom.c` y `eeprom.h`: Funciones de lectura y escritura de la memoria EEPROM de la tarjeta.
- `gpio.c`: Funciones encargadas de las operaciones entradas y salida de propósito general.
- `initvals.c`: Funciones encargadas de cargar en las estructuras correspondientes los valores iniciales para cada uno de los *chipsets* a los que da soporte el *driver*.
- `led.c`: Funciones encargadas de controlar los LEDs de la tarjeta.
- `mac80211-ops.c`: Código que contiene las operaciones MAC como añadir/remover interfaz de red o configurar antenas.
- `pci.c`: Funciones para servicios bus PCI.
- `pcu.c`: Funciones de la unidad de control de protocolo o PCU (*Protocol Control Unit*), que es la encargada de mantener varias propiedades del protocolo antes de que un paquete sea enviado o recibido hacia o desde una banda base. Más específicamente, se encarga de tareas tales como el control de los *buffers* de paquetes de entrada y salida, encriptado

y desencriptado, generación de ACKS, mantenimiento de TSF (*Timing Synchronization Function*), el control de FCS (*Frame Check Sequence*), el actualizado de los datos de las balizas, el filtrado de recepción de paquetes, el filtrado BSSID y el control de los diferentes modos de operación.

- `phy.c`: Código donde se manejan las funciones de bajo nivel relacionadas con las bandas base y las partes análogas del *frontend*. Contiene las partes más complejas de código que actúa sobre el *hardware* como el fijado y cambio de canales, la calibración automática de la ganancia, la calibración del umbral de ruido, la inicialización en capa PHY de los dispositivos para los varios modos de operación y modos de ancho de banda, el control de antena y el control de potencia de transmisión por tipo de canal, *rate* o paquete. Es código candidato a ser modificado para conseguir una implementación completa del protocolo IEEE 802.11p.

- `qcu.c`: Funciones de la unidad de control de colas o QCU (*Queue Control Unit*). Es aquí donde se fijan los parámetros para las doce colas de transmisión disponibles.

- `reg.h`: Cabecera que contiene los valores de registro para Atheros 5210/-5211/5212 del *driver* `ar5k` de OpenBSD. También contiene valores de registro encontrados en un volcado de memoria de un programa de testeo de radio Atheros o ART (*Atheros Radio Test*) para `ath9k` liberado por Atheros.

- `reset.c`: Funciones y ayudas de reseteo. Aquí se implementan las rutinas de reseteo principal usadas para hacer funcionar la tarjeta y conseguir que esté preparada para la recepción de paquetes. También se manejan rutinas que no encajan en otro sitio como el reloj, los estados de reposo (*sleep*) y el control de encendido.

- `rfbuffer.h`: Cabecera que contiene los registros de *buffer* de radio frecuencia o RF (*Radio Frequency*), registros especiales que controlan varias configuraciones de operación relacionadas en su mayor parte con canales, ajustes de ganancias, etc.

- `rfgain.h`: Cabecera que contiene una tabla de ganancias de RF para RF5111/5112 por motivos de optimización.

- `rfkill.c`: Código que da soporte a `RFKILL` para `ath5k`. `RFKILL` es una herramienta de consulta del estado de los interruptores, botones e interfaces de subsistemas RF.

- `sysfs.c`: Código relacionado con el almacenamiento de información en algún tipo de sistema de archivos propio.

- `trace.h`: Cabecera necesaria para operaciones de trazado.

## 6.1. Resumen de base.c

El fichero base.c contiene la parte más relevante del flujo de ejecución del *driver*. Al no haberse encontrado documentación sobre la estructura y funcionamiento del código se decidió realizar un estudio estático sobre él. En dicho estudio, hecho sobre el código en crudo, destacó la existencia de las funciones *ath5k_setup_channels*, *ath5k_setup_bands*, *ath5k_init_ah* y *ath5k_init*.

- ath5k_setup_channels: Esta función es la encargada de configurar los canales en los que la tarjeta es capaz de emitir para cada uno de los modos físicos de emisión de los que dispone y para los que está configurada. Lo primero que se hace es comprobar el modo en el que se ha iniciado la tarjeta, el cual tiene que coincidir con AR5K_MODE_11A, AR5K_MODE_11B o AR5K_MODE_11G, constantes definidas en el fichero ath5k.h que hacen referencia a los modos de operación de los que dispone la tarjeta. Dentro de una instrucción switch sobre el modo físico de operación se establece el número máximo de canales para dicho modo, a partir de ahora tamaño, y el ancho de banda para cada caso.

  Una vez establecido el tamaño, un bucle for inicializa el canal, ch, a 1 y aumenta su valor en 1 en cada vuelta del bucle. La salida de bucle está regida por las variables de control tamaño y el parámetro de entrada max. Si se llega a un número de canales mayor al contenido en tamaño o el contador, variable que se explicará más adelante, llega al valor contenido en max, el bucle acaba. En cada vuelta se obtiene la frecuencia asociada al canal mediante una llamada a ieee80211_channel_to_frequency(ch, band). Entonces, se evalúa el valor de la frecuencia, si no es cero, la carga en un *array* de canales, cuyo puntero ha sido pasado por parámetro junto con su frecuencia central, su banda, y su valor *hardware*. Se comprueba si ese canal definido es soportado por el *chipset* y si es estándar. Si se cumplen ambas condiciones aumenta el contador usado como variable de control del bucle. En caso contrario se continúa sin aumentar el contador del *array* de canales, por lo que el canal cargado anteriormente acabará siendo sobrescrito. Al finalizar se devuelve el contador, indicando cuantos canales válidos existen. Como puede observarse por su uso, dicho contador representa el número de canales válidos cargados hasta el momento en cada vuelta del bucle.

- ath5k_setup_bands: Esta función se encarga de preparar las estructuras de datos que representan las bandas y crea la lista de canales disponibles atendiendo a los modos de operación soportados por el *hardware*. Las capas IEEE 802.11 son las responsables filtrar esta lista basándose en parámetros del modo PHY y en las restricciones regulatorias de dominio.

  Inicialmente, se hace apuntar la variable sband a la dirección de memoria de la estructura de la banda de los 2 GHz y se le asigna a su campo band el valor IEEE80211_BAND_2GHZ y a bitrates se lo hace apuntar a &ah->rates[IEEE80211_BAND_2GHZ][0].

Posteriormente, se evalúa si el modo G es soportado por el *hardware*, según la información contenida en `ah->ah_capabilities.cap_mode`, y si lo es se procede a cargar en `sband->bitrates` los primeros doce valores de *bitrates* contenidos en la estructura constante `ath5k_rates`. Al campo `sband->channels` se lo hace apuntar a `ah->channels` y se hace una llamada a `ath5k_setup_channels(ah, sband->channels, AR5K_MODE_11G, max_c)`, función cuyo funcionamiento ya ha sido explicado. A continuación se fija el campo `hw->wiphy->bands[IEEE80211_BAND_2GHZ]` a `sband`, para que los cambios se conserven, dado que `sband` seguirá siendo utilizada como variable intermedia para el resto de modos de operación.

Si en la evaluación sobre el soporte de la tarjeta del modo G el resultado fuera `FALSE`, se procedería a evaluar directamente si la tarjeta soporta el modo B. El comportamiento para este caso es análogo, con la salvedad de un ajuste específico en los *bitrates* para las tarjetas Atheros 5211. Dado que no se han usado estas tarjetas en el desarrollo de este TFG, no merece la pena ahondar más en el tema.

La función continúa realizando la comprobación de soporte del modo A, perteneciente a la banda de los 5 GHz, y, en caso afirmativo, la posterior carga de *bitrates* y canales independientemente de si la tarjeta soporta o no los modos B y G.

- `ath5k_init_ah`: Esta función se encarga de comenzar la inicialización de la información privada del *driver* sobre el *hardware*, sus modos de operación y sus modos de interfaz, entre otras cosas. Se inicializa el dispositivo físico y se llama a `ath5k_init` para acabar la inicialización.

- `ath5k_init`: Esta función se encarga de finalizar la inicialización de la información privada del *driver* sobre el *hardware*. Desde aquí se llama a `ath5k_setup_bands`.

## 6.2. Flujo de llamadas a funciones en la carga del módulo

Se aclara que este trazado tiene sobre todo en cuenta el código reflejado en `base.c` que, como ya se ha dicho, contiene la parte más relevante del *driver* en lo que a funcionamiento general respecta. Para trazar el flujo de llamadas a funciones se ha marcado la entrada y la salida de cada función y, posteriormente, se ha estudiado la salida de `dmesg` para comprender el funcionamiento del *driver* en su carga. Se decidió centrar el estudio en el código ejecutado en la carga del *driver*, puesto que el análisis estático reflejado en el apartado anterior dejó claro que es en esta sección donde se centra la mayor parte de las configuraciones de parámetros implicados en la implementación del protocolo IEEE 802.11p.

A continuación se detalla la información obtenida del marcado hecho y de la salida de *debug* con las que ya contaban los módulos involucrados en este estudio.

La primera llamada trazada que se ve en `demsg` es `ath5k_pci_probe` y se encuentra dentro de `pci.c`. Esta función realiza una inicialización PCI y se encarga de registrar la tarjeta e identificarla como una *wiphy* o una interfaz física inalámbrica.

Según la información obtenida, la primera función llamada dentro de `ath5k` en la carga del módulo es `ath5k_init_ah`, encargada de inicializar el *hardware* Atheros y configurar parámetros comunes y privados mediante una llamada a `ath5k_init`. El parámetro más relevante de la función es la variable `ah`, una estructura `ath5k_hw` que representa el estado del *driver* asociado a una instancia de *hardware* Atheros. Se ha añadido como anexo la definición de dicha estructura, la cual puede encontrarse en `drivers/net/wireless/ath/ath5k/ath5k.h`.

Dentro de las tareas realizadas por esta función, `ath5k_init_ah`, se encuentra la inicialización de los datos privados del *driver* y la inicialización del dispositivo.

Los datos privados del *driver* se inicializan al fijarse determinados campos del campo `ah->hw`, una estructura `ieee80211_hw` que contiene la configuración e información de un dispositivo físico IEEE 802.11. En concreto se fijan la dirección de la estructura con `ah->dev`, `hw->flags` con ciertas banderas (`IEEE80211_HW_RX_INCLUDES_FCS`, `IEEE80211_HW_HOST_BROADCAST_PS_BUFFERING`, `IEEE80211_HW_SIGNAL_DBM`, `IEEE80211_HW_MFP_CAPABLE`, `IEEE80211_HW_REPORTS_TX_ACK_STATUS`), `hw->wiphy->interface_mode` con los modos de interfaz de red admitidos (i.e., *Access Point*, *Station*, *Ad-hoc* y *Mesh*), `hw->wiphy->iface_combinations`, el *array* de combinaciones posibles modos interfaces, con la dirección del *array* y `hw->wiphy->n_iface_combinations`, el número máximo de modos de interfaz combinados, al valor uno.

La inicialización del dispositivo se realiza mediante una llamada a `ath5k_hw_init`, función definida en `attach.c`. A grandes rasgos puede decirse que en esta función se comprueba si el *hardware* concreto es soportado mediante una instrucción `case` y se inicializan las estructuras necesarias para cada caso.

La inicialización de los datos privados del *driver* la acaba la llamada `ath5k_init`. En su flujo de llamadas a otras funciones las más relevantes son `ath5k_setup_bands`, `ath5k_desc_alloc`, `ath5k_beaconq_setup`, `ath5k_txq_setup` y `ath5k_update_bssid_mask_and_opmode`, cuyo objetivo será desarrollado en adelante.

