UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN TECNOLOGÍAS ESPECÍFICAS DE TELECOMUNICACIÓN
MENCIÓN EN TELEMÁTICA

Implementación del Protocolo de Transporte Básico en un Software de Comunicación V2X

Autor:

D. Daniel Monje González

Tutor:

Dr. D. Juan Carlos Aguado Manzano

Valladolid, noviembre 2021

TÍTULO:	**Implementación del Protocolo de Transporte Básico en un Software de Comunicación V2X**
AUTOR:	**D. Daniel Monje González**
TUTOR:	**Dr. D. Juan Carlos Aguado Manzano**
DEPARTAMENTO:	**Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL

PRESIDENTE:	**Dr. D. Ignacio de Miguel Jiménez**
VOCAL:	**Dr. D. Ramón José Durán Barroso**
SECRETARIO	**Dr. D. Juan Carlos Aguado Manzano**
FECHA:	
CALIFICACIÓN:	

Resumen

El propósito de este proyecto es la implementación del Protocolo de Transporte Básico del estándar ITS G5 de la ETSI en el software de comunicación V2X conocido como OpenC2X, en el que el Grupo de Comunicaciones Ópticas lleva trabajando varios años. Para lograr esta meta se han realizado diversas tareas y estudios que complementaban diferentes ángulos del problema a resolver.

Durante el trabajo de fin de grado se realizó una investigación sobre como había evolucionado la implementación de 802.11p en Linux y la búsqueda de nuevos dispositivos comerciales que podrían mejorar el rendimiento del sistema. También se estudió el funcionamiento interno del programa OpenC2X para poder añadirle las características necesarias concretadas en el estándar del BTP. Una vez realizadas las modificaciones pertinentes, se comprobó que el funcionamiento era el esperado, y el trafico ITS se generaba y se recibía correctamente.

Palabras clave

ETSI ITS G5, BTP, 802.11p, OpenC2X, vehículo conectado, V2X.

Abstract

The purpose of this project is the implementation of the Basic Transport Protocol of ETSI ITS G5 standard in the V2X communication software known as OpenC2X. The research group “Grupo de Comunicaciones Ópticas” has been working with this software for several years. In order to achieve the main goal, different studies and tasks which attack the same problem from different angles were carried out.

During the final degree project, an investigation was done about how the 802.11p implementation in Linux has evolved and new commercial devices that could improve system performance of radio connection with 802.11p were looked for. However, the main task of this work has been to understand the structure of the OpenC code in order to add the necessary characteristics specified in the BTP standard. Once the pertinent modifications had been made, it was verified that the operation was as expected, and the ITS traffic was generated and received correctly.

Keywords

ETSI ITS G5, BTP, 802.11p, OpenC2X, connected vehicle, V2X.

Índice

Índice v

Índice de Figuras vii

Glosario de Términos xi

Capítulo 1. Introducción 1

1.1. Antecedentes y motivaciones 1

1.2. Objetivos 3

1.3. Fases del proyecto 4

Capítulo 2. Estado del arte 7

2.1. Actualización del estado de la implementación de 802.11p en el kernel de Linux 7

2.1.1. Modificaciones implementadas a partir de la versión 4.4 del kernel de Linux 8

2.1.2. Modificaciones aún no implementadas 11

2.2. Actualización del kernel de Linux 14

2.3. Problemas con el CRDA 16

2.4. Opciones de dispositivos 802.11p en el mercado 17

2.5. Mejora en el sistema GPS 18

Capítulo 3. Estudio del software OpenC2X 21

3.1. Arquitectura 22

3.2. Funcionamiento y estructura de comunicación 24

3.3. Encapsulamiento de Datos 26
3.4. Ruta del mensaje 28
3.4.1. Envío 29
3.4.2. Recepción 32
Capítulo 4. El protocolo BTP 35
4.1. Estructura del Paquete BTP 36
4.2. Cabecera del BTP 37
4.2.1. Cabecera BTP tipo A 37
4.2.2. Cabecera BTP tipo B 38
4.3. Servicios de datos BTP 38
4.3.1. La primitiva BTP-Data.request 38
4.3.2. La primitiva BTP-Data.indication 40
4.4. Operaciones del protocolo 40
4.4.1. Operaciones en el origen 40
4.4.2. Operaciones en el destino 41
Capítulo 5. Implementación del BTP en OpenC2X 43
5.1. Revisión general del fichero btp.cpp 46
5.2. Primitivas BTP_REQUEST y BTP_INDICATION 47
5.3. Configuración del servicio BTP 48
5.4. Envío de PDUs ITS por OpenC2X 50
5.5. Recepción de PDUs ITS por OpenC2X 52
5.6. Problema implementación original OpenC2X 54
Capítulo 6. Pruebas de comunicación 57
Capítulo 7. Conclusiones y Líneas Futuras 61
Referencias 63

Índice de Figuras

Figura 1. Ejemplo de comunicación V2X 1

Figura 2. Imagen conceptual de una red V2XEjemplo teórico de una red V2X.............. 2

Figura 3. Modificación del fichero drivers/net/wireless/ath/ath9k/ath9k.h [5]................ 8

Figura 4. Modificación del fichero drivers/net/wireless/ath/ath9k/debug.c [5] 9

Figura 5. Modificación del fichero drivers/net/wireless/ath/ath9k/htc_drv_init.c [5]...... 9

Figura 6. Modificación del fichero drivers/net/wireless/ath/ath9k/hw.c [5] 10

Figura 7. Modificación del fichero drivers/net/wireless/ath/ath9k/hw.c [5] 10

Figura 8. Modificación del fichero drivers/net/wireless/ath/ath9k/recv.c [5] 11

Figura 9. Modificación del fichero drivers/net/wireless/ath/ath9k/common-init.c [5] .. 12

Figura 10. Modificación del fichero drivers/net/wireless/ath/ath9k/ani.c [5] 12

Figura 11. Modificación del fichero drivers/net/wireless/ath/ath9k/hw.h [5]................ 13

Figura 12. Modificación del fichero drivers/net/wireless/ath/ath9k/main.c [5] 13

Figura 13. Modificación del fichero drivers/net/wireless/ath/regd.c [5]........................ 14

Figura 14. Posición del vehículo en el carril calculada mediante Dead Reckoning....... 18

Figura 15. Protocolos de la pila ITS en cada proyecto. [18] 21

Figura 16. Arquitectura OpenC2X derivada de la pila de protocolos del estándar ETSI ITS G5. 22

Figura 17. Esquema de comunicación de los diferentes módulos en OpenC2X............ 24

Figura 18. Fichero common/utility/CommunicationSender.cpp [6] 25

Figura 19. Fichero common/utility/CommunicationReceiver.cpp [6] 25

Figura 20. Receptor del DENM para el DCC en el constructor de la clase DenService, /denm/src/denservice.cpp. [6] 26
Figura 21. Receptor del CAM para el DCC en el constructor de la clase CaService, /cam/src/caservice.cpp. [6] 26
Figura 22. Emisor del DCC para CAM y DENM en el constructor de la clase DCC, /dcc/src/dcc.cpp. [6] 26
Figura 23. Fichero /common/buffers/data.proto 27
Figura 24. Fichero /denm/src/denservice.cpp 28
Figura 25. Función send() del fichero cam/src/caservice.cpp 29
Figura 26. Diagrama comunicación entre CAM y DCC 30
Figura 27. Inicialización de threads que ejecutan las funciones de recepción de otros módulos, fichero dcc/src/dcc.cpp 30
Figura 28. Función receiveFromCa2() en dcc/src/dcc.cpp. 30
Figura 29. Función sendQueuedPackets() en el fichero dcc/src/dcc.cpp 31
Figura 30. Función sendWithGeoNet() en el fichero dcc/src/SendToHardwareViaMac.cpp 31
Figura 31. Función receiveFromHw2() en el fichero dcc/src/dcc.cpp 33
Figura 32. Función receiveWithGeoNetHeader() en el fichero dcc/src/ReceiveFromHardwareViaMac.cpp 34
Figura 33. Función receive() en el fichero denm/src/denservice.cpp 34
Figura 34. SAPs, SDUs y PDUs relevantes para el BTP [8] 36
Figura 35. Estructura del paquete BTP. [8] 36
Figura 36. Codificación de los tipos de cabecera BTP dentro del campo Next Header en la cabecera GeoNetworking. [8] 37
Figura 37. Formato de la Cabecera BTP-A 37
Figura 38. Formato de la Cabecera BTP-A 38
Figura 39. Formato de la Cabecera BTP-B 38
Figura 40. Formato de la Cabecera BTP-B 38
Figura 41. Parámetros de la primitiva BTP-Data.request 39

Figura 42. Parámetros de la primitiva BTP-Data.indication 40
Figura 43. Configuración de los campos para una cabecera BTP-A 41
Figura 44. Configuración de los campos para una cabecera BTP-B 41
Figura 45. Configuración de parámetros para la primitiva GN-Data.request para solicitar el envío de un paquete de GeoNetworking.. 41
Figura 46. Configuración de parámetros para la primitiva GN-Data.request para solicitar el envío de un paquete de GeoNetworking.. 42
Figura 47. Nuevo esquema de comunicación de los diferentes módulos en OpenC2X con el BTP 44
Figura 48. Diagrama de puertos de OpenC2X con el BTP. 45
Figura 49. Constructor de la clase BTPService, btp/src/btp.cpp 46
Figura 50. Función init() en el BTP, btp/src/btp.cpp... 46
Figura 51. Función main() en el BTP, btp/src/btp.cpp 47
Figura 52. Ficheros common/buffers/btpRequest.proto y common/buffers/btpIndication.proto. 48
Figura 53. Fichero de configuración del CAM cam/config/config.xml 49
Figura 54. Estructura para la configuración del servicio CAM..................................... 49
Figura 55. Función modificada send() del fichero cam/src/caservice.cpp 50
Figura 56. Función receiveFromCamSendToDcc() del fichero btp/src/btp.cpp 52
Figura 57. Modificación en *receiveWithGeoNetHeader()* 53
Figura 58. Función receiveFromDccSendToServices() del fichero btp/src/btp.cpp 54
Figura 59. Estructura de comunicación de la capa transversal del DCC (DCC_CROSS) 55
Figura 60. Datos correspondientes al control de congestión 56
Figura 61. Interfaz del OpenC2X con el servicio del BTP a la derecha........................ 58
Figura 62. Tráfico ITS generado con OpenC2X 58
Figura 63. Cabecera BTP construida por el módulo BTPService de OpenC2X para CAM 59

Figura 64. Cabecera BTP construida por el módulo BTPService de OpenC2X para DENM 59

Glosario de Términos

BTP. Basic Transport Protocol

BTP-PDU. Basic Transport Protocol Protocol Data Unit

BTP-SAP. Basic Transport Protocol Service Access Point

CRDA. Central Regulatory Domain Agent

DSRC. Digital Short Range Communications

FL. Facilities Layer

GN-PDU. GeoNetworking protocol PDU exchanged between peer entities of the GeoNetworking protocol

GN-SAP. GeoNetworking Service Access Point

ITS. Intelligent Transport Systems

ITS-FL-PCI. ITS Facilities layer Protocol Control Information

ITS-FL-PDU. ITS Facilities layer Protocol Data Unit

ITS-FL-SDU. ITS Facilities layer Service Data Unit

LL. Link Layer

OCB. Outside the Context of a BSSID

PCI. Protocol Control Information

PDU. Protocol Data Unit

V2I. Vehicle to infrastructure

V2V. Vehicle to vehicle

V2X. Vehicle to everything

Capítulo 1. Introducción

1.1. Antecedentes y motivaciones

Durante los últimos años los medios de comunicación han prestado una gran atención a los vehículos inteligentes. El despliegue técnico que requiere un vehículo autónomo actual, con innumerables cámaras y sensores repartidos por todo el vehículo, hace que sea un producto muy vendible desde el punto de vista de los medios.

Sin embargo, esta forma de ver los vehículos autónomos solo abarca una pequeña parte de su compleja realidad. Al fin y al cabo, los sensores que incorporan los vehículos les ayudan a tener una idea de lo que está ocurriendo en su entorno, pero debido a las limitaciones técnicas actuales dicha percepción suele tener todavía importantes lagunas. Por ello hay una serie de iniciativas como el estándar ITS G5 de la ITU-T que promueven una tecnología menos llamativa que la del vehículo autónomo en sí, pero igual de importante para su futuro desarrollo: se trata de la comunicación entre vehículos de información relevante captada por sus sensores, o acerca de su situación concreta, que permita al resto de vehículos tener una percepción mucho más definida de lo que está ocurriendo a su alrededor.

Figura 1. Ejemplo de comunicación V2X

El funcionamiento de esta tecnología se basa en el intercambio de información a través de DSRC (digital short-range communications) o 5G, esto es, cualquier conexión inalámbrica de corto alcance con ancho de banda y latencia aceptables. Cuando el vehículo se cruce en la carretera con otro vehículo o con una infraestructura conectada, podrá tanto recibir la información de eventos registrados por el otro extremo de la comunicación como enviarle los propios eventos detectados por sí mismo. Si todos los vehículos llevaran implementadas funciones V2X (comunicación vehículo a todo), se podría formar redes de comunicación vehicular locales en cada una de las carreteras y vías en las que circulen los vehículos, compartiendo todos ellos los diferentes eventos detectados que pueden ser de interés para el resto. De esta manera pueden mostrarse avisos al conductor mucho antes para cualquier circunstancia que pueda ocurrir en la carretera, como por ejemplo evitar accidentes en cruces con baja visibilidad a través de la transmisión de presencia de los diferentes vehículos, inicio de maniobras evasivas o frenado de emergencia en el caso de que un vehículo que vaya por delante tengo un accidente o cambie bruscamente de velocidad, y un largo etcétera.

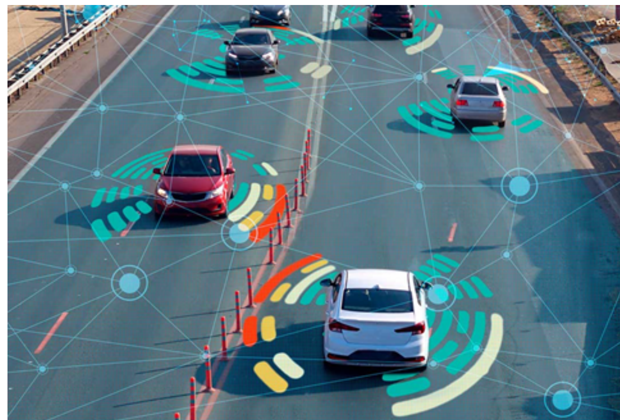

Figura 2. Imagen conceptual de una red V2XEjemplo teórico de una red V2X

El Grupo de Comunicaciones Ópticas lleva varios años trabajando en un proyecto V2X. Concretamente se han realizado diversos trabajos para estudiar la pila del estándar ETSI-ITS G5. Originalmente este estándar trabajaba con 802.11p como capa física y de enlace, un estándar del popularmente conocido como Wi-Fi que soporta el modo OCB (Outside the Context of a BSSID). Este modo permite realizar el envío de comunicación sin tener que pasar por las fases de autenticación, lo que reduce mucho los tiempos de comunicación, permitiendo comunicaciones de muy baja latencia con retardos del orden de milisegundos. La baja latencia en la mayoría de casos prácticos en comunicaciones vehiculares es un requisito fundamental. Como ejemplo, dos coches que se cruzan a 120

km/h tienen un tiempo para intercambiar la información muy escaso. El modo OCB permite que cualquier dispositivo se una a la red, sin identificarse previamente, y la comunicación se espera que sea básicamente broadcast, de tal manera que todo el que la escuche está interesado en ella.

Este proyecto parte del trabajo realizado por mis compañeros de años anteriores, Javier Fernández Pastrana [1], Pilar Sánchez Martín [2], Alejandro Lobo González [3] y Roberto Herreras Babón [4]. Javier se encargó de estudiar la implementación de 802.11p en el kernel de Linux y de instalar todo el software necesario para llevar a cabo el proyecto. Pilar y Alejandro se encargaron del estudio del estándar ETSI-ITS G5 y la búsqueda de un proyecto de código abierto que funcionara sobre Linux, con la que se optó por la utilización de OpenC2X. Roberto se encargó de hacer las instalaciones correspondientes de software y herramientas necesarios en ordenadores reducidos, y la realización de diferentes pruebas de campo con antenas Wi-Fi y dispositivos compatibles con 802.11p.

Partiendo del trabajo de mis compañeros el objetivo que me planteo en este trabajo fin de grado es realizar mejoras en diferentes aspectos, aumentar la funcionalidad del software sobre el que se ha estado trabajando en los últimos años y hacer una revisión de la evolución de algunos aspectos de este tipo de comunicación, dado que desde la finalización del primer proyecto realizado por Javier Fernández Pastrana [1] en el año 2017, es razonable pensar que se han producido cambios significativos tanto en los sistema operativos como en la comercialización de dispositivos que incorporan esta tecnología.

1.2. Objetivos

El objetivo principal de este proyecto ha sido la integración del protocolo BTP (Basic Transport Protocol) según es definido por el estándar ETSI ITS G5 en OpenC2X. OpenC2X es un software de código abierto que dispone de una implementación incompleta de la pila ETSI-ITS G5, pues faltan protocolos por añadir, como son el Protocolo de Transporte Básico (BTP) y el GeoNetworking.

Durante los trabajos realizados en cursos anteriores se llevaron a cabo múltiples estudios sobre diferentes proyectos de código abierto para comunicación V2X, para

determinar con cual se podría completar antes la pila ITS G5. Se llegó a la conclusión de que la mejor opción era la utilización de OpenC2X, pero no se realizaron mejoras en su funcionamiento. Por esta razón, se decidió que para este proyecto se comenzaría con las mejoras en el proyecto. La primera mejora sería la adición del BTP para ir completando la capa de transporte, posteriormente se realizarían la integración del GeoNetworking y otros servicios.

Aunque el desarrollo del BTP es el objetivo principal, no es el único. Esta es la lista de objetivos completa del proyecto:

- Actualización del estado de la implementación de 802.11p en Linux, pues con el paso del tiempo se ha ido completando y es necesario saber el estado actual por si hubiera que realizar modificaciones en el kernel.
- La búsqueda de opciones de mercado para mejorar el equipamiento utilizado en años anteriores.
- La propuesta de mejoras en el sistema GPS, pues en proyectos anteriores se ha trabajado con dispositivos de baja calidad y es necesaria la adquisición de un sistema más preciso y eficiente.
- Finalmente, la integración del estándar BTP en OpenC2X y la realización de pruebas de comunicación para comprobar el correcto funcionamiento.

