UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DISCA
DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

Departamento de Informática de Sistemas y Computadores

Desarrollo de un sistema de transmisión de contenidos entre drones y vehículos mediante difusión

Trabajo Fin de Master

Master en Ingeniería de Computadores y Redes

Autor: Sergio Ortiz Mayordomo

Tutor: Carlos Tavares Calafate

2016/2017

Resumen

En los últimos años el uso de drones se está extendiendo rápidamente debido a los muchos campos de aplicación que tienen. Por otro lado, las redes vehiculares están evolucionando muy rápido debido a que los vehículos están equipados con dispositivos de comunicación inalámbrica. Si integramos estos dos elementos, drones y redes vehiculares, se pueden realizar trabajos muy interesantes y altamente novedosos.

El objetivo de este proyecto es el desarrollo de un sistema para entrega de contenidos desde un dron equipado con Raspberry Pi hacia vehículos terrestres (coches, autobuses, camiones, etc.). La información (imagen, voz, imagen+voz o video corto) será codificada a nivel de aplicación usando códigos Raptor (*Forward Error Correction*), y después difundida usando tecnología IEEE 802.11a, siendo alcanzada simultáneamente por un número cualquiera de vehículos receptores, y a continuación reconstruida y visualizada en un terminal Android.

El sistema de entrega de contenido se ha completado en su totalidad y ha sido probado experimentalmente con éxito. Las pruebas han sido realizadas utilizando diferentes dispositivos móviles, lo cual nos ha permitido realizar un estudio comparativo para ver qué dispositivos se adaptan mejor a nuestro sistema.

Palabras clave: Android, Raptor, Raspberry Pi, difusión de contenidos, drones.

Abstract

In recent years, the use of drones has rapidly spread due to their wide scope of application. On the other hand, vehicular networks have evolved very fast because the vehicles are equipped with wireless communication devices. If we integrate these two elements, drones and vehicular networks, very interesting and highly novel solutions can be achieved.

In this work, we develop a system for the delivery of the contents from a drone equipped with Raspberry Pi to terrestrial vehicles (cars, buses, trucks, etc). The information (image, voice, image+voice or short video) will be encoded at the application level using Raptor codes (Forward Error Correction), and then broadcasted using IEEE 802.11a technology, being simultaneously received by any number of the vehicles, and then rebuilt and displayed in an Android terminal.

The proposed Content Delivery System has reached its final development status, and has successfully been tested experimentally. Test have been performed using different mobile devices, allowing us to perform a comparative study to determine which devices are better adapted to our system.

Keywords: Android, Raptor, Raspberry Pi, content diffusion, drones.

Tabla de contenidos

Resumen.....3

Abstract.....4

1. Introducción.....8

1.1. Motivación.....8

1.2. Objetivos.....9

1.3. Estructura del documento.....9

2. Trabajos relacionados.....11

2.1. Difusión de contenidos en redes vehiculares.....11

2.2. Uso de códigos Raptor.....13

3. RaptorQ.....15

3.1. Introducción.....15

3.1.1. Forward Error Correction.....15

3.1.2. Erasure code.....15

3.1.3. Fountain code.....15

3.1.4. Raptor code.....16

3.2. Conceptos.....16

3.3. Diagrama.....17

3.4. OpenRQ.....18

4. Arquitectura Propuesta.....20

4.1. Descripción del entorno.....20

4.2. Detalles sobre el sistema de entrega.....20

4.3. Hardware.....23

4.3.1. Raspberry Pi.....23

4.3.2. Configuración Punto de Acceso.....24

5. Detalles de Implementación.....27

5.1. Tecnologías utilizadas.....27

5.1.1. Java.....27

5.1.2. Android.....28
5.2. Servidor.....32
5.3. Cliente Android.....35
6. Resultados Experimentales.....39
6.1. Perdida paquetes totales.....39
6.2. Tiempo de transmisión.....40
6.3. Tiempo de decodificación.....41
7. Conclusiones.....44
7.1. Trabajo Futuro.....44

Bibliografía.....45

Índice de figuras

Figura 1: Diagrama de RaptorQ.....17
Figura 2: Escenario General.....20
Figura 3: Arquitectura del sistema.....21
Figura 4: Diagrama de flujo remitente/receptor.....22
Figura 5: Raspberry Pi 3 modelo B.....23
Figura 6: Arquitectura de Android.....30
Figura 7: Pantalla Principal.....36
Figura 8: Pantalla Reproducción de Contenido.....37
Figura 9: Pantalla de información.....38
Figura 10: Tiempo total de recepción y decodificación para diferentes tasas de pérdida.....39
Figura 11: Tiempo de transmisión para diferentes tamaños de fichero, y para diferentes modos de transmisión.....40
Figura 12: Gráfica Tiempos de decodificación en el PC y en el móvil para diferentes tamaños de mensaje.....41
Figura 13: Gráfica Tiempos de decodificación en el PC y en el móvil para diferentes tamaños de mensaje I.....42
Figura 14: Tiempos de decodificación en diferentes móviles para diferentes tamaños de mensaje.....43

1 Introducción

1.1. Motivación

En la actualidad el uso de drones ha aumentado de una forma considerable. Su uso ya tiene una función más profesional que recreativo, ya que nos ofrece un gran número de posibilidades en muchos sectores, como puede ser la fotografía, la publicidad, la industria, la medicina y las comunicaciones. Por otra parte, los dispositivos móviles han experimentado un gran avance tecnológico en los últimos años, ya que el rendimiento que nos ofrecen los microprocesadores que llevan integrados es comparable al de un ordenador, y además nos ofrecen otros aspectos como la movilidad y la conectividad.

La combinación de drones, smartphones y vehículos nos permite lograr un amplio abanico de posibilidades, como por ejemplo avisar a los vehículos cercanos de que hay algún obstáculo en la carretera o un accidente cercano. Para ello conviene difundir algún contenido de tamaño reducido, ya sea imagen, sonido o video, de forma continua para que todos los vehículos que pasen por esa zona lo reciban y lo puedan ver en los dispositivos móviles. Todo esto es posible gracias a la movilidad que tienen tanto los drones como los móviles ubicados dentro de los vehículos.

La propuesta que se pretende llevar a cabo en este Trabajo Fin de Máster no consiste únicamente en el envío y recepción de información entre el dron y el dispositivo móvil, sino que pretende ir más allá, aplicando una técnica de corrección de errores en la transmisión de los datos de manera a evitar tener transmisiones independientes para cada vehículo, y asegurando al mismo tiempo que los contenidos a transmitir se reciben correctamente y en su totalidad. La técnica que vamos a usar para ello es la adopción de códigos correctores de errores del tipo FEC conocidos como RaptorQ [1]. De esta forma, lo que buscamos es que la información transmitida llegue y sea reproducida en el móvil lo antes posible.

El objetivo último que se persigue es ayudar a los conductores, avisándoles de antemano de cosas que están ocurriendo en la carretera y que ellos no pueden ver porque están a una distancia lejana, como puede ser un accidente, un obstáculo, o un incendio. Esto les ayudará a tener más tiempo para poder reaccionar y tomar decisiones más óptimas.

1.2. Objetivos

El objetivo de este Trabajo Fin de Máster es el desarrollo de un sistema de comunicación entre drones y vehículos. Para ello el dron irá equipado con una Raspberry Pi que transmitirá la información mediante broadcast usando la tecnología IEEE 802.11a. La información será recibida, reconstruida y reproducida en un terminal Android que actúa como unidad de a bordo del vehículo.

El objetivo general se puede detallar en 3 objetivos específicos:

1. Desarrollo del código servidor, que será el que se ejecute en la Raspberry Pi. Hemos optado por hacerlo en Java, ya que la librería de código abierto OpenRQ, que implementa la funcionalidad de los códigos RaptorQ, está desarrollada en este lenguaje. Esta librería es la que se encarga de codificar y decodificar los datos que vamos a transmitir.

   La función que tiene el servidor es la de codificar los datos y transmitirlos de forma continua.

2. Realización de una aplicación para smartphones. Nos hemos decantado por realizarla en Android ya que, como en el caso anterior, la librería OpenRQ está diseñada en Java, y la podemos adaptar a este sistema operativo para dispositivos móviles.

   La aplicación Android lo que hace es recibir los datos que envía el servidor, y cuando tiene datos suficientes, los decodifica y los reproduce, de manera a poder ser visualizado inmediatamente por el usuario.

3. Una vez desarrollados el código servidor y la aplicación para smartphone, el tercer objetivo será medir las prestaciones alcanzadas con distintos terminales móviles. Es decir, medir los tiempos necesarios para realizar la decodificación de los datos usando móviles con diferentes características.

1.3. Estructura del documento

En el capítulo 2 se hablará sobre los trabajos previos que se han realizado en el área de las redes vehiculares, más concretamente en lo que respecta a difusión de contenido en este tipo de redes. También se mencionarán algunos trabajos en los que se han usado los códigos Raptor.

En el capítulo 3 se hará una introducción al funcionamiento de los algoritmos de corrección de errores, centrándonos en los Raptor codes, y más profundamente en RaptorQ.

En el capítulo 4 se comentará en profundidad la propuesta implementada en el proyecto, detallando la arquitectura y describiendo el sistema de entrega de contenidos. En esta sección también comentamos el hardware que hemos usado, en este caso la Raspberry Pi, y los pasos que hemos seguido para su configuración.

En el capítulo 5 detallaremos la implementación. Primero mencionaremos las tecnologías utilizadas en el proyecto. Después haremos una descripción técnica del servidor y el cliente implementados.

En el capítulo 6 se expondrán los resultados experimentales que se han obtenido mediante la realización de diferentes pruebas con el sistema desarrollado.

Finalmente, en el séptimo y último capítulo, se presentarán las conclusiones y los posibles futuros trabajos que se pueden realizar a partir del sistema desarrollado.

2 Trabajos relacionados

En esta sección vamos a ofrecer una visión general de los trabajos relacionados en las dos áreas que se combinan en este trabajo: difusión de contenido en redes vehiculares y uso de códigos Raptor.

2.1. Difusión de contenido en redes vehiculares

Los avances en la tecnología de las comunicaciones vehiculares están provocando que la distribución de contenido a los vehículos sea más eficaz y cada vez más popular. Como consecuencia, podemos encontrar muchos trabajos relacionados con este tema en la literatura.

Mario Gerla et al. [2] nos presentan un trabajo en el que describen las tecnologías y los protocolos para la distribución de contenidos en redes vehiculares ad-hoc (VANETs). También cubren algunos aspectos como la coexistencia de WiFi y LTE, la aplicación de la codificación en la red, la protección contra los ataques de contaminación, y el soporte a la calidad de servicio en aplicaciones de streaming de video.

Fabricio Aguiar Silva et al. [3] proponen una clasificación de soluciones para la entrega de contenidos en VANETs, al tiempo que describen sus características y el diseño de su arquitectura. Una de las soluciones que nos proponen es la adaptación de las CDN (Content Delivery Network) a las redes vehiculares, lo que nos ofrecería algunas ventajas como la disminución del ancho de banda consumido, y la reducción de la latencia en la entrega del contenido.

Irina Tal y Gabriel-Miro Muntean [4] proponen una solución de entrega de contenido multimedia basada en clústeres y orientada al usuario a través de las VANETs. El sistema es capaz de abordar las preferencias de los pasajeros de los vehículos y ofrecer contenido multimedia de su interés. Las pruebas basadas en simulaciones demuestran cómo esta solución aumenta la estabilidad del sistema, dando como resultado una mejora en la calidad de los servicios de trasmisión multimedia.