La primera llamada realizada dentro de `ath5k_init` es a `ath5k_setup_bands`, en ella se evalúa el bit de modo, `ah_capabilities.cap_mode`, de la estructura `ah` y se comprueba que cada uno de los modos a los que el *driver* da soporte son admitidos por la tarjeta física. Los modos disponibles para esta tarjeta son A y G, identificados por los enteros 2 y 0 respectivamente. Para aquellos modos que pasan la evaluación se copia la configuración de *bitrates* y de canales para cada una de las bandas *standard* disponibles para la tarjeta y se guarda en las variables, `sband->n_bitrates` y `sband->n_channeles`, el número total de *bitrates* y canales registrados, respectivamente.

Dentro de ath5k_setup_bands se llama a ath5k_setup_channels(*ah, -*channels, mod, max), también contenida en base.c. La función recibe cuatro parámetros, la estructura ah que contiene información del *hardware* Atheros que el *driver* controla, i.e., la tarjeta que se está usando; channels, una lista de estructuras que representa los canales posibles para cada banda de frecuencias; mod, un entero que indica el modo de emisión; y max, que es el límite de canales que pueden ser fijados. Dicha función consta básicamente de un switch sobre el modo pasado que se usa para fijar dos valores bases requeridos más adelante. Dichos valores son el número de canales con los que cuenta esa banda, size, y el identificador de la banda, band.

Con esos valores fijados se entra en un bucle que recorre los canales con dos banderas de salida relacionadas por un operador &&, ch<=size y count<max. En cada vuelta se rellenan los valores de un canal de la lista de estructuras de canal channels: frecuencia central del canal (center_freq), la banda (band) y el modo de operación (hw_value) y se comprueba que el canal sea soportado por el *chipset* y sea estándar con las funciones ath5k_channel_ok y ath5k_-is_standard_channel, respectivamente. Por último se aumenta el valor de count. Si algunas de las comprobaciones sobre el canal falla, la frecuencia no es soportada por el *chipset* o no es estándar, se continua con el bucle y por tanto deja de evaluarse, quedando la variable count sin aumentar. Al final se devuelve el valor count dando información sobre cuántos canales de los pertenecientes al modo pasado pueden emitir porque tienen frecuencias correctas, son soportados por el *chipset* y son estándar.

Como resumen de estas comprobaciones en el entorno actual, los canales probados por defecto del modo 2 (modo G a 2 GHz), con un máximo de 314 canales y en la banda 0 que pasen la comprobación ath5k_is_channel_ok son los que van del 1 al 14. Los que van del 15 al 16 tienen frecuencia 0, lo que implica que el mapeo inicial fue incorrecto y se los toma como no estándar.

Los canales probados por defecto del modo 0 (modo A a 5 GHz), con un máximo de 300 canales tras la vuelta del modo 2 y en la banda 1 con frecuencia válida y soportados por el *chipset* son los que van del 1 al 181 y del 197 al 220. De ellos el conjunto de los que de base son considerados estándar es el siguiente: 8, 12, 16, 36, 40, 44, 48, 52, 56, 60, 64, 100, 104, 108, 112, 116, 120, 124, 128, 132, 136, 140, 149, 153, 157, 161 y 165.

Los canales probados para el modo 0, max 300 y banda 1 del 182 al 197 tienen todos frecuencia válida, pero no son soportados por el *chipset*.

Acabada la ejecución de ath5k_setup_channels se devuelve el control a ath5k_setup_bands y se llama a ath5k_setup_rate_idx y ath5k_debug_dump_-bands, la primera función fija los valores de los *bitrates* de la banda de frecuencia pasada por parámetro y la segunda se encarga de que se envíen los mensajes de *debug*.

Tras resolverse la llamada ath5k_setup_bands desde ath5k_init se llama a ath5k_desc_alloc, donde se reserva la memoria para los descriptores de transmisión y receptor.

El control se devuelve a ath5k_init y se llama a ath5k_beaconq_setup, donde se reserva la memoria para la colas de transmisión. Una cola por marco

de baliza (*beacon frame*) y una cola de datos por cada prioridad de calidad de servicio o QoS (*Quality of Service*). Las funciones *hardware* se encargan de resetear estas colas a su debido tiempo.

El control se devuelve a `ath5k_init` y se llama a `ath5k_txq_setup`, donde se fijan los valores de las colas de transmisión en la estructura correspondiente. La mayor parte de las llamadas a funciones y la más importantes dentro `ath5k_txq_setup` se encuentra en `qcu.c`, fuera del alcance de este trabajo, por lo que no se ha ahondado en su funcionamiento.

El control se devuelve a `ath5k_init` y se llama a `ath5k_update_bssid_mask_and_opmode` que recibe como parámetros las estructuras `ah` y `vif` que contienen información de la tarjeta y de la interfaz virtual, respectivamente, para actualizar la máscara BSSID y modo de operación de la interfaz de red. Se usa la dirección MAC del *hardware* que obtiene desde `ah` como referencia y se usa junto con la máscara BSSID al hacer el *matching* de direcciones. Primero se fija la información de la interfaz con la dirección MAC, se establece como inactiva, como necesitada de fijado de dirección *hardware*, como *opmode* no especificado y como número de *stations* 0. Se obtiene un listado de todas las direcciones MAC activas. Establece el `opmode` de `ah` en función del valor obtenido de la lista. Se evalúa `iter_data.need_set_hw_addr && iter_data.found_active`, y si la expresión es cierta se fija el valor del ID de la estación con el de la MAC activa de la lista. Además, si `ah` tiene máscara BSSID se fija esa máscara en la estructura `ah`. Por último se establecen los filtros RX de recepción.

La siguiente llamada capturada es `ath_regd_init`, definida en `drivers/net/wireless/ath/regd.c` y encargada de evaluar la información relativa a la zona, a las restricciones de transmisión derivadas de la misma y registrar la *wiphy* en consecuencia.

Lo que hace el *driver* es consultar por la tarjeta y registrarla como una interfaz física `phyN` siendo N un número que se va incrementando desde el 0 cada vez que se relanza el módulo, esta llamada viene desde un principio desde la función `ath5k_pci_probe` cuyo código está en `pci.c`. Tras eso, el módulo `ath`, genérico para todas las tarjetas Atheros, comprueba el *regdomain*, i.e. el código de registro de dominio, un código que indica en qué zona esa tarjeta estaba configurada para emitir y por tanto bajo qué restricciones debe emitir; y el código de país por defecto a usar. En este caso la memoria EEPROM indica que el *regdomain* es `0x0` y el código de país es `0x3a`, lo que significa que la restricción de *regdomain* es nula o universal pero el código de país es el de EE.UU., por lo que se configurará la tarjeta para que emita bajo las condiciones de frecuencia y *bitrates* propias de ese país. Con esos datos comprueba el *country alpha*, un código de dos letras asociado al país y el *regpair*, su contrapartida numérica, si encuentra coincidencias usa esa restricción, sino se sirve de los resultados de una función llamada `regulatory_hint()`, que contiene valores precargados de restricciones de dominio. Con esta información como parámetro el módulo `ieee80211` selecciona el algoritmo de control de tasa de transferencia.

# Capítulo 7

# Modificaciones Propuestas

## 7.1. Añadir nuevo modo de operación

Dentro del fichero de cabecera `ath5k.h` puede verse un `enum` de nombre `ath5k_driver_mode` que contiene los identificadores de los modos PHY de operación. Se estima oportuno añadir un nuevo modo de operación, `AR5K_MODE_11P`, para la implementación del protocolo IEEE 802.11p. Esta propuesta de implementación pretende conservar el modo A sin modificar, a diferencia de lo que se ha hecho en los ejemplos de éxito encontrados, donde la implementación se ha basado en la modificación de los parámetros físicos del modo de operación A.

Código 7.1: Definición del `enum` de modos de operación del driver en `ath5k.h`

```
1
2 enum ath5k_driver_mode {
3         AR5K_MODE_11A           =       0,
4         AR5K_MODE_11B           =       1,
5         AR5K_MODE_11G           =       2,
6         AR5K_MODE_MAX           =       3
7 };
```

Dado que `ath5k_driver_mode` es sólo un `enum` y en la documentación queda reflejado que al usarse como índice en ciertos casos de uso, su alteración puede ocasionar problemas si no se tienen en cuenta todas las dependencias. Por ésto, la inclusión de un nuevo modo implica un estudio pormenorizado del código. Puesto que no pueden conocerse *a priori* las dependencias existentes, no puede hacerse una previsión totalmente realista del tiempo a dedicar a esta tarea.

## 7.2. Frecuencias de emisión

El más importante de los cambios propuestos para satisfacer los requisitos consiste en modificar el *driver* para que la tarjeta sea capaz de emitir en las frecuencias que van desde 5,85 MHz a 5,925 MHz. El primer paso es asegurarse de que el *chipset* de la tarjeta en uso da soporte a las frecuencias para IEEE 802.11p, el intervalo 5,825 GHz a 5,95 GHz. Lo que, según la información del *datasheet*, ocurre.

Centrando la atención en el código, las comprobaciones que se hacen en la carga de canales en los que el *driver* permite a la tarjeta emitir tienen dos fases, en la primera se evalúa que el canal sea estándar y en la segunda se evalúa que el canal esté definido de forma correcta. La función encargada de evaluar si un canal es estándar tiene dos definiciones en el código de `base.c` y se usa una u otra dependiendo si se ha activado o no la opción de `CONFIG_ATH5K_TEST_CHANNELS` al compilar el módulo. Si se ha activado se consideran todos los canales como estándar y, si no, se hace un evaluación en función del número de canal. La opción sencilla y la propuesta para las primeras pruebas es recompilar el módulo con dicha opción habilitada. Sin embargo, no es una solución completa dado que provoca retardos en la inicialización y funcionamientos erróneos cuando se usan herramientas como `iw` para listar la información de la interfaz física. La falta de límite en el número de canales considerados estándar ralentiza las inicializaciones en una pequeña medida y provoca que el *buffer* usado por comandos como `iw <phy> show` sea demasiado grande y acabe por desbordarse, imposibilitando el visionado de esa información. Con esta modificación las frecuencias necesarias ya serían consideradas estándar por el *driver*.

No obstante, la mejor alternativa para un desarrollo de esta características es añadir los canales necesarios para IEEE 802.11p a la lista de canales estándar definidos en la función de comprobación. A continuación se muestra un ejemplo de modificación para esta propuesta.

Código 7.2: Propuesta de modificación en `base.c` para habilitar los canales usados en IEEE 802.11p

```
1  static bool ath5k_is_standard_channel(short chan, enum ieee80211_band band)
2  {
3          if (band == IEEE80211_BAND_2GHZ && chan <= 14)
4                  return true;
5
6          return  /* UNII 1,2 */
7                  (((chan & 3) == 0 && chan >= 36 && chan <= 64) ||
8                  /* midband */
9                  ((chan & 3) == 0 && chan >= 100 && chan <= 140) ||
10                 /* UNII-3 */
11                 ((chan & 3) == 1 && chan >= 149 && chan <= 165) ||
12                 /* 802.11j 5.030-5.080 GHz (20MHz) */
13                 (chan == 8 || chan == 12 || chan == 16) ||
14                 /* 802.11j 4.9GHz (20MHz) */
15                 (chan == 184 || chan == 188 || chan == 192 || chan == 196) ||
16
17                 //MOD Propuesta de modificacion
18                 /* 802.11p (5.9MHz) */
19                 (chan >= 170 && chan <= 190)
20                 //DOM Propuesta de modificacion
21
22                 );
23 }
```

Tras la evaluación del canal como estándar mediante `ath5k_is_standard_channel`, se carga su información en la estructura encargada de llevar el registro de canales disponibles para ese *hardware* Atheros.