1.3. Fases del proyecto

Para la consecución de todos los objetivos planteados se dividió el proyecto en diferentes fases. La primera fase fue la consulta de los cambios necesarios en el kernel de Linux para permitir la compatibilidad con el modo OCB de 802.11p, para esto se siguió el trabajo realizado por Javier Fernández Pastrana [1] y se hizo un estudio de las nuevas versiones del kernel de Linux en las que se han incluido algunos de los cambios que se proponen en el trabajo ya mencionado. Además de esto, también se buscaron soluciones hardware en el mercado para mejorar los dispositivos que el GCO ya tenía en su poder y se buscaron alternativas para el sistema GPS actual.

La segunda fase del proyecto fue la compilación e instalación del kernel 5.4, junto con todas las herramientas software especificadas en el proyecto de Roberto Herreras [4], en un ordenador x86 y una HummingBoard-2 (ARM) proporcionados por el Grupo de

Comunicaciones Ópticas de la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universidad de Valladolid.

En la tercera fase se realizó un estudio profundo de estándar BTP [9] y de los estándares con los que se veía implicado, con el fin de poder posteriormente realizar una implementación ajustada al estándar en OpenC2X.

La cuarta fase fue la del estudio profundo de OpenC2X para comprender los aspectos fundamentales en los que se vería implicado el BTP.

La quinta fase fue íntegramente de programación, pues se modificó el código fuente de OpenC2X para que el BTP funcionara correctamente. No solo se modificaron ficheros, sino que también se crearon algunos nuevos.

La sexta y última fase consistió en la realización de las pruebas de comunicación entre los ordenadores utilizados en años anteriores con el nuevo software desarrollado.

Capítulo 2. Estado del arte

Antes de la implementación del estándar BTP en el software de comunicación V2X conocido como OpenC2X, es necesario mencionar diferentes actualizaciones de los trabajos que realizaron mis compañeros en años anteriores. Se revisará el estado de la implementación de 802.11p en Linux, se revisarán algunos problemas que surgieron durante el desarrollo de este proyecto y se propondrán mejoras a algunas cuestiones sugeridas en anteriores proyectos.

2.1. Actualización del estado de la implementación de 802.11p en el kernel de Linux

Como ya explicó Javier Fernández Pastrana [1], con el fin de poder utilizar un sistema Linux con 802.11p hay que realizar una serie de modificaciones en el kernel del sistema operativo. Concretamente, se debe modificar el driver para tarjetas Wi-Fi Atheros ATH9K, que permiten la utilización del modo OCB con estos dispositivos. Las tarjetas Wi-Fi Atheros eran las únicas que en su momento implementaban a nivel hardware las frecuencias y el modo OCB en ellas. Es por esta razón que la modificación solo se hace de este driver, y también explica el interés en buscar nuevas tarjetas que se incluyen más adelante en este capítulo.

Nuestro punto de partida es la situación en la que acabaron los anteriores trabajos fin de grado y máster. Concretamente, Javier [1] realizó un estudio de las diferentes modificaciones necesarias para la versión 3.15 del kernel de Linux. Roberto, Pilar y Alejandro [4, 2, 3] trabajaron a partir de lo que él hizo y revisaron versiones más recientes en busca de nuevos cambios, particularmente la versión que hay actualmente instalada en los ordenadores del laboratorio es el kernel 4.2. En mi caso, el trabajo consistió en una

revisión de todas la versiones de kernel desde el 4.2 hasta el 5.4 comprobando cuáles han sido la modificaciones incorporadas respecto al modo OCB.

Una vez realizado dicho estudio, se pudo comprobar que las modificaciones se habían realizado de manera parcial, hasta la versión 4.4 no había ninguna modificación implementada, pero a partir de esta, se habían incluido algunas que ya quedaron integradas de forma permanente.

2.1.1. Modificaciones implementadas a partir de la versión 4.4 del kernel de Linux

Las siguientes modificaciones que se describen del código del kernel, están permanentemente integradas en cualquier kernel Linux que se haya basado en la rama principal, la de Torvalds [1], concretamente, están disponibles a partir de la versión 4.4 incluida. Los desarrolladores de esta versión modificaron líneas de código en diferentes ficheros con el objetivo de preparar el kernel para la futura llegada del modo OCB, dado que no está implementado completamente, se tratan de ajustes mínimos en el driver para que tenga en cuenta que ahora hay un modo de operación nuevo.

2.1.1.1. Modificación del fichero drivers/net/wireless/ath/ath9k/ath9k.h:

```
struct ath9k_vif_iter_data {
    u8 hw_macaddr[ETH_ALEN]; /* address of the first vif */
    u8 mask[ETH_ALEN]; /* bssid mask */
    bool has_hw_macaddr;
    u8 slottime;
    bool beacons;

    int naps;      /* number of AP vifs */
    int nmeshes;   /* number of mesh vifs */
    int nstations; /* number of station vifs */
    int nadhocs;   /* number of adhoc vifs */
++      int nocbs;      /* number of OCB vifs */
    int nbcnvifs;  /* number of beaconing vifs */
    struct ieee80211_vif *primary_beacon_vif;
    struct ieee80211_vif *primary_sta;
};
```

Figura 3. Modificación del fichero drivers/net/wireless/ath/ath9k/ath9k.h [5]

Una modificación bastante simple pero importante, al añadir esa variable podremos configurar nuestra tarjeta con varias interfaces virtuales para operar en modo OCB. Esta estructura contiene datos importantes de las interfaces virtuales que creemos para nuestra tarjeta, tales como el número de puntos de acceso, la dirección MAC, la máscara del BSSID, etc.

2.1.1.2. Modificación del fichero drivers/net/wireless/ath/ath9k/debug.c:

```
static int read_file_misc(struct seq_file *file, void *data)
{
    ...
    iter_data.nstations, iter_data.nmeshes, iter_data.nwds);
--      seq_printf(file, " ADHOC: %i TOTAL: %hi BEACON-VIF: %hi\n",
--      iter_data.nadhocs, sc->cur_chan->nvifs,sc->nbcnvifs);
++      seq_printf(file, " ADHOC: %i OCB: %i TOTAL: %hi BEACON-VIF: %hi\n",
++      iter_data.nadhocs, iter_data.nocbs, sc->cur_chan->nvifs,
++      sc->nbcnvifs);
}
```

Figura 4. Modificación del fichero drivers/net/wireless/ath/ath9k/debug.c [5]

El fichero “drivers/net/wireless/ath/ath9k/debug.c” es un fichero que aporta depuración, en este caso, ya viene modificado para recibir realimentación del número de interfaces de tipo OCB.

2.1.1.3. Modificación del fichero drivers/net/wireless/ath/ath9k/htc_drv_init.c:

```
static void ath9k_set_hw_capab(struct ath9k_htc_priv *priv, struct
ieee80211_hw *hw){
    ...
    hw->wiphy->interface_modes =
      BIT(NL80211_IFTYPE_STATION) | BIT(NL80211_IFTYPE_ADHOC) |
      BIT(NL80211_IFTYPE_AP) | BIT(NL80211_IFTYPE_P2P_GO) |
      BIT(NL80211_IFTYPE_P2P_CLIENT) | BIT(NL80211_IFTYPE_MESH_POINT)
++      | BIT(NL80211_IFTYPE_OCB);
    ...
}
```

Figura 5. Modificación del fichero drivers/net/wireless/ath/ath9k/htc_drv_init.c [5]

Este es un fichero de inicialización del driver, en este caso simplemente hay que tener en cuenta el modo OCB en el modo de funcionamiento que se le asigna a la interfaz Wi-Fi.

2.1.1.4. Modificación del fichero drivers/net/wireless/ath/ath9k/hw.c:

```
static void ath9k_hw_set_operating_mode(struct ath_hw *ah, int opmode)
{
  switch(opmode){
      ...
++        case NL80211_IFTYPE_OCB:
    case NL80211_IFTYPE_ADHOC:
      ...
}
```

Figura 6. Modificación del fichero drivers/net/wireless/ath/ath9k/hw.c [5]

Lo mismo que en el caso del fichero anterior, modificación para tener en cuenta el modo OCB.

2.1.1.5. Modificación del fichero drivers/net/wireless/ath/ath9k/init.c:

```
static void ath9k_set_hw_capab(struct ath_softc *sc, struct ieee80211_hw
*hw)
{
  ...
  if (!config_enabled(CONFIG_ATH9K_TX99)) {
        hw->wiphy->interface_modes =
            BIT(NL80211_IFTYPE_P2P_GO) |
            BIT(NL80211_IFTYPE_P2P_CLIENT) |
            BIT(NL80211_IFTYPE_AP) |
            BIT(NL80211_IFTYPE_STATION) |
            BIT(NL80211_IFTYPE_ADHOC) |
            BIT(NL80211_IFTYPE_MESH_POINT) |
            BIT(NL80211_IFTYPE_WDS) |
++            BIT(NL80211_IFTYPE_OCB);
  ...
}
```

Figura 7. Modificación del fichero drivers/net/wireless/ath/ath9k/hw.c [5]

Mismo procedimiento que los 2 anteriores.

2.1.1.6. Modificación del fichero drivers/net/wireless/ath/ath9k/recv.c:

```
u32 ath_calcrxfilter(struct ath_softc *sc) {
  ...
  rfilt |= ATH9K_RX_FILTER_MYBEACON;
--  else
++  else if (sc->sc_ah->opmode != NL80211_IFTYPE_OCB)
    rfilt |= ATH9K_RX_FILTER_BEACON;
  ...
}
```

Figura 8. Modificación del fichero drivers/net/wireless/ath/ath9k/recv.c [5]

Lo mismo que los 3 anteriores.

2.1.2. Modificaciones aún no implementadas

Los siguientes ajustes no se encuentran aún en la rama principal del kernel, dado que no se consideran modificaciones críticas para el funcionamiento de este. Más bien, son características que se añaden para poder adaptar nuestro software y que sea compatible con el estándar ITS G5 y no tanto con el funcionamiento del modo OCB.

2.1.2.1. Modificación del fichero drivers/net/wireless/ath/ath9k/common-init.c:

En este fichero se definen los canales en los que puede trabajar nuestra tarjeta Wi-Fi, no podremos conectarnos a ningún canal que no esté definido aquí. Los canales se definen otorgándoles el valor de la frecuencia como primer parámetro, y un índice para identificarlos como segundo parámetro. Lo que conseguimos con esto es añadir todos los canales definidos en el estándar ITS-G5, y los diferentes grupos que están destinados a proporcionar diferentes servicios, como el G5A, el G5B y el G5D.

```
static const struct ieee80211_channel ath9k_5ghz_chantable[] = {
  ...
  CHAN5G(5785, 35), /* Channel 157 */
  CHAN5G(5805, 36), /* Channel 161 */
  CHAN5G(5825, 37), /* Channel 165 */

  ++    CHAN5G(5850, 38), /* Channel 170 */
  ++    /* ITA-G5B */
  ++    CHAN5G(5855, 39), /* Channel 171 */
  ++    CHAN5G(5860, 40), /* Channel 172 */
  ++    CHAN5G(5865, 41), /* Channel 173 */
  ++    CHAN5G(5870, 42), /* Channel 174 */
  ++    /* ITS-G5A */
  ++    CHAN5G(5875, 43), /* Channel 175 */
  ++    CHAN5G(5880, 44), /* Channel 176 */
  ++    CHAN5G(5885, 45), /* Channel 177 */
  ++    CHAN5G(5890, 46), /* Channel 178 */
  ++    CHAN5G(5895, 47), /* Channel 179 */
  ++    CHAN5G(5900, 48), /* Channel 180 */
  ++    CHAN5G(5905, 49), /* Channel 181 */
  ++    /* ITS-G5D */
  ++    CHAN5G(5910, 50), /* Channel 182 */
  ++    CHAN5G(5915, 51), /* Channel 183 */
  ++    CHAN5G(5920, 52), /* Channel 184 */
  ++    CHAN5G(5925, 53), /* Channel 185 */
};
```

Figura 9. Modificación del fichero drivers/net/wireless/ath/ath9k/common-init.c [5]

2.1.2.2. Modificación del fichero drivers/net/wireless/ath/ath9k/ani.c:

```
if (is_scanning ||
        (ah->opmode != NL80211_IFTYPE_STATION &&
++          ah->opmode != NL80211_IFTYPE_OCB &&
         ah->opmode != NL80211_IFTYPE_ADHOC)) {
```

Figura 10. Modificación del fichero drivers/net/wireless/ath/ath9k/ani.c [5]

Este a pesar de estar relacionado con el modo OCB, aún no está implementado, por lo que hay que añadir la posibilidad de que la tarjeta se encuentre en modo OCB cuando está escaneando las posibles redes a las que conectarse.

2.1.2.3. Modificación del fichero drivers/net/wireless/ath/ath9k/hw.h:

```
#define ATH9K_RSSI_BAD          -128
--  #define ATH9K_NUM_CHANNELS  38
++  #define ATH9K_NUM_CHANNELS  54
```

Figura 11. Modificación del fichero drivers/net/wireless/ath/ath9k/hw.h [5]

Al haber añadido nuevos canales, debemos modificar también la constante que establece cuantos hay.

2.1.2.4. Modificación del fichero drivers/net/wireless/ath/ath9k/main.c:

```
static void ath9k_vif_iter(struct ath9k_vif_iter_data *iter_data,
switch (vif->type) {
++  case NL80211_IFTYPE_OCB:
++      iter_data->nocbs++;
++      break;

void ath9k_calculate_summary_state(struct ath_softc *sc,
++      else if (iter_data.nocbs)
++          ah->opmode = NL80211_IFTYPE_OCB;
                ...
++  if ((iter_data.nstations + iter_data.nadhocs + iter_data.nmeshes +
++         iter_data.nocbs) > 0)
    ah->imask |= ATH9K_INT_TSFOOR;

static void ath9k_bss_info_changed(struct ieee80211_hw *hw,
++  if ((changed & BSS_CHANGED_IBSS) ||
++        (changed & BSS_CHANGED_OCB)) {
++      memcpy(common->curbssid, bss_conf->bssid, ETH_ALEN);
++      common->curaid = bss_conf->aid;
++      ath9k_hw_write_associd(sc->sc_ah);
++  }
```

Figura 12. Modificación del fichero drivers/net/wireless/ath/ath9k/main.c [5]

El caso de este fichero es un poco especial, dado que hay modificaciones que sí están integradas y otras que no. No obstante, es recomendable revisarlo bien para comprobar qué tenemos que añadir y qué no. Igual que los casos anteriores, hay que tener en cuenta el modo OCB para varias comprobaciones y de eso se encarga el código añadido.

2.1.2.5. Modificación del fichero drivers/net/wireless/ath/regd.c:

```
static int __ath_regd_init(struct ath_regulatory *reg);
--  #define ATH9K_5GHZ_5470_5850    REG_RULE(5470-10, 5850+10, 80, 0, 30,\
++  #define ATH9K_5GHZ_5470_5925    REG_RULE(5470-10, 5925+10, 80, 0, 30,\
                NL80211_RRF_NO_IR)
--  #define ATH9K_5GHZ_5725_5850    REG_RULE(5725-10, 5850+10, 80, 0, 30,\
++  #define ATH9K_5GHZ_5725_5925    REG_RULE(5725-10, 5925+10, 80, 0, 30,\
                  NL80211_RRF_NO_IR)
#define ATH9K_5GHZ_ALL      ATH9K_5GHZ_5150_5350, \
--              ATH9K_5GHZ_5470_5850
++              ATH9K_5GHZ_5470_5925
#define ATH9K_5GHZ_NO_MIDBAND   ATH9K_5GHZ_5150_5350, \
--              ATH9K_5GHZ_5725_5850
++              ATH9K_5GHZ_5470_5925
```

Figura 13. Modificación del fichero drivers/net/wireless/ath/regd.c [5]

Aquí lo que se hace es modificar la banda de 5GHz que estaba especificada para la tarjeta, en lugar de llegar hasta los 5850 MHz ahora deberá llegar hasta los 5925 MHz que marca el estándar.

2.2. Actualización del kernel de Linux

Los últimos trabajos realizados en el grupo con las tarjetas Atheros se han llevado a cabo con la versión de kernel 4.2. De hecho, originalmente Javier Fernández Pastrana [1] comenzó con la versión 3.15 que se actualizó rápidamente a la 4.2. Pero como hemos visto, a partir de la versión 4.4 se introducen de forma permanente algunas de las modificaciones que requieren el estándar 802.11p, y además es importante actualizar los kernels una vez estos quedan excesivamente anticuados.

Por ello, una vez se estudiaron los cambios necesarios en el kernel de Linux para soportar el modo OCB, se decidió que se instalaría la versión 5.4. La versión de kernel se instaló en dos modelos de ordenador totalmente diferente:

1. Se procedió con la compilación del kernel 5.4 en un x86 en el laboratorio del GCO. En este tipo de arquitectura no se encontraron problemas relevantes durante la compilación.
2. Se intento también compilar para un miniordenador HummingBoard 2 de SolidRun que utiliza una arquitectura ARM. La idea de utilizar este

dispositivo era la versatilidad que nos ofrecía, un ordenador pequeño, que podría ponerse en cualquier zona del coche sin ser un estorbo. Sin embargo, en este caso la compilación del kernel resultó problemática.

La compilación del kernel en la Hummingboard plantea una serie de problemas que no tienen las arquitecturas x86. Por un lado, al ser un producto ARM de una marca concreta hay que descargar la versión del kernel de la página del proveedor, en este caso SolidRun. Posteriormente se deben realizar las modificaciones pertinentes en el código fuente del kernel, y seguir los pasos mencionados en [4] y [1] para la compilación. En mi caso se intentó realizar una compilación cruzada, desde un x86 para un ARM, por lo que a la hora de ejecutar la compilación se ha de especificar la arquitectura del dispositivo y el compilador cruzado que se va a utilizar. Se añaden los campos *ARCH=arm CROSS_COMPILE=arm-linux-gnueabifh-* al comando *make* que se realiza en el proceso. Con el primer campo se especifica la arquitectura de destino, y con el segundo el compilador que se va a utilizar, en este caso un compilador cruzado que ya estaba instalado en el PC.

La última versión de Linux estable disponible para este dispositivo en la página web del fabricante cuenta con un kernel 4.9, bastante más antiguo que la versión que queríamos introducir. El proceso de compilación que debería haber sido inmediato resultó estar plagado de complicaciones. Finalmente se realizó una consulta a Javier Fernández Patrana, que actualmente tiene gran experiencia en compilaciones con Yocto y en compilación de kernels de Linux.