Ikecukwu K. Azogu et al. [5] proponen un modelo de Markov (APLM) para medir el nivel de integridad de los esquemas de seguridad para la difusión de contenido en las VANETs. La ventaja del modelo APLM que proponen es su enfoque de caja negra, que mide el nivel de integridad sin necesidad de examinar los detalles de implementación de un esquema de seguridad particular, haciendo el modelo factible para las aplicaciones del mundo real. Teniendo en cuenta las características especiales de los esquemas de integridad en las VANETs, prueban el modelo APLM en cinco escenarios diferentes, demostrando así la significatividad, repetibilidad y viabilidad del modelo APLM.

Marco Fiore et al. [6] proponen y analizan una aplicación para compartir información en VANETs. La aplicación se llama Infoshare, y aprovecha la naturaleza *broadcast* del medio inalámbrico para lograr la máxima difusión de consultas de información entre los vehículos. Para conseguir esto usan una política de caché inteligente que limita la sobrecarga de consultas inútiles y

respuestas duplicadas. Esta propuesta se evalúa utilizando la herramienta de simulación ns-2, destacando el impacto de varios parámetros del sistema sobre la dinámica de difusión de la información.

Jakob Eriksson et al. [7] describen el diseño, la implementación y la evaluación experimental de Cabernet. Cabernet es un sistema de entrega de datos desde y hacia vehículos en movimiento, usando puntos de acceso 802.11 abiertos que encuentran durante el viaje. La conectividad de red en Cabernet es fugaz e intermitente, lo que ocasiona que haya altas tasas de pérdida de paquetes a través del canal. Para mejorar este aspecto los autores introducen dos nuevos componentes. En primer lugar, proponen QuickWifi, que reduce el tiempo medio de conexión a menos de 400 ms, un valor significativamente inferior a los más de 10 segundos alcanzados cuando se utiliza software estándar de red inalámbrica. Lo segundo que proponen es CTP, un protocolo de transporte que distingue la congestión en la parte cableada de la ruta de las pérdidas por el enlace inalámbrico. Con esto se obtiene el doble de rendimiento respecto a TCP.

Huang et al. [8] proponen un esquema de difusión de información para el sistema infotainment de vehículo, permitiendo asegurar la posibilidad de comunicaciones heterogéneas en entornos vehiculares. También proponen un framework para la entrega de servicios en tiempo real a través de una red IP para asegurar la interoperabilidad, roaming y la gestión de la sesión extremo a extremo. La eficacia de su solución se muestra utilizando una herramienta de simulación desarrollada por ellos mismos.

Uichin Lee et al. [9] introducen FleaNet, que consiste en un mercado virtual donde los usuarios pueden realizar transacciones de compra/venta a través de una VANET. Los clientes expresan sus demandas/ofertas para comprar/vender artículos a través de consultas de radio. Estas consultas se difunden de manera oportunista, aprovechando la movilidad de otros clientes, hasta que se encuentre un cliente/proveedor. Los autores identifican las métricas clave de rendimiento – latencia, escalabilidad, movilidad y agitación –, y evalúan su impacto en el rendimiento utilizando modelos analíticos y simulación.

Carlos T. Calafate et al. [10] nos presentan una solución eficiente y robusta de un sistema de entrega de contenido basado en broadcast. Su objetivo es el de reducir el tiempo en la entrega de contenido, y para ello lo que hacen es optimizar el rendimiento buscando el tamaño óptimo del paquete. El objetivo se logra combinando resultados analíticos y de simulación, y considerando receptores estáticos y móviles a diferentes distancias del transmisor. Además, desarrollan una arquitectura que integra el protocolo FLUTE con diferentes esquemas FEC para lograr una distribución de contenido eficiente.

2.2. Uso de códigos Raptor

Los códigos Raptor son una de las técnicas de corrección de errores (FEC) más eficientes. Se usan para controlar los errores en la transmisión de datos por canales de comunicación poco fiables o con mucho ruido. El acrónimo Raptor viene de Rapid Tornado, y representa una evolución de los primeros tipos de "Erasure Codes". La técnica fue inventada por Amin Shokrollahi en 2001, y se publicó por primera vez en 2004 [11].

Posteriormente, los códigos Raptor se han usado para realizar algunos trabajos relevantes.

Ufuk DEMIR et al. [12] realizan un estudio en el que hacen una comparación de dos códigos de corrección de errores alternativos a nivel de paquete, Raptor code y Reed Solomon. El objetivo del estudio es averiguar qué casos son apropiados para el uso de cada uno de estos códigos.

Michael Luby et al. [13] nos presentan un trabajo en el que analizan la entrega de archivos a través de redes móviles de difusión. Para ello usan los códigos Raptor para la entrega del contenido multimedia. Primero, comienzan describiendo los códigos LT, para luego, usando una notación de álgebra lineal fácil de entender, describir los códigos Raptor como una poderosa extensión de los códigos LT. Después proporcionan información de cómo implementar los codificadores y decodificadores usando estos códigos. Finalmente, muestran los resultados de algunas simulaciones en las que verifican el buen rendimiento de la distribución de archivos usando los códigos Raptor.

Saejoon Kim et al. [14] nos describen un algoritmo eficiente para la máxima probabilidad de decodificación en los códigos Raptor usados a través del canal de borrado binario. El algoritmo está inspirado en el esquema de decodificación sugerido en el 3GPP Multimedia Broadcast/Multicast Services, y es una versión mejorada de esta.

Todor Mladenov et al. [15] nos ofrecen un trabajo en el que analizan los trade-offs involucrados en la implementación de sistemas que emplean MBMS (Multimedia Broadcast Multicast Service). Para ello muestran el rendimiento para varias tasas de borrado de paquetes con un overhead fijo y variable del símbolo del código Raptor. También analizan la calidad del servicio, para diferentes tamaños de bloques y símbolos, y el rendimiento del sistema. Y para finalizar, investigan los límites de rendimiento del decodificador Raptor en receptores móviles.

Más recientemente surgió la última generación de los códigos Raptor conocida como RaptorQ, que también se ha usado en algunos trabajos.

Linjia Hu et al. [16] nos muestran en su artículo la idoneidad de usar la GPU con el código RaptorQ para procesar grandes bloques y tamaños de símbolos en las transmisiones FEC (Forward Error Correction). Lo primero que hacen es explorar las implementaciones en serie y paralelo del código RaptorQ en la CPU y la GPU. Para ello realizan simulaciones que han permitido averiguar el grado de cumplimiento con requisitos de tiempo real en servicios Broadcast y Multicast, y en la difusión de video digital. Finalmente concluyen con que una paralelización eficiente usando la GPU puede mejorar significativamente el rendimiento del decodificador.

Todor Mladenov et al. [17] hacen una comparación de las propiedades de codificación, el rendimiento de decodificación y la eficiencia energética de los dos códigos, Raptor y RaptorQ, en un sistema embebido. Las simulaciones son realizadas en un escenario práctico de MBMS (Multimedia Broadcast/Multicast Services). Finalmente concluyen que en el rendimiento de codificación RaptorQ es mejor, pero consume más energía que la codificación Raptor.

Michael Luby [18] describe las mejores técnicas prácticas para el uso de los códigos de corrección de errores y el intercalado de los datos en la difusión de contenido multimedia. También nos muestra la metodología que debe seguir el receptor para la decodificación de los datos recibidos. Todo ello lo realiza incorporando el código RaptorQ junto con técnicas de difusión entrelazadas. Con todo lo anterior Luby nos garantiza que los usuarios van a obtener una calidad en la visualización de los contenidos multimedia y una buena duración de la batería con un uso mínimo de los recursos de la CPU en los dispositivos móviles.

3 RaptorQ

En este apartado vamos a describir los códigos RaptorQ. Para ello comenzaremos con una introducción de los principales tipos de técnicas en este ámbito; seguidamente veremos los conceptos más importantes, el diagrama de funcionamiento, y finalizaremos explicando OpenRQ.

3.1. Introducción

Antes de definir RaptorQ vamos a hacer una introducción donde veremos una serie de definiciones que nos ayudarán a entenderlo mejor.

3.1.1. Forward Error Correction

El FEC (Forward Error Correction) es una técnica usada para contrarrestar los efectos de los errores en la transmisión de datos a través de canales pocos fiables y con ruido. La idea principal consiste en que el transmisor codifique el mensaje de una forma redundante usando algún código de corrección de errores. El primero de estos códigos fue el de Hamming, que lo inventó en 1950 el matemático americano Richard Hamming.

Los códigos FEC le dan al receptor la capacidad de corregir errores sin necesidad de un canal inverso para solicitar la retransmisión de datos, pero a costa de un ancho de banda de canal ligeramente superior, y un aumento significativo en la complejidad del codificador y del decodificador. Por este motivo los códigos FEC son aplicados en situaciones en las que las retransmisiones son costosas o imposibles, como los enlaces de comunicación unidireccionales, o cuando la transmisión se realiza a múltiples receptores en simultáneo vía *broadcast*.

3.1.2. Erasure codes

Los Erasure codes son un código FEC con la capacidad de recuperarse de las pérdidas en las comunicaciones. Los datos se dividen en k símbolos, que se transforman en un mensaje más largo con n símbolos, de forma que el mensaje original pueda ser recuperado con un subconjunto de los n símbolos.

3.1.3. Fountain codes

Los Fountain code son una clase de Erasure code. Los Fountain code tienen dos propiedades importantes:

- se puede generar un número arbitrario de símbolos de codificación al vuelo, simplificando la adaptación a diferentes tasas de pérdida.
- los datos pueden ser reconstruidos con una alta probabilidad desde cualquier subconjunto de símbolos codificados.

Los LT code fueron la primera implementación práctica de los Fountain codes. Posteriormente se introdujeron los Raptor codes, con los que se logró una codificación en tiempo lineal, y una complejidad de decodificación más reducida a través de una etapa de pre-codificación de los símbolos de entrada.

3.1.4. Raptor codes

Los Raptor codes son la primera clase conocida de Fountain codes con codificación y decodificación en tiempo lineal. Fueron inventados por Amin Shokrollahi en el 2001, y son una mejora teórica y práctica sobre los códigos LT.

Los Raptor codes codifican un mensaje dado generando una secuencia de k símbolos de codificación. Cuando en el receptor conocemos k o más símbolos codificados mediante Raptor codes es posible recuperar el mensaje original con una probabilidad no nula. La probabilidad de que el mensaje se pueda recuperar aumenta con el número de símbolos recibidos. Por encima de k símbolos la probabilidad está muy cerca de 1, siendo nula si tenemos menos de k símbolos.

Por ejemplo, teniendo los datos divididos en k símbolos, el decodificador RaptorQ puede:

- Con un conjunto de k símbolos codificados, decodificar con éxito los datos originales con una probabilidad del 99%.
- Con un conjunto de k+1 símbolos codificados, decodificar con éxito los datos originales con una probabilidad del 99.99%.
- Con un conjunto de k+2 símbolos codificados, decodificar con éxito los datos originales con una probabilidad del 99.9999%.

Hay dos tipos de Raptor codes: los sistemáticos y los no sistemáticos. En el caso de los sistemáticos, los símbolos del mensaje original están incluidos dentro del conjunto de símbolos de codificación. Un ejemplo de código sistemático es el definido por el 3rd Generation Partnership Project para su uso en la difusión de contenido en redes celulares móviles.