El tratamiento de la regulación de emisión es otro punto a tener en cuenta a la hora de dar una implementación del protocolo IEEE 802.11p, ya que también limitará las frecuencia en las que puede funcionar la tarjeta. Por ello, se ha con-

siderado que es necesario redefinir mediante la herramienta `wireless-regdb` la base de datos de regulación creando una nueva definición regulatoria de país que contemple la banda de frecuencia usada en WAVE. Con ella se deberá compilar un nuevo `regulatory.bin` e importarlo a `/usr/lib/crda`. Para que los cambios se hagan efectivos por el *driver* debe aceptarlos el agente de regulación de dominio `crda`. Para que esta modificación tenga efecto, el *driver* tiene que aceptar bases de datos de regulación personalizadas y ello implica activar la opción de configuración `CONFIG_CFG80211_CERTIFICATION_ONUS` en la compilación del *kernel*.

A continuación puede verse una propuesta de nueva definición regulatoria para `db.txt`.

Código 7.3: Propuesta de modificación sobre `db.txt` acorde a las restricciones de 802.11p

```
1  ...
2  #(start_freq MHz - end_freq MHz @ bandwidth MHz), (max_antenna_gain, max_eirp)
3  country HX:
4          (2402 - 2472 @ 40), (3, 27)
5          (5170 - 5250 @ 40), (3, 17)
6          (5250 - 5330 @ 40), (3, 20), DFS
7          (5490 - 5600 @ 40), (3, 20), DFS
8          (5650 - 5710 @ 40), (3, 20), DFS
9          (5735 - 5835 @ 40), (3, 30)
10         (5840 - 5930 @ 10), (3, 30)
11 ...
```

Esta propuesta toma como base la configuración regulatoria de Estados Unidos, `US`, pero amplía la última banda de emisión hasta los 6100 MHz para que contenga las frecuencias requeridas por IEEE 802.11p y cambia el ancho de banda a 10 MHz para corresponder a las restricciones de la implementación.

**Nota:** A pesar de que todos los cambios arriba detallados son necesarios para conseguir modificar las frecuencias de emisión de tarjeta y se ha realizado, no son suficientes. Lo que quiere decir que no se ha conseguido que la tarjeta emita en las frecuencias deseadas.

## 7.3. Ancho de banda

Para cumplir con los requisitos es necesario que el ancho de banda usado sea de 10 MHz, la mitad que el usado en IEEE 801.11a. Como se explicó anteriormente, ésto se hace para reducir lo tiempos de espera y conseguir comunicaciones más rápidas como son necesarias en comunicaciones intervehiculares. Dicho modo se encuentra definido ya en el *driver*, como puede verse en `ath5k.h`.

Código 7.4: Definición de `enum` de los anchos de banda admitidos en `ath5k.h`

```
1  /**
2   * enum ath5k_bw_mode - Bandwidth operation mode
3   * @AR5K_BWMODE_DEFAULT: 20MHz, default operation
4   * @AR5K_BWMODE_5MHZ: Quarter rate
5   * @AR5K_BWMODE_10MHZ: Half rate
6   * @AR5K_BWMODE_40MHZ: Turbo
```

```
 */
enum ath5k_bw_mode {
        AR5K_BWMODE_DEFAULT     = 0,
        AR5K_BWMODE_5MHZ        = 1,
        AR5K_BWMODE_10MHZ       = 2,
        AR5K_BWMODE_40MHZ       = 3
};
```

El cambio propuesto es el de modificar el *channel width* inicializado por defecto en la función `ath5k_hw_init` de `attach.c`.

Código 7.5: Función de inicialización del *hardware* Atheros en `attach.c`

```
int ath5k_hw_init(struct ath5k_hw *ah)
{
        static const u8 zero_mac[ETH_ALEN] = { };
        struct ath_common *common = ath5k_hw_common(ah);
        struct pci_dev *pdev = ah->pdev;
        struct ath5k_eeprom_info *ee;
        int ret;
        u32 srev;

        /*
         * HW information
         */
        ah->ah_bwmode = AR5K_BWMODE_DEFAULT;
        ah->ah_txpower.txp_tpc = AR5K_TUNE_TPC_TXPOWER;
        ah->ah_imr = 0;
        ah->ah_retry_short = AR5K_INIT_RETRY_SHORT;
        ah->ah_retry_long = AR5K_INIT_RETRY_LONG;
        ah->ah_ant_mode = AR5K_ANTMODE_DEFAULT;
        ah->ah_noise_floor = -95;       /* until first NF calibration is run
            */
        ah->ani_state.ani_mode = ATH5K_ANI_MODE_AUTO;
        ah->ah_current_channel = &ah->channels[0];
```

## 7.4. Nueva lista de *bitrates*

El nuevo *driver* deberá ser capaz de hacer que la tarjeta emita con los siguientes *bitrates* en Mbits: 3; 4,5; 6; 9; 12; 18; 24; 27. Estos valores son la mitad de los valores de *bitrate* del protocolo IEEE 802.11a. Su definición se encuentra en `ath5k.h`, como puede verse en el siguiente extracto de código.

Código 7.6: Definición de *bitrates* y propuesta de modificación para el *driver* `ath5k` en `ath5k.h`

```
 #define AR5K_MAX_RATES 32

 /* B */
#define ATH5K_RATE_CODE_1M      0x1B
#define ATH5K_RATE_CODE_2M      0x1A
#define ATH5K_RATE_CODE_5_5M    0x19
#define ATH5K_RATE_CODE_11M     0x18
/* A and G */

//MOD Propuesta de modificacion
#define ATH5K_RATE_CODE_3M      0x01
#define ATH5K_RATE_CODE_4_5M      0x00
//DOM Propuesta de modificacion

#define ATH5K_RATE_CODE_6M      0x0B
```

```
16 #define ATH5K_RATE_CODE_9M      0x0F
17 #define ATH5K_RATE_CODE_12M     0x0A
18 #define ATH5K_RATE_CODE_18M     0x0E
19 #define ATH5K_RATE_CODE_24M     0x09
20 #define ATH5K_RATE_CODE_36M     0x0D
21 #define ATH5K_RATE_CODE_48M     0x08
22 #define ATH5K_RATE_CODE_54M     0x0C
```

Según la documentación interna la tarjeta está preparada para registrar 32 *rates* distintos indexados del 1 al 32. De los *bitrates* necesarios todos salvo 3 Mbits y 4,5 Mbits ya se encontraban definidos. El primero es definible con el identificador `0x01` atendiendo la información de la tabla. El segundo no, pero puede definirse con valor 0x00 para considerarlo en la lista. Para conseguir una implementación completa será necesario estudiar cómo puede escribirse en la EEPROM de la tarjeta la información necesaria para fijar ese valor.

| **Rate code** | **Rate (Kbps)** |
|---|---|
| 0x01 | 3000 (XR) |
| 0x02 | 1000 (XR) |
| 0x03 | 250 (XR) |
| 0x04 - 05 | -Reservado- |
| 0x06 | 2000 (XR) |
| 0x07 | 500 (XR) |
| 0x08 | 48000 (OFDM) |
| 0x09 | 24000 (OFDM) |
| 0x0A | 12000 (OFDM) |
| 0x0B | 6000 (OFDM) |
| 0x0C | 54000 (OFDM) |
| 0x0D | 36000 (OFDM) |
| 0x0E | 18000 (OFDM) |
| 0x0F | 9000 (OFDM) |
| 0x10 - 17 | -Reservado- |
| 0x18 | 11000L (CCK) |
| 0x19 | 5500L (CCK) |
| 0x1A | 2000L (CCK) |
| 0x1B | 1000L (CCK) |
| 0x1C | 11000S (CCK) |
| 0x1D | 5500S (CCK) |
| 0x1E | 2000S (CCK) |
| 0x1F | -Reservado- |

Tabla 7.1: Tabla de *bitrates* de la tarjeta Atheros

Con esta definición hecha, lo siguiente es añadir los nuevos *bitrates* a la lista de *bitrates* posibles para `ath5k` definidos dentro de `base.c`. Esta lista se encuentra en forma de una estructura constante de tipo `ieee80211_rate` de nombre `ath5k_rates`. A continuación puede verse un ejemplo de propuesta de modificación.

Código 7.7: Definición de `ath5k_rates` en `base.h`

```
static const struct ieee80211_rate ath5k_rates[] = {
        { .bitrate = 10,
          .hw_value = ATH5K_RATE_CODE_1M, },
        { .bitrate = 20,
          .hw_value = ATH5K_RATE_CODE_2M,
          .hw_value_short = ATH5K_RATE_CODE_2M | AR5K_SET_SHORT_PREAMBLE,
          .flags = IEEE80211_RATE_SHORT_PREAMBLE },
          ...
        { .bitrate = 540,
          .hw_value = ATH5K_RATE_CODE_54M,
          .flags = 0 },

        //MOD Propuesta de modificaciones
        { .bitrate = 30,
          .hw_value = ATH5K_RATE_CODE_3M},
        { .bitrate = 40,
          .hw_value = ATH5K_RATE_CODE_4_5M},
        //DOM Propuesta de modificaciones
};
```

## 7.5. Duración de símbolo y preámbulo

Este cambio fue propuesto en un principio como requisito atendiendo al documento de ejemplo de éxito de la Universidad de Sri Lanka, pero se descubrió en el análisis del código que el *driver* ya estaba preparado para emitir con una duración de símbolo de 8 $\mu$s y de preámbulo 32 $\mu$s, desde el momento en que acepta comunicaciones *half-rate* o, lo que es lo mismo, desde que es capaz de emitir con un ancho de banda de 10 MHz. Ésto se ve reflejado en el siguiente fragmento de código.

Código 7.8: Fragmento de `pcu.c` donde se establecen los valores de preámbulo y tiempo de símbolo para cada ancho de banda

```
int ath5k_hw_get_frame_duration(struct ath5k_hw *ah, enum ieee80211_band band,
                int len, struct ieee80211_rate *rate, bool shortpre)
{

...

 switch (ah->ah_bwmode) {
        case AR5K_BWMODE_40MHZ:
                sifs = AR5K_INIT_SIFS_TURBO;
                preamble = AR5K_INIT_OFDM_PREAMBLE_TIME_MIN;
                break;
        case AR5K_BWMODE_10MHZ:
                sifs = AR5K_INIT_SIFS_HALF_RATE;
                preamble *= 2;
                sym_time *= 2;
                break;
        case AR5K_BWMODE_5MHZ:
                sifs = AR5K_INIT_SIFS_QUARTER_RATE;
                preamble *= 4;
                sym_time *= 4;
                break;
        default:
                sifs = AR5K_INIT_SIFS_DEFAULT_BG;
                break;
        }

...
```

```
28
29 }
```

Por tanto, no hay que adaptar el código para que ajustar los parámetros de tiempo de símbolo y preámbulo, sino establecer métodos para comprobar que se consigue que el *driver* emita en modo *half-rate* y dichos tiempos se respetan.

## 7.6. Intervalo de guarda

El intervalo de guarda es el tiempo de espera entre la trasmisión de cada símbolo por el canal con el objetivo de evitar interferencias derivadas ecos de la señal. En la siguiente imagen se puede apreciar con mayor claridad este problema.