Una de las posibles razones por las que aparecen tantos errores, es que al ser un kernel mucho más actual que el que ya se tenía, la parte de usuario y el nuevo kernel no eran capaces de comunicarse eficiente debido a que no eran compatibles. Esto es, el Linux del fabricante era un 4.9, mientras que el que se quería compilar era un 5.4. Para que todo funcionara bien, se debía respetar la parte de usuario del Linux original del fabricante y hacerlo funcionar con el kernel 5.4. Este procedimiento se puede realizar con la mayoría de los Linux, siempre y cuando ambas partes sean compatibles, lo cual es cierta para versiones suficientemente cercanas.

Antes de proceder con el descarte de actualizar el Linux de la Hummingboard, se probó la instalación de un sistema operativo Linux con un kernel ya modificado para

soportar 802.11p, que se había generado con la herramienta Yocto [29]. Esta herramienta permite la creación de una distribución Linux personalizada, añadiendo solo las características que el usuario necesite y así creando un sistema operativo mucho más ligero. Una vez se instaló la imagen nos dimos cuenta de que, por el momento, no era posible la ejecución de OpenC2X, dado que el sistema no contaba con el suficiente espacio para almacenar el programa y, aún más importante, no disponía de las librerías necesarias para ejecutarlo. Al ser un Linux creado a partir de Yocto la instalación de nuevas librerías y herramientas suponía un reto largo y tedioso. Debido esto, se decidió que no se debía proseguir por esta línea de acción, que dado que no era uno de los objetivos fundamentales del proyecto y se disponía de equipos sustitutivos de sobra., debido fundamentalmente.

Por tanto, una vez decidida la no utilización de la HummingBoard o al menos su utilización con un Linux actualizado, se procedió a continuar el proyecto con los ordenadores que ya se había utilizado en proyectos anteriores en los que tampoco se actualizó Linux dado que no que queríamos empezar ya con el desarrollo de OpenC2X. [2, 3, 4].

2.3. Problemas con el CRDA

El CRDA (Central Regulatory Domain Agent) es el agente central de los dominios reguladores de Linux. En él se especifican las frecuencias inalámbricas en las que está permitido trabajar en cada país. El CRDA se encarga de denegar o permitir el acceso de un dispositivo Wi-Fi a una determinada frecuencia en función de si está o no permitida. Para poder trabajar con las frecuencias especificadas en el estándar ETSI ITS G5, es necesario realizar algunas modificaciones en este servicio que nos permitan habilitar el espectro de frecuencias requerido. En el Trabajo de Fin de Máster de Roberto Herreras [4] se explica detalladamente como realizar estas modificaciones.

Un resumen del procedimiento sería el siguiente: modificar el fichero db.txt con los rangos de las frecuencias que queremos, en nuestro país o en el país que precisemos. Realizar las compilaciones y ejecuciones necesarias que se explican en [4] y asegurarnos de tener actualizada la fecha y hora del sistema para que no haya conflictos con las claves de cifrado.

Sin embargo, este procedimiento aquí explicado solo es válido para versiones del kernel 4.15 o anteriores. A partir de la versión 4.15 Linux cambia la forma de cargar el fichero *regulatory.bin* del directorio */lib/crda/* que contiene la codificación de los diferentes países y las frecuencias disponibles para cada uno. Ahora este fichero se llama *regulatory.db* y debe cargarse en el kernel como firmware. En teoría esto debería realizarse durante la compilación del kernel y se deberían seleccionar opciones específicas de compilación para que funcione. En nuestro caso, tras múltiples intentos, no fuimos capaces de realizar estas modificaciones correctamente pues nuestro *regulatory.db* nunca se actualizaba. Dado que no era un problema central en el proyecto se decidió abandonar esta línea de trabajo. En cualquier caso, en [28] se referencia al tutorial que se siguió para realizar este proceso, que puede ser de utilidad en futuros proyectos.

2.4. Opciones de dispositivos 802.11p en el mercado

Una de las primeras tareas que realicé cuando comencé el proyecto, fue la búsqueda de posibles dispositivos que funcionaran con 802.11p. Concretamente, el objetivo era buscar un chip compatible con 802.11p para poder sustituir las tarjetas ath9k, y un kit de evaluación para sistemas 802.11p.

En cuanto al chip, se buscó en diferentes marcas y tiendas, pero el único que se encontró específico para 802.11p, fue el chip de uBlox UBX-P3 [23] y el transceptor VERA-P1 [24], que se diseñaron exclusivamente para comunicaciones V2X. En [30] se hace referencia al documento EOL (End Of Life) de estos dispositivos, en él se especifica que se dejan de fabricar dado que se busca que los dispositivos sustituyentes estén orientados hacia uso del 5G en lugar de 802.11p. Esto se debe a la falta de demanda que existe en estos productos. De este documento, se puede deducir que será imposible en el futuro encontrar dispositivos diseñados exclusivamente para esta tecnología.

El kit de evaluación que se encontró fue el EVK-3300 de Kapsch [25], que se trata de un dispositivo muy completo, dispone incluso de una aplicación web que nos permite configurarlo. También trae implementada la pila ITS (capa de *facilities*, red y transporte, y enlace con 802.11p), con la capacidad de generar los mensajes CAM, DENM, etc. Incluso, lleva implementado LDM. Los datos de posicionamiento se pueden obtener a través de una interfaz externa dedicada.

En resumen, los dispositivos que ofrece uBlox están en fase de ser descatalogados y de no tener sustitutos y por otro lado el dispositivo EVK-3300 es un producto cerrado sobre el que no podríamos trabajar. Parece que el futuro va orientado hacia C-V2X y por lo tanto es hacia esta tecnología de acceso hacia donde se deberían orientar los nuevos trabajos.

2.5. Mejora en el sistema GPS

Otra tarea que se había planteado en proyectos anteriores era la mejora del sistema GPS, pues lo dispositivos que se habían comprado tenían una precisión muy baja (de unos 40 metros) y para realizar comunicaciones V2X esto no es para nada útil.

Para ello se buscaron dispositivos que nos aportaran precisiones con un margen de error lo más bajo posible. La mejor opción que se encontró fueron los módulos uBlox ZED-F9K [27] para comunicación V2X, que aportan precisiones con un margen de error menor de 0.2 metros y disponen de un sistema de Dead Reckoning.

Este sistema es un proyecto de última generación que combina los sistemas tradicionales de posicionamiento GPS y GNNS utilizando algoritmos de procesado de cambios de distancia, dirección y elevación que ocurren durante la perdida de la señal. Esto permite estimar la posición del vehículo en situaciones de lo más adversas, pues mejora notablemente la precisión del posicionamiento con climatología extrema e incluso permite el cálculo de la posición en túneles y zonas muertas donde no llega la cobertura con el satélite.

Figura 14. Posición del vehículo en el carril calculada mediante Dead Reckoning

Estas características convierten a estos dispositivos en ideales para la comunicación V2X, o sistemas ADAS, debido a la gran precisión que permite incluso calcular la posición del vehículo dentro de un único carril de la carretera. [26]

Capítulo 3. Estudio del software OpenC2X

En los anteriores proyectos de fin de carrera en los que se trabajó en la comunicación V2X y el estándar ITS ETSI G5, se llevó a cabo un estudio de diferentes proyectos y herramientas software basados en este estándar para ver cuáles de estos estaban más cerca de una posible implementación real que cumpliera con los requisitos de la pila ITS.

Lo que se buscaba era un proyecto de código abierto que integrara los protocolos definidos en el estándar ETSI y además que se pudiera ejecutar en un sistema Linux. Entonces no se pudo encontrar ninguno que tuviera todos los protocolos y aplicaciones desarrollados, por lo que se optó por elegir uno de entre estos 3: Github Vanetza, Github GeoNetworking y OpenC2X. Finalmente se optó por la opción de utilizar OpenC2X dado que era el que tenía más capas de la pila implementadas, tal y como se puede ver en la Figura 15Figura 15. Protocolos de la pila ITS en cada proyecto. [18].

ETSI	Standard	OpenC2X	GeoNet Stack[2]	Vanetza[3]
DCC	102 687	✓	–	✓
CAM	102 637-2	✓	✓	–
DENM	102 637-3	✓	✓	–
LDM	102 863	✓	–	–
GeoNet/BTP	102 636-4-2/-5-1	–	✓	✓
Security		–	–	✓
Management		✓	✓	✓
GPS		✓	✓	✓
OBD-II		✓	–	–

Figura 15. Protocolos de la pila ITS en cada proyecto. [18]

Como podemos observar OpenC2X era el que disponía de más protocolos y aplicaciones desarrollados. Particularmente interesante era que estuviera desarrollado el DCC [16], una de las partes más complejas del estándar ETSI ITS G5 que es el que se encarga de realizar el control de flujo de la comunicación. Además, también disponía de implementación de CAM y DENM, lo que permitía hacer pruebas casi de manera inmediata. Solamente le faltan la capa de seguridad, la capa del GeoNetworking y el BTP.

Justamente, uno de los principales objetivos de este proyecto es completar la capa del BTP. En los siguientes apartados, se intentará avanzar en este problema con la integración del BTP, pero para eso es necesario conocer la arquitectura, el funcionamiento y el código detrás de OpenC2X.

3.1. Arquitectura

Como ya se ha mencionado anteriormente, OpenC2X es un proyecto de código abierto que funciona sobre Linux, fue desarrollado en el lenguaje C++ en los CCS Labs en la Universidad de Paderborn, Alemania. Su arquitectura está diseñada en un modelo de capas tomando como referencia la pila de protocolos establecida en el estándar ETSI ITS G5 (Los servicios en línea discontinua aún no están desarrollados):

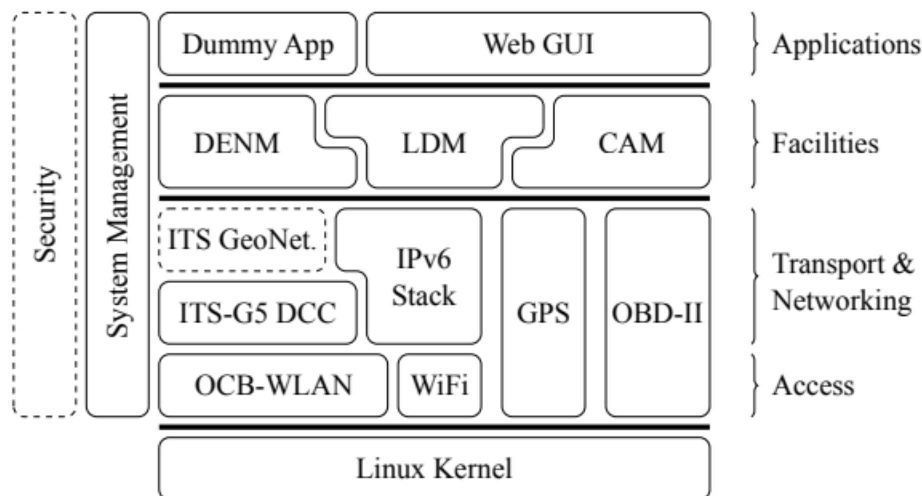

Figura 16. Arquitectura OpenC2X derivada de la pila de protocolos del estándar ETSI ITS G5.

Aunque mis compañeros Pilar [2] y Alejandro [3] ya explicaron en sus trabajos fin de grado y master los servicios ofrecidos por OpenC2X, por facilitar una lectura más compresiva y dado que haré múltiples referencias en el futuro a distintas cuestiones de la

implementación de OpenC2X para justificar mis decisiones, se repasarán algunos de los servicios de este software que tienen relación con el desarrollo de BTP.

En la capa de *Applications* de la Figura 16 Figura 16. Arquitectura OpenC2X derivada de la pila de protocolos del estándar ETSI ITS G5.se encontrarán las aplicaciones que hacen de interfaz con el usuario, como es la aplicación Web que nos deja controlar, mediante una interfaz amigable, el envío de mensajes y además visualizado los mensajes que enviamos y los que recibimos.

En la capa de *facilities* se encuentran 3 servicios:

- CAM (Cooperative Awareness): Su función principal es la del envío de mensajes relacionados con los vehículos que están utilizando la vía y la situación del tráfico. Puede enviar información sobre la velocidad, la posición GPS, etc. [10]
- DENM (Decentralized Enviromental Notification): Este protocolo se encarga del manejo de la información relacionada con determinados eventos en la vía, como por ejemplo un accidente. Este protocolo proporcionará información de ese determinado evento, tal como su posición geográfica, la gravedad de este, etc. [11]
- LDM (Local Dynamic Map): Este servicio es una base de datos que almacena información sobre diferentes vehículos y eventos. Almacena mensajes CAM y DENM para su uso posterior. [19]

En la capa de acceso y transporte se encuentran:

- DCC (Decentralized Congestion Control): Es el protocolo más complejo e importante de las redes ITS, se encarga de mantener la estabilidad y el rendimiento de las comunicaciones, realizando un exhaustivo control de flujo para evitar sobresaturar el medio, de tal manera que la combinación del modo OCB con el control de la saturación del medio permita que el intercambio de información se realice en tiempos muy pequeños.
- GPS: Este servicio se encarga de obtener la posición geográfica de las estaciones ITS de forma periódica, puede utilizarse para esto tanto un

sistema GPS real como coger la información de un fichero externo para mandar datos predeterminados.

- OBD-II (On Board Diagnosis): Este protocolo es un sistema que se encargar de recoger información del propio vehículo, tanto información de su funcionamiento como posibles averías o errores.

Finalmente, en el último nivel de la pila se encuentra el kernel de Linux que es el que se encarga de realizar la comunicación con la tarjeta Wi-Fi para enviar datos o recibirlos por la interfaz OCB de esta.

Una vez repasada la arquitectura, explicaré el funcionamiento de OpenC2X desde un punto de vista más práctico, analizando principalmente su código y resaltando los aspectos más importantes.

3.2. Funcionamiento y estructura de comunicación

En OpenC2X todos los servicios explicados en el apartado anterior funcionan como módulos independientes, que se comunican entre sí para ir formando las tramas ITS que se enviarán a la red o, en caso de estar recibiendo, desencapsular el mensaje y pasarlo a las capas superiores. La estructura de comunicación se representa en la Figura 17.Figura 17. Esquema de comunicación de los diferentes módulos en OpenC2X

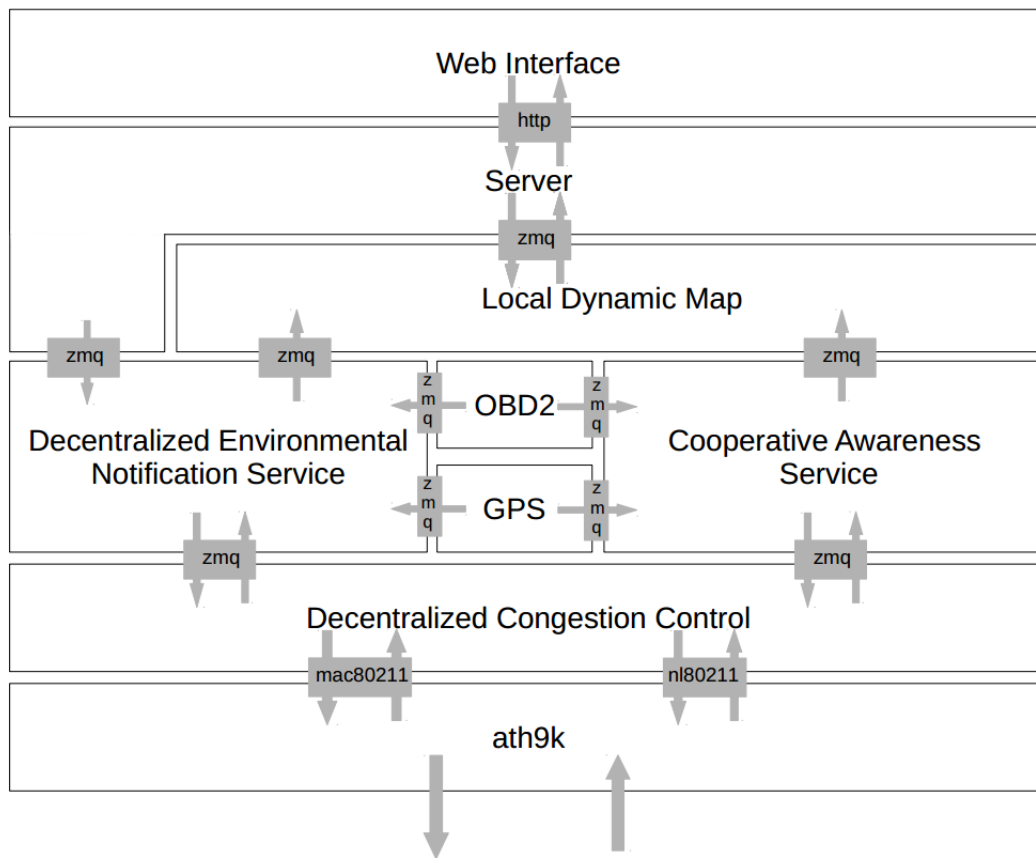

Figura 17. Esquema de comunicación de los diferentes módulos en OpenC2X

Los módulos se comunican entre sí a través de zmq (ZeroMQ). Se trata de una librería de sockets asíncronos de alto rendimiento para el desarrollo de aplicaciones distribuidas o locales. Por medio de estos sockets los diferentes módulos envían o reciben información con latencia casi nula, lo que permite una comunicación rápida y eficiente. Para más información sobre esta herramienta se puede consultar [20] y [21].