Actualmente la versión más avanzada de Raptor code es RaptorQ, la cual vamos a describir en las secciones siguientes.

3.2. Conceptos

A continuación, vamos a ver los conceptos más importantes relacionados con RaptorQ.

Datos origen: es el objeto que va a ser codificado/decodificado por RaptorQ. Puede ser cualquier secuencia de bits, como un archivo o un flujo de datos.

Bloques origen: los datos de origen primero se particionan en porciones contiguas, llamadas bloques origen, para permitir una decodificación eficiente.

Símbolos: son la unidad de datos más básica de tamaño fijo utilizada en RaptorQ. Se pueden dividir en dos tipos:

Símbolos origen: son una porción contigua de un bloque origen.

Símbolos de reparación: es una pieza de datos generada que contiene la redundancia necesaria para asegurar la recuperación de errores con alta probabilidad.

Parámetros FEC: es un conjunto de información que se envía desde el remitente al receptor antes de que el receptor comience a recopilar los símbolos. Los parámetros FEC están formados por:

Longitud datos origen: la longitud de los datos origen en número de bytes. Una vez que los datos codificados pueden contener bytes de relleno adicionales, este valor permite que el decodificador infiera el tamaño real de los datos decodificados.

Tamaño del símbolo: el tamaño de un símbolo en número de bytes. Este valor representa el tamaño de un símbolo de codificación (origen o de reparación), excepto el tamaño del último símbolo de origen, que puede ser menor.

Número de bloques origen: el número de bloques origen en los que se dividen los datos de origen, en nuestro caso va a tener un valor de 1.

Longitud del intercalado: este valor por defecto es siempre 1.

3.3. Diagrama

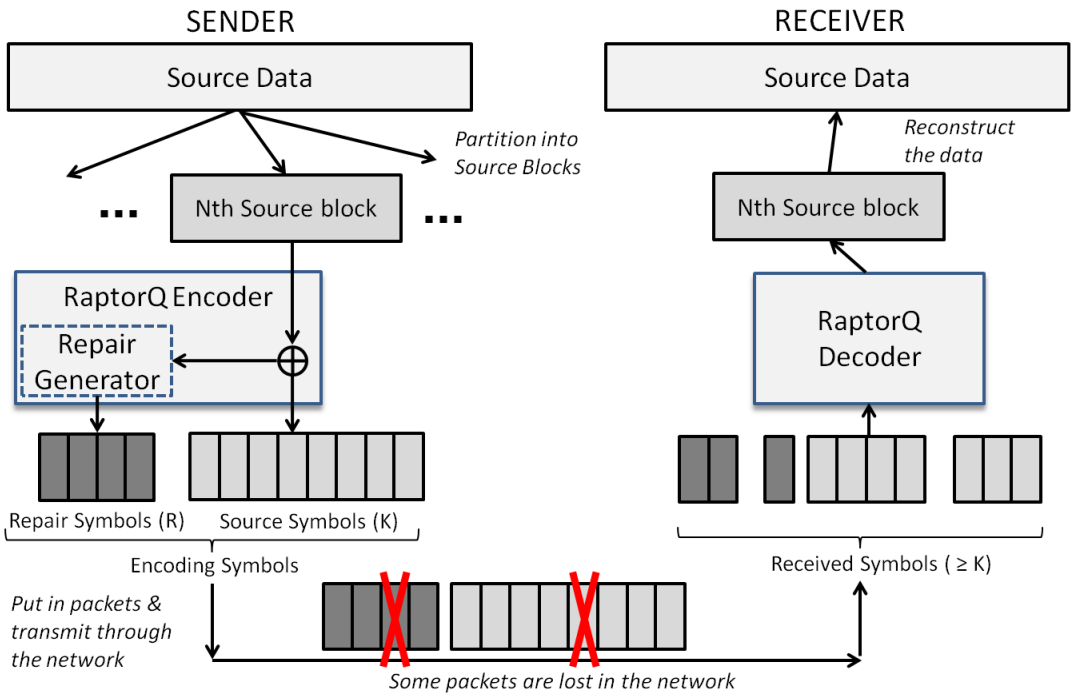

Figura 1: Diagrama de RaptorQ

En el diagrama podemos ver los pasos que se siguen al usar RaptorQ.

1. Los datos se dividen en varios bloques de origen, en nuestro caso solo lo vamos a dividir en un bloque ya que los datos de origen que usamos no son de gran tamaño.
2. A partir del bloque de origen se generan los símbolos de origen y los de reparación.
3. Los símbolos se ponen en paquetes y se transmiten a través de la red. Cada paquete estará formado por un único símbolo.
4. Cuando en el receptor hay suficientes paquetes, se decodifican y se reconstruyen los datos origen.

3.4. OpenRQ

En el proyecto hemos usado la librería OpenRQ, ya que ofrece una API Java de código abierto.

OpenRQ es una implementación de RaptorQ que sigue exactamente las instrucciones del RFC 6330.

Proporciona las siguientes características:

- Divide los datos en bloques, donde cada bloque puede ser codificado/decodificado independientemente.
- Codifica bloques en paquetes individuales que pueden ser transmitidos independientemente al receptor.
- Decodifica bloques desde paquetes individuales recibidos en cualquier orden.
- Permite configurar parámetros de codificación/decodificación tales como número de bloques y tamaño de paquete.
- Ofrece múltiples formas de transmitir paquetes o parámetros de configuración.

OpenRQ está diseñado para ser usado en canales de comunicación con mucho ruido o poco fiables, ya que nos ofrece un alto porcentajes de recuperación de errores. Sin embargo, OpenRQ también tiene algunas limitaciones que se concentran en el rendimiento. Concretamente, observamos experimentalmente que el rendimiento no es óptimo debido a que la función de decodificación consume muchos recursos, convirtiéndose en cuello de botella.

La última versión que hay es la OpenRQ API 3.3.2. La API está dividida en cuatro paquetes:

net.fec.openrq: es el paquete principal de la API de OpenRQ.

net.fec.openrq.decoder: contiene las clases relacionadas con el decodificador de RaptorQ.

net.fec.openrq.encoder: contiene las clases relacionadas con el codificador de RaptorQ.

net.fec.openrq.parameters: contiene las clases para manejar los parámetros FEC.

4 Arquitectura propuesta

En este apartado vamos a describir la arquitectura propuesta en nuestro trabajo. Primero describiremos el entorno de nuestra aplicación, y después detallaremos el sistema de entrega de contenido. Acabaremos hablando del hardware que hemos usado para la realización de esta arquitectura.

4.1. Descripción del entorno

El entorno de este trabajo se centra en un escenario específico de una Red de Área Vehicular donde, por un lado, tenemos el dron que está enviando la información, y por otro lado tenemos los vehículos que la están recibiendo.

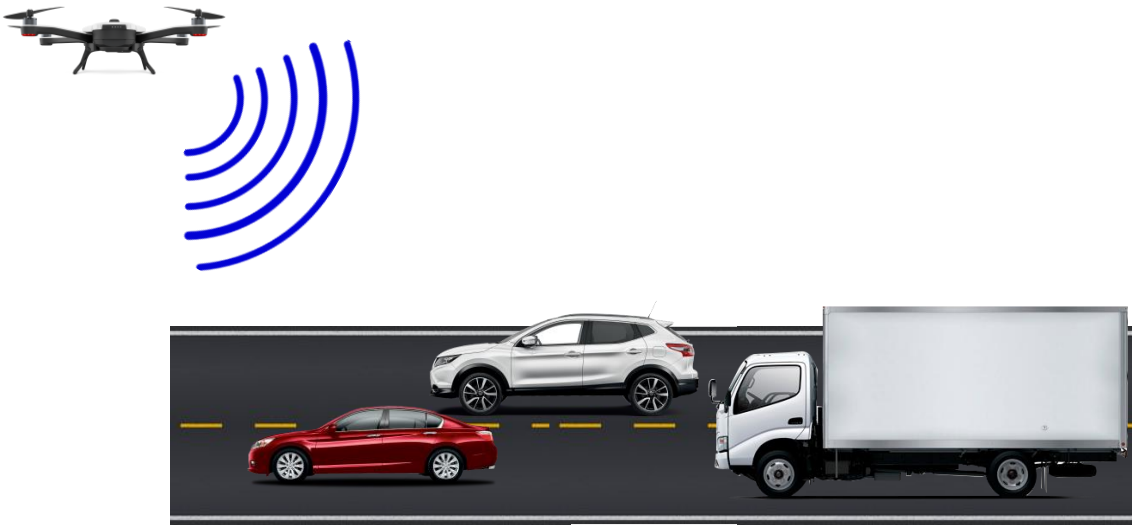

Figura 2: Escenario general.

Para poder realizar la comunicación hemos configurado la Raspberry Pi que se encuentra en el dron como un punto de acceso. Los dispositivos móviles, para poder recibir los datos, se conectan a la red que crea la Raspberry Pi. A parte de hacer la función de punto de acceso, la Raspberry Pi también hace la función de servidor, transmitiendo los datos en broadcast usando la tecnología 802.11a.

4.2. Detalles sobre el sistema de entrega

El sistema desarrollado usas las características de los esquemas FEC para proporcionar un sistema de entrega de contenido robusto y fiable para el entorno de las redes vehiculares.

En la siguiente figura 3 podemos ver la arquitectura en capas del sistema propuesto.

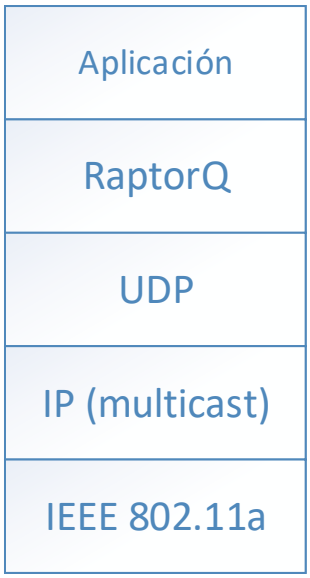

Figura 3: Arquitectura del sistema.

En la capa de aplicación hemos desarrollado dos componentes diferentes: el servidor, que está ubicado en el dron, y el receptor, que está localizado en el lado del cliente móvil. Ambas aplicaciones usan el esquema FEC basado en RaptorQ para hacer la transferencia más rápida, más robusta y más confiable.

Respecto a las capas física y de enlace, usamos la tecnología 802.11a. Esta tecnología nos permite operar en la banda de 5 GHz, que está menos congestionada que la de 2.4 GHz, lo que ocasiona que las interferencias sean menores. Otra ventaja que también nos ofrece es la de lograr, en general, mayor velocidad, pero en contra se da la situación que aún no todos los dispositivos son compatibles con esta frecuencia.

En la figura 4 podemos ver el diagrama de flujo del servidor y cliente (receptor). El servidor realiza las siguientes operaciones:

1. Trata los datos para transmitirlos en modo binario.
2. A partir de los datos se crea el bloque origen. En nuestro caso solo vamos a dividir los datos en un solo bloque origen.
3. Con el bloque origen se crean los símbolos origen y los de reparación.
4. Cada símbolo se pone en un paquete y se transmite a través de la red.

El receptor hace las operaciones de forma inversa a las del servidor:

1. Espera a recibir los paquetes.
2. Analiza el paquete para ver el contenido que ha recibido.
3. Almacena los símbolos hasta que tiene los suficientes para poder decodificar los datos.
4. Reconstruye de los datos.