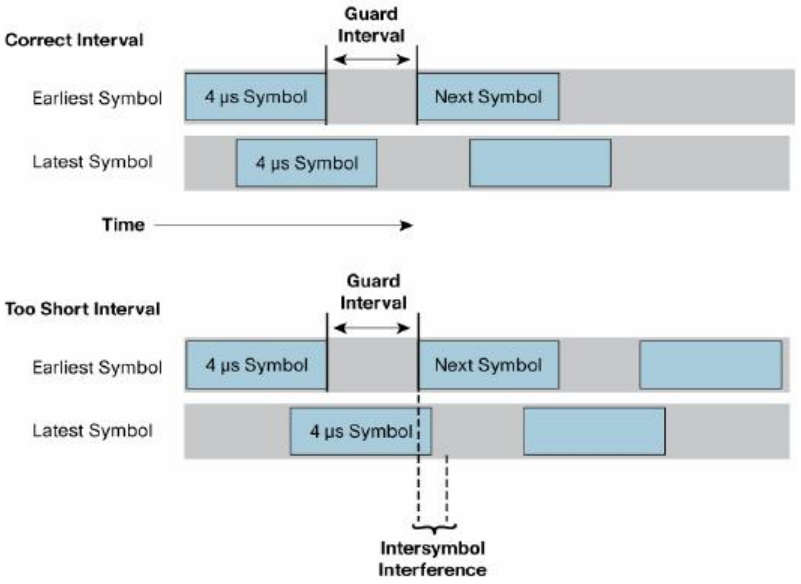


Figura 7.1: Explicación guard interval

A pesar de que es necesario modificar el intervalo de guarda de los 0,8 $\mu$s a los 1,6 $\mu$s dadas las características de la comunicación intervehicular no se ha encontrado en todo el código fuente del *kernel* un sitio en el que se tenga la certeza de que se fije dicho valor para ser añadido a los posibles valores de intervalo de guarda con los que puede operar la tarjeta. No obstante, se han hallado numerosas apariciones de banderas que permiten el uso del llamado *Short Guard Interval*, un intervalo de guardia corto, 0,4 $\mu$s, usado para las comunicaciones donde hay menor posibilidad de colisiones, razón por la cual pueden permitirse intervalos de guardia menores.

## 7.7. Duración de señal de campo

La señal de campo o L-SIG es usada por IEEE 802.11a para describir la tasa de transferencia de datos y la longitud en *bytes* del paquete. Los dispositivos

receptores la utilizan para calcular el tiempo de duración de la transmisión del paquete.

No se han encontrado referencias de este parámetro en el código de `ath5k`, pero sí en el código de `ath9k`. En el siguiente fragmento puede verse la definición de una constante nombrada como `L_SIG` y un uso de su valor para calcular la duración de la transmisión de un paquete. Se han buscado similitudes dentro de `ath5k` sin resultado. Cabe destacar que en el caso de `ath9k` para el cálculo de este tiempo de transmisión sólo se toman en cuenta dos anchos de banda posibles 20 MHz y 40 MHz. Por lo tanto, de intentarse llegar a una implementación desde `ath9k` habría que prestar atención a su uso, puesto que existe la posibilidad de que se necesite modificar su funcionamiento para contemplar el uso de un ancho de banda de 10 MHz.

Código 7.9: Fragmento de código de `xmit.c` en el driver `ath9k` donde pude verse una defínción de `L_SIG` y un uso de él

```
1  #define BITS_PER_BYTE           8
2  #define OFDM_PLCP_BITS          22
3  #define HT_RC_2_MCS(_rc)        ((_rc) & 0x0f)
4  #define HT_RC_2_STREAMS(_rc)    ((((_rc) & 0x78) >> 3) + 1)
5  #define L_STF                   8
6  #define L_LTF                   8
7  #define L_SIG                   4
8  #define HT_SIG                  8
9  #define HT_STF                  4
10 #define HT_LTF(_ns)             (4 * (_ns))
11 #define SYMBOL_TIME(_ns)        ((_ns) << 2) /* ns * 4 us */
12 #define SYMBOL_TIME_HALFGI(_ns) (((_ns) * 18 + 4) / 5)  /* ns * 3.6 us */
13 #define NUM_SYMBOLS_PER_USEC(_usec) (_usec >> 2)
14 #define NUM_SYMBOLS_PER_USEC_HALFGI(_usec) (((_usec*5)-4)/18)
15
16 ...
17
18 /*
19  * rix - rate index
20  * pktlen - total bytes (delims + data + fcs + pads + pad delims)
21  * width  - 0 for 20 MHz, 1 for 40 MHz
22  * half_gi - to use 4us v/s 3.6 us for symbol time
23  */
24 static u32 ath_pkt_duration(struct ath_softc *sc, u8 rix, struct ath_buf *bf,
25                             int width, int half_gi, bool shortPreamble)
26 {
27         u32 nbits, nsymbits, duration, nsymbols;
28         int streams, pktlen;
29
30         pktlen = bf_isaggr(bf) ? bf->bf_al : bf->bf_frmlen;
31
32         /* find number of symbols: PLCP + data */
33         nbits = (pktlen << 3) + OFDM_PLCP_BITS;
34         nsymbits = bits_per_symbol[rix][width];
35         nsymbols = (nbits + nsymbits - 1) / nsymbits;
36
37         if (!half_gi)
38                 duration = SYMBOL_TIME(nsymbols);
39         else
40                 duration = SYMBOL_TIME_HALFGI(nsymbols);
41
42       /* addup duration for legacy/ht training and signal fields */
43         streams = HT_RC_2_STREAMS(rix);
44         duration += L_STF + L_LTF + L_SIG + HT_SIG + HT_STF + HT_LTF(streams);
45
46         return duration;
47 }
```

## 7.8. Duración SIFS

Este parámetro es el tiempo en microsegundos requerido por una interfaz inalámbrica para procesar un paquete recibido y enviar otro paquete de respuesta. Se toma como la diferencia de tiempo entre el primer símbolo de respuesta emitido y el último símbolo del paquete recibido.

Dicho parámetro ya se encuentra definido correctamente para comunicaciones *half-rate* (con un ancho de banda de 10 MHz) en el código de `ath5k`, en concreto en el fichero de cabeceras `ath5k.h` y puede verse un ejemplo de uso en fichero `pcu.c` del mismo módulo, más concretamente en la función `ath5k_hw_get_default_sifs`. Ambos fragmentos de código se muestran a continuación.

Código 7.10: Ejemplo de uso de la constante AR5K_INIT_SIFS_HALF_RATE en `pcu.c`

```
/**
 * ath5k_hw_get_default_sifs - Get the default SIFS for current mode
 *
 * @ah: The &struct ath5k_hw
 */
unsigned int ath5k_hw_get_default_sifs(struct ath5k_hw *ah)
{
        struct ieee80211_channel *channel = ah->ah_current_channel;
        unsigned int sifs;

        switch (ah->ah_bwmode) {
        case AR5K_BWMODE_40MHZ:
                sifs = AR5K_INIT_SIFS_TURBO;
                break;
        case AR5K_BWMODE_10MHZ:
                sifs = AR5K_INIT_SIFS_HALF_RATE;
                break;

...
```

Código 7.11: Definición de la constante AR5K_INIT_SIFS_HALF_RATE en `ath5k.h`

```
/* SIFS */
#define AR5K_INIT_SIFS_TURBO                    6
#define AR5K_INIT_SIFS_DEFAULT_BG               10
#define AR5K_INIT_SIFS_DEFAULT_A                16
#define AR5K_INIT_SIFS_HALF_RATE                32
#define AR5K_INIT_SIFS_QUARTER_RATE             64
```

Por lo tanto no es necesario realizar ninguna modificación en el código sobre el tiempo SIFS, sino tan sólo comprobar que se realizan correctamente el cambio de ancho de banda y que con él se llega a un correcto fijado de la variable `sifs` encargada de contener dicho parámetro.

## 7.9. Espacio de subportadoras

En términos generales una subportadora es una señal analógica o digital contenida dentro de una transmisión de radio principal que contiene información

extra tales como voz o datos. En palabras técnicas, es un señal ya modulada que posteriormente será modulada en otra señal de mayor frecuencia y ancho de banda.

El estándar 802.11 reserva 64 subportadoras para su uso con OFDM, aunque en la práctica sólo 52 de ellas son usadas. Tomando en cuenta que las comunicaciones *half-rate* tienen un ancho de banda de 10 MHz y se cuenta con 64 subportadores, el espaciado de subportadores resultante es de 0.15625 MHz. Desgraciadamente no se ha encontrado ninguna referencia a este parámetro en el código fuente del `kernel`, por lo que lo único que queda es comprobar que con una correcta modificación del ancho de banda, este parámetro, sabidamente dependiente, cambia con él al valor esperado.

# Capítulo 8

# Pruebas y validación

Este capítulo se estructura en dos partes. La primera parte describe las pruebas que se han realizado para validar las modificaciones hechas sobre el código del driver. La segunda parte describe las pruebas que han de realizarse una vez modificado el modo de emisión del driver, lo que requiere información adicional suministrada por el fabricante y que no ha sido posible obtener.

## 8.1. Pruebas realizadas

### 8.1.1. Prueba P1 sobre frecuencias de emisión

La primera prueba se realizó mediante la herramienta `iw` en su versión 3.17. Esta utilidad de línea de comando basada en el nuevo `nl80211` para la configuración de dispositivos *WiFi* soporta todos los *drivers* que han sido recientemente añadidos al *kernel*. La antigua herramienta `iwconfing` usaba la interfaz `wext` o *Wireless Extensions* ya obsoleta, por lo que ahora se recomienda el uso de `iw`.

La opción usada en esta prueba es `iw <phy> info`, la cual muestra las *capabilities* o capacidades potenciales del dispositivo inalámbrico `<phy>`.

Siendo `<phy>` el identificador de dispositivo Atheros dado en la carga del *driver* se espera que la salida muestre en el apartado de frecuencias las frecuencias en las que puede emitir la tarjeta y que ese conjunto contenga a aquellas correspondientes a los canales que van del 171 al 184.

Para esta prueba las condiciones iniciales son que el *driver* cargado es el modificado `ath5kp` y el *regdomain* usado es `HX`, el propuesto en este TFG.

Código 8.1: Ejemplo de salida deseada al ejecutar `iw <phy> info` tras la modificación sobre las frecuencias de emisión permitidas

```
Wiphy phy0
...
                Frequencies:
...
                        * 5850 MHz [170] (30.dBm)
                        * 5855 MHz [171] (30.dBm)
                        * 5860 MHz [172] (30.dBm)
                        * 5865 MHz [173] (30.dBm)
                        * 5870 MHz [174] (30.dBm)
```

```
                        * 5875 MHz [175] (30.dBm)
                        * 5880 MHz [176] (30.dBm)
                        * 5885 MHz [177] (30.dBm)
                        * 5890 MHz [178] (30.dBm)
                        * 5895 MHz [179] (30.dBm)
                        * 5900 MHz [180] (30.dBm)
                        * 5905 MHz [181] (30.dBm)
                        * 5910 MHz [182] (30.dBm)
                        * 5915 MHz [183] (30.dBm)
                        * 5920 MHz [184] (30.dBm)
...
```

La salida obtenida fue la siguiente.

Código 8.2: Salida obtenida al ejecutar `iw <phy> info` tras la modificación sobre las frecuencias de emisión permitidas

```
Wiphy phy0
...
                Frequencies:
...
                        * 5850 MHz [170] (30.dBm) (passive scanning, no IBSS)
                        * 5855 MHz [171] (disabled)
                        * 5860 MHz [172] (disabled)
                        * 5865 MHz [173] (disabled)
                        * 5870 MHz [174] (disabled)
                        * 5875 MHz [175] (disabled)
                        * 5880 MHz [176] (disabled)
                        * 5885 MHz [177] (disabled)
                        * 5890 MHz [178] (disabled)
                        * 5895 MHz [179] (disabled)
                        * 5900 MHz [180] (disabled)
                        * 5905 MHz [181] (disabled)
                        * 5910 MHz [182] (disabled)
                        * 5915 MHz [183] (disabled)
                        * 5920 MHz [184] (disabled)
...
```

Como puede verse todos los nuevos canales han sido añadidos como estándar y son contemplados como posibles por la tarjeta. No obstante han sido deshabilitados todos salvo el canal 170 (5850 GHz) perteneciente al inicio de banda IEEE 802.11p.