Para ver como utiliza esta herramienta podemos consultar el código de OpenC2X [6]. En el directorio *OpenC2X/common/utility* podemos encontrar los ficheros de código fuente que la implementan. En nuestro caso nos centraremos en los ficheros que contienen las clases *CommunicationSender* y *CommunicationReceiver*.

```cpp
CommunicationSender::CommunicationSender(string ownerModule, string portOut, int expNo, string loggingConf, string statisticConf) {
    mOwnerModule = ownerModule;

    mContext = new zmq::context_t(1);
    mPublisher = new zmq::socket_t(*mContext, ZMQ_PUB);
    mPublisher->bind(("tcp://*:" + portOut).c_str());

    mLogger = new LoggingUtility(mOwnerModule, expNo, loggingConf, statisticConf);
}

void CommunicationSender::send(string envelope, string message) {
    s_sendmore(*mPublisher, envelope);
    s_send(*mPublisher, message);

    mLogger->logDebug(envelope + " sent");
}
```

Figura 18. Fichero common/utility/CommunicationSender.cpp [6]

```cpp
CommunicationReceiver::CommunicationReceiver(string ownerModule, string portIn, string envelope, int expNo, string loggingConf, string statisticConf) {
    mOwnerModule = ownerModule;

    mEnvelope = envelope;
    mContext = new zmq::context_t(1);
    mSubscriber = new zmq::socket_t(*mContext, ZMQ_SUB);
    mSubscriber->connect(("tcp://localhost:" + portIn).c_str());
    if (envelope == "") {
        mSubscriber->setsockopt(ZMQ_SUBSCRIBE, "", 0);//subscribe to all messages
    } else {
        mSubscriber->setsockopt(ZMQ_SUBSCRIBE, envelope.c_str(), 1);
    }

    mLogger = new LoggingUtility(mOwnerModule, expNo, loggingConf, statisticConf);
}

pair<string, string> CommunicationReceiver::receive() {
    string envelope = s_recv(*mSubscriber);
    string message = s_recv(*mSubscriber);

    mLogger->logDebug(envelope + " received");

    return make_pair(envelope, message);
}
```

Figura 19. Fichero common/utility/CommunicationReceiver.cpp [6]

Estas clases sirven para controlar los extremos de la comunicación entre módulos. *CommunicationSender* es el emisor y manda el mensaje para el otro módulo en forma de string al puerto *portOut* (resaltado en amarillo), que se le pasa como parámetro en el constructor. Lo que se hace es asociar el socket zmq con el puerto de salida *portOut* para que cada vez que se llame a la función *send()* esta envíe el mensaje a ese puerto. El mensaje se envía junto con otro string llamado *envelope* que representa el módulo al que va dirigido, por ejemplo, si el mensaje lo genera el CAM el *envelope* será "CAM".

En el caso *CommunicationReceiver* es un caso similar, pero en vez de asociarlo a un puerto para que envíe lo hace para que escuche en el *portIn* (también señalado en amarillo). Además, a esta se le pasa el *envelope* del módulo del que se quiere recibir un mensaje. Esto es útil, dado que si tenemos un módulo que envía mensajes a otros dos por el mismo puerto, los receptores configurando su *CommunicationReceiver* con su propio *envelope* recibirán solo los mensajes que contengan el que coincida con su nombre. Este es el caso del DCC, CAM y DENM. El DCC envía datos a CAM y DENM a través de un mismo puerto y ellos al configurar su receptor con su *envelope* predeterminado solo reciben los mensajes que son para ellos.

```
51      mReceiverFromDcc = new CommunicationReceiver(module, "5555", "DENM", mGlobalConfig.mExpNo, loggingConf, statisticConf);
```

Figura 20. Receptor del DENM para el DCC en el constructor de la clase DenService, /denm/src/denservice.cpp. [6]

```
55      mReceiverFromDcc = new CommunicationReceiver("CaService", "5555", "CAM", mGlobalConfig.mExpNo, loggingConf, statisticConf);
```

Figura 21. Receptor del CAM para el DCC en el constructor de la clase CaService, /cam/src/caservice.cpp. [6]

```
59      mSenderToServices = new CommunicationSender(module, "5555", mGlobalConfig.mExpNo, loggingConf, statisticConf);
```

Figura 22. Emisor del DCC para CAM y DENM en el constructor de la clase DCC, /dcc/src/dcc.cpp. [6]

Hay que resaltar una importante limitación de este mecanismo. Por cómo están desarrolladas estas clases en OpenC2X y los propios sockets zmq ahora mismo no se puede enviar a un mismo puerto desde diferentes módulos dado que salta un error con la dirección mac. En cambio, la opción de que dos módulos escuchen en el mismo puerto sí es posible.

3.3. Encapsulamiento de Datos

Otro aspecto importante de OpenC2X es cómo envía los datos para formar las tramas entre módulos, pues los sockets zmq implementados en el programa están preparados para enviar solamente un string.

Para ello, el programa utiliza la herramienta *Protocol Buffers* de Google [22], que se trata de un mecanismo para serializar los datos. Esta utilidad nos permite definir estructuras de datos personalizadas, que con una simple línea de código pueden serializarse en un string de datos. Para ello el usuario solo tiene que definirlas en un fichero con extensión *.proto* y mediante un compilador se generara un fichero *.c* que se puede importar a nuestro proyecto como una clase más. Con esto podemos crear clases para poder enviar diferentes mensajes de una manera rápida y sin código excesivamente complicado.

Dentro de OpenC2X los ficheros *.proto* que contienen la definición de los datos se encuentran en el directorio */common/buffers/*. Si nos fijamos existen ficheros para cada estructura de datos necesaria, tanto mensajes CAM, como mensajes DENM, como cabeceras ITS. La estructura que más nos interesa es la definida en *data.proto*, pues es el tipo de dato que se envía entre módulos.

```
package dataPackage;

message DATA {
	enum Type {
		CAM = 1;
		DENM = 2;
	}
	enum Priority {
		BK = 1;
		BE = 0;
		VI = 4;
		VO = 6;
	}
	required int32 id = 1;
	required Type type = 2;
	required Priority priority = 3;
	optional int64 createTime = 4;
	optional int64 validUntil = 5;
	optional double txPower = 6;
	optional double bitRate = 7;
	required bytes content = 8;
}
```

Figura 23. Fichero /common/buffers/data.proto

En las estructuras de datos lo único que necesitamos definir son los campos que contienen y obligatoriamente la posición que ocupan dentro de estas. Se puede elegir si un determinado campo siempre debe tener asociado un valor o por el contrario puede ser que no se especifique (para eso sirven las sentencias *required* y *optional*). También se puede especificar qué valores puede tomar un campo con la sentencia *enum*.

Una vez definido el fichero *.proto*, OpenC2X trae incorporado un script que genera los ficheros *.c* con las clases creadas a partir de este, se trata del fichero */common/buffers/generate.sh*. Por ejemplo, *data.proto* nos creará en la carpeta *build/* dos ficheros *data.pb.cc* y *data.pb.h* con una clase DATA que tiene como atributos los campos mencionados en *data.proto*. Además, también se generan funciones para poder modificar esos atributos o para poder extraerlos, y también para poder serializar la clase en un string, y viceversa, parsear un string en una nueva clase DATA. Para ver mejor esto, consultar dichos ficheros en [6] pues son demasiado extensos como para mostrarlos en una figura.

En el siguiente código del módulo del DENM se puede observar como se utiliza la clase DATA. El objetivo de este código es crear un mensaje DENM que se envía al DCC para ser transmitido.

```
165  void DenService::send(triggerPackage::TRIGGER trigger) {
166      string serializedData;
167      dataPackage::DATA data;
168
169      // create denm
170      DENM_t* denm = generateDenm();
171      // asn_fprint(stdout, &asn_DEF_DENM, denm);
172      vector<uint8_t> encodedDenm = mMsgUtils->encodeMessage(&asn_DEF_DENM, denm);
173      string strDenm(encodedDenm.begin(), encodedDenm.end());
174      mLogger->logDebug("Encoded DENM size: " + to_string(strDenm.length()));
175
176      data.set_id(messageID_denm);
177      data.set_type(dataPackage::DATA_Type_DENM);
178      data.set_priority(dataPackage::DATA_Priority_VI);
179
180      int64_t currTime = Utils::currentTime();
181      data.set_createtime(currTime);
182      data.set_validuntil(currTime + 2*1000*1000*1000);   //2s TODO: conform to standard? -> specify using CLI
183      data.set_content(strDenm);
184      data.SerializeToString(&serializedData);
185      mLogger->logInfo("send new DENM " + to_string(data.id()) + " to DCC and LDM");
186      mSenderToDcc->send("DENM", serializedData);     //send serialized DATA to DCC
```

Figura 24. Fichero /denm/src/denservice.cpp

La función en la línea 170, *generateDenm(),* devuelve un puntero a estructura tipo *DENM_t* que luego se codifica en el string *strDenm* en la líneas 172 y 173. Ese string contiene el paquete DENM generado. Entre las líneas 176 a 183 se rellenan los distintos campos que incluye la clase DATA, donde podemos prestar especial atención a la línea 183 en la cual se encapsula el string generado en la línea 173 (y que contiene el paquete DENM en sí) dentro del campo *content* de DATA. Finalmente, en la línea 184 se serializa todo el objeto y se copia en el string *serializedData*, y en la línea 186 se envía por el socket al DCC como se explicó anteriormente.

3.4. Ruta del mensaje

Una vez entendido como se comunican los diferentes módulos es interesante ver como se van formando las diferentes tramas que se enviarán por la red y como se van desencapsulando las que lleguen a nuestra interfaz de red. Comprender estos procesos es necesario para para luego comprender como se ha realizado la implementación del BTP. Ambos casos se explicarán mediante un ejemplo, el envío de un mensaje CAM y la recepción de un DENM.

3.4.1. Envío

Los mensajes CAM están configurados para enviarse periódicamente si se cumplen unas determinadas condiciones. Este proceso es controlado por la función *alarm()* definida en el fichero *cam/src/caservice.cpp*, que dependiendo de si cambia la posición GPS, la velocidad, etc; llamará a la función *trigger()* que a su vez llama a la

función *send()*. Esta última genera el mensaje CAM y lo envía al DCC. Esto se puede entender mucho mejor si se consulta el fichero en cuestión.

```
330  //generate CAM and send to LDM and DCC
331  void CaService::send() {
332      string serializedData;
333      dataPackage::DATA data;
334
335      // Standard compliant CAM
336      CAM_t* cam = generateCam();
337      vector<uint8_t> encodedCam = mMsgUtils->encodeMessage(&asn_DEF_CAM, cam);
338      string strCam(encodedCam.begin(), encodedCam.end());
             ...
341      data.set_id(messageID_cam);
342      data.set_type(dataPackage::DATA_Type_CAM);
             ...
348      data.set_content(strCam);
349
350      data.SerializeToString(&serializedData);
351      mLogger->logInfo("Send new CAM to DCC and LDM\n");
352
353      mSenderToDcc->send("CAM", serializedData);  //send serialized DATA to DCC
```

Figura 25. Función send() del fichero cam/src/caservice.cpp

Una vez que se genera el mensaje CAM (línea 336) el programa lo encapsula en un objeto DATA como se mencionó anteriormente, y se utiliza la función *send()* del objeto mSenderToDcc (destinado al DCC) que se crea desde la clase *CommunicationSender* (se define en el constructor del CAService igual que en la Figura 22Figura 22. Emisor del DCC para CAM y DENM en el constructor de la clase DCC, /dcc/src/dcc.cpp. [6], pero con el puerto 6666), para enviarlo al puerto en el que escucha el *CommunicationReceiver* configurado en el DCC (igual que en la Figura 21Figura 21. Receptor del CAM para el DCC en el constructor de la clase CaService, /cam/src/caservice.cpp. [6], pero con el puerto 6666 y el *envelope* “CAM”) para el CAM. Tal y como se muestra en el siguiente diagrama (esto se aplica para todas las comunicaciones entre módulos):

Figura 26. Diagrama comunicación entre CAM y DCC

Para recibir el mensaje del CAM el dcc ejecuta continuamente la función *receiveFromCa2()* por medio de un thread configurado en la función *init()*.

```
139  void DCC::init() {
        ...
148      //create and start threads
149      mThreadReceiveFromCa = new boost::thread(&DCC::receiveFromCa2, this);
150      mThreadReceiveFromDen = new boost::thread(&DCC::receiveFromDen, this);
151      mThreadReceiveFromHw = new boost::thread(&DCC::receiveFromHw2, this);
```

Figura 27. Inicialización de threads que ejecutan las funciones de recepción de otros módulos, fichero dcc/src/dcc.cpp

La función *receiveFromCa2()* está ejecutando continuamente un bucle (líneas 219 a 239) para recibir del CAM, una vez que le llega el mensaje (línea 220, en esta línea se llama a la función *receive* que contiene un bucle while del que no sale hasta que llega un paquete), lo mete a una cola en función de la prioridad que traiga su objeto DATA (línea 233). Y finalmente lo envía llamando a *sendQueuedPackets()* en la línea 236.

```
215  void DCC::receiveFromCa2() {
216      string encodedData;              //serialized DATA
217      dataPackage::DATA* data;         //deserialized DATA
218
219      while (1) {
220          pair<string, string> received = mReceiverFromCa->receive();
221          encodedData = received.second;
222
223          data = new dataPackage::DATA();
224          data->ParseFromString(encodedData);     //deserialize DATA
225
226          string encodedCam = data->content();
227
228          Channels::t_access_category ac = (Channels::t_access_category) data->priority();
229          int64_t nowTime = Utils::currentTime();
230          mBucket[ac]->flushQueue(nowTime);
231
232          mLogger->logInfo("");                        //for readability
233          bool enqueued = mBucket[ac]->enqueue(data, data->validuntil());
234          if (enqueued) {
235              mLogger->logInfo("AC "+ to_string(ac) + ": received and enqueued CAM " + to_string(data->id()) + ", queue length: " + to_string(mBucl
236              sendQueuedPackets(ac);
237          }
238          else {
239              mLogger->logInfo("AC "+ to_string(ac) + ": received and dropped CAM " + to_string(data->id()) + ", queue full -> length: " + to_stri
```

Figura 28. Función receiveFromCa2() en dcc/src/dcc.cpp.

Esta última comprueba si el paquete está disponible para ser transmitido (línea 540) y si los datos encapsulados siguen siendo válidos (línea 542). En el caso de que se cumplan estas condiciones llamará a la función *sendWithGeoNet()*, definida en el fichero *dcc/src/SendToHardwareViaMac.cpp* para que haga el envío final a la red. Si las condiciones no se cumplen simplemente descarta el paquete.

```
538  //sends queued packets from specified LeakyBucket to hardware until out of packets or tokens
539  void DCC::sendQueuedPackets(Channels::t_access_category ac) {
540      while(dataPackage::DATA* data = mBucket[ac]->dequeue()) {      //true if packet and token available -> pop 1st packet f
541          int64_t nowTime = Utils::currentTime();
542          if(data->validuntil() >= nowTime) {                         //message still valid
543              setMessageLimits(data);
544
545              string byteMessage;
546              byteMessage = data->content();
547              mSenderToHw->sendWithGeoNet(&byteMessage, ac, data->type());
548              mLogger->logInfo("AC " + to_string(ac) + ": Sent data " + to_string(data->id()) + " to HW -> queue length: " + t
549              delete data;
```

Figura 29. Función sendQueuedPackets() en el fichero dcc/src/dcc.cpp

```
195  void SendToHardwareViaMAC::sendWithGeoNet(string* msg, int priority, int type) {
196      unsigned int geoHdrLen;
197      uint8_t* geoHdr;
198      switch(type) {
199          case dataPackage::DATA_Type_DENM:
200              fillGeoNetBTPheaderForDenm(msg->size());
                 ...
204          case dataPackage::DATA_Type_CAM:
205              fillGeoNetBTPheaderForCam(msg->size());
206              geoHdrLen = sizeof(struct GeoNetworkAndBTPHeaderCAM);
207              geoHdr = reinterpret_cast<uint8_t*>(&mGeoBtpHdrForCam);
208              break;
209          default:
210              mLogger->logError("Queued packet has invalid type: " + to_string(type));
211              break;
212      }
213      unsigned int packetsize = sizeof(struct ether_header) + geoHdrLen + msg->size();
214      unsigned char packet[packetsize];
215      unsigned char* payload = packet + sizeof(struct ether_header) + geoHdrLen;
216      unsigned char* geoNetHdr = packet + sizeof(struct ether_header);
217
218      //copy header to packet
219      memcpy(&packet,&mEth_hdr,sizeof(struct ether_header));
220
221      // Fill in geo networking and btp header
222      memcpy(geoNetHdr, geoHdr, geoHdrLen);
223
224      //copy payload to packet
225      memcpy(payload,msg->c_str(),msg->size());
226      //dumpBuffer(reinterpret_cast<const uint8_t*>(msg->c_str()), msg->size());
         ...
247      if(send_to_socket != -1){
248          /** @todo record/put into dcc info/extend to fixed size packetsize */
249          if ((sendto(send_to_socket,packet,packetsize,0,(struct sockaddr* )&mTo_sock_addr,
250                      sizeof(struct sockaddr_ll))) == -1)
```

Figura 30. Función sendWithGeoNet() en el fichero dcc/src/SendToHardwareViaMac.cpp

sendWithGeoNet() hace varias cosas. Se encarga de rellenar una estructura con los datos que debería tener la cabecera GeoNetworking y BTP (línea 200 y 205, la función de relleno de cabeceras se explicará con más detalle en el capítulo 5); esto lo hace con valores genéricos, pues estos protocolos no están implementados en el programa y es necesario enviar estas cabeceras para que sea tráfico ITS; añade las cabeceras (líneas 213 a 225) ethernet, GeoNetworking, y BTP a la carga (que es el mensaje CAM) y la envía, en la línea 249, a la tarjeta ATH9K para que la mande por su interfaz OCB.

fillWithGeoNetBTPheaderForCam() se encarga de rellenar los datos de las cabeceras, su uso se explicará en el capítulo 5, pues se ha adaptado su uso para la integración con el BTP. En cuanto a la adición de las cabeceras a la trama CAM, esta se realiza entre las líneas 213 y 225. Lo primero es definir una cadena de caracteres que llevará almacenada la trama completa (*packet[packetsize]*), posteriormente se definen dos punteros:

- *payload*: apunta a la dirección de memoria donde empezará la carga útil del paquete, es decir, el mensaje cam. Para llegar a esta dirección basta con sumarle a la dirección donde empieza *packet* el tamaño de la cabecera ethernet y el de la cabecera GeoNetworkig + BTP (*geoHdrLen*).
- *geoNetHdr*: apunta a la dirección de memoria donde empieza la cabecera de GeoNetworking, en este caso solo hay que sumarle a *packet* el tamaño de la cabecera ethernet.

Con los punteros ya definidos, solo necesitamos copiar en esas direcciones de memoria el contenido de las cabeceras. En la línea 219 se copia en la dirección donde empieza *packet* el contenido de la cabecera ethernet, en la 222 se copia en la dirección apuntada por *geoNetHdr* el contenido de la cabecera GeoNetworking + BTP y, por último, en la línea 225 se copia la carga (mensaje CAM) en la dirección apuntada por *payload*. Finalmente, en la línea 249 se envía el mensaje.

Una vez realizados estos pasos, OpenC2X ya ha enviado un paquete perfectamente encapsulado a la red ITS.

3.4.2. Recepción

La parte de recepción es un poco más sencilla que la de envío, pues están involucradas menos funciones.