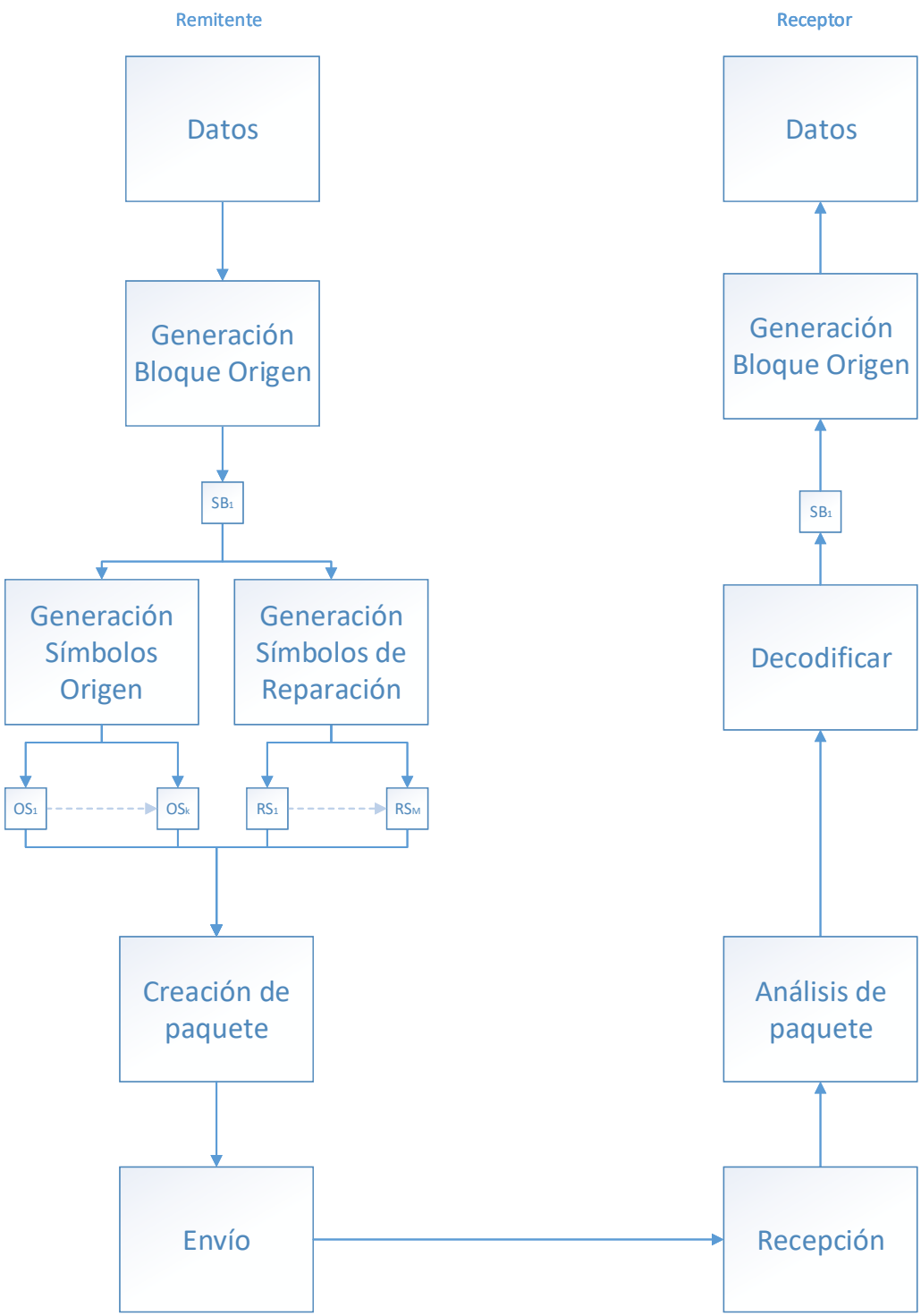

Figura 4: Diagrama de flujo remitente/receptor

4.3. Hardware

A continuación, vamos a hacer una breve descripción del hardware utilizado en este proyecto.

4.3.1. Raspberry Pi

La Raspberry Pi es un ordenador de bajo coste desarrollado en Reino Unido por la Fundación Raspberry Pi, con el objetivo de estimular la enseñanza de ciencias de la computación en las escuelas. Tiene el tamaño de una tarjeta de crédito, que se conecta a un monitor de computador o TV, y utiliza un teclado estándar y un ratón. El sistema operativo oficial para la Raspberry Pi es una versión oficial adaptada de Debian, denominada Raspbian, aunque permite otros sistemas operativos, incluido una versión de Windows 10.

Figura 5: Raspberry Pi 3 modelo B.

En la figura podemos ver una Raspberry Pi 3 modelo B, idéntica a la que hemos utilizado en este proyecto. Sus detalles técnicos son los siguientes:

- Procesador ARMv8 1.2GHz 64-bit Quad-core
- 1 GB LPDDR2 SDRAM (2x memorias)
- 802.11n Wireless LAN

- Bluetooth 4.1
- Bluetooth Low Energy (BLE)
- 4 puertos USB
- 1 puerto Ethernet (IEEE 802.3)
- 1 lector de tarjetas MicroSD
- Salida de Video y Audio – HDMI y AV 3.5mm Jack
- Conector para GPIOs con 2x20 pines
- Conector para Cámara
- Conector para Pantalla

La Raspberry Pi 3 tiene un peso aproximado de 45 g y dimensiones de 8.6 cm de largo y 5.7 cm de ancho.

4.3.2. Configuración en modo Punto de Acceso

A continuación, vamos a detallar los pasos para poder configurar la Raspberry Pi como un punto de acceso inalámbrico.

Lo primero que tenemos que hacer es instalar el servidor DHCP y el servicio de creación de puntos de acceso, ya que no vienen instalados por defecto en Raspbian. Para ello tecleamos:

sudo apt-get install isc-dhcp-server hostapd

Después de este punto es necesario reiniciar la Raspberry, para poder empezar con la configuración.

Lo primero que configuramos es el servidor DHCP. Para ello editamos el siguiente archivo:

sudo nano /etc/dhcp/dhcpd.conf

En este archivo buscamos las siguientes líneas y las comentamos:

#option domain-name "example.org";

#option domain-name-servers ns1.example.org, ns2.example.org;

Después descomentamos el elemento #authoritative, que por defecto está comentado:

authoritative;

Para finalizar esta parte configuramos la red en la que funcionará el servidor DHCP, por ejemplo, en la red 192.168.2.0. Para ello añadimos al final del fichero lo siguiente:

subnet 192.168.2.0 netmask 255.255.255.0 {

range 192.168.2.2 192.168.2.30;

option broadcast-address 192.168.2.255;

option routers 192.168.2.1;

default-lease-time 600;

max-lease-time 7200;

option domain-name "local";

option domain-name-servers 8.8.8.8, 8.8.4.4;

}

A continuación, editamos el siguiente archivo de configuración del servidor:

sudo nano /etc/default/isc-dhcp-server

Buscamos en el archivo la línea INTERFACES="" y la cambiamos por:

INTERFACES="wlan0"

Con esto el servidor DHCP ya está configurado. Ahora configuramos la conexión WLAN. Para ello desconectamos la tarjeta Wi-Fi mediante el siguiente comando:

sudo ifdown wlan0

Después editamos el archivo "interfaces":

nano /etc/network/interfaces

Y lo configuramos de la siguiente manera:

auto lo

iface lo inet loopback

iface eth0 inet dhcp

allow-hotplug wlan0

iface wlan0 inet static

address 192.168.2.1

netmask 255.255.255.0

Las demás líneas las comentamos. Para aplicar los cambios de manera inmediata tecleamos:

sudo ifconfig wlan0 192.168.2.1

Ahora configuramos el punto de acceso. Para ello editamos el archivo:

sudo nano /etc/hostapd/hostapd.conf

En este archivo sustituimos lo que haya por lo siguiente:

interface=wlan0

ssid=RaspAP

hw_mode=a

channel=46

macaddr_acl=0

auth_algs=1

ignore_broadcast_ssid=0

wpa=2

wpa_passphrase=password

wpa_key_mgmt=WPA-PSK

wpa_pairwise=TKIP

rsn_pairwise=CCMP

Tanto el SSID, como el wpa_passphrase y el canal lo podemos configurar con los valores que nosotros queramos.

Para finalizar con la configuración editamos el archivo:

sudo nano /etc/default/hostapd

Descomentamos y cambiamos la línea #DAEMON_CONF="" por:

DAEMON_CONF="/etc/hostapd/hostapd.conf"

Por último, para que el punto de acceso y el servidor DHCP arranquen de forma automática con nuestro dispositivo, tenemos que teclear lo siguiente:

sudo update-rc.d hostapd enable

sudo update-rc.d isc-dhcp-server enable

Ahora solamente queda reiniciar la Raspberry Pi. Cuando vuelva a arrancar automáticamente creará el punto de acceso y podrá asignar direcciones IP a los hosts que se conecten a él.

5 Detalles de implementación

En este apartado vamos a describir las tecnologías que hemos utilizado para la implementación de nuestro proyecto. Seguidamente detallaremos la implementación del Servidor y de la aplicación Android que se usará en el lado del cliente.

5.1. Tecnologías utilizadas

5.1.1. Java

Java es un lenguaje de programación de propósito general, concurrente y orientado a objetos que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo, lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra.

Java se ha convertido, a partir de 2012, en uno de los lenguajes de programación más populares, especialmente para aplicaciones de cliente-servidor web, con unos 10 millones de usuarios reportados.

Java fue originalmente desarrollado por James Gosling de Sun Microsystems, y publicado en 1995 como un componente fundamental de la plataforma Java de Sun Microsystems. Su sintaxis deriva en gran medida de C y C++, pero tiene menos utilidades de bajo nivel que cualquiera de ellos. Las aplicaciones de Java son generalmente compiladas a bytecode (clase de java) que puede ejecutarse en cualquier máquina virtual Java (JVM) sin importar la arquitectura de la computadora.

En nuestro proyecto nos hemos decantado por usar el lenguaje de programación Java porque nos permite ejecutar el código en diferentes plataformas. Esto nos ha permitido realizar el código del Servidor primero en un ordenador y después exportarlo a la Raspberry Pi, sin tener por ello que realizar ninguna modificación.

5.1.2. Android

Android es un sistema operativo basado en el núcleo Linux. Fue diseñado principalmente para dispositivos móviles con pantalla táctil, como teléfonos inteligentes, tablets, y también para relojes inteligentes, televisores y automóviles. Inicialmente fue desarrollado por Android Inc., empresa que Google respaldó económicamente y que, más tarde, en el año 2005, terminó comprando. Android fue presentado en 2007 junto a la fundación del Open Handset Alliance, que era un consorcio de compañías de hardware, software y telecomunicaciones, para avanzar en los estándares abiertos de los dispositivos móviles. En ese momento se proporcionó la primera versión de Android, junto con el SDK correspondiente, para que los programadores empezaran a crear sus aplicaciones para este sistema.

El sistema permite programar aplicaciones para una variante de Java llamada Dalvik hasta la versión 5.0, pero luego cambió al entorno Android Runtime (ART), el cual trae mejoras en el rendimiento y aumenta la duración de la batería.

Android proporciona todas las interfaces necesarias para desarrollar aplicaciones que accedan a las funciones del teléfono (como el almacenamiento interno, las llamadas, la agenda, etc.) de una forma muy sencilla en el lenguaje de programación Java. Android permite controlar dispositivos mediante bibliotecas desarrolladas o adaptadas por Google mediante Java.