Por motivos comparativos se realizó la misma prueba con el módulo `ath5k` y *regdomain* `HX`, con el módulo `ath5k` y *regdomain* `US` y, por último, con el módulo `ath5kp` y *regdomain* `US`. A continuación se muestran las salidas obtenidas.

Código 8.3: Salida de `iw <phy> info` con el módulo `ath5k` y *regdomain* `HX`

```
Wiphy phy0
...
                Frequencies:
                        * 5040 MHz [8] (disabled)
                        * 5060 MHz [12] (disabled)
                        * 5080 MHz [16] (disabled)
                        * 5180 MHz [36] (17.0 dBm)
...
                        * 5640 MHz [128] (disabled)
                        * 5660 MHz [132] (20.0 dBm) (passive scanning, no IBSS
                            , radar detection)
                        * 5680 MHz [136] (20.0 dBm) (passive scanning, no IBSS
                            , radar detection)
                        * 5700 MHz [140] (20.0 dBm) (passive scanning, no IBSS
                            , radar detection)
```

```
                        * 5745 MHz [149] (30.dBm)
                        * 5765 MHz [153] (30.dBm)
                        * 5785 MHz [157] (30.dBm)
                        * 5805 MHz [161] (30.dBm)
                        * 5825 MHz [165] (30.dBm)
                Bitrates (non-HT):
...
```

Código 8.4: Salida de `iw <phy> info` con el módulo `ath5k` y *regdomain* `US`

```
Wiphy phy0
...
                Frequencies:
                        * 5040 MHz [8] (disabled)
                        * 5060 MHz [12] (disabled)
                        * 5080 MHz [16] (disabled)
                        * 5180 MHz [36] (17.0 dBm)
...
                        * 5640 MHz [128] (disabled)
                        * 5660 MHz [132] (20.0 dBm) (passive scanning, no IBSS
                            , radar detection)
                        * 5680 MHz [136] (20.0 dBm) (passive scanning, no IBSS
                            , radar detection)
                        * 5700 MHz [140] (20.0 dBm) (passive scanning, no IBSS
                            , radar detection)
                        * 5745 MHz [149] (30.dBm)
                        * 5765 MHz [153] (30.dBm)
                        * 5785 MHz [157] (30.dBm)
                        * 5805 MHz [161] (30.dBm)
                        * 5825 MHz [165] (30.dBm)
                Bitrates (non-HT):
...
```

Código 8.5: Salida de `iw <phy> info` con el módulo `ath5kp` y *regdomain* `US`

```
Wiphy phy0
...
                Frequencies:
...
                        * 5850 MHz [170] (disabled)
                        * 5855 MHz [171] (disabled)
                        * 5860 MHz [172] (disabled)
                        * 5865 MHz [173] (disabled)
                        * 5870 MHz [174] (disabled)
                        * 5875 MHz [175] (disabled)
                        * 5880 MHz [176] (disabled)
                        * 5885 MHz [177] (disabled)
                        * 5890 MHz [178] (disabled)
                        * 5895 MHz [179] (disabled)
                        * 5900 MHz [180] (disabled)
                        * 5905 MHz [181] (disabled)
                        * 5910 MHz [182] (disabled)
                        * 5915 MHz [183] (disabled)
                        * 5920 MHz [184] (disabled)
...
```

En las pruebas se puede apreciar que en ningún caso los canales que van del 170 al 184 se encuentran presentes en la lista de canales posibles para el *driver* sin modificar, `ath5k`, sea cual sea su configuración de región. En el caso probado al usar el módulo `ath5kp` y la configuración de región por defecto, `US`, los canales del 170 al 184 se encuentran presentes en la lista, pero todos están deshabilitados.

De ésto se puede concluir que la adición de los canales necesarios para la

implementación a la lista de canales estándar se ha realizado correctamente. Y la modificación regulatoria es una condición necesaria, pero no suficiente para conseguir una implementación completa del protocolo. Existe aún alguna restricción no eliminada para conseguir emitir en las frecuencias deseadas.

### 8.1.2. Prueba P2 sobre frecuencias de emisión

La siguiente prueba consistió en crear entre dos máquinas idénticas una red *ad-hoc* con las frecuencias problema. A continuación se muestran los pasos necesarios para crear una red *ad-hoc*, siendo `<interfaz>` el identificador de interfaz de red y `<direccion ip>` la dirección IP que se quiera dar en cada máquina. Este procedimiento debió realizarse en ambas máquinas,

Código 8.6: Pasos para crear una red ad-hoc

```
$ sudo ifconfig <interfaz> down
$ sudo iw <interfaz> set type ibss
$ sudo ifconfig wlan0 <direccion ip> up
$ sudo iw <interfaz> ibss join foss-adhoc 2412
$ sudo iw <interfaz> info
$ sudo iw <interfaz> link
```

Se esperaba que tras el tiempo tardado en realizar los procedimientos de sincronización y configuración se creara una red *ad-hoc* entre las dos máquinas y se pudiera realizar un *ping* de una máquina a otra y viceversa.

Sin embargo con ninguna frecuencia fue posible conseguir una comunicación exitosa. La salida obtenida fue la codificada con el número `-22`, código que corresponde al error argumento inválido. Este error se ha trazado en el código fuente de `iw` hasta `ibss.c` donde se ha perdido el rastro en una llamada a `NLA_PUT_U32(msg, NL80211_ATTR_WIPHY_FREQ, freq);`. Este tipo de llamadas son llamadas de fijación de atributos `netlink`, el cual es una interfaz de *sockets* para la comunicación entre procesos del *kernel* y de usuario. Lo que da idea de que el problema no está en la herramienta usada para la configuración de la red *adhoc*, sino en alguna restricción del *hardware* o el *driver* que no ha sido eliminada aún.

### 8.1.3. Prueba P3 sobre frecuencias de emisión

A raíz de la prueba anterior se intentó crear otro tipo de red distinta a la *adhoc*. Se consiguió entonces configurar la interfaz de red inalámbrica en modo monitor con la frecuencia 5850 GHz. A continuación se listan la serie de comandos utilizados para conseguir esta configuración.

Código 8.7: Configuración del modo monitor

```
$ sudo iw phy phy0 interface add mon0 type monitor
$ sudo iw dev wlan0 del
$ sudo ifconfig mon0 up
$ sudo iw dev mon0 set freq 2437
$ tcpdump -i mon0 -n -w wireless.cap
```

El último comando sirve para establecer una escucha desde la interfaz en modo monitor, pero dado que no existen comunicaciones de momento en esa frecuencia, la salida obtenida fue nula.

### 8.1.4. Prueba P4 sobre nueva lista de *bitrates*

La primera prueba a realizar es que el *driver* ha identificado y permitido la lista de *bitrates* para el nuevo modo de emisión. Para ello se puede ejecutar `iw list`, comando que sirve para listar todos los dispositivos inalámbricos junto a sus capacidades.

La salida esperada y obtenida es la siguiente:

Código 8.8: Salida esperada de `iw list` en una prueba sobre la modificación de los *bitrates*

```
Wiphy phy1:
        Band 1:
                Bitrates:
                        ...
                Frequencies:
                        ...
        Band 2:
                Bitrates (non-HT):
                        * 3.0 Mbps
                        * 4.5 Mbps
                        * 6.0 Mbps
                        * 9.0 Mbps
                        * 12.0 Mbps
                        * 18.0 Mbps
                        * 24.0 Mbps
                        * 27.0 Mbps
                Frequencies:
                        ...
```

De ésto se puede concluir que la lista de *bitrates* necesaria para la implementación ha sido añadida y reconocida por el *driver* modificado.

## 8.2. Pruebas propuestas

### 8.2.1. Prueba PF1 sobre frecuencias de emisión

La última prueba sobre las frecuencias de emisión consiste en comprobar efectivamente que la comunicación entre las dos máquinas se produce en la frecuencia requerida. Para ello se ha escrito un pequeño programa C que abre un *socket* y envía datos y otro que los consume. La prueba consiste en lanzar ambos programa y analizar con un espectrómetro la comunicación para verificar que se ha realizado la comunicación de manera exitosa. El código de ambos programas se ha añadido como anexo a esta memoria. Se espera que fijando como frecuencia central la frecuencia elegida para crear la red ad-hoc y estableciendo un *span* o ventana de frecuencia de 50 MHz, pueda verse la onda de la transmisión entre ambas máquinas.

Dado que aún no se ha conseguido crear una red *ad-hoc* entre dos máquinas idénticas en ninguna de las frecuencias usadas por el protocolo IEEE 802.11p, esta prueba aún no puede ser realizada. No obstante, sí se ha comprobado que el código funciona correctamente en otras frecuencias.

### 8.2.2. Prueba PF2 sobre nuevo modo de emisión

Dado que no parece haber una herramienta para averiguar el modo físico de operación, entiéndase por ésto, diferenciar si se está usando el protocolo IEEE 802.11a, IEEE 802.11b, etc; se propone como prueba para asegurar que se ha conseguido introducir un nuevo modo PHY de operación, realizar un sembrado de sentencias en la zona del código donde se carguen la información de los canales en los que puede emitir la tarjeta al usar ese modo de operación. A continuación se muestra un fragmento del código del fichero `base.c` para dejar claro dónde debería realizarse dicha prueba.

Código 8.9: Fragmento de código de `base.c` donde se cargan los canales para cada modo PHY de operación

```
static unsigned int
ath5k_setup_channels(struct ath5k_hw *ah, struct ieee80211_channel *channels,
                unsigned int mode, unsigned int max)
{
        unsigned int count, size, freq, ch;
        enum ieee80211_band band;

        switch (mode) {
        case AR5K_MODE_11A:
                /* 1..220, but 2GHz frequencies are filtered by check_channel
                    */
                size = 220;
                band = IEEE80211_BAND_5GHZ;
                break;

        //MOD Propuesta de modificacion y prueba
        case AR5K_MODE_11P:
                        /*172..184  */
                        size = 13;
                        band = IEEE80211_BAND_10GHZ;
                        break;
                        printk("ATH5KP_DEBUG: Cargando canales de 11p\n");
        //DOM Propuesta de modificacion y prueba

        case AR5K_MODE_11B:
        case AR5K_MODE_11G:
                size = 26;
                band = IEEE80211_BAND_2GHZ;
                break;
        default:
                ATH5K_WARN(ah, "bad mode, not copying channels\n");
                return 0;
        }

...
}
```

Se espera que en los mensajes de *debug* visibles con el comando `dmesg` pueda encontrarse la línea `ATH5KP_DEBUG: Cargando canales de 11p`.

### 8.2.3. Prueba PF3 sobre ancho de banda

Las pruebas a realizar para comprobar que la modificación del ancho de banda se ha realizado correctamente requieren el uso de un analizador de espectro. Ha de fijarse la frecuencia central y un *span* de 59 MHz para capturar la imagen de la onda.

Lo que se espera ver es que la ancho de banda detectado por el analizador de espectro sea de 10 MHz.

### 8.2.4. Prueba PF4 sobre duración de símbolo, preámbulo y duración SIFS

En el caso de estos parámetros su valor se fija en cada transmisión en `pcu.c`, por lo que una prueba de que se usan los tiempos de retardo correctos es realizar un sembrado de sentencias en esa parte del código tras la asignación, para comprobar que los anchos de banda usados son correctos y los tiempos corresponden a lo dictado por el protocolo.