En el módulo del DCC se encuentra la función *receiveFromHw2()*, que se encarga de escuchar hasta que le llegue un paquete desde el socket *mReceiverFromHw*. Para ello llama a la función *receiveWithGeoNetHeader()* (línea 305) que se encarga de leer directamente del socket que se comunica con la interfaz de la tarjeta de red.

Esta función también se encarga de eliminar las cabeceras Ethernet y GeoNetworking + BTP, utilizando el mismo método que se usó para añadirlas, pero a la inversa. Aunque aquí hay que resaltar un detalle importante. OpenC2X aprovecha que las funciones *fillWithGeoNetBTPheaderForCam()* y *fillWithGeoNetBTPheaderForDenm()* rellenan las cabeceras del GeoNetworking siempre con los mismos valores para saber cuál va dirigido al CAM y al DENM. Esto no debería ser así según el estándar, pues de esto se debería encargar el BTP, pero como no está implementado, los programadores improvisaron de esta manera una solución para poder saber a qué puerto va destinado el

mensaje. De esta manera, se sabe que el byte 5 de la cabecera del GeoNetworking siempre tendrá el valor 80 cuando se trate de un CAM y el 66 cuando sea un DENM. En función de esto, *receiveWithGeoNetHeader()* devolverá el mensaje sin cabeceras y con información de si es un CAM o un DENM.

Volvemos a *receiveFromHw2()*. Sabiendo como diferenciar a qué servicio va cada una, se encarga de mandarlas por el socket con un *envelope* correspondiente con el servicio al que va dirigido, como se explicó anteriormente. Aunque en este caso el socket está preparado para enviar siempre al puerto 5555, y los dos servicios escuchan en él, en vez de hacer un socket para cada servicio.

```
299  void DCC::receiveFromHw2() {
300      pair<ReceivedPacketInfo, string> receivedData;      //MAC Sender, serialized DATA
301      ReceivedPacketInfo* pktInfo = &receivedData.first;
302      string* serializedData = &receivedData.second;
303      mLogger->logInfo("start receiving via Hardware");
304      while (1) {
305          receivedData = mReceiverFromHw->receiveWithGeoNetHeader();  //receive serialized DATA
                     ...
314          switch(pktInfo->mType) {                          //send serialized DATA to corresponding module
315              case dataPackage::DATA_Type_CAM:
316                  mSenderToServices->send("CAM", *serializedData);
317                  mLogger->logInfo("forward received CAM from source "+ pktInfo->mSenderMac +" to services");
318                  break;
319              case dataPackage::DATA_Type_DENM:
320                  mSenderToServices->send("DENM", *serializedData);
321                  mLogger->logInfo("forward received DENM from source "+ pktInfo->mSenderMac +" to services");
322                  break;
323              default:
324                  break;
```

Figura 31. Función receiveFromHw2() en el fichero dcc/src/dcc.cpp

```
90   pair<ReceivedPacketInfo, string> ReceiveFromHardwareViaMAC::receiveWithGeoNetHeader() {
91       while(1) {
92           // receive package, blocking
93           mBytes = read(mSocket, mPacket, sizeof(mPacket));
94           if (mBytes == -1) {
95               mLogger->logPError("reading from Socket failed");
                     ...
102          int geoNetPDULen = mBytes - mLinkLayerLength;
103          // convert sender Mac from network byte order to char
104          string senderMac = ether_ntoa((struct ether_addr*)mEth_hdr->ether_shost);
105          // Hack! As of now, we are looking for very specific bits in the GeoNetworking header
106          char* geoNetPDU = mPacket + mLinkLayerLength;
107          if (geoNetPDU[5] == 80) {
108              // CAM
                     ...
116          } else if (geoNetPDU[5] == 66) {
117              // DENM
118              int denmPDULen = geoNetPDULen - sizeof(struct GeoNetworkAndBTPHeaderDENM);
119              char* denmPDU = geoNetPDU + sizeof(struct GeoNetworkAndBTPHeaderDENM);
120              string msg(denmPDU, denmPDULen);
121              ReceivedPacketInfo info;
122              info.mSenderMac = senderMac;
123              info.mType = dataPackage::DATA_Type_DENM;
124              return make_pair(info, msg);
125          } else {
```

Figura 32. Función receiveWithGeoNetHeader() en el fichero dcc/src/ReceiveFromHardwareViaMac.cpp

Finalmente, el mensaje llega a la función *receive* del DENM que simplemente se encarga de pasar el mensaje al módulo LDM para que lo registre en la base de datos.

```
94   //receive DENM from DCC and forward to LDM
95   void DenService::receive() {
96       string envelope;           //envelope
97       string serializedAsnDenm;
98       string serializedProtoDenm;
99
100      while(1) {
101          pair<string, string> received = mReceiverFromDcc->receive();
102          envelope = received.first;
103          serializedAsnDenm = received.second;
104
105          DENM_t* denm = 0;
106          int res = mMsgUtils->decodeMessage(&asn_DEF_DENM, (void **)&denm, serializedAsnDenm);
107          if (res != 0) {
108              mLogger->logError("Failed to decode received DENM. Error code: " + to_string(res));
109              continue;
110          }
111          //asn_fprint(stdout, &asn_DEF_DENM, denm);
112          denmPackage::DENM denmProto = convertAsn1toProtoBuf(denm);
113          denmProto.SerializeToString(&serializedProtoDenm);
114
115          mLogger->logInfo("forward incoming DENM " + to_string(denm->header.stationID) + " to LDM");
116          mSenderToLdm->send(envelope, serializedProtoDenm);
117      }
118  }
```

Figura 33. Función receive() en el fichero denm/src/denservice.cpp

Capítulo 4. El protocolo BTP

El protocolo de transporte básico (BTP) proporciona un servicio de transporte extremo a extremo no orientado a conexión en redes ad hoc para sistemas ITS. El propósito principal de este protocolo es la multiplexación de los mensajes de la capa de *facilities* o servicios del modelo ITS (servicios CAM y DENM principalmente), para su transmisión mediante el protocolo GeoNetworking. También permite su demultiplexado en el destino. BTP también permite a las entidades de la capa de *facilities* acceder a diferentes servicios del GeoNetworking y enviar información de control entre esta capa y este protocolo.

La multiplexación/demultiplexación de los mensajes se basa en puertos, representados con direcciones de 16 bits internas en cada estación. Cada puerto representa un punto final de la comunicación que identifica la entidad de protocolo de la estación ITS, ya sea en el origen o bien en el destino. El uso de estos puertos es bastante similar al transporte en dos etapas de la pila de protocolos IP, donde IP encamina los paquetes del origen al destino y el protocolo de transporte (TCP o UDP) realiza la multiplexación/demultiplexación de mensajes de/hacia los procesos de aplicación. En el caso de BTP, el protocolo de GeoNetworking transporta los paquetes entre las estaciones ITS y el protocolo BTP reparte los paquetes a las entidades a nivel de capa de servicios ITS. BTP también adopta el concepto de "well-known ports" (puertos bien conocidos) donde asigna puertos predefinidos a cada entidad de la capa de servicios ITS. La definición de los puertos se puede consultar en [9].

Este protocolo es ligero, consiste en una cabecera de 4 bytes y requiere un procesamiento mínimo. El servicio es no fiable, es decir, los paquetes pueden llegar desordenados, duplicados o simplemente, no llegar. Por lo tanto, el diseño de BTP asume

que las entidades que hacen uso del servicio son capaces de tolerar estos errores o que por lo menos presentan mecanismos para proporcionar comunicación fiable en sus protocolos.

Estos servicios son proporcionados mediante el Punto de Acceso al Servicio BTP (BTP-SAP) usando primitivas de servicio de diferentes tipos que contienen parámetros, p.ej. Información de Control de Protocolo (ITS-FPCI) y la PDU de la entidad de protocolo superior (p.ej. ITS-FPCI). BTP utiliza los servicios del protocolo GeoNetworking para proporcionar sus servicios de transporte de paquetes.

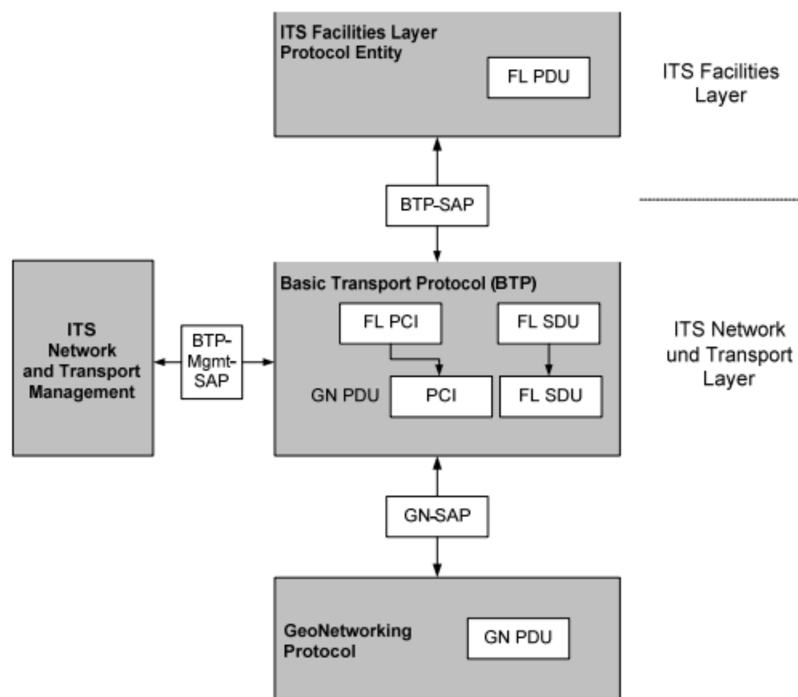

Figura 34. SAPs, SDUs y PDUs relevantes para el BTP [8]

4.1. Estructura del Paquete BTP

Un paquete BTP debe estar encapsulado con las cabeceras de protocolo y la carga descritas en la Figura 35Figura 35. Estructura del paquete BTP. [8].

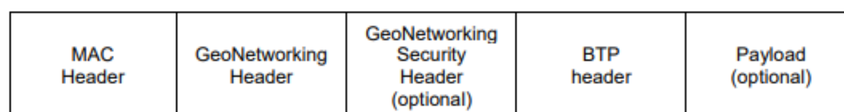

Figura 35. Estructura del paquete BTP. [8]

- La cabecera MAC pertenece al protocolo de capa de acceso del estándar ITS.

- La cabecera de GeoNetworking pertenece a la capa inmediatamente inferior al BTP y se le puede añadir una cabecera adicional perteneciente a la capa de seguridad definida en este protocolo.
- La cabecera BTP es la cabecera del protocolo de transporte básico.
- Finalmente, la carga representa los datos de aplicación generados por las capas superiores de la tecnología ITS que se pasan a la entidad BTP para su transmisión.

Es importante hacer notar que el paquete se muestra tal y como lo vería el protocolo de capa de acceso del estándar ITS.

4.2. Cabecera del BTP

Dentro del BTP se definen dos tipos posibles de cabeceras:

- Cabecera BTP-A: Para transporte interactivo de paquetes.
- Cabecera BTP-B: Para transporte no interactivo de paquetes.

El tipo de cabecera utilizado se puede distinguir según el campo Next Header en la cabecera del GeoNetworking.

Next Header (NH)	Encoding	Description
BTP-A	1	BTP-A header
BTP-B	2	BTP-B header

Figura 36. Codificación de los tipos de cabecera BTP dentro del campo Next Header en la cabecera GeoNetworking. [8]

4.2.1. Cabecera BTP tipo A

La cabecera de tipo A lleva encapsulados los puertos de origen y destino para el BTP. El puerto de destino identifica a la entidad del protocolo de la capa de servicios ITS en el destino de una BTP-PDU (paquete BTP). El puerto de origen identifica el protocolo de la capa de servicios que generó el FL-SDU. El puerto de origen también representa el puerto al que habría que redireccionar una respuesta a la BTP-PDU en la ausencia de más información.

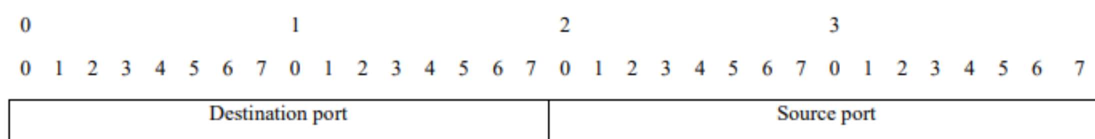

Figura 37. Formato de la Cabecera BTP-A

Field #	Field name	Octet position		Type	Unit	Description
		First	Last			
1	*Destination port*	0	1	16 bit integer		Identifies the protocol entity at the destination's ITS Facility layer.
2	*Source port*	2	3	16 bit integer		Identifies the port of the protocol entity at the source's ITS Facility layer in the source of the BTP PDU.

Figura 38. Formato de la Cabecera BTP-A

4.2.2.Cabecera BTP tipo B

La cabecera BTP-B incluye el puerto destino, pero no puerto de origen. El puerto de destino identifica la entidad de protocolo en la capa de servicios ITS en el destino de un BTP-PDU. Para puertos bien conocidos, debe ser fijada a un valor correspondiente al servicio de capa de servicios identificado según está especificado en [9]. El protocolo en la capa de servicios ITS origen proporciona información adicional si el puerto destino es un puerto bien conocido. Si se especifica en [9], el campo "información de puerto Destino (Destination Port Info)" debe de ser fijado a un valor correspondiente al servicio de la capa de *facilities* identificado en [9]. El valor por defecto para "Información de puerto Destino" es 0.

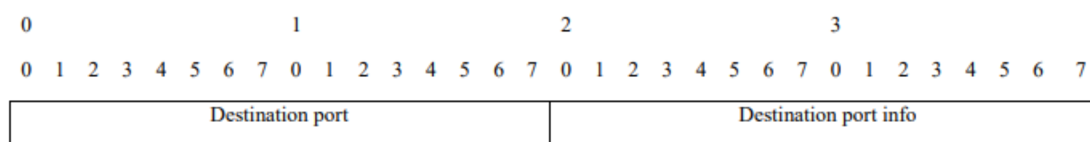

Figura 39. Formato de la Cabecera BTP-B

Field #	Field name	Octet position		Type	Unit	Description
		First	Last			
1	*Destination port*	0	1	16 bit integer		Identifies the protocol entity at the ITS Facility layer in the destination.
2	*Destination port info*	2	3	16 bit integer		Provides additional info if *Destination port* is a well-known port. Setting is beyond the scope of the present document, default setting is 0.

Figura 40. Formato de la Cabecera BTP-B

4.3. Servicios de datos BTP

Las primitivas de servicio de datos BTP permiten a entidades de los protocolos de la capa de *facilities* del modelo ITS enviar y recibir PDUs a través del punto de acceso BTP-SAP.

4.3.1. La primitiva BTP-Data.request

Es utilizada por la entidad de algún protocolo de la capa de *facilities* para solicitar el envío de una BTP-PDU. Tras la recepción de esta primitiva, la entidad BTP entregará

un BTP-SDU al protocolo GeoNetworking a través del punto de acceso GN-SAP. Aquí se presenta la primera limitación práctica para la implementación del estándar. En la sección 3.2 (y en la sección 5 se explicará con más detalle) se mencionó que *zmq* no permite que dos servicios envíen al mismo puerto. Esto provocará que la BTP_data.request deba enviarse por puertos diferentes al mismo servicio de capa inferior (este problema que es de implementación, se verá con más detalle en la sección 5). Los parámetros de dicha primitiva son los siguientes:

BTP-Data.request (
BTP type,
Source port, (optional)
Destination port,
Destination port info, (optional)
GN Packet transport type,
GN Destination,
GN Communication profile,
GN Maximum packet lifetime, (optional)
GN Repetition interval, (optional)
GN Traffic class,
Length,
Data
)

Figura 41. Parámetros de la primitiva BTP-Data.request

Es importante hacer notar que cada parámetro con "GN" delante se pasa al GeoNetworking por el GN-SAP, pero el BTP no hace nada con él.

- BTP type especifica si la comunicación BTP es interactiva (BTP-A) o no interactiva (BTP-B).
- Source port, o puerto origen, indica el puerto BTP que se utiliza para mandar un FL-SDU, y además indica el puerto al que se deberá mandar una respuesta. Este parámetro es opcional y solo se usa para comunicación interactiva.
- Destination port identifica el puerto BTP en el destino.
- Destination port info aporta información adicional para el campo anterior en el caso de que se trate de uno de los puertos bien conocidos.
- Length indica la longitud de "Data".
- Data representa la carga del paquete BTP.
- Para saber la función de los demás parámetros se puede consultar [8], pero de momento no nos son necesarios.

4.3.2. La primitiva BTP-Data.indication

Esta primitiva es la inversa a la anterior, indica a la entidad de la capa de *facilities* que ha llegado un paquete BTP. Y se pasa la FL-SDU a la capa superior.

Contiene los siguientes parámetros:

BTP-Data.indication (
Source port, (optional)
Destination port,
Destination port info, (optional)
GN Packet transport type,
GN Destination,
GN Source position vector,
GN Traffic class,
GN Maximum packet lifetime,
Length,
Data
)

Figura 42. Parámetros de la primitiva BTP-Data.indication

Los parámetros significan lo mismo que en el caso de la primitiva anterior.

4.4. Operaciones del protocolo

En este apartado se explica cómo deben operar las entidades BTP para enviar y recibir BTP-PDUs.

4.4.1. Operaciones en el origen

Para que la entidad BTP reciba una primitiva BTP-Data.request, el origen (una entidad de la capa de *facilities*) debe realizar las siguientes operaciones:

1. Crear una BTP-PDU con el ITS-FL-SDU. O lo que es lo mismo, una unidad de datos del protocolo BTP con la información recibida de la capa de *facilities* como carga, y una cabecera de protocolo BTP:
   a. Si el parámetro tipo BTP de la primitiva BTP-Data.request es BTP-A, la cabecera BTP se configura con un puerto origen y otro destino, tal y como aparece en la Figura 43.

Field name	Field setting	Description
Source port	Value of Source port parameter from the BTP-Data.request primitive	Identifies the port of the protocol entity at the source's ITS Facility layer in the source of the BTP PDU.
Destination port	Value of Destination port parameter from the BTP-Data.request primitive	Identifies the protocol entity at the destination ITS Facility layer.

Figura 43. Configuración de los campos para una cabecera BTP-A

b. Si el parámetro tipo BTP de la primitiva BTP-Data.request es BTP-B, la cabecera BTP se configura con un puerto destino y otro campo con información sobre el puerto destino, tal y como aparece en la siguiente figura.