5.1.2.1. Características de Android

La gran ventaja que tiene Android es que tiene su propio entorno de desarrollo integrado llamado Android Studio, el cual, nos permite programar y analizar el código de una manera muy cómoda. También incluye un emulador de dispositivos, herramientas para la depuración, y un editor de diseño que permite arrastrar y soltar componentes de la interfaz de usuario.

Las principales características de Android son:

- Diseño de dispositivo: La plataforma es adaptable a pantallas de mayor resolución, VGA, biblioteca de gráficos 2D, biblioteca de gráficos 3D basada en las especificaciones del OpenGL ES 2.0 y diseño de teléfonos tradicionales.
- Almacenamiento: SQLite, una base de datos liviana, que es usada para propósitos de almacenamiento de datos.
- Conectividad: Android soporta las siguientes tecnologías de conectividad: GSM/EDGE, IDEN, CDMA, EV-DO, UMTS, Bluetooth, Wi-Fi, LTE, HSDPA, HSPA+, NFC y WiMAX, GPRS, UMTS y HSDPA+.
- Mensajería: SMS y MMS son formas de mensajería, incluyendo mensajería de texto, además del servicio de Firebase Cloud Messaging (FCM), siendo la nueva versión de Google Cloud Messaging (GCM) basada en Firebase con nuevos SDK para hacer el desarrollo de mensajería en la nube mucho más sencillo.
- Navegador web: El navegador web incluido en Android está basado en el motor de renderizado de código abierto WebKit, emparejado con el motor

JavaScript V8 de Google Chrome. El navegador por defecto de Ice Cream Sandwich obtiene una puntuación de 100/100 en el test Acid3.

- Soporte de Java: Aunque la mayoría de las aplicaciones están escritas en Java, no hay una máquina virtual Java en la plataforma. El bytecode Java no es ejecutado, sino que primero se compila en un ejecutable Dalvik y se ejecuta en la Máquina Virtual Dalvik. Dalvik es una máquina virtual especializada, diseñada específicamente para Android, y optimizada para dispositivos móviles que funcionan con batería y que tienen memoria y procesador limitados. A partir de la versión 5.0, se utiliza el Android Runtime (ART), aunque los principios de funcionamiento son similares.
- Soporte multimedia: Android soporta los siguientes formatos multimedia: WebM, H.263, H.264 (en 3GP o MP4), MPEG-4 SP, AMR, AMR-WB, AAC, HE-AAC, MP3, MIDI, Ogg Vorbis, WAV, JPEG, PNG, GIF y BMP.
- Soporte para streaming: Streaming RTP/RTSP (3GPP PSS, ISMA), descarga progresiva de HTML (HTML5 <video> tag). Adobe Flash Streaming (RTMP) es soportado mediante el Adobe Flash Player. Se planea el soporte de Microsoft Smooth Streaming con el port de Silverlight a Android. Adobe Flash HTTP Dynamic Streaming estará disponible mediante una actualización de Adobe Flash Player.
- Soporte para hardware adicional: Android soporta cámaras de fotos, de vídeo, pantallas táctiles, GPS, acelerómetros, giroscopios, magnetómetros, sensores de proximidad y de presión, sensores de luz, gamepad, termómetro, y aceleración por GPU 2D y 3D.
- Google Play: es un catálogo de aplicaciones gratuitas o de pago en el que pueden ser descargadas e instaladas en dispositivos Android sin la necesidad de un PC.
- Multi-táctil: Android tiene soporte nativo para pantallas capacitivas con soporte multi-táctil. La funcionalidad fue originalmente desactivada a nivel de kernel (posiblemente para evitar infringir patentes de otras compañías). Más tarde, Google publicó una actualización para el Nexus One y el Motorola Droid que activa el soporte multi-táctil de forma nativa.
- Bluetooth
- Videollamada
- Multitarea: La multitarea real de aplicaciones está disponible, es decir, las aplicaciones que no estén ejecutándose en primer plano reciben ciclos de reloj.
- Tethering: Android soporta Tethering, lo cual permite al teléfono ser usado como un punto de acceso cableado o inalámbrico.

5.1.2.2. Arquitectura de Android

La arquitectura interna de Android se puede dividir en cinco grandes componentes: Núcleo de Linux, bibliotecas, entorno de ejecución, marco de aplicación y aplicaciones.

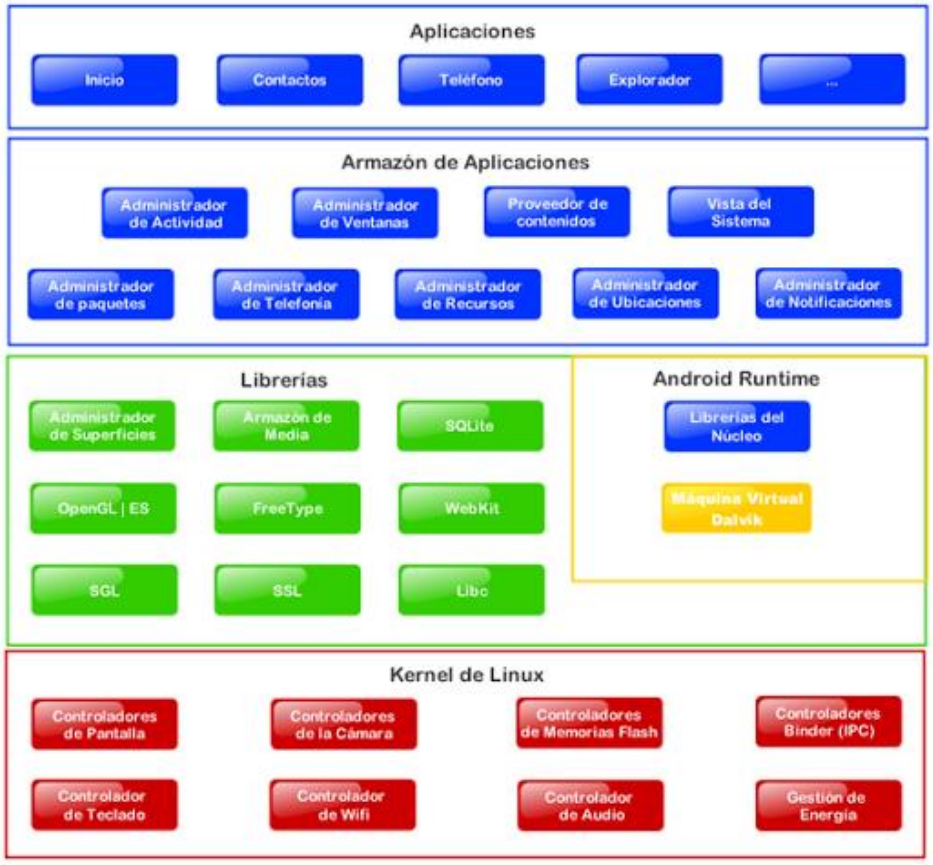

Figura 6: Arquitectura de Android

- Núcleo de Linux: Android utiliza el núcleo de Linux 2.6 como una capa de abstracción para el hardware disponible en los dispositivos móviles. Esta capa contiene los drivers necesarios para que cualquier componente hardware pueda ser utilizado mediante las llamadas correspondientes. Siempre que un fabricante incluye un nuevo elemento de hardware, lo primero que se debe realizar para que éste pueda ser utilizado desde Android es crear las librerías de control y los drivers necesarios dentro del kernel de Linux embebido en el propio Android. Además de proporcionar controladores hardware, el kernel se encarga de gestionar los diferentes recursos del teléfono (energía, memoria, etc.) y del sistema operativo (procesos, elementos de comunicación, etc.).
- Bibliotecas: Android incluye un conjunto de bibliotecas que están escritas en C o C++, y que están compiladas para la arquitectura hardware específica del teléfono. Se encargan de proporcionar funcionalidad a las aplicaciones, para tareas que se repiten con frecuencia, evitando tener que

codificarlas cada vez, y garantizando que se llevan a cabo de la forma más eficiente.

- Entorno de ejecución: Android incluye un conjunto de bibliotecas base que proporcionan la mayor parte de las funciones disponibles en las bibliotecas base del lenguaje Java. Cada aplicación Android corre su propio proceso, con su propia instancia de la máquina virtual Dalvik. Dalvik ha sido escrito de forma que un dispositivo puede correr múltiples máquinas virtuales de forma eficiente. Dalvik ejecutaba hasta la versión 5.0 archivos en el formato Dalvik Executable (.dex), el cual está optimizado para memoria mínima. La Máquina Virtual está basada en registros y corre clases compiladas por el compilador de Java que han sido transformadas al formato .dex por una herramienta incluida llamada "dx". Desde la versión 5.0 utiliza el ART, que realiza la compilación desde cero en el momento de instalación de la aplicación.
- Marco de aplicación: los desarrolladores tienen un acceso completo a los mismos APIs del framework usados por las aplicaciones base. La arquitectura está diseñada para simplificar la reutilización de componentes; cualquier aplicación puede publicar sus capacidades, y cualquier otra aplicación puede luego hacer uso de esas capacidades (sujeto a reglas de seguridad del framework). Este mismo mecanismo permite que los componentes sean reemplazados por el usuario.
- Aplicaciones: este nivel contiene tanto las aplicaciones incluidas por defecto con Android, como aquellas que el usuario vaya añadiendo posteriormente, ya sean de terceras empresas o de su propio desarrollo. Todas estas aplicaciones utilizan los servicios, las APIs y las bibliotecas de los niveles anteriores.

5.2. Servidor

En este apartado nos vamos a centrar en la descripción del código elaborado para el lado del servidor, el cual se encarga de codificar los datos y enviarlos de forma continua mediante broadcast, para que todos los vehículos puedan recibirlos.

A continuación, vamos a detallar los pasos que hemos realizado en la implementación del servidor, destacando las partes de código más importantes.

Primero, definimos el host, el puerto y el tamaño de los símbolos.

```
public static String host = "192.168.2.255";
public static int port = 4445;
private static final int SYMB_SIZE = 1400;
```

El host tiene la ip “192.168.2.255” porque el punto de acceso que hemos configurado está en la red “192.168.2.0”, y añadimos al final el 255 para configurarlo en modo broadcast. El tamaño de símbolo lo configuramos a 1400 bytes para que, a la hora de transmitirlo, la trama de nivel 2 resultante no supere los 1500 bytes, en un intento de evitar al máximo los efectos negativos relacionados con la fragmentación a nivel IP.

Lo siguiente que hacemos es convertir el fichero de datos que vamos a transmitir en un array de bytes.

```
FileInputStream fis = new FileInputStream(fichero);
ByteArrayOutputStream bos = new ByteArrayOutputStream();
byte[] buf = new byte[1024];
for (int readNum; (readNum = fis.read(buf)) != -1;) {
    bos.write(buf, 0, readNum);
}
datosF = bos.toByteArray();
fis.close();
```

Después, extraemos la extensión del fichero que vamos a enviar.

```
public static String getExtension(String filename) {
    int index = filename.lastIndexOf('.');
    if (index == -1) {
          return "";
    } else {
          return filename.substring(index + 1);
    }
```

A continuación, pasamos a crear los parámetros FEC. Para ello necesitamos el tamaño de los datos, el tamaño de símbolo, que lo hemos definido antes y el número de bloques origen, que en nuestro caso es 1.

```
int numSBs = minAllowedNumSourceBlocks(datosF.length, SYMB_SIZE);
FECParameters parametrosFEC = FECParameters.newParameters(datosF.length, SYMB_SIZE, numSBs);
```

Lo siguiente es crear el codificador. Para ello necesitamos los datos y los parámetros FEC.

```
ArrayDataEncoder arrayDataEn = OpenRQ.newEncoder(datosF, parametrosFEC);
SourceBlockEncoder sbn = arrayDataEn.sourceBlock(0);
```

Una vez tenemos el codificador general, creamos el codificador para el único bloque origen que tenemos, a partir del cual crearemos los símbolos origen y de reparación.