Código 8.10: Fragmento de `pcu.c` donde se establecen los valores de preámbulo y tiempo de símbolo para cada ancho de banda

```
int ath5k_hw_get_frame_duration(struct ath5k_hw *ah, enum ieee80211_band band,
                int len, struct ieee80211_rate *rate, bool shortpre)
{

...

 switch (ah->ah_bwmode) {
        case AR5K_BWMODE_40MHZ:
                sifs = AR5K_INIT_SIFS_TURBO;
                preamble = AR5K_INIT_OFDM_PREAMBLE_TIME_MIN;
                break;
        case AR5K_BWMODE_10MHZ:
                sifs = AR5K_INIT_SIFS_HALF_RATE;
                preamble *= 2;
                sym_time *= 2;
                break;
        case AR5K_BWMODE_5MHZ:
                sifs = AR5K_INIT_SIFS_QUARTER_RATE;
                preamble *= 4;
                sym_time *= 4;
                break;
        default:
                sifs = AR5K_INIT_SIFS_DEFAULT_BG;
                break;
 }
 //MOD Sembrado de sentencias de prueba sobre sym_time, preamble y sifs
 printk("ATH5KP_DEBUG: sym_time=%d, preamble=%d, sifs=%d\n",sym_time, preamble
     , sifs);
 //DOM Sembrado de sentencias de prueba sobre sym_time, preamble y sifs

...

}
```

# Capítulo 9

# Conclusiones

Este Trabajo de Fin de Grado busca desentrañar el funcionamiento de la interfaz `ath5k`, con el fin de adaptarla para el cumplimiento del estándar 802.11p. El primer requisito sobre el que se trabajó en este TFG fue la adición de un nuevo modo de operación con en el que fuera, en la medida de lo posible, minimizar las modificaciones sobre el funcionamineto normal del driver base `ath5k`. Las acciones emprendidas fueron las siguientes:

- Se ha buscado el modo de añadir al driver un nuevo modo de operación a los ya existentes. Para ello se realizó un estudio pormenorizado del funcionamiento del driver. Este objetivo se ha cumplido sólo parcialmente, al persistir accesos no controlados a memoria que hacen sospechar que la interacción entre los modos de funcionamiento y el driver provocan comportamientos no deseados.

- Una vez descartada la idea anterior, se optó por modificar los parámetros del modo A para que correspondan a los exigidos por el modo P, tal y como hicieron los equipos recogidos en este documento como ejemplos de éxito.

- Se ha trabajado en la determinación de las acciones necesarias para permitir la emisión en la banda de frecuencias de 5,9 Ghz. Tras llevar a cabo las modificaciones propuestas en el capítulo 7, la herramienta `iw` permitió comprobar que los canales que van del 170 (5,85 GHz) al 181 (5,905 GHz) fueron correctamente añadidos a la lista de canales estándar y que su definición era correcta, tanto en la lista de *bitrates* que contempla como en la frecuencia central que los define. Tras ello, se detectó que existe una restricción no documentada en el *firmware* de la tarjeta wifi utilizada que deshabilita todos los canales salvo el 170 (frecuencia central 5,85 GHz). Esa restricción es un problema hardware que no invalida el desarrollo realizado. Los intentos de comunicarse con el fabricante de la tarjeta han sido infructuosos.

- Se ha consegido configurar la tarjeta en modo escucha sobre él en la primera frecuencia de la banda, lo cual será de gran utilidad en futu-

ras pruebas, dado que servirá de herramienta para comprobar que las comunicaciones se producen de manera correcta.

- El ancho de banda por defecto ha sido modificado en su inicialización y el sembrado de sentencias comprueba que la modificación se ha realizado correctamente. A falta de comprobar con un analizador de espectro que la información modificada en el *driver* es interpretada correctamente por la tarjeta y realmente se respeta la restricción del ancho de banda, se puede considerar que este parámetro ha sido modificado con éxito.

- En cuanto a la lista de *bitrates* permitida, las pruebas realizadas con la herramienta `iw <phy> list` demuestran que han sido añadidos al *driver* correctamente, por lo que se deduce que, una vez realizado el paso de un ancho de banda a otro sin incidencias, también lo hará el cambio de *bitrates* en cada momento.

- Respecto de la modificación de los parámetros temporales (duración de símbolo, prémabulo y duración SIFS), se ha determinado que ya está resuelto dentro de la implementación de comunicaciones *half-rate* suministrada por el *driver* `ath5k`. Sin embargo, no pueden realizarse las pruebas propuestas debido a la imposibilidad, descrita arriba, de realizar comunicaciones entre dos máquinas en las frecuencias deseadas.

- Para el resto de parámetros (intervalo de guarda, espacio de subportadores y duración de señal de campo) no se han encontrado referencias en el código de `ath5k` suficientes que permitan dar una idea de qué tipo de modificaciones son necesarias para hacer cumplir las restricciones del protocolo en esos aspectos. No ocurre así en el caso del *driver* `ath9k`, tal y como quedó reflejado en el capítulo 7.

En definitiva, lo que el análisis llevado a cabo permite concluir es que la forma más sencilla de conseguir una implementación del protocolo IEEE 802.11p pasa por la modificación de los parámetros de una implemantación del protocolo IEEE 802.11a sobre un driver *Atheros*, ya que de los conocidos, es el más flexible. No obstante, por la experiencia obtenida en este TFG y declaraciones de gente que se ha enfrentado al mismo problema, podemos concluir que es preferible intentarlo con tarjetas *WiFi* de la serie 9000, más abierta a cambio, más documentada y con una comunidad de desarrolladores más activa a día de hoy.

El principal escollo al que nos hemos enfrentado, y el que ha llevado más tiempo, ha sido la modificación de la frecuencia de emisión. Aunque no se ha conseguido dar una solución completa a este problema, se ha conseguido información suficiente como para tener una buena base de la que partir en futuros proyectos relacionados.

## 9.1. Trabajo futuro

Habida cuenta de la información obtenida con este Trabajo de Fin de Grado se proponen los siguientes trabajos futuros.

- Desarrollo de un *driver* para comunicaciones 802.11p basado en `ath9k`
- Estudio sobre las comunicaciones IEEE WAVE/1609 en las capas de enlace y superiores

El primer tema se propone por los motivos explicado arriba, por lo que ha podido aprenderse en el desarrollo de este Trabajo de Fin de Grado el *driver* `ath9k` está más abierto a modificaciones y tiene una comunidad de desarrolladores más activa en estos momentos.

El segundo tema se propone porque, dado que este Trabajo parte de la idea de llegar a la construcción de sistema capaz de realizar comunicaciones WAVE, es necesario no sólo dar una implementación no sólo al estándar IEEE 802.11p, sino también a la serie de estándares IEEE 1609 que contribuyen a la pila WAVE.

# Bibliografía


[1] 802.11-2012 - IEEE Standard for Information technology–Telecommunications and information exchange between systems Local and metropolitan area networks–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. Technical Report IEEE Std 802.11$^{TM}$-2012, IEEE-Inst.

[2] Philippe Agostini, Raymond Knopp, Jérôme Härri, and Nathalie Haziza. Implementation and test of a DSRC prototype on OpenAirInterface SDR platform. In *ICC 2013, IEEE International Conference on Communications Workshops, 9-13 June 2013, Budapest, Hungary*, Budapest, HUNGARY, 06 2013.

[3] O. Altintas, W. Chen, G.J. Heijenk, F. Dressler, E. Ekici, F. Kargl, H. Shigeno, and S. Dietzel, editors. *2011 IEEE Vehicular Networking Conference (VNC): Demo Summaries*. Number WP 11-04 in CTIT Workshop Proceedings Series. University of Twente, Enschede, the Netherlands, November 2011.

[4] Duarte Carona, António Serrador, Pedro Mar, Ricardo Abreu, Nuno Ferreira, Tiago Meireles Paixão, João Nuno Matos, and Jorge Alves Lopes. A 802.11p prototype implementation. In *Intelligent Vehicles Symposium*, pages 1116–1121. IEEE, 2010.

[5] Sotfware Freedom Law Center. Coda analysis of the linux wireless team's ath5k driver, Septiembre 2007.

[6] Sotfware Freedom Law Center. mac80211 overview, Febrero 2009. Resumen en pdf del funcionamiento y estructuras de mac80211.

[7] Luís Miguel Faria de Oliveira. Vehicular networks: Ieee 802.11p analysis and integration into heterogeneous wmn. Master's thesis, Faculdade de Engenharia Da Universidade Do Porto, Junio 2012.

[8] Dr. Hans-Joachim Fisher. Basic set of communication standars (calm) enabling one commin its platform as requested by the ec, 2008. Resumen en pdf de los estándar WAVE, DSRC e IEEE 802.11p.

[9] Andreas Geiger, Martin Lauer, Frank Moosmann, Benjamin Ranft, Holger Rapp, Christoph Stiller, and Julius Ziegler. Team annieway's entry to

the grand cooperative driving challenge 2011. *Transactions on Intelligent Transportation Systems (TITS)*, 2012.

[10] Daniel Jiang and Luca Delgrossi. Ieee 802.11p: Towards an international standard for wireless access in vehicular environments. In *VTC Spring*, pages 2036–2040. IEEE, 2008.

[11] Wei-Yen Lin, Mei-Wen Li, Kun chan Lan, and Chung-Hsien Hsu. A comparison of 802.11a and 802.11p for v-to-i communication: a measurement study. November 2010.

[12] Filipe Neves, Andre Cardote, Ricardo Moreira, and Susana Sargento. Real-world evaluation of ieee 802.11p for vehicular networks. In *Proceedings of the Eighth ACM International Workshop on Vehicular Internetworking*, VANET '11, pages 89–90, New York, NY, USA, 2011. ACM.

[13] J. R. Kodagoda y I. U. Liyanage R. S. A. De Silva, D. P. G. S. R. Fernando. Driver development for ieee wave (802.11p/1609.x) based wireless module with embedded networking services, Diciembre 2012. Presentación de diapositivas de un proyecto realizado por la Universidad de Moratuwa para el desarrollo de un módulo WiFi basado en WAVE.

[14] Dr. Michele Weigle. Standars: Wave/dsrc/802.11p, 2008. Resumen en pdf de los estándar WAVE, DSRC e IEEE 802.11p.

[15] Xi Zhang and Daji Qiao, editors. *Quality, Reliability, Security and Robustness in Heterogeneous Networks - 7th International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness, QShine 2010, and Dedicated Short Range Communications Workshop, DSRC 2010, Houston, TX, USA, November 17-19, 2010, Revised Selected Papers*, volume 74 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. Springer, 2012.

# Anexo A

# Contenido del soporte digital

En el soporte digital adjunto con el TFG se encuentran cuatro directorios, **Codigo**, **Herramientas**, **Memoria**, y **SO**.

En el primero, **Codigo**, se encuentra el código modificado `ath5kp`. En el segundo, **Herramientas**, se encuentran las herramientas externas utilizadas en el TFG. En el tercero, **Memoria**, se encuentra el PDF `memoria.pdf` con esta memoria. En el cuarto y último, **SO**, se encuentra la imagen ISO del CD de instalación del sistema operativo usado, Slackware 14.1 y la imagen de *kernel* 3.10 comprimida.

Dentro de **Código** se encuentran dos directorios, `ath5kp` y `extras`. El primero contiene el código de la propuesta de módulo adaptado `ath5kp` y `extras` contiene *scripts* que modifican archivos relacionados con la implementación que modifican otros módulos distintos a `ath5k`, junto a ficheros necesarios en la instalación del entorno.