Field name	Field setting	Description
Destination port	Value of Destination port parameter from the BTP-Data.request primitive	Identifies the protocol entity at the destination ITS Facility layer.
Destination port info	Value of Destination port info parameter from the BTP-Data.request primitive	Provides additional info if Destination port is a well-known port. Setting is beyond the scope of the present document, default setting is 0.

Figura 44. Configuración de los campos para una cabecera BTP-B

2. Pasar la BTP-PDU a la entidad del protocolo GeoNetworking a través del punto de acceso GN-SAP por medio de la primitiva GN-Data.request según la configuración de parámetros de la siguiente figura.

Parameter name	Parameter setting
Upper protocol entity	BTP
Packet transport type	Value of *BTP-Data.request* parameter *GN Packet transport type*
Destination	Value of *BTP-Data.request* parameter *GN Destination*
Communication profile	Value of *BTP-Data.request* parameter *Communication profile*
Maximum packet lifetime	Value of *BTP-Data.request* parameter *Maximum packet lifetime*. Omitted if not used in *BTP-Data.request*.
Repetition interval	Value of *BTP-Data.request* parameter *Repetition interval* Omitted if not used in *BTP-Data.request*
Traffic class	Value of *BTP-Data.request* parameter
Length	Length of [FL-SDU +4]
Data	BTP-SDU payload

Figura 45. Configuración de parámetros para la primitiva GN-Data.request para solicitar el envío de un paquete de GeoNetworking

4.4.2. Operaciones en el destino

Cuando se recibe una BTP-PDU a través de una primitiva GN-Data.indication, el destino deberá pasar la carga de dicha PDU por el BTP-SAP con una primitiva BTP-Data.indication según la siguiente configuración de parámetros (como se ve en la sección 5.2 esta GN-Data.indication en realidad no es necesaria):

Parameter name	Parameter setting
Source port	If BTP header type is BTP-A, set to Source port in the BTP-A header. If BTP header type is BTP-B, omit this parameter.
Destination port	Set to Destination port in the BTP-A or BTP-B header.
Destination port info	If BTP header type is BTP-A, omit this parameter. If BTP header type is BTP-B, set to Destination port info in BTP-B header.
GN Packet transport type	Value of GN-Data.indication parameter GN Packet transport type.
GN Destination	Value of GN-Data.indication parameter GN Destination.
GN Source position vector	Value of GN-Data.indication parameter Source position vector.
GN Traffic class	Value GN-Data.indication parameter Traffic class. Omitted if not used in GN-Data.indication.
GN Remaining packet lifetime	Value of GN-Data.indication parameter Remaining packet lifetime. Omitted if not used in GN-Data.indication.
Length	Length of [GN-PDU payload - 4].
Data	BTP-PDU payload.

Figura 46. Configuración de parámetros para la primitiva GN-Data.request para solicitar el envío de un paquete de GeoNetworking

Capítulo 5. Implementación del BTP en OpenC2X

En el capítulo 3 se explicó la arquitectura de OpenC2X y la pila de protocolos del estándar ITS que lleva implementados. En la capa de Red y Transporte se encontraba el protocolo GeoNetworking y el BTP como un mismo módulo, pero los programadores de OpenC2x no lo habían llegado a desarrollar. En este apartado se explicará cómo se diseñó y posteriormente se programó la capa BTP en este trabajo fin de grado.

Aunque en la arquitectura de OpenC2X el GeoNetworking y el BTP se especificaran como un servicio conjunto, estos se deben implementar de forma separada para cumplir el estándar. Como de momento solo queríamos introducir el BTP, se implementó el servicio de manera independiente a la espera de que en el futuro se introduzca el GeoNetworking. En OpenC2X, originalmente, se incluyeron como una función llamada desde el DCC que se encargaba de rellenar las cabeceras, pero sin realizar realmente la función del protocolo, solo como mero sistema que asegura la compatibilidad de los mensajes enviados con el estándar ITS G5.

Para introducir el BTP tenemos que hacer varios cambios en la estructura de comunicación y en el diagrama de puertos de ZMQ, puesto que ahora existirá un nuevo módulo que necesitará comunicarse. Modificando la Figura 17 que vimos en la sección 3.2, quedaría tal y como se ve en la siguiente figura.

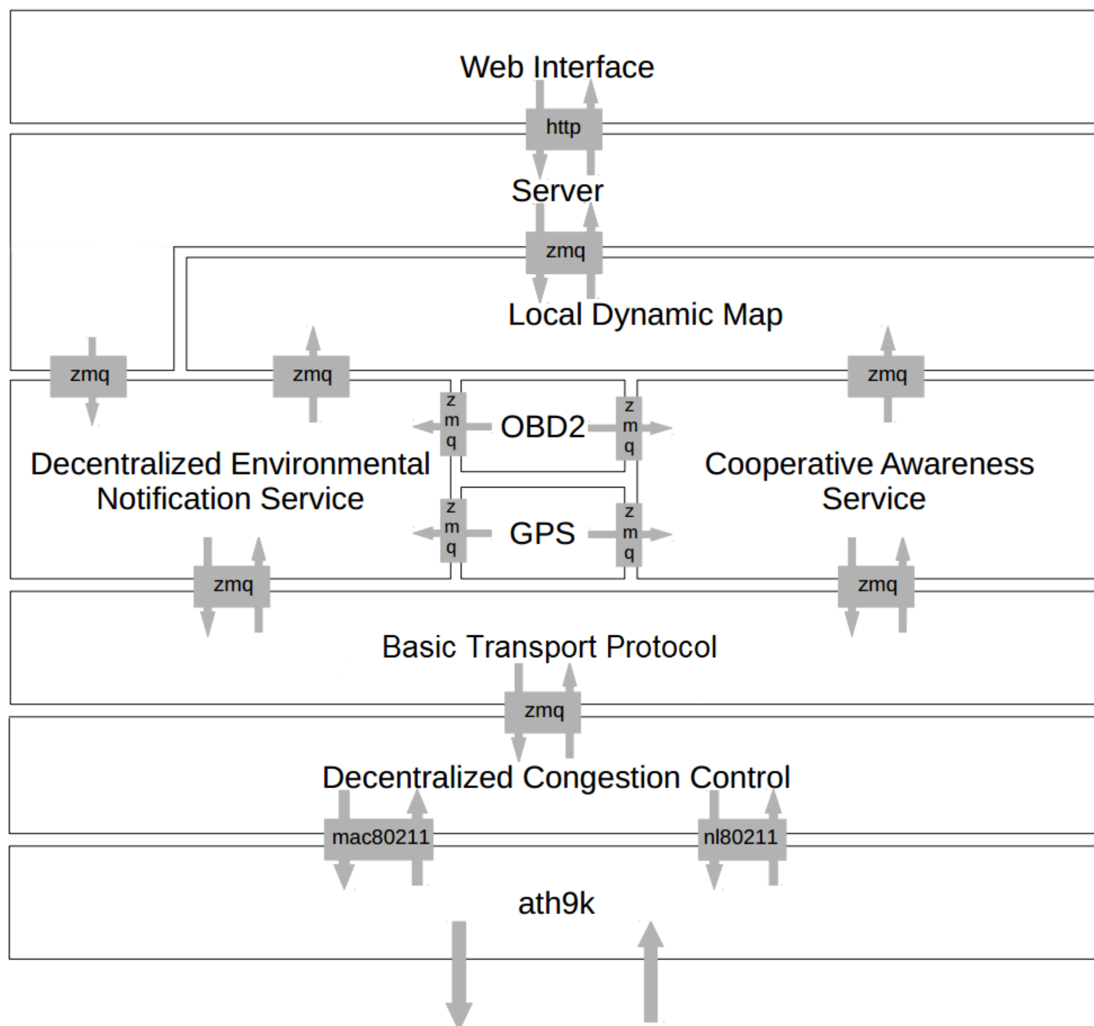

Figura 47. Nuevo esquema de comunicación de los diferentes módulos en OpenC2X con el BTP

Como se explicó en el apartado 4, el BTP funciona como un multiplexor para los módulos de la capa de servicios. Para cumplir esta función se decidió integrar el servicio como se muestra en la anterior figura, como un intermediario entre el DCC y el CAM y el DENM.

Anteriormente, el DCC se encargaba de hacer la multiplexación, lo hacía mediante un trampantojo aprovechando que las cabeceras de GeoNetworking eran siempre iguales. Pero ahora, será el BTP el que se encargará de, según el *Destination Port* que le llegue, mandar el mensaje a un servicio o a otro. Las cabeceras de GeoNetworking seguirán siendo rellenadas por una función de relleno invocada desde el DCC.

El diagrama de puertos ahora quedará de esta forma según se puede ver en la Figura 48. En esta figura es importante fijarse que los servicios DENM y CAM envían por puertos diferentes a BTP la BTP.request, actuando contrariamente a lo que señala el

estándar. Esto se debe a que ZMQ tiene una importante limitación, no permite que dos servicios envíen a un mismo puerto, dado que se crea un conflicto con las direcciones MAC asignadas a estos servicios. En cambio, si permite que varios servicios reciban desde el mismo puerto. Por eso, el CAM y el DENM envía información a los puertos 6666 y 6677 respectivamente y escuchan ambos en el mismo, el 5555. Esto presenta un problema para la implementación, pues habrá que desarrollar dos funciones prácticamente idénticas en el BTP que se ejecuten continuamente para enviar en cada puerto, pues cada una deberá utilizar un *CommunicationSender* diferente.

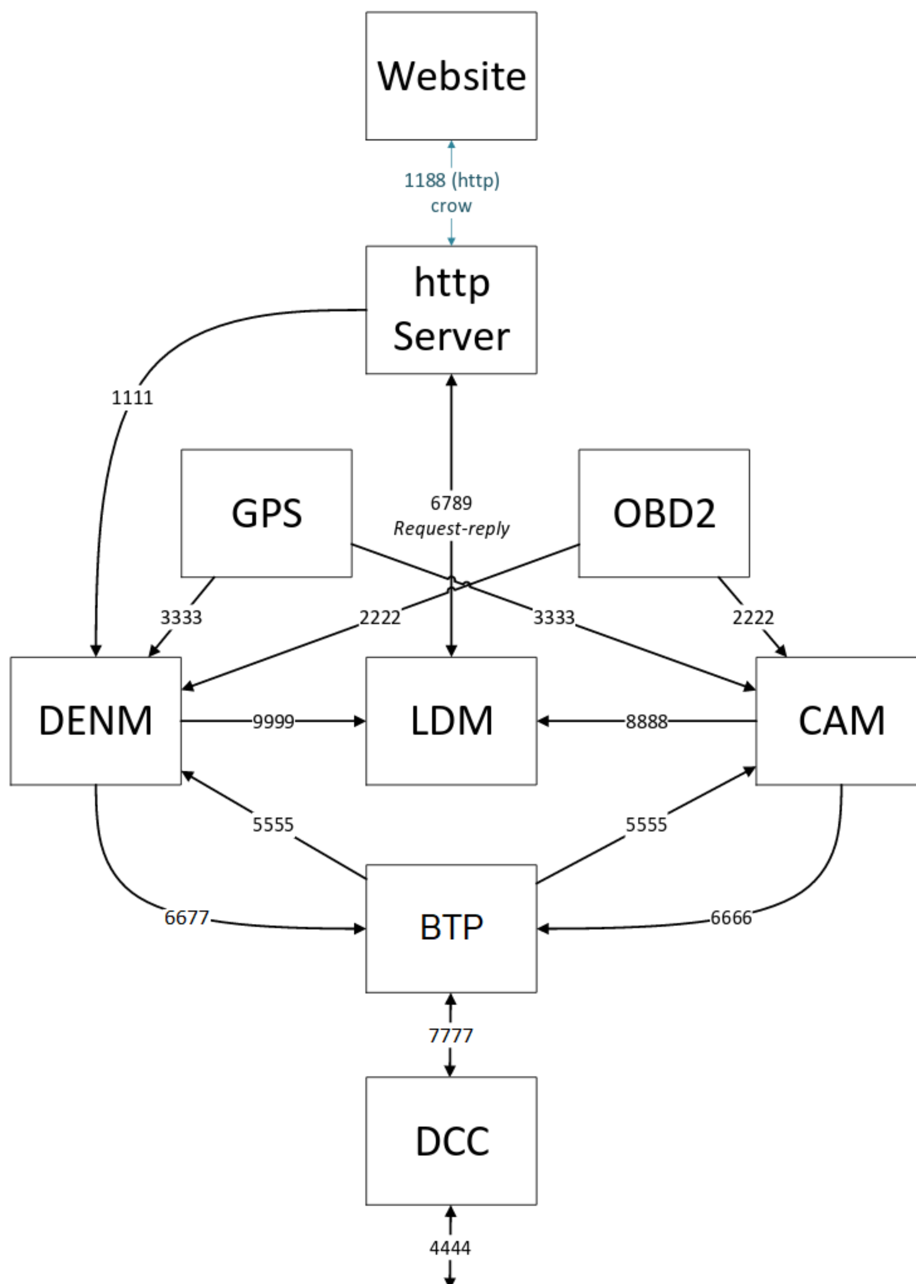

Figura 48. Diagrama de puertos de OpenC2X con el BTP.

5.1. Revisión general del fichero btp.cpp

El fichero *btp/src/btp.cpp* (también es importante el fichero de cabecera *btpsrc/btp.h*) es el núcleo del servicio BTP, en él se encuentran todas las funciones y clases necesarias para su funcionamiento.

La clase principal del servicio es *BTPService*, su constructor se encarga de cargar la configuración global de OpenC2X (línea 43), para obtener el número de experimento, y definir los *CommunicationReceivers* y *CommunicationSenders* correspondientes a los demás servicios (líneas 51 a 56). Además, también crea la herramienta de logging que más adelante nos aportará realimentación (línea 49).

```
40  BTPService::BTPService(string globalConfig, string loggingConf, string statisticConf){
41
42      try {
43          mGlobalConfig.loadConfigXML(globalConfig);
44      }
45      catch (std::exception &e) {
46          cerr << "Error while loading the global config.xml: " << e.what() << endl << flush;
47      }
48
49      mLogger = new LoggingUtility("BTPService", mGlobalConfig.mExpNo, loggingConf, statisticConf);
50
51      mReceiverFromCam = new CommunicationReceiver("BTPService", "6666", "CAM", mGlobalConfig.mExpNo, loggingConf, statisticConf);
52      mReceiverFromDenm = new CommunicationReceiver("BTPService", "6677", "DENM", mGlobalConfig.mExpNo, loggingConf, statisticConf);
53      mReceiverFromDcc = new CommunicationReceiver("BTPService", "5555", "", mGlobalConfig.mExpNo, loggingConf, statisticConf);
54
55      mSenderToDcc = new CommunicationSender("BTPService", "7777", mGlobalConfig.mExpNo, loggingConf, statisticConf);
56      mSenderToServices = new CommunicationSender("BTPService", "5566", mGlobalConfig.mExpNo, loggingConf, statisticConf);
57
58  }
```

Figura 49. Constructor de la clase BTPService, btp/src/btp.cpp

La función *init()* es muy importante, ya que en ella se crean los threads que ejecutan las funciones de recepción y envío que escuchan y envían datos a los servicios contiguos al BTP.

```
77  void BTPService::init(){
78
79      mThreadReceiveFromCam = new boost::thread(&BTPService::receiveFromCamSendToDcc, this);
80      mThreadReceiveFromDenm = new boost::thread(&BTPService::receiveFromDenmSendToDcc, this);
81      mThreadReceiveFromDcc = new boost::thread(&BTPService::receiveFromDccSendToServices, this);
82
83  }
```

Figura 50. Función init() en el BTP, btp/src/btp.cpp

La función *main()* se encarga de lanzar el constructor de la clase *BTPService* y llamar a la función *init()* que a su vez ejecuta las funciones necesarias para lograr un flujo correcto del programa.

```
259 int main(int argc, const char* argv[]) {
260
261     BTPService btp(argv[1], argv[2], argv[3]);
262     btp.init();
263
264     return EXIT_SUCCESS;
265 }
```

Figura 51. Función main() en el BTP, btp/src/btp.cpp

El resto de funciones se explicarán en apartados posteriores junto con la ruta que sigue el mensaje con las nuevas modificaciones al programa.

5.2. Primitivas BTP_REQUEST y BTP_INDICATION

Uno de los detalles más interesantes del estándar BTP es cómo se solicitan los envíos de datos y cómo se indica que ha llegado nueva información. Para ello, en el estándar, se utilizan las primitivas BTP-Data.request y BTP-Data.indication.

Dentro del código decidí que la mejor forma de introducir estos mecanismos era utilizar la herramienta *Protocol Buffers*[22], explicada en la sección 3.3. Para ello se escribieron los ficheros *common/buffers/btpRequest.proto* y *common/buffers/btpIndication.proto.*

Ambos ficheros implementan todos los datos que se especifican en el estándar para las primitivas, aunque se ha optado por poner los datos relacionados con el GeoNetworking de tipo opcional hasta que se integre un servicio que implemente ese estándar en OpenC2X.

Como datos, se utiliza una de las clases DATA definidas anteriormente. Que ya llevan las tramas CAM y DENM encapsuladas.

Un detalle importante es que, aunque en el estándar del BTP se hable del *BTP.indication* en el cual se envía información de control hacia la capa de *facilities,* en el estándar BTP tampoco se especifica muy bien qué hace la capa de *facilities* con esos datos (por lo menos con los que pertenecen al BTP), y si consultamos el estándar CAM y DENM, a la hora de enviar los datos al BTP sí se habla de necesidad de enviar información de control (que coincide con el *BTP.request*) pero en el sentido contrario el estándar dice que solo es necesario enviar hacia la capa de *facilities* la PDU de *facilities*

recibida desde la capa BTP. [10,11]. Por esta razón, se ha decidido que en la parte de recepción en vez de formar una *BTP.indication* se envíen los datos directamente, pero aun así, la clase del *indication* que desarrollada por si en algún futuro fuera necesaria para el código.

```
1  package btp;
2
3  import "data.proto";
4
5  message BTP_REQUEST {
6      enum btp_Type {
7          A = 1;
8          B = 2;
9      }
10     enum GN_Type {
11         GeoUnicast = 1;
12         SHB = 2;
13         TSB = 3;
14         GeoBroadcast = 4;
15         GeoAnycast = 5;
16     }
17     enum GN_Traffic {
18         Relevance = 1;
19         Reliability = 2;
20         Latency = 3;
21     }
22     required btp_Type btpType = 1;
23     optional int32 sourcePort = 2;
24     required int32 destinationPort = 3;
25     optional int32 destinationPortInfo = 4;
26     optional GN_Type gnType = 5;
27     optional int64 gnDestination = 6;
28     optional int32 gnComProfile = 7;
29     optional int64 gnPackLifetime = 8;
30     optional int64 gnRepInterval = 9;
31     optional GN_Traffic gnTraffic = 10;
32     required int32 length = 11;
33     required dataPackage.DATA data = 12;
34 }
```

```
1  package btp;
2
3  import "data.proto";
4
5  message BTP_INDICATION {
6      enum GN_Type {
7          GeoUnicast = 1;
8          SHB = 2;
9          TSB = 3;
10         GeoBroadcast = 4;
11         GeoAnycast = 5;
12     }
13     enum GN_Traffic {
14         Relevance = 1;
15         Reliability = 2;
16         Latency = 3;
17     }
18     optional int32 sourcePort = 1;
19     required int32 destinationPort = 2;
20     optional int32 destinationPortInfo = 3;
21     optional GN_Type gnType = 4;
22     optional int64 gnDestination = 5;
23     optional int64 gnSourcePositionVector = 6;
24     optional GN_Traffic gnTraffic = 7;
25     optional int64 gnPackLifetime = 8;
26     required int32 length = 9;
27     required dataPackage.DATA data = 10;
28 }
```

Figura 52. Ficheros common/buffers/btpRequest.proto y common/buffers/btpIndication.proto.