Procede también enviar los parámetros FEC a los receptores para que puedan percatarse del procedimiento de codificación, y ajustar el decodificador de manera a hacerlo compatible con los datos transmitidos. Para ello creamos un paquete UDP con la información más relevante a nivel FEC y lo enviamos.

```
System.out.println("Enviando los parametros FEC");
DatagramPacket pFEC = new DatagramPacket(byteFEC,byteFEC.length,aHost,port);
socket.send(pFEC);
System.out.println("Enviados los parametros FEC");
```

A continuación, enviamos la extensión del fichero.

```
System.out.println("Enviando la extension");
DatagramPacket pExt = new DatagramPacket(byteExt,byteExt.length,aHost,port);
socket.send(pExt);
System.out.println("Enviada la extension");
```

Este paso es necesario para que la aplicación Android del cliente sepa qué tipo de contenido tiene que reproducir.

Finalmente, creamos y enviamos los símbolos de reparación.

```
EncodingPacket repairPacket = sbn.repairPacket(j);
System.out.println("Creado el repairPacket j: " + j);
byte[] repairArray = repairPacket.asArray();
DatagramPacket rSource = new DatagramPacket(repairArray,repairArray.length,aHost,port);
socket.send(rSource);
System.out.println("Enviado el repairSymbol j: " + j);
```

Para ello primero creamos un paquete de reparación a partir del codificador de bloque origen (sbn). Después convertimos el paquete a un array de bytes y, por último, creamos el paquete UDP correspondiente y lo enviamos.

Para ejecutar el código anterior, lo que hacemos es crear desde el lado del servidor una aplicación de consola. Para arrancar el servidor hay que introducir el siguiente comando:

java -cp ".:/home/ServerTFM/openrq-3.3.2.jar" Servidor "imagen.jpg"

El parámetro –cp, seguido de la ruta donde se encuentra la librería, es necesario para poder incluir en el código las clases necesarias de OpenRQ.

El siguiente parámetro del comando es la clase Servidor, la cual acepta como parámetro de entrada el nombre del fichero que vamos a transmitir. En caso de que no introduzcamos ningún parámetro, se nos mostrará un error indicándonos qué nos falta introducir, y el programa detendrá su ejecución.

5.3. Cliente Android

El cliente Android recibe los datos que le llegan del servidor, y cuando tiene datos suficientes los decodifica y los reproduce.

A continuación, vamos a detallar los pasos que hemos realizado en la implementación del cliente, mostrando las partes de código más importantes.

Cuando ejecutamos la aplicación, ésta empieza a escuchar todos los paquetes que le llegan. No obstante, hasta que no recibe los parámetros FEC, no comienza realmente a almacenar los datos, ya que para poder hacerlo necesita conocer dichos parámetros.

A continuación, se ilustra la recepción de los parámetros FEC.

```
DatagramPacket pac = new DatagramPacket(aux, aux.length);
socket.receive(pac);
Parsed<FECParameters> parametrosFECParse = FECParameters.parse(aux);
if (parametrosFECParse.isValid()) {
    FECParameters parametrosFEC = parametrosFECParse.value();
```

Destacar que este código permite comprobar que el paquete contiene los parámetros FEC porque, al crear el objeto, el método *isValid()* permite comprobar si el valor es el correcto.

Una vez se reciben los parámetros FEC, se crea el decodificador general y el decodificador para el bloque origen.

```
ArrayDataDecoder arrayDataDe = OpenRQ.newDecoder(parametrosFEC, 0);
SourceBlockDecoder dbn = arrayDataDe.sourceBlock(0);
```

A continuación, lo que hacemos es recibir la extensión del fichero. La extensión siempre se envía a continuación de los parámetros FEC, lo que nos permite saber que el siguiente paquete se tiene que tratar de una manera diferente.

```
DatagramPacket pac = new DatagramPacket(aux, aux.length);
socket.receive(pac);
ext = new String(aux);
```

Una vez ya hemos recibido los parámetros FEC y la extensión, ya podemos comenzar a recibir los símbolos. A continuación, podemos ver la recepción de un paquete que contiene un símbolo de reparación.

```
} else if (!parametrosFECParse.isValid()) {
    Parsed<EncodingPacket> packetParse = EncodingPacket.parsePacket(arrayDataDe, aux, false);
    if(packetParse.isValid()){
        EncodingPacket genPacket = packetParse.value();
```

Para comprobar si el paquete contiene un símbolo de reparación volvemos a hacer uso del método *isValid()*.

Conforme vamos recibiendo los símbolos, éstos se van almacenando, hasta que tenemos suficientes para realizar la decodificación.

```
if (dbn.putEncodingPacket(genPacket) == SourceBlockState.DECODED) {
    time_end = System.currentTimeMillis();
    result = true;
    break;
} else if (dbn.putEncodingPacket(genPacket) == SourceBlockState.INCOMPLETE) {
        log("Decodificación incompleta, son necesarios más símbolos");
    } else {
            log("Fallo de Decodificación");
    }
```

Una vez se han decodificado los datos, se reproduce el contenido.

A continuación, vamos a ver el funcionamiento de la aplicación de forma gráfica.

La aplicación está divida en varios módulos, típicamente como cualquier aplicación para dispositivos móviles (cada una de las pantallas corresponde con cada uno de los módulos de la aplicación).

Al iniciar la aplicación aparece la pantalla principal en la que podemos ver que hay una barra de progreso. Al mismo tiempo la aplicación nos indica que comienza a escuchar los paquetes que puedan llegar del servidor.

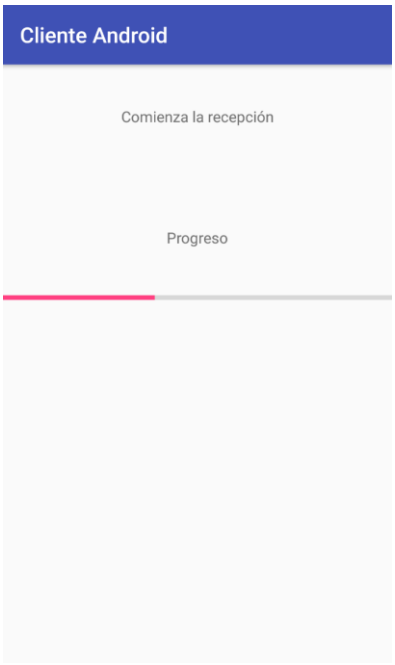

Figura 7: Pantalla Principal

Esta barra va aumentando conforme vamos recibiendo los datos desde el servidor. Cuando la barra se completa del todo significa que los datos ya se han recibido y se han decodificado, con lo que nos pasará a otra pantalla en la que se reproducirá el contenido que hayamos recibido. En la figura 8 podemos ver la pantalla que nos muestra la imagen que hemos recibido.

Figura 8: Pantalla Reproducción de Contenido

Una vez hemos reproducido el contenido que hemos recibido del servidor si volvemos a la pantalla de inicio podemos ver algunos datos, como el tiempo que ha tardado en recibir los paquetes necesarios para la decodificación, el tiempo que ha tardado en decodificar los datos, y el tiempo total (recepción + decodificación) de todo el proceso.

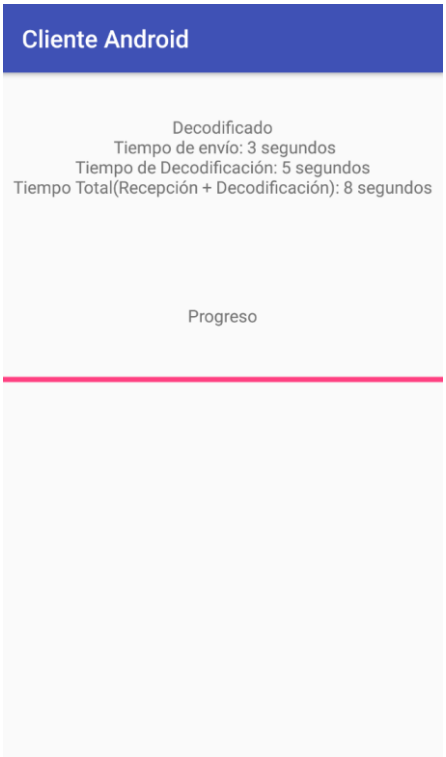

Figura 9: Pantalla de información.

6 Resultados experimentales

En este apartado vamos a exponer los resultados obtenidos para las pruebas que hemos realizado con el sistema desarrollado. El objetivo es determinar si el sistema funciona de manera eficiente, y si los tiempos de codificación y decodificación son asumibles en situaciones reales.

6.1. Prestaciones variando la pérdida de paquetes

Con esta prueba lo que buscamos es obtener el tiempo total que tarda el cliente en recibir y decodificar los datos que envía el servidor ante una tasa de pérdida de paquetes variable. Para realizarla hemos lanzado el código servidor y el código cliente en la misma máquina, ejecutándolos en paralelo, y hemos simulado un canal con una tasa de pérdidas regulable. Hemos lanzado la prueba varias veces simulando diferentes porcentajes de pérdidas, y para archivos de diferente tamaño. En la figura 10 podemos ver los resultados obtenidos:

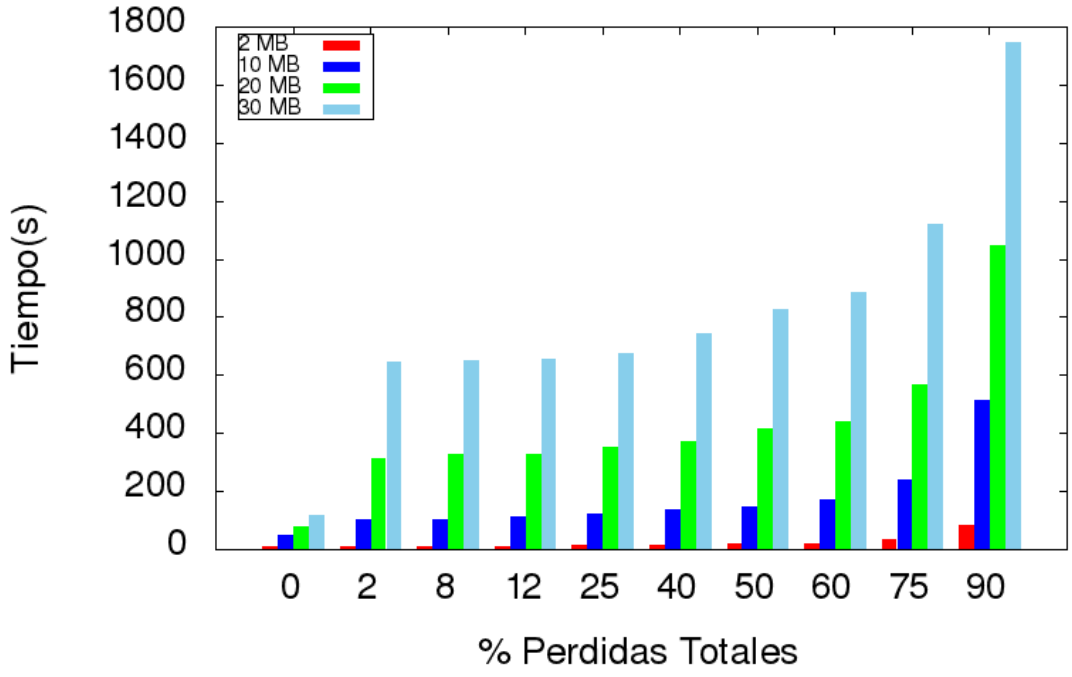

Figura 10: Tiempo total de recepción y decodificación para diferentes tasas de pérdida.