En más detalle, `extras` contiene el directorio `codMod` con los ficheros de código externos a `ath5k` modificados y necesarios para esta adaptación; `config`, el fichero de configuración recomendado para la compilación del *kernel*; `db.txt`, la base de datos regulatoria de dominio; `leeme.txt`, un fichero de texto que detalla el contenido del directorio; y `setup.sh`, el *script* bash usado para modificar los archivos relacionados con la implementación que modifican otros módulos distintos a `ath5k`.

Dentro de **Herramientas** están contenidos los archivos comprimidos `crda--latest.tar`, que contiene el código fuente de la herramienta *Central Regulatory Domain Agent*; y `wireless-regdb_2009.11.25-1.tar`, que contiene la herramienta para compilar las bases de datos regulatorias.

- Codigo/
  - ath5kp/
  - extras/
    - codMod/
    - config
    - db.txt
    - leeme.txt

        - setup.sh
- Herramientas/
    - crda-latest.tar
    - wireless-regdb_2009.11.25-1.tar
- Memoria/
    - memoria.pdf
- SO
    - linux-3.10.tar.gz
    - slackware-14.1-install-dvd.iso.tar.gz

# Anexo B

# Instalación del entorno de desarrollo

## B.1. Instalar Slackware

La versión de Slackware usada en este TFG es la 14.1 de 32 *bits* con una instalación estándar.

## B.2. Compilar *kernel*

Para este trabajo se usó el *kernel* Linux 3.10.0 descargado desde el sitio *web* `kernel.org`. El código fuente (`linux-3.10.tar.gz`) se encuentra comprimido en el directorio `SO` del soporte digital. Dicha imagen hay que descomprimirla en `/usr/src`.

Dentro del directorio `Codigo/extras` contenido en el soporte digital adjunto se encuentra el fichero de configuración `config`. A continuación se explicará el procedimiento de despliegue de un *kernel*, apartado que puede obviarse si ya se conoce.

Aunque no es necesario, se recomienda modificar en el archivo `Makefile`, contenido dentro del directorio del código fuente, el campo `EXTRAVERSION` por una cadena que permita identificar esta compilación personalizada de otras compilaciones. A continuación se muestra un ejemplo de renombrado.

```
EXTRAVERSION = .TFG-00
```

Una vez descargado y descomprimido el directorio del código de fuente, es necesario configurarlo, compilarlo e instalarlo. Para ello, por cuestiones de simplicidad, primero se borra el enlace simbólico `linux` y se crea otro enlace simbólico `linux` que apunte al directorio del nuevo código fuente. En ese directorio debe copiarse el fichero de configuración `config` contenido en el directorio `extras` y renombrarse como `.config`. A continuación se muestran los comandos a ejecutar para compilar el *kernel*, sus módulos e instalarlos.

Código B.1: Lista de comandos a ejecutar para compilar el *kernel* y sus módulos

```
$ make
$ make modules
$ make modules_install
```

El nuevo *kernel* ha quedado guardado en `/usr/src/linux/arch/i386/boot` con el nombre `bzImage`, esa imagen debe copiarse dentro del directorio `/boot` y se recomienda que sea renombrada de forma reconocible. Hecho ésto, es necesario configurar el gestor de arranque `lilo` para que ofrezca el nuevo *kernel* como opción en el arranque. Para ello hay que editar el fichero `/etc/lilo.conf`. A continuación se muestra un ejemplo de modificación donde pueden verse cuatro líneas de las cuales interesa prestar atención a las tres primeras: en la primera se especifica la ruta de la imagen; en la segunda, la ruta del dispositivo que se usará como raíz y en la tercera, la etiqueta por la que se identificará la imagen de *kernel*.

Código B.2: Ejemplo de modificación del fichero de configuración `lilo.conf`

```
1 ...
2   image = /boot/vmlinuz-3.10.TFG-00
3   root = /dev/sda4
4   label = 3_10_TFG
5   read-only
6 ...
```

El fragmento anterior es un ejemplo de configuración donde pueden verse los campos a rellenar. El primero es `image`, el cual debe contener la ruta a la imagen a cargar. El segundo es `label` y contiene la etiqueta que se va a mostrar en arranque para identificar a esa imagen. El último campo es `root` y contiene la ruta del dispositivo que se usará como raíz al cargarse esa imagen de *kernel*. En este caso, se recomienda usar los valores de la imagen cargada por defecto que puede encontrarse como última entrada en dicho fichero si no se ha añadido ningún *kernel* antes. Para hacer efectivos los cambios en la configuración es necesario ejecutar el comando `lilo`.

Una vez hecho ésto, para usar la imagen del *kernel* recién compilada es necesario reiniciar la máquina y seleccionar la nueva imagen en el gestor de arranque.

## B.3. Instalar *driver* modificado

Lo primero será situarse el directorio `drivers/net/wireless/ath` contenido en el directorio del código fuente del *kernel* descargado. Es en este directorio donde se encuentra el *driver* `ath5k`, aquel que sirvió de base para la propuesta de desarrollo de `ath5kp`, el nuevo *driver* propuesto en este TFG. La nueva versión que implemente el protocolo IEEE 802.11p deberá colocarse entonces en `drivers/net/wireless/ath`. En este directorio ha de colocarse el directorio `ath5k` contenido en el directorio `Codigo` del soporte digital.

Dentro de `extras` se encuentran una serie de ficheros que contienen código de otros módulos que guardan alguna relación de dependencia con el *driver* Atheros modificado. Se han añadido al repositorio porque necesitan ser

sustituidos para conseguir una implementación del protocolo IEEE 802.11p. Junto a los ficheros de código y cabeceras C se encuentra un fichero de texto, `leeme.txt`, que contiene la lista de ficheros C modificados y la ubicación donde deben ser colocados, y un *script* Bash, `setup.sh`, que se encarga de remplazarlos con una sola ejecución.

Dado que se han modificado varios módulos con la ejecución del *script* `setup.sh`, se recomienda por cuestiones de simplicidad compilar todos los módulos nuevamente. Para ello basta con situarse en el directorio `/usr/src/linux` ejecutar lo siguiente.

```
$ make
$ make modules
$ make modules_install
```

Al intentar compilar tras los nuevos cambios se han añadido nuevas opciones de compilación relacionadas con el nuevo módulo `ath5kp`. La configuración necesaria para estas nuevas opciones se listan a continuación.

- **Atheros 5xxx wireless cards support (for 802.11p) (ATH5KP) [N/m/?] (NEW):** m
- **Atheros 5xxx debuggin (for 802.11p) (ATH5KP_DEBUG) [N/y/?] (NEW):** y
- Atheros 5xxx debuggin (for 802.11p) (ATH5KP_DEBUG) [N/y/?] (NEW): y
- **Atheros 5xxx PCI bus support (for 802.11p) (ATH5KP_PCI) [N/y/?] (NEW):** y
- **Enables testing channels on ath5k (for 802.11p) (ATH5K_TEST_CHANNELS) [N/y/?] (NEW):** n

Tras reiniciar la máquina todos los cambios se habrán hecho efectivos.

## B.4. Modificar, compilar e instalar nuevo *regdomain*

Para las modificaciones sobre la base de datos de registro de dominio se han usado las herramientas `crda` y `wireless-regdb`. Estas herramientas se encuentran adjuntas en el soporte digital que acompaña este TFG.

Lo primero que hay que hacer, una vez copiados y desplegados ambos códigos fuentes, es generar una nueva base de datos de regulación personalizada. Para ello se ha de modificar el documento `db.txt` contenido en el directorio del proyecto `wireless-regdb` y añadir una nueva definición de país que respete las necesidades de una implementación 802.11p. En el directorio `extras` se encuentra el fichero `db.txt` con la configuración usada en este proyecto.

Sólo hay que copiar sobreescribiendo con ese fichero el fichero `db.txt` contenido en el código fuente de `wireless-regbd`. Con esa base de datos se ha de crear el nuevo `regulatory.bin`, un binario que contiene la base de datos de forma que el servicio `crda` pueda leerla y manejarla, para ello hay que ejecutar `make` en el directorio del proyecto `wireles-regdb` donde tenemos el fichero de texto de la base de datos. Esto no sólo crea el fichero `regulatory.bin`, sino también el par de claves pública y privada con las que se firmará la base de datos. Dichas claves se almacenan en `$BUILDDIR/<username>.key.pub.pem` y `/.wireless-regdb-<username>.key.priv.pem`, respectivamente.

Antes de sustituir la anterior base de datos por la nueva creada, se recomienda hacer una copia de seguridad de la antigua. Dicha base de datos se encuentra en el directorio `/usr/lib/crda` y es ahí donde debe copiarse la nueva.

Ahora ha de instalarse el nuevo `crda` descargado. Primero se copia la clave pública desde el directorio del proyecto `wireless-regdb` a `crda/pubkeys` y se ejecutan los comandos `make` y `sudo make install`.

Para que los cambios se hagan efectivos es necesario relanzar el módulo `cfg80211`, para ello se pueden cargar y descargar todos los módulos de red inalámbrica incluido `cfg80211` o bien reiniciar el sistema.

Para cargar el nuevo perfil regulatorio personalizado hay que ejecutar el comando `sudo iw reg set <código>` y su correcto funcionamiento puede comprobarse con `iw reg get`.