5.3. Configuración del servicio BTP

En el BTP existen diferentes tipos de parámetros que el usuario debería ser capaz de controlar en función de cómo quiera configurar su estación ITS. En el resto de servicios se ofrece también la posibilidad de configurar diferentes parámetros, por ejemplo, el tiempo de generación de los CAMs, las coordenadas GPS, etc.

En el caso del BTP es necesario que el usuario pueda elegir si desea utilizar una comunicación interactiva o no interactiva (en función del tipo de cabecera) o que información se quiere enviar sobre el puerto de destino. Esta configuración, aunque sea referente al servicio BTP, debe integrarse en el servicio de la capa de *facilities* correspondiente, dado que dicha configuración es enviada directamente por los servicios

superiores como el CAM y el DENM como parte del BTP-DATA.request. Me gustaría señalar que en el presente proyecto solo se implementó la configuración del servicio CAM, modificando el fichero de configuración de este servicio. Para ello, se modificó el fichero *cam/config/config.xml,* de esta manera se puede especificar la configuración BTP para este servicio. El servicio DENM no cuenta con un fichero de configuración, por lo que se dejó para cambios a futuro.

```
1  <cam>
2      <!-- General configuration -->
3      <generateMsgs>true</generateMsgs>
4      <expirationTime>1</expirationTime> <!-- in seconds ! -->
5      <maxGpsAge>10</maxGpsAge> <!-- in seconds ! -->
6      <maxObd2Age>10</maxObd2Age> <!-- in seconds ! -->
7      <thresholdRadiusForHeading>0.30</thresholdRadiusForHeading> <!-- in metres ! -->
8
9      <!--BTP CONFIG-->
10     <!-- True: Header Type A    False: Header Type B -->
11     <typeHeader>false</typeHeader>
12     <!-- Default: 0 -->
13     <destInfo>0</destInfo>
14
15     <!-- Basic Container in CAM -->
16     <isRSU>0</isRSU> <!-- 0 for Vehicle, 1 for RSU -->
17
18 </cam>
```

Figura 53. Fichero de configuración del CAM cam/config/config.xml

```
54  struct CaServiceConfig {
55      bool mGenerateMsgs;
56      int mExpirationTime;
57      int mMaxGpsAge;
58      int mMaxObd2Age;
59      double mThresholdRadiusForHeading;
60      bool mTypeHeader;
61      int mDestInfo;
62      bool mIsRSU;
63
64      void loadConfigXML(const std::string &filename) {
65          boost::property_tree::ptree pt;
66          read_xml(filename, pt);
67
68          mGenerateMsgs = pt.get("cam.generateMsgs", true);
69          mExpirationTime = pt.get("cam.expirationTime", 1);
70          mMaxGpsAge = pt.get("cam.maxGpsAge", 10);
71          mMaxObd2Age = pt.get("cam.maxObd2Age", 10);
72          mThresholdRadiusForHeading = pt.get("cam.thresholdRadiusForHeading", 0.3);
73          mTypeHeader = pt.get("cam.typeHeader", false);
74          mDestInfo = pt.get("cam.destInfo", 0)
75          mIsRSU = pt.get("cam.isRSU", 0);
76      }
77  };
```

Figura 54. Estructura para la configuración del servicio CAM

En el fichero *cam/src/caservice.h* se incluye una estructura y una función para obtener y almacenar los valores de la configuración, similar a la que tienen los demás módulos de OpenC2X. Dicha estructura ha sido convenientemente modificada para que también pueda leer la configuración del servicio BTP.

5.4. Envío de PDUs ITS por OpenC2X

Para conseguir una plena implementación del estándar BTP en OpenC2X, no solo hay que editar los ficheros de código del módulo BTPService, también es necesario modificar los módulos que se encuentran en los extremos de este, el CAM, DENM y el DCC. En el caso del CAM y el DENM las modificaciones son exactamente las mismas, solo habría que rellenar la cabecera BTP con los puertos definidos en [9].

```
331  void CaService::send() {
332      string serializedRequest;
333      dataPackage::DATA* data = new dataPackage::DATA;
334      btp::BTP_REQUEST btpRequest;
335
336      // Standard compliant CAM
337      CAM_t* cam = generateCam();
338      vector<uint8_t> encodedCam = mMsgUtils->encodeMessage(&asn_DEF_CAM, cam);
339      string strCam(encodedCam.begin(), encodedCam.end());
340      mLogger->logDebug("Encoded CAM size: " + to_string(strCam.length()));
341
342      data->set_id(messageID_cam);
343      data->set_type(dataPackage::DATA_Type_CAM);
344      data->set_priority(dataPackage::DATA_Priority_BE);
345
346      int64_t currTime = Utils::currentTime();
347      data->set_createtime(currTime);
348      data->set_validuntil(currTime + mConfig.mExpirationTime*1000*1000*1000);
349      data->set_content(strCam);
350      if(mConfig.mTypeHeader == false){
351          btpRequest.set_btptype(btp::BTP_REQUEST_btp_Type_B);
352      } else {
353          btpRequest.set_btptype(btp::BTP_REQUEST_btp_Type_A);
354      }
355      btpRequest.set_sourceport(2001);
356      btpRequest.set_destinationport(2001);
357      btpRequest.set_destinationportinfo(mConfig.mDestInfo);
358      btpRequest.set_length(strCam.length());
359      btpRequest.set_allocated_data(data);
360
361      btpRequest.SerializeToString(&serializedRequest);
362      mLogger->logInfo("Send new CAM to BTP and LDM\n");
363
364      mSenderToBtp->send("CAM", serializedRequest);   //send serialized DATA to BTP
365
366      camPackage::CAM camProto = convertAsn1toProtoBuf(cam);
367      string serializedProtoCam;
368      camProto.SerializeToString(&serializedProtoCam);
369      mSenderToLdm->send("CAM", serializedProtoCam); //send serialized CAM to LDM
370      asn_DEF_CAM.free_struct(&asn_DEF_CAM, cam, 0);
371  }
```

Figura 55. Función modificada send() del fichero cam/src/caservice.cpp

Al igual que se hizo en el capítulo 3 del proyecto, la primera función que se necesita conocer es la de envío del servicio de la capa de *facilities*, en este caso el CAM. Se ha modificado la función *send()* para que implemente la primitiva BTP.request definida en el estándar. Básicamente, se rellena el objeto DATA (líneas 342 a 349) igual que se hacía anteriormente, y a mayores, se rellena un objeto BTP_REQUEST con los datos que se especifican en el fichero de configuración del cam (en la parte referente a la configuración BTP), por defecto viene definida la configuración con cabecera tipo B, y destination info 0. También es necesario añadirle a la BTP_REQUEST los puertos correspondientes con el CAM (2001) (líneas nuevas líneas). Finalmente, se le encapsula al request el objeto DATA rellenado anteriormente (línea 359), se serializa el request en un string y se envía por *mSenderToBtp* al BTPService.

En el módulo del BTP (BTPService) la función que recogerá el request enviado por el CAM es *receiveFromCamSendToDcc().* Una vez que lo recibe lo deserializa en un nuevo objeto BTP_REQUEST, y extrae todo lo necesario, incluido el objeto DATA que contiene la PDU CAM.

Para rellenar la cabecera se utiliza la función *fillBTPHeader(),* que rellena una estructura de tipo *BTPHeader* con los parámetros que se le pasen. Los datos se extraen del objeto BTP_REQUEST recibido del servicio CAM.

Lo siguiente es añadir la cabecera BTP a la PDU CAM, para ello se utiliza el mismo procedimiento visto anteriormente. Se copia la cabecera en las direcciones de memoria correspondientes. Finalmente se sustituye el contenido anterior del objeto DATA (la PDU CAM sola) por el nuevo (cabecera BTP + PDU CAM) y se serializa en un string.

Por último, se comprueba que el *envelope* recibido coincide con el servicio CAM, y si lo hace se envía. Esto último no es realmente necesario, pues solo se recogen del socket con el CAM los datos que vengan con su *envelope*, pero se realiza esta comprobación por si en una futura implementación del GeoNetworking se modificara algún punto del código que pudiera conducir a error en este punto.

La función *receiveFromDenmSendToDcc()* es idéntica, solo cambia que se rellenan los parámetros con los que corresponde con el DENM.

En el código del DCC no es necesario realizar ningún cambio significativo, solamente en la función *fillGeoNetBTPHeaderForCam()* del fichero SendToHardwareViaMac.cpp, para que no rellene la cabecera BTP, solo la de GeoNetworking.

```cpp
void BTPService::receiveFromCamSendToDcc() {
    string encodedRequest;              // serialized BTP_REQUEST
    string envelope;
    dataPackage::DATA data;             // data received via BTP_REQUEST
    btp::BTP_REQUEST* btpRequest;           // deserialized BTP_REQUEST
        unsigned int btpHdrLen;
    uint8_t* btpHdr;
    size_t camsize;
    string encodedData;

    while (1) {
        pair<string, string> received = mReceiverFromCam->receive();
        envelope = received.first;
        encodedRequest = received.second;

        btpRequest = new btp::BTP_REQUEST();
        btpRequest->ParseFromString(encodedRequest);    // deserialize BTP_REQUEST

        data = btpRequest->data();          // data encapsulated in the BTP_REQUEST
        string encodedCam = data.content();         // CAM string
        camsize = encodedCam.size();

        btp::BTP_REQUEST_btp_Type headerType = btpRequest->btptype();
        int sourcePort = btpRequest->sourceport();
        int destinationPort = btpRequest->destinationport();
        int destinationPortInfo = btpRequest->destinationportinfo();
        fillBTPheader(headerType, sourcePort, destinationPort, destinationPortInfo);

        btpHdrLen = sizeof(struct BTPHeader);
        btpHdr = reinterpret_cast<uint8_t*>(&mBtpHdr);

        unsigned int packetsize = btpHdrLen + camsize;
        unsigned char packet[packetsize];
        unsigned char* payload = packet + btpHdrLen;

        memcpy(&packet, btpHdr, btpHdrLen);         // copy BTP header at the memory address where is located the packet

        memcpy(payload,encodedCam.c_str(),camsize);     // copy payload to packet

        data.set_content(packet, packetsize);       // set the content of the DATA: CAM + BTPHeader
        data.SerializeToString(&encodedData);

        if (envelope.compare("CAM") == 0) {
            mLogger->logInfo("Forward incoming CAM to DCC");
                mSenderToDcc->send("CAM", encodedData);     //Send btp packet to DCC
        } else {
            mLogger->logInfo("INVALID DATA TYPE! EXPECTED CAM BUT " + envelope + " ARRIVED");
        }

    }
}
```

Figura 56. Función receiveFromCamSendToDcc() del fichero btp/src/btp.cpp

5.5. Recepción de PDUs ITS por OpenC2X

En el caso de la recepción pasa como en la anterior, el código del DCC apenas se ha visto afectado. Solo cambia que ya no se eliminan las cabeceras BTP antes de pasarlo a la capa superior, como si se hacía antes. Esto sucedía en la función *receiveWithGeoNetHeader()* del fichero *receiveFromHardwareViaMac.cpp*. Ahora simplemente se elimina la cabecera GeoNetworking y se pasa al BTP la cabecera BTP y la PDU CAM/DENM.

```
90   pair<ReceivedPacketInfo, string> ReceiveFromHardwareViaMAC::receiveWithGeoNetHeader() {
91       while(1) {
92           // receive package, blocking
93           mBytes = read(mSocket, mPacket, sizeof(mPacket));
                   ...
102          int geoNetPDULen = mBytes - mLinkLayerLength;
103          // convert sender Mac from network byte order to char
104          string senderMac = ether_ntoa((struct ether_addr*)mEth_hdr->ether_shost);
105          // Hack! As of now, we are looking for very specific bits in the GeoNetworking header
106          char* geoNetPDU = mPacket + mLinkLayerLength;
107          if (geoNetPDU[5] == 80) {
108              // CAM
109              int btpCamPDULen = geoNetPDULen - sizeof(struct GeoNetHeaderCAM);
110              char* btpCamPDU = geoNetPDU + sizeof(struct GeoNetHeaderCAM);
111              string msg(btpCamPDU, btpCamPDULen);
112              ReceivedPacketInfo info;
113              info.mSenderMac = senderMac;
114              info.mType = dataPackage::DATA_Type_CAM;
115              return make_pair(info, msg);
```

Figura 57. Modificación en *receiveWithGeoNetHeader()*

En el código del BTPService entra en acción la función *receiveFromDccSendToServices()*, que se encarga de recibir del DCC la trama BTP. Una vez la recibe conserva el *envelope* para después. Después extrae la cabecera BTP de los datos recibidos. Y mediante la función *memcpy()* se queda con los dos campos de la cabecera para su posterior evaluación.

Finalmente comprueba el puerto de destino, y en función de si se corresponde con el CAM o con el DENM, enviará los datos a uno o a otro. El otro campo recibido será un puerto origen o información del puerto de destino en función del tipo de cabecera BTP recibida. En el estándar no se concreta cual es la utilidad de estos datos, pueden ser para utilidad del usuario o para una futura aplicación, por lo que se ha decidido que de momento solo se mostrarán al usuario cuando se reciban.

En el caso de la recepción no hace falta realizar ningún cambio en los servicios CAM y DENM, pues simplemente se pasan los datos a la capa superior. Podría utilizarse una primitiva *BTP.indication* para pasar los datos e información de control, pero en ambos estándares no se menciona nada de esta información y lo único necesario es la PDU CAM/DENM.

```
85   void BTPService::receiveFromDccSendToServices() {
86       string envelope;
87       string serializedData;
88       unsigned int dataLen;
89       const char* data;
90       uint16_t btpDestination;
91       uint16_t btpSecondary;
92
93       while (1) {
94           pair<string, string> received = mReceiverFromDcc->receive();
95           envelope = received.first;
96           serializedData = received.second;
97           unsigned int btpPDULen = serializedData.size();
98
99           dataLen = btpPDULen - sizeof(struct BTPHeader);
100          data = serializedData.c_str() + sizeof(struct BTPHeader);
101
102          char btpPDU[btpPDULen];
103          memcpy(&btpPDU, serializedData.c_str(), btpPDULen);
104
105          memcpy(&btpDestination, btpPDU, sizeof(uint16_t));
106
107          memcpy(&btpSecondary, btpPDU + sizeof(uint16_t), sizeof(uint16_t));
108
109          string dataForUpperLayers = string(data, dataLen);
110
111          switch (ntohs(btpDestination)) {
112              case 2001:
113                  mSenderToServices->send("CAM", dataForUpperLayers);
114                  mLogger->logInfo("forward received CAM from source to services");
115                  mLogger->logInfo("Source/Destination Info = " +  to_string(ntohs(btpSecondary)));
116                  mLogger->logInfo("Data = " +  dataForUpperLayers);
117                  break;
118              case 2002:
119                  mSenderToServices->send("DENM", dataForUpperLayers);
120                  mLogger->logInfo("forward received DENM from source to services");
121                  mLogger->logInfo("Source/Destination Info = " +  to_string(ntohs(btpSecondary)));
122                  mLogger->logInfo("Data = " +  dataForUpperLayers);
123                  break;
124              default:
125                  break;
126          }
127      }
128  }
```

Figura 58. Función receiveFromDccSendToServices() del fichero btp/src/btp.cpp

5.6. Problema implementación original OpenC2X

Existe un problema con el desarrollo de OpenC2X en cuanto a la compatibilidad con los estándares de la pila ITS, concretamente con el DCC.

Al no haberse implementado el GeoNetworking + BTP en su día, en la estructura de comunicación, el DCC se comunicaba directamente con la capa de *facilities*, entonces los desarrolladores no vieron necesaria la implementación de la capa cruzada del DCC. Esta capa independiente hace de intermediario con el resto de capas de la pila, para ello utiliza 3 entidades:

1. La entidad de *facilities*, que se encarga de comunicarse con la capa del mismo nombre.
2. La entidad de red, que se encarga de comunicarse con la capa de red y transporte, es decir, con el GeoNetworking y el BTP.
3. Por último, la entidad de enlace, que se comunica con el kernel y la capa de enlace.

La utilidad de esta capa es la intercambiar información de control entre las diferentes capas (tiempos de creación de datos, tiempos de validez de mensajes, etc.) para que, a la hora de realizar la asignación de colas a los diferentes mensajes, no tenga que encargarse de ello la capa principal del DCC, que ya se encarga de realizar el control de congestión de la red (con los datos de la capa transversal) y de enviar los paquetes ITS.