Podemos observar que los resultados son los esperados, ya que cuanto más alto es el porcentaje de pérdida de paquetes, más altos son los tiempos obtenidos. Lo mismo ocurre si hacemos una comparación en base al tamaño del archivo, verificándose que cuanto más grande es el archivo más se tarda en recibirlo y decodificarlo, siendo muchas veces esta diferencia no lineal.

Con esta prueba nos dimos cuenta de que los tiempos obtenidos para archivos de 20 y 30 MB eran muy altos. Con un porcentaje del 90% de pérdida de paquetes para 30 MB casi llega a treinta minutos todo el proceso de recepción y decodificación. Por esto decidimos usar archivos de tamaño más pequeño para continuar con las pruebas.

6.2. Tiempo de transmisión

En esta prueba lo que realizamos es una comparación de los tiempos de transmisión, es decir, el tiempo que el cliente tarda en recibir los paquetes necesarios para poder decodificar el paquete, sin contar con el tiempo de decodificación. Para ello comparamos el tiempo teórico que debería de tardar, con el tiempo que tarda ejecutando el servidor y el cliente en el mismo ordenador, y el tiempo a través de una red real, en este caso en la red Wi-fi del laboratorio.

El tiempo teórico lo podemos calcular porque sabemos cuántos paquetes son necesarios para poder decodificar los datos. Entonces con el dato anterior, y sabiendo que cada paquete se envía cada 5 ms, podemos calcular el tiempo de recepción teórico.

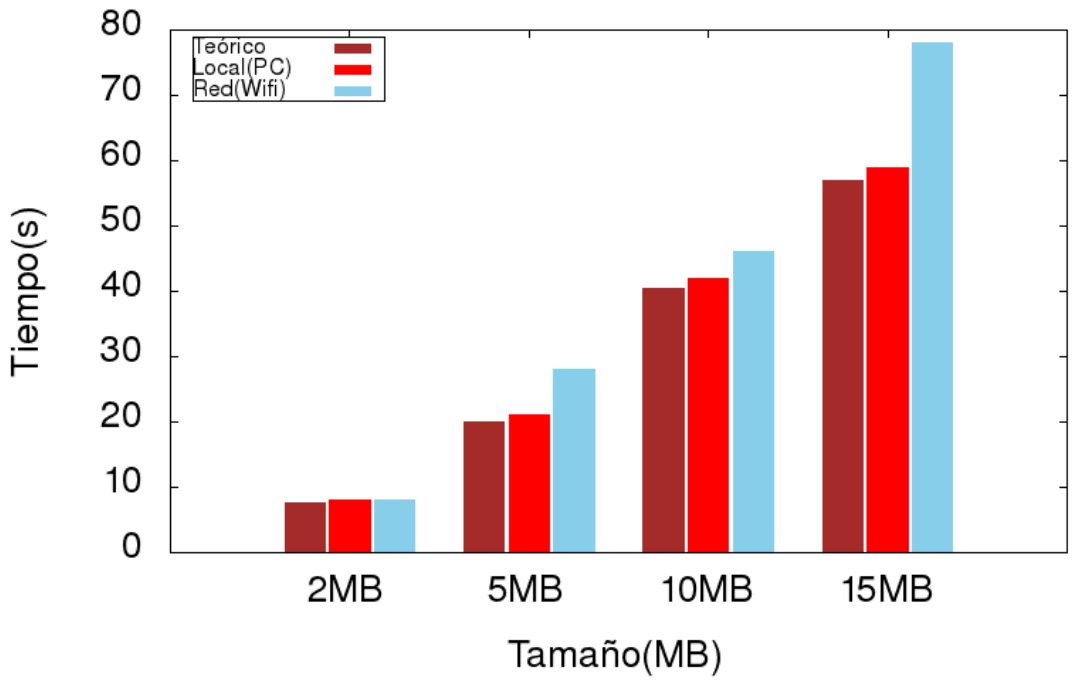

Figura 11: Tiempo de transmisión para diferentes tamaños de fichero, y para diferentes modos de transmisión.

Como podemos observar en la gráfica de la figura 11, los tiempos de transmisión para el archivo de tamaño de 2 MB, son muy similares en los tres escenarios. Conforme el tamaño de archivo va aumentando se hace más clara la diferencia. Entre el caso teórico y la transmisión realizada internamente en el PC no hay mucha diferencia en los tiempos, pero sí la hay a través de la red inalámbrica. El

motivo de que aumenten más los tiempos a través de la red Wifi puede ser debido a que en el laboratorio este medio esté saturado, lo que de origen a pérdida de paquetes.

6.3. Tiempo de decodificación

En esta prueba lo que realizamos es una comparación de los tiempos de decodificación en diferentes dispositivos.

Primero mostramos la diferencia entre un ordenador y un teléfono móvil. Después mostraremos la comparativa entre diferentes modelos de teléfono móvil.

El ordenador con el que vamos a realizar la prueba tiene un procesador Intel i7-5700HQ Quad Core 2.7 GHz y 8 GB de memoria RAM.

El teléfono móvil tiene un procesador Qualcomm Snapdragon 820 Quad Core 2x2.2 GHz 2x1.6 GHz y 6 GB de memoria RAM.

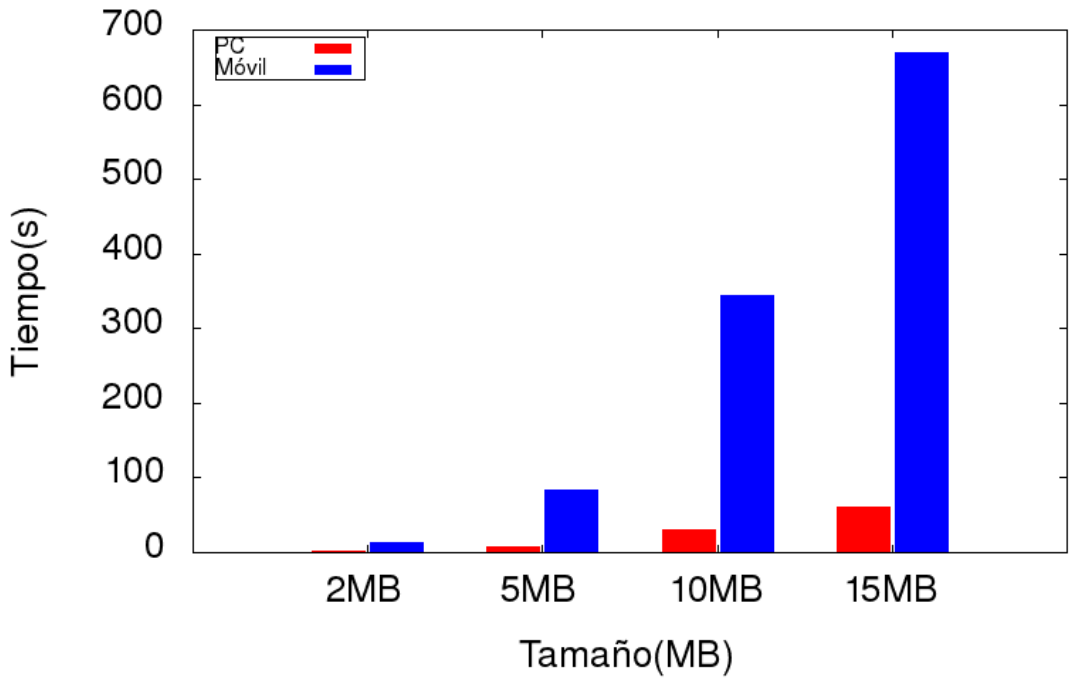

Figura 12: Tiempos de decodificación en el PC y en el móvil para diferentes tamaños de mensaje.

Como podemos ver en la gráfica de la figura 12, cuanto mayor es el tamaño del archivo, mayor es la diferencia de tiempo entre el dispositivo móvil y el ordenador. A partir de 5MB, los tiempos de decodificación en el dispositivo móvil son muy altos, lo cual se deba posiblemente a limitaciones en términos de memoria RAM y memoria caché, por lo que hemos decidido que vamos a usar archivos de menor tamaño.

En la figura 13 se muestran los resultados variando el tamaño del archivo, ahora usado archivos de menor tamaño para la comparación.

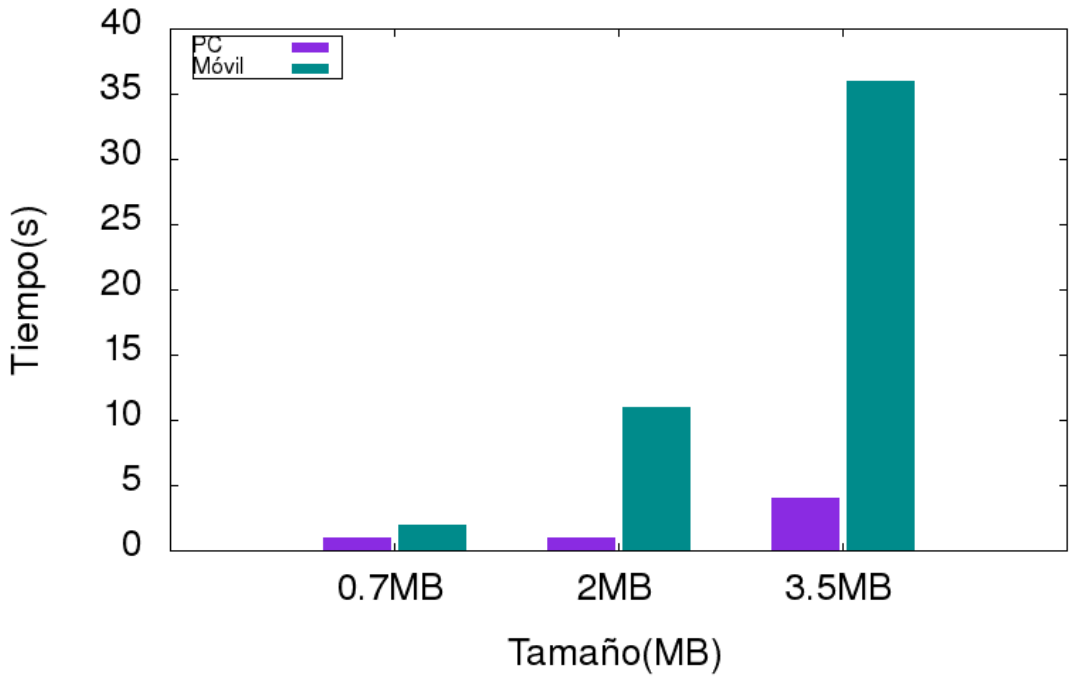

Figura 13: Tiempos de decodificación en el PC y en el móvil para diferentes tamaños de mensaje I.

Podemos observar que los tiempos disminuyen bastante respecto a los de la gráfica de la figura 12. Ahora, cuanto menor es el tamaño de archivo, menor es la diferencia en el tiempo de decodificación para el ordenador y el móvil. Y aunque para el dispositivo móvil la decodificación del archivo de 3.5MB está por debajo de los 40 segundos, aún sigue siendo bastante alto el tiempo. Por esta razón nos decantamos por usar en nuestro proyecto archivos de menos de 2MB de tamaño.