# Anexo C

# Definición de la estructura `ath5k_hw`

## C.1. `socketServer.c`

Código C.1: Código fuente de la prueba PF1 del lado del servidor

```
1  /* Driver state associated with an instance of a device */
2  struct ath5k_hw {
3          struct ath_common       common;
4
5          struct pci_dev          *pdev;
6          struct device           *dev;           /* for dma mapping */
7          int irq;
8          u16 devid;
9          void __iomem            *iobase;        /* address of the device */
10         struct mutex            lock;           /* dev-level lock */
11         struct ieee80211_hw     *hw;            /* IEEE 802.11 common */
12         struct ieee80211_supported_band sbands[IEEE80211_NUM_BANDS];
13         struct ieee80211_channel channels[ATH_CHAN_MAX];
14         struct ieee80211_rate   rates[IEEE80211_NUM_BANDS][AR5K_MAX_RATES];
15         s8                      rate_idx[IEEE80211_NUM_BANDS][AR5K_MAX_RATES];
16         enum nl80211_iftype     opmode;
17
18 #ifdef CONFIG_ATH5K_DEBUG
19         struct ath5k_dbg_info   debug;          /* debug info */
20 #endif /* CONFIG_ATH5K_DEBUG */
21
22         struct ath5k_buf        *bufptr;        /* allocated buffer ptr */
23         struct ath5k_desc       *desc;          /* TX/RX descriptors */
24         dma_addr_t              desc_daddr;     /* DMA (physical) address */
25         size_t                  desc_len;       /* size of TX/RX descriptors
               */
26
27         DECLARE_BITMAP(status, 4);
28 #define ATH_STAT_INVALID        0               /* disable hardware accesses
    */
29 #define ATH_STAT_PROMISC        1
30 #define ATH_STAT_LEDSOFT        2               /* enable LED gpio status */
31 #define ATH_STAT_STARTED        3               /* opened & irqs enabled */
32
33         unsigned int            filter_flags;   /* HW flags, AR5K_RX_FILTER_*
               */
34         struct ieee80211_channel *curchan;      /* current h/w channel */
35
36         u16                     nvifs;
37
38         enum ath5k_int          imask;          /* interrupt mask copy */
```

```
39
40          spinlock_t              irqlock;
41          bool                    rx_pending;     /* rx tasklet pending */
42          bool                    tx_pending;     /* tx tasklet pending */
43
44          u8                      bssidmask[ETH_ALEN];
45
46          unsigned int            led_pin,        /* GPIO pin for driving LED */
47                                  led_on;         /* pin setting for LED on */
48
49          struct work_struct      reset_work;     /* deferred chip reset */
50          struct work_struct      calib_work;     /* deferred phy calibration */
51
52          struct list_head        rxbuf;          /* receive buffer */
53          spinlock_t              rxbuflock;
54          u32                     *rxlink;        /* link ptr in last RX desc */
55          struct tasklet_struct   rxtq;           /* rx intr tasklet */
56          struct ath5k_led        rx_led;         /* rx led */
57
58          struct list_head        txbuf;          /* transmit buffer */
59          spinlock_t              txbuflock;
60          unsigned int            txbuf_len;      /* buf count in txbuf list */
61          struct ath5k_txq        txqs[AR5K_NUM_TX_QUEUES];       /* tx queues
                */
62          struct tasklet_struct   txtq;           /* tx intr tasklet */
63          struct ath5k_led        tx_led;         /* tx led */
64
65          struct ath5k_rfkill     rf_kill;
66
67          spinlock_t              block;          /* protects beacon */
68          struct tasklet_struct   beacontq;       /* beacon intr tasklet */
69          struct list_head        bcbuf;          /* beacon buffer */
70          struct ieee80211_vif    *bslot[ATH_BCBUF];
71          u16                     num_ap_vifs;
72          u16                     num_adhoc_vifs;
73          u16                     num_mesh_vifs;
74          unsigned int            bhalq,          /* SW q for outgoing beacons
                */
75                                  bmisscount,     /* missed beacon transmits */
76                                  bintval,        /* beacon interval in TU */
77                                  bsent;
78          unsigned int            nexttbtt;       /* next beacon time in TU */
79          struct ath5k_txq        *cabq;          /* content after beacon */
80
81          bool                    assoc;          /* associate state */
82          bool                    enable_beacon;  /* true if beacons are on */
83
84          struct ath5k_statistics stats;
85
86          struct ath5k_ani_state  ani_state;
87          struct tasklet_struct   ani_tasklet;    /* ANI calibration */
88
89          struct delayed_work     tx_complete_work;
90
91          struct survey_info      survey;         /* collected survey info */
92
93          enum ath5k_int          ah_imr;
94
95          struct ieee80211_channel *ah_current_channel;
96          bool                    ah_iq_cal_needed;
97          bool                    ah_single_chip;
98
99          enum ath5k_version      ah_version;
100         enum ath5k_radio        ah_radio;
101         u32                     ah_mac_srev;
102         u16                     ah_mac_version;
103         u16                     ah_phy_revision;
104         u16                     ah_radio_5ghz_revision;
105         u16                     ah_radio_2ghz_revision;
106
```

```
107 #define ah_modes                ah_capabilities.cap_mode
108 #define ah_ee_version           ah_capabilities.cap_eeprom.ee_version
109
110         u8                      ah_retry_long;
111         u8                      ah_retry_short;
112
113         u32                     ah_use_32khz_clock;
114
115         u8                      ah_coverage_class;
116         bool                    ah_ack_bitrate_high;
117         u8                      ah_bwmode;
118         bool                    ah_short_slot;
119
120         /* Antenna Control */
121         u32                     ah_ant_ctl[AR5K_EEPROM_N_MODES][AR5K_ANT_MAX];
122         u8                      ah_ant_mode;
123         u8                      ah_tx_ant;
124         u8                      ah_def_ant;
125
126         struct ath5k_capabilities ah_capabilities;
127
128         struct ath5k_txq_info   ah_txq[AR5K_NUM_TX_QUEUES];
129         u32                     ah_txq_status;
130         u32                     ah_txq_imr_txok;
131         u32                     ah_txq_imr_txerr;
132         u32                     ah_txq_imr_txurn;
133         u32                     ah_txq_imr_txdesc;
134         u32                     ah_txq_imr_txeol;
135         u32                     ah_txq_imr_cbrorn;
136         u32                     ah_txq_imr_cbrurn;
137         u32                     ah_txq_imr_qtrig;
138         u32                     ah_txq_imr_nofrm;
139
140         u32                     ah_txq_isr_txok_all;
141         u32                     ah_txq_isr_txurn;
142         u32                     ah_txq_isr_qcborn;
143         u32                     ah_txq_isr_qcburn;
144         u32                     ah_txq_isr_qtrig;
145
146         u32                     *ah_rf_banks;
147         size_t                  ah_rf_banks_size;
148         size_t                  ah_rf_regs_count;
149         struct ath5k_gain       ah_gain;
150         u8                      ah_offset[AR5K_MAX_RF_BANKS];
151
152
153         struct {
154                 /* Temporary tables used for interpolation */
155                 u8              tmpL[AR5K_EEPROM_N_PD_GAINS]
156                                         [AR5K_EEPROM_POWER_TABLE_SIZE];
157                 u8              tmpR[AR5K_EEPROM_N_PD_GAINS]
158                                         [AR5K_EEPROM_POWER_TABLE_SIZE];
159                 u8              txp_pd_table[AR5K_EEPROM_POWER_TABLE_SIZE *
                        2];
160                 u16             txp_rates_power_table[AR5K_MAX_RATES];
161                 u8              txp_min_idx;
162                 bool            txp_tpc;
163                 /* Values in 0.25dB units */
164                 s16             txp_min_pwr;
165                 s16             txp_max_pwr;
166                 s16             txp_cur_pwr;
167                 /* Values in 0.5dB units */
168                 s16             txp_offset;
169                 s16             txp_ofdm;
170                 s16             txp_cck_ofdm_gainf_delta;
171                 /* Value in dB units */
172                 s16             txp_cck_ofdm_pwr_delta;
173                 bool            txp_setup;
174                 int             txp_requested;  /* Requested tx power in dBm
                        */
```

```
175         } ah_txpower;
176
177         struct ath5k_nfcal_hist ah_nfcal_hist;
178
179         /* average beacon RSSI in our BSS (used by ANI) */
180         struct ewma             ah_beacon_rssi_avg;
181
182         /* noise floor from last periodic calibration */
183         s32                     ah_noise_floor;
184
185         /* Calibration timestamp */
186         unsigned long           ah_cal_next_full;
187         unsigned long           ah_cal_next_short;
188         unsigned long           ah_cal_next_ani;
189
190         /* Calibration mask */
191         u8                      ah_cal_mask;
192
193         /*
194          * Function pointers
195          */
196         int (*ah_setup_tx_desc)(struct ath5k_hw *, struct ath5k_desc *,
197                 unsigned int, unsigned int, int, enum ath5k_pkt_type,
198                 unsigned int, unsigned int, unsigned int, unsigned int,
199                 unsigned int, unsigned int, unsigned int, unsigned int);
200         int (*ah_proc_tx_desc)(struct ath5k_hw *, struct ath5k_desc *,
201                 struct ath5k_tx_status *);
202         int (*ah_proc_rx_desc)(struct ath5k_hw *, struct ath5k_desc *,
203                 struct ath5k_rx_status *);
204 };
```

# Anexo D

# Código fuente para la prueba PF1

## D.1. socketServer.c

Código D.1: Código fuente de la prueba PF1 del lado del servidor

```
1  #include <stdio.h>
2  #include <sys/socket.h>
3  #include <arpa/inet.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <unistd.h>
7  #include <netinet/in.h>
8
9  #define MAXPENDING 5    /* Max connection requests */
10 #define BUFFSIZE 32
11 void Die(char *mess) { perror(mess); exit(1); }
12 void HandleClient(int sock) {
13         char buffer[BUFFSIZE];
14         int received = -1;
15         /* Receive message */
16         if ((received = recv(sock, buffer, BUFFSIZE, 0)) < 0) {
17                 Die("Failed to receive initial bytes from client");
18         }
19         /* Send bytes and check for more incoming data in loop */
20         while (received > 0) {
21                 /* Send back received data */
22                 if (send(sock, buffer, received, 0) != received) {
23                         Die("Failed to send bytes to client");
24                 }
25                 /* Check for more data */
26                 if ((received = recv(sock, buffer, BUFFSIZE, 0)) < 0) {
27                         Die("Failed to receive additional bytes from client");
28                 }
29         }
30         close(sock);
31 }
32
33 int main(int argc, char *argv[]) {
34         int serversock, clientsock;
35         struct sockaddr_in echoserver, echoclient;
36
37         if (argc != 2) {
38         fprintf(stderr, "USAGE: echoserver <port>\n");
39         exit(1);
40         }
41         /* Create the TCP socket */
```

```
42          if ((serversock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
43          Die("Failed to create socket");
44          }
45          /* Construct the server sockaddr_in structure */
46          memset(&echoserver, 0, sizeof(echoserver));       /* Clear struct */
47          echoserver.sin_family = AF_INET;                  /* Internet/IP */
48          echoserver.sin_addr.s_addr = htonl(INADDR_ANY);   /* Incoming addr */
49          echoserver.sin_port = htons(atoi(argv[1]));       /* server port */
50          /* Bind the server socket */
51          if (bind(serversock, (struct sockaddr *) &echoserver,
52                                  sizeof(echoserver)) < 0) {
53                  Die("Failed to bind the server socket");
54          }
55          /* Listen on the server socket */
56          if (listen(serversock, MAXPENDING) < 0) {
57                  Die("Failed to listen on server socket");
58          }
59          /* Run until cancelled */
60          while (1) {
61                  unsigned int clientlen = sizeof(echoclient);
62                  /* Wait for client connection */
63                  if ((clientsock =
64                     accept(serversock, (struct sockaddr *) &echoclient,
65                            &clientlen)) < 0) {
66                          Die("Failed to accept client connection");
67                  }
68                  fprintf(stdout, "Client connected: %s\n",
69                                  inet_ntoa(echoclient.sin_addr));
70                  HandleClient(clientsock);
71          }
72  }
```

## D.2. socketClient.c

Código D.2: Código fuente de la prueba PF1 del lado del cliente

```
1  #include <stdio.h>
2  #include <sys/socket.h>
3  #include <arpa/inet.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <unistd.h>
7  #include <netinet/in.h>
8
9  #define BUFFSIZE 32
10 void Die(char *mess) { perror(mess); exit(1); }
11 int main(int argc, char *argv[]) {
12          int sock;
13          struct sockaddr_in echoserver;
14          char buffer[BUFFSIZE];
15          unsigned int echolen;
16          int received = 0;
17
18          if (argc != 4) {
19                  fprintf(stderr, "USAGE: TCPecho <server_ip> <word> <port>\n");
20                  exit(1);
21          }
22          /* Create the TCP socket */
23          if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
24                  Die("Failed to create socket");
25          }
26          /* Construct the server sockaddr_in structure */
27          memset(&echoserver, 0, sizeof(echoserver));       /* Clear struct */
28          echoserver.sin_family = AF_INET;                  /* Internet/IP */
29          echoserver.sin_addr.s_addr = inet_addr(argv[1]);  /* IP address */
```

```
30          echoserver.sin_port = htons(atoi(argv[3]));       /* server port */
31          /* Establish connection */
32          if (connect(sock,
33                      (struct sockaddr *) &echoserver,
34                      sizeof(echoserver)) < 0) {
35                  Die("Failed to connect with server");
36          }
37          /* Send the word to the server */
38          echolen = strlen(argv[2]);
39          if (send(sock, argv[2], echolen, 0) != echolen) {
40                  Die("Mismatch in number of sent bytes");
41          }
42          /* Receive the word back from the server */
43          fprintf(stdout, "Received: ");
44          while (received < echolen) {
45                  int bytes = 0;
46                  if ((bytes = recv(sock, buffer, BUFFSIZE-1, 0)) < 1) {
47                          Die("Failed to receive bytes from server");
48                  }
49                  received += bytes;
50                  buffer[bytes] = '\0';        /* Assure null terminated string
                        */
51                  fprintf(stdout, buffer);
52          }
53          fprintf(stdout, "\n");
54          close(sock);
55          exit(0);
56  }
```