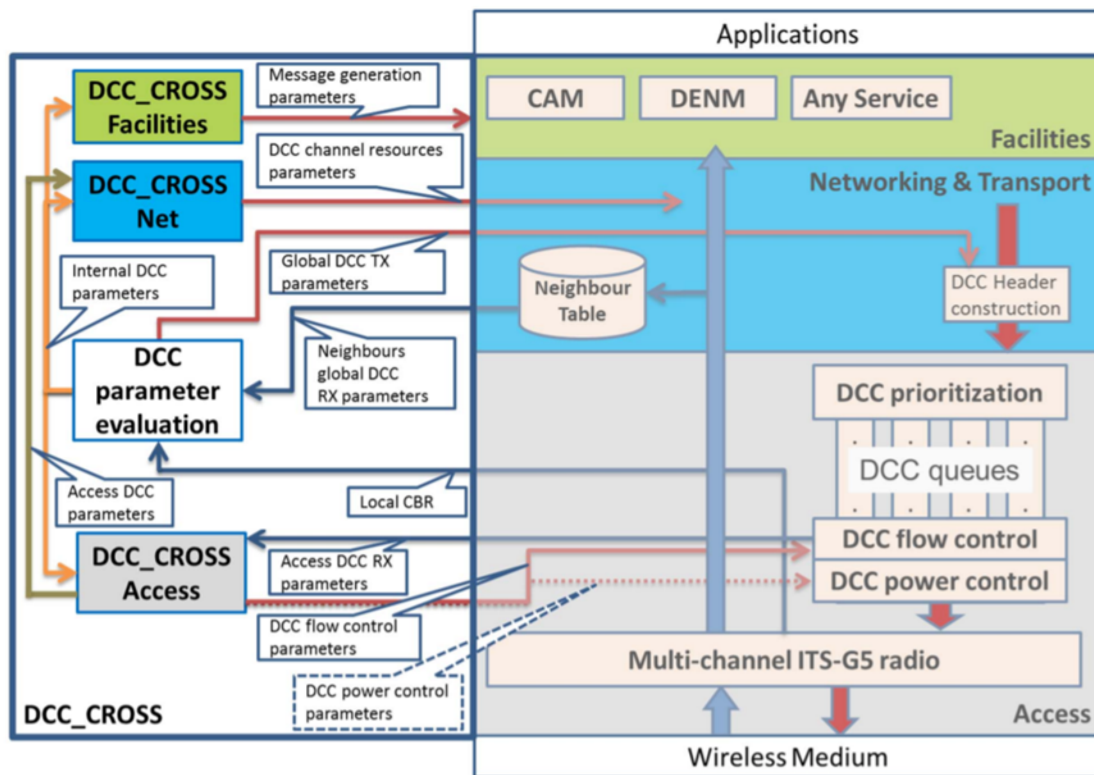

Figura 59. Estructura de comunicación de la capa transversal del DCC (DCC_CROSS)

Volviendo a OpenC2X, si nos fijamos en la estructura de datos DATA, definida para el envío de mensajes de la capa de *facilities* en el fichero *common/buffers/data.proto*, se incluyen diferentes parámetros relacionados con la congestión y el flujo de la comunicación, cuando en teoría solo debería llevar datos correspondientes con las PDUs de los servicios de la capa de *facilities*. Debido a esto, en las BTP.Request, que en su

campo Data solo debería ir contenida la PDU de *facilities*, también se incluye información de control para el DCC que no debería enviar el BTP. Por ello, esto es un problema que debería solucionarse en próximos proyectos: hacer que la estructura DATA solo lleve los datos de las capas superiores y que la capa cruzada del DCC (que también se debe implementar) se encargue de la información de control. El objetivo es que el paquete se encapsule igual que lo hace actualmente, el CAM lo pasa al BTP, este le añade su cabecera y posteriormente se pasa al DCC para que haga control de flujo. Ahora la información de control se pasa por la capa cruzada antes mencionada, por lo que seguirán rutas diferentes y llegaran al DCC por puertos diferentes, con esta modificación se cumpliría lo que especifica el estándar del DCC.

```
1   package dataPackage;
2
3   message DATA {
4       enum Type {
5           CAM = 1;
6           DENM = 2;
7       }
8       enum Priority {
9           BK = 1;
10          BE = 0;
11          VI = 4;
12          VO = 6;
13      }
14      required int32 id = 1;
15      required Type type = 2;
16      required Priority priority = 3;
17      optional int64 createTime = 4;
18      optional int64 validUntil = 5;
19      optional double txPower = 6;
20      optional double bitRate = 7;
21      required bytes content = 8;
22  }
```

Figura 60. Datos correspondientes al control de congestión

Este es un pequeño paso, pero muy necesario, para conseguir que, en un futuro, OpenC2X consiga una implementación completa de la pila ITS.

Capítulo 6. Pruebas de comunicación

Una vez desarrollado todo el código del BTP, lo primero que debemos hacer es compilar de nuevo OpenC2X, para esto se pueden seguir las instrucciones del fichero *README*. Aun así, si es la primera vez que añadimos los ficheros del BTP, hay que configurar el *cmake* para que los incluya en la ruta de compilación.

Cmake utiliza ficheros *.txt* para establecer las rutas de compilación y realizar las configuraciones de dependencias. En el directorio *btp/* hay que añadir un fichero *CMakeLists.txt* con una única instrucción: *add_subdirectory(src)*. Después dentro del directorio */btp/src/* hay que añadir otro *CMakeLists.txt* que contenga lo siguiente:

```
set(btp_SRCS btp.cpp)
add_executable (btp ${btp_SRCS})
target_link_libraries (btp zmq protobuf boost_system boost_thread asn
proto messages utility)
```

Una vez hecho esto ya tendremos configurado *cmake* para que incluya los nuevos ficheros. El siguiente paso es añadir al script de inicio de OpenC2X las siguientes líneas:

```
tmux send-keys "cd $BUILD_DIR/btp/src" C-m
tmuxsend-keys "./btp $GLOBAL_CONFIG $OPENC2X/btp/$LOGGING_CONF
$OPENC2X/btp/$STATISTICS_CONF" C-m
```

Esto nos permitirá lanzar el servicio del BTP y cargar las configuraciones pertinentes. Una vez hecho esto ya solo hace falta ejecutar el script. Así, simplemente se lanzó el nuevo OpenC2X en los ordenadores con los que trabajó Roberto [4], y se cambiaron diferentes parámetros del BTP para ver si los enviaba y recibía bien.

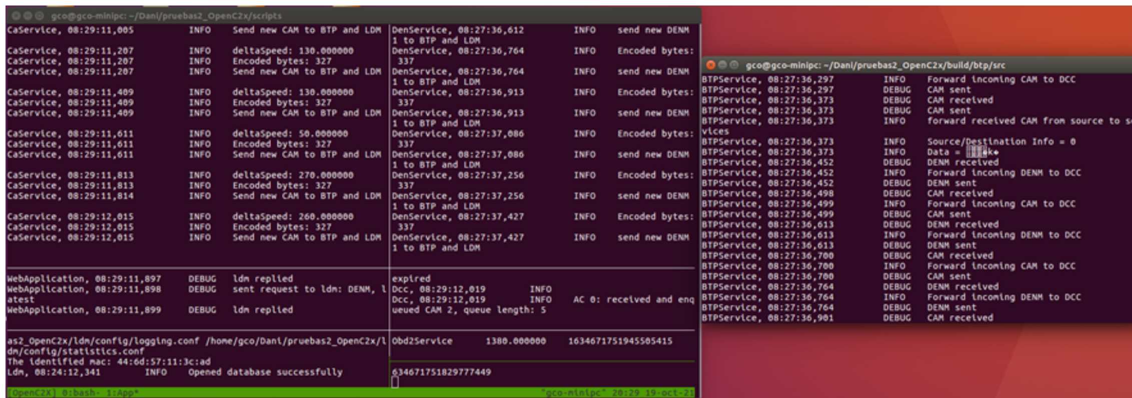

Figura 61. Interfaz del OpenC2X con el servicio del BTP a la derecha.

Al servicio del BTP se le configuró un logger para que mostrara realimentación de cuando enviaba y recibía paquetes. Este servicio corresponde con el terminal pequeño, de la derecha. Se puede observar como es capaz de enviar los mensajes CAM y DENM, e incluso como recibe un CAM del otro PC y muestra su información de destino y los datos recibidos.

También se puede observar el correcto funcionamiento utilizando Wireshark para ver los paquetes correctamente encapsulados. Podemos ver en la siguiente figura como se envían diferentes paquetes CAM y DENM con las diferentes cabeceras necesarias. Si los paquetes estuvieran mal construidos, Wireshark mostraría errores de malformación en las tramas. En amarillo se representan los mensajes generados por el ordenador A y en verde por el ordenador B.

No.	Time	Source	Destination	Protocol	Length	Info
68	48.029058265	1.5.505.44:6d:57:11:3c:ad	Broadcast	CAMv1	99	CAMv1
69	49.946457557	1.5.505.20:16:d8:8b:a2:5a	Broadcast	CAMv1	99	CAMv1
70	50.028955736	1.5.505.44:6d:57:11:3c:ad	Broadcast	CAMv1	99	CAMv1
71	51.857451176	1.5.505.44:6d:57:11:3c:ad	Broadcast	DENMv1	117	DENMv1
72	51.944476428	1.5.505.20:16:d8:8b:a2:5a	Broadcast	CAMv1	99	CAMv1
73	52.010126494	1.5.505.44:6d:57:11:3c:ad	Broadcast	DENMv1	117	DENMv1
74	52.028891044	1.5.505.44:6d:57:11:3c:ad	Broadcast	CAMv1	99	CAMv1
75	52.175182732	1.5.505.44:6d:57:11:3c:ad	Broadcast	DENMv1	117	DENMv1

```
> Frame 1: 99 bytes on wire (792 bits), 99 bytes captured (792 bits) on interface ocb0, id 0
> Ethernet II, Src: LiteonTe_8b:a2:5a (20:16:d8:8b:a2:5a), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
> GeoNetworking
> BTP-B
> Intelligent Transport Systems
```

Figura 62. Tráfico ITS generado con OpenC2X

Si nos vamos a la cabecera de BTP, podemos ver que se envía correctamente con los campos rellenados que especifica el estándar por defecto. Si se modificara el fichero de configuración del BTP podrían enviarse con diferentes valores. En los mensajes CAM se puede observar como se envía con el puerto de destino con el valor 2001 y el puerto de destino a 0, en los DENM se envía el puerto 2002 correspondiente con él.

```
70 50.028955736  1.5.505.44:6d:57:11:3c:ad    Broadcast    CAMv1    99 CAMv1

> Frame 70: 99 bytes on wire (792 bits), 99 bytes captured (792 bits) on interface ocb0, id 0
> Ethernet II, Src: LiteonTe_11:3c:ad (44:6d:57:11:3c:ad), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
> GeoNetworking
v BTP-B
    Destination Port: 2001
    Destination Port info: 0x0000
> Intelligent Transport Systems
```

Figura 63. Cabecera BTP construida por el módulo BTPService de OpenC2X para CAM

```
71 51.857451176  1.5.505.44:6d:57:11:3c:ad    Broadcast    DENMv1    117 DENMv1

> Frame 71: 117 bytes on wire (936 bits), 117 bytes captured (936 bits) on interface ocb0, id 0
> Ethernet II, Src: LiteonTe_11:3c:ad (44:6d:57:11:3c:ad), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
> GeoNetworking
v BTP-B
    Destination Port: 2002
    Destination Port info: 0x0000
> Intelligent Transport Systems
```

Figura 64. Cabecera BTP construida por el módulo BTPService de OpenC2X para DENM

Para realizar las capturas de tráfico en Wireshark es imprescindible utilizar las últimas versiones del programa disponibles, pues en versiones más antiguas, como es el caso de la que está instalada en los ordenadores A y B, no está implementada la pila ITS y el programa no será capaz de interpretar el tráfico.

Capítulo 7. Conclusiones y Líneas Futuras

Como se comentó al principio de este proyecto, el propósito final de este trabajo era la implementación correcta del Protocolo de Transporte Básico (BTP) en el software OpenC2X. Pero también existían algunos otros objetivos que intentar cumplir.

El primero que se exploró fue la actualización del estado de la implementación de 802.11p en Linux, del cual se consiguió concretar a partir de qué versiones había que realizar cambios en el kernel, y en qué consistían estos cambios.

Otro objetivo fue la búsqueda de opciones de mercado y la propuesta de mejora del sistema GPS, los cuales se cumplieron también, pues se encontraron distintos dispositivos que cumplían las especificaciones necesarias para ambos propósitos.

También se consiguió hacer un estudio completo de OpenC2X y del estándar del BTP, de los cuales se llegó a tener un conocimiento bastante avanzado. Por lo que en este aspecto los objetivos se cumplieron con creces.

Finalmente queda el objetivo final, hacer que OpenC2X funcione con una capa del BTP correctamente, este objetivo se ha demostrado en el apartado de pruebas de comunicación. En este apartado se demostró aportando capturas de pantalla del propio software y a partir de captura de red, que los paquetes se formaban correctamente y el programa era capaz de interpretar los menajes enviados y recibidos correctamente.

Aunque se cumplieron los objetivos establecidos al principio, también se encontraron diferentes propuestas de mejora y líneas futuras de trabajo:

- La modificación del módulo del DCC y el desarrollo de la capa cruzada del DCC para que se cumpla el estándar correctamente. Así como la modificación de la clase DATA para que no envíe información de control cuando no tendría que hacerlo.
- La implementación del protocolo GeoNetworking para poder posicionar a los vehículos geográficamente en la red V2X y tener completa la capa de red y transporte especificada en la pila ITS.
- La adquisición de un dispositivo con Dead Reckoning para conseguir el posicionamiento en un solo carril.
- Las pruebas con otras tecnologías inalámbricas como el 5G para comprobar si se obtienen mejores resultados.
- Explorar la compilación de nuevos kernels de Linux en las Hummingboards, resolviendo los múltiples problemas encontrados a lo largo del proyecto: actualización a kernels más nuevos, modificación del CRDA.

Referencias

[1]. Fernández Pastrana, Javier. "802.11p standard and V2X applications on comercial Wi-Fi cards", Universidad de Valladolid, 2017.

[2]. Sánchez Martín, Pilar. "Implementación de una unidad de a bordo de comunicación entre vehículos según el estándar ETSI ITS G5", Universidad de Valladolid, 2017.

[3]. Lobo González, Alejandro. "Implementación de Road Side Unit de comunicación infraestructura vehículos según el estándar ETSI ITS G5", Universidad de Valladolid, 2017.

[4]. Herreras Babón, Roberto. "Integración, testado y propuestas de mejora de una unidad de comunicación V2V", Universidad de Valladolid, 2018.

[5]. Torvalds, Linus. "Official Linux kernel Github Repository", disponible en: https://github.com/torvalds/linux

[6]. CSS Labs, "OpenC2X", disponible en: https://www.ccs-labs.org/software/openc2x/

[7]. Rielb, R., "Vanetza", Technische Hochschule Ingolstadt, 2013. Disponible en: https://github.com/riebl/vanetza

[8]. ETSI TS 102 636-5-1, "Intelligent Transport Systems (ITS); Vehicular Communications; GeoNetworking; Part 5: Transport Protocols; Sub-part 1: Basic Transport Protocol", ETSI, 2021.

[9]. ETSI TS 103 248, “Intelligent Transport Systems (ITS); GeoNetworking; Port Numbers for the Basic Transport Protocol (BTP)”, ETSI, 2018.

[10]. ETSI EN 302 637-2, “Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 2: Specification of Cooperative Awareness Basic Service”, ETSI, 2014.

[11]. ETSI EN 302 637-3, “Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 3: Specifications of Decentralized Environmental Notification Basic Service”, ETSI, 2014.

[12]. ETSI TS 102 871-1, “Intelligent Transport Systems (ITS); Testing; Conformance test specifications for GeoNetworking ITS-G5; Part 1: Test requirements and Protocol Implementation Conformance Statement (PICS) pro forma”, ETSI, 2017.

[13]. ETSI TS 102 871-2, “Intelligent Transport Systems (ITS); Testing; Conformance test specifications for GeoNetworking ITS-G5; Part 2: Test Suite Structure and Test Purposes (TSS & TP)”, ETSI, 2017.

[14]. ETSI TS 102 871-3, “Intelligent Transport Systems (ITS); Testing; Conformance test specifications for GeoNetworking ITS-G5; Part 3: Abstract Test Suite (ATS) and Protocol Implementation eXtra Information for Testing (PIXIT)”, ETSI, 2017.

[15]. ETSI TS 102 636-4-2, “Intelligent Transport Systems (ITS); Vehicular Communications; GeoNetworking; Part 4: Geographical addressing and forwarding for point-to-point and point-to-multipoint communications; Sub-part 2: Media-dependent functionalities for ITS-G5”, ETSI, 2021.

[16]. ETSI TS 103 175, “Intelligent Transport Systems (ITS); Cross Layer DCC Management Entity for operation in the ITS G5A and ITS G5B medium”, ETSI, 2015.

[17]. ETSI TR 101 613, “Intelligent Transport Systems (ITS); Cross Layer DCC Management Entity for operation in the ITS G5A and ITS G5B medium; Validation set-up and results”, ETSI, 2015.

[18]. Sven Laux, Gurjashan Singh Pannu, Stefan Schneider, Jan Tiemann, Florian Klingler, Christoph Sommer, Falko Dressler, “Demo: OpenC2X – An Open Source Experimental and Prototyping Platform Supporting ETSI ITS-G5”, Paderborn University, Germany.

[19]. ETSI EN 302 895, “Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Local Dynamic Map (LDM)”, ETSI, 2014.

[20]. Pieter Hintjens, “ZeroMQ”, O’Reilly Media, 2013.

[21]. Andy Oram, “ØMQ - The Guide”, última vez consultado en 2021, accesible en: https://zguide.zeromq.org/

[22]. Google, “Protocol Buffers”, Google Developer, accesible en: https://developers.google.com/protocol-buffers

[23]. uBlox, “UBX-P3 series, 802.11p V2X chip”, accessible en : https://www.u-blox.com/en/product/ubx-p3-series

[24]. uBlox, “VERA-P1 series, Host-based transceiver modules”, accesible en : https://www.u-blox.com/en/product/vera-p1-series

[25]. Kapsch, “EVK-3300 Evaluation Kit for V2X communications and 802.11p”, accesible en: https://connectedvehicles.kapsch.net/download/Kapsch%20Evaluation%20Kit%20EVK-3300.pdf

[26]. uBlox, “Automotive Dead Reckoning Technology”, Julio de 2021, accesible en: https://www.u-blox.com/en/technologies/automotive-dead-reckoning-technology

[27]. uBlox, “ZED-F9K Module”, accessible en: https://www.u-blox.com/en/product/zed-f9k-module

[28]. H&D Wireless, “Wifi Regulatory Domains in Linux”, accesible en: http://docker.hd-wireless.com/Support/WifiRegulatoryDomainsinLinux

[29]. Yocto Project Website, accesible en: https://www.yoctoproject.org/

[30]. Ezra Stein, Ublox, “VERA-P1 V2X modules end-of-life notification”, 19 de Marzo de 2021, accessible en: https://www.u-blox.com/sites/default/files/VERA-P1_EOL_UBX-21008871.pdf