A parte de la comparativa anterior, también hemos realizado otra similar, pero utilizando diferentes teléfonos móviles. Esto nos ayuda a tener una visión más general de los tiempos de decodificación.

Los teléfonos móviles que vamos a comparar son los siguientes:

- OnePlus 3: Qualcomm Snapdragon 820 Quad Core 2x2.2 GHz 2x1.6 GHz y 6 GB de RAM.
- Samsung Galaxy S7 edge: Exynos 8890 Octa Core 2.15 GHZ y 4 GB de RAM.
- Samsung J5: Qualcomm Snapdragon 410 Quad Core 1.2 GHz y 1.5 GB de RAM.
- Bq Aquaris A4.5: MediaTek MT6735M Quad Core Cortex A53 hasta 1 GHz y 1 GB de RAM

En la figura 14 podemos ver los resultados de esta comparación. Como podemos observar, hay una gran diferencia entre el OnePlus3 y el Samsung S7 respecto al Samsung J5 y el Bq A4.5.

Los tiempos de decodificación para el OnePlus3 y el S7 son los esperados para el tipo de sistema que hemos desarrollado, ya que, para un archivo de tamaño 0.7 MB, tardan sobre 2 segundos en decodificarlo. En cambio, el tiempo del Samsung J5 para el mismo fichero asciende a 264 segundos, y el tiempo para el Bq A4.5 alcanza los 376 segundos, un tiempo considerado excesivo.

La gran diferencia de tiempo puede ser debida a varios factores. Uno de estos factores puede ser la versión de Android instalada en cada móvil. El OnePlus3 y el S7 llevan la última versión de Android, la número 7, en cambio el J5 y el Bq tienen versiones anteriores, la 5 y la 6, respectivamente. Otros factores determinantes pueden ser el procesador, la memoria RAM y la memoria caché, ya que, como hemos visto antes, cada móvil tiene características diferentes.

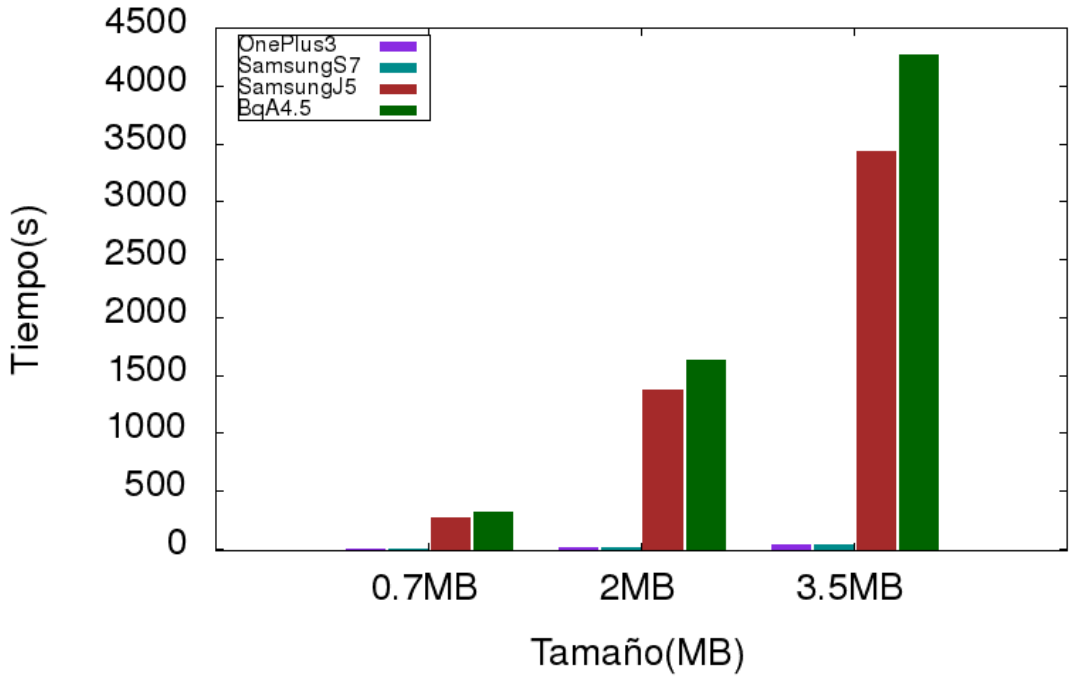

Figura 14: Tiempos de decodificación en diferentes móviles para diferentes tamaños de mensaje.

7 Conclusiones

El uso cada vez más extendido de drones genera una serie de nuevos desafíos en diferentes ámbitos profesionales. Uno de estos desafíos es el de integrar los drones en el ámbito de las comunicaciones, y para ello será necesario dotarles de algún sistema embebido, como puede ser la Raspberry Pi.

En este trabajo se demuestra que es posible integrar la movilidad de los drones en las redes vehiculares. Esto nos permite que el dron sea capaz de transmitir contenido a todos los vehículos que estén a su alcance. El contenido puede ser información de confort para los pasajeros como, por ejemplo, que avise que hay atascos un poco más adelante. La información también puede servir para la seguridad, ya que el dron puede avisar de un accidente que ocasione que la carretera sea cortada, o que la calzada se encuentre en un mal estado.

Globalmente, se han cumplido los objetivos del proyecto, pues se ha desarrollado completamente el sistema de entrega de contenido. Se ha verificado su correcto funcionamiento realizando diferentes pruebas en diferentes entornos. Para algunas pruebas se ha utilizado como servidor la Raspberry Pi o el PC, lo que nos ha permitido evaluar su rendimiento en diferentes entornos, y usando diferentes modelos de red. También se han usado como clientes diferentes dispositivos móviles con diferentes características, con lo que hemos podido evaluar y comparar su rendimiento.

7.1. Trabajo Futuro

Como trabajo futuro, en primer lugar, se plantea realizar las pruebas en un entorno real. Para eso se procederá a equipar un dron con una Raspberry Pi, y usar varios coches para recibir la información transmitida. Esto nos permitirá evaluar si los tiempos globales son adecuados para un correcto funcionamiento del sistema. También podremos ver cuanta información podemos transmitir, midiendo los tiempos con diferentes tamaños de fichero.

Como trabajo futuro también se plantea incluir el protocolo FLUTE en nuestro sistema. FLUTE es un protocolo para la transmisión de datos en broadcast, y no requiere comunicación bidireccional, lo que permite que se pueda integrar en nuestro sistema sin ningún problema. FLUTE se puede combinar con diferentes esquemas FEC, lo que nos permitiría hacer una comparación del rendimiento usando diferentes códigos de corrección de errores.

Bibliografía

[1] https://tools.ietf.org/html/rfc6330

[2] M. Gerla, C. Wu, G. Pau, X. Zhu, Content distribution in VANETs, Vehicular Communications, Volume 1, Issue 1, 2014, Pages 3-12, ISSN 2214-2096, http://dx.doi.org/10.1016/j.vehcom.2013.11.001.

[3] F. A. Silva, A. Boukerche, T. R. M. Braga Silva, L. B. Ruiz, E. Cerqueira, A. F. Loureiro, 2015. Vehicular Networks: A New Challenge for Content Delivery-based Applications.

[4] I. Tal and G. M. Muntean, "User-oriented cluster-based solution for multimedia content delivery over VANETs," IEEE international Symposium on Broadband Multimedia Systems and Broadcasting, Seoul, 2012, pp. 1-5. doi: 10.1109/BMSB.2012.6264290

[5] I. K. Azogu, M. T. Ferreira and H. Liu, "A security metric for VANET content delivery," 2012 IEEE Global Communications Conference (GLOBECOM), Anaheim, CA, 2012, pp. 991-996. doi: 10.1109/GLOCOM.2012.6503242

[6] M. Fiore, C. Casetti, C.-F. Chiasserini, M. Garetto, Analysis and simulation of a content delivery application for vehicular wireless networks, Performance Evaluation, Volume 64, Issue 5, 2007, Pages 444-463, ISSN 0166-5316, http://dx.doi.org/10.1016/j.peva.2006.08.008.

[7] J. Eriksson, H. Balakrishnan, S. Madden, Cabernet: vehicular content delivery using WiFi, in: 14th ACM International Conference on Mobile computing and networking, 2008.

[8] C.-J. Huang, Y.-J. Chen, I.-F. Chen, T.-H. Wu, An intelligent infotainment dissemination scheme for heterogeneous vehicular networks, Elsevier Expert Systems with Applications 2009.

[9] U. Lee, J. Lee, J.-S. Park, M. Gerla, FleaNet: A Virtual Market Place on Vehicular Networks, IEEE Transaction on Vehicular Technology 59 (1) (2010) 344–355.

[10] C. T. Calafate, G. Fortino, S. Fritsch, J. Monteiro, J.C. Cano, P. Manzoni (2012). An efficient and robust content delivery solution for IEEE 802.11p vehicular environments. Journal of Network and Computer Applications. 35(2):753-762. doi: 10.1016/j.jnca.2011.11.008.

[11] A. Shokrollahi, "Raptor codes," in IEEE Transactions on Information Theory, vol. 52, no. 6, pp. 2551-2567, June 2006

[12] U. Demir and O. Aktas, "Raptor versus Reed Solomon forward error correction codes," *2006 International Symposium on Computer Networks*, Istanbul, 2006, pp. 264-269. doi: 10.1109/ISCN.2006.1662545

[13] M. Luby, M. Watson, T. Gasiba, T. Stockhammer and Wen Xu, "Raptor codes for reliable download delivery in wireless broadcast systems," CCNC 2006. 2006 3rd IEEE Consumer Communications and Networking Conference, 2006., 2006, pp. 192-197. doi: 10.1109/CCNC.2006.1593014

[14] S. Kim, S. Lee and S. Y. Chung, "An efficient algorithm for ML decoding of raptor codes over the binary erasure channel," in IEEE Communications Letters, vol. 12, no. 8, pp. 578-580, Aug. 2008. doi: 10.1109/LCOMM.2008.080599

[15] T. Mladenov, S. Nooshabadi and K. Kim, "MBMS raptor codes design trade-offs for IPTV," in IEEE Transactions on Consumer Electronics, vol. 56, no. 3, pp. 1264-1269, Aug. 2010. doi: 10.1109/TCE.2010.5606257

[16] L. Hu, S. Nooshabadi and T. Mladenov, "Forward error correction with RaptorQ code on GPU," 2013 IEEE International Symposium on Circuits and Systems (ISCAS2013), Beijing, 2013, pp. 281-284. doi: 10.1109/ISCAS.2013.6571837

[17] T. Mladenov, K. Kim and S. Nooshabadi, "Forward error correction with RaptorQ Code on embedded systems," 2011 IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS), Seoul, 2011, pp. 1-4. doi: 10.1109/MWSCAS.2011.6026424

[18] M. Luby, "Best practices for mobile broadcast delivery and playback of multimedia content," IEEE international Symposium on Broadband Multimedia Systems and Broadcasting, Seoul, 2012, pp. 1-7. doi: 10.1109/BMSB.2012.6264233

[19] http://openrq-team.github.io/openrq/

[20] https://github.com/openrq-team/OpenRQ/wiki

[21] www.wikipedia.com

[22] www.android.com

[23] https://www.redeszone.net/raspberry-pi/manual-para-configurar-raspberry-pi-como-un-router-wi-fi/