UNIVERSIDAD POLITÉCNICA DE VALENCIA
DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

Desarrollo de un protocolo para entrega fiable de datos en entornos inalámbricos basado en Raptor codes

TESINA DE MÁSTER

Miguel Báguena Albaladejo

Director
Carlos Tavares Calafate

14 de Junio de 2011

Índice

1.- Introducción....4
1.1.- Visión general....4
1.2.- Objetivos del trabajo....6
1.3.- Estructura del documento....6
2.- Trabajos previos....8
3.- Codificación FEC (Forward Error Correction)....11
3.1.- La corrección de errores en la transmisión de información....11
3.2.- Sistemas de codificación FEC....12
3.2.1.- Familia Reed-Solomon....12
3.2.2.- Turbo codes....12
3.2.3.- Tornado codes y el paradigma de las fuentes digitales....13
3.3.- Funcionamiento y características de los Raptor Codes....13
3.3.1.- Base teórica....13
3.3.2.- La implementación práctica: DF Raptor de Digital Fountain....14
3.3.2.1- El codificador DF Raptor R11....17
3.3.2.2- El decodificador DF Raptor R11....20
4.- RDT: protocolo para la entrega fiable de datos en redes inalámbricas basado en Raptor Codes. 23
4.1.- Descripción del protocolo....23
4.2.- Diseño....23
4.2.1.- El esqueleto: la librería UDT....23
4.2.1.1.- Ofreciendo un servicio orientado a conexión....26
4.2.1.2.- Herramientas de soporte al desarrollo sobre UDT....27
4.2.2.- Diseño básico....29
4.2.3.- Diseño avanzado....31
4.3.- El control de la tasa de transmisión....36
4.4.- Funcionamiento....37
4.5.- Funcionamiento alternativo....38
5.- Detalles de implementación....40
5.1.- Codificación y decodificación Raptor....40
5.2.- Integración con la librería UDT....51
5.3.- Eliminación del sistema de envío UDT....55
5.4.- Inclusión del sistema de control de flujo RDT....57
5.5.- Codificación en paralelo....63
5.6.- Gestión del tiempo de espera entre paquetes....69
5.7.- Cambio en la estrategia de reconocimiento de bloques....70
5.8.- Optimizaciones a la codificación: desarrollos evaluados....71
5.9.- Decodificación en paralelo 74
5.10.- Implementación del protocolo avanzado....77
5.11.- Filtrado....79
6.- Validación y pruebas....83
6.1.- Validación....83
6.2.- Evaluación del rendimiento para el diseño básico....87
6.3.- Evaluación comparativa de las mejoras propuestas....91
6.4.- Evaluación comparativa de los sistemas de filtrado....98
6.5.- Evaluación del rendimiento para el diseño avanzado....104
6.6.- Pruebas de concurrencia....108
7.- Conclusiones....114
8.- Bibliografía....115

ANEXO I: API del codificador DF Raptor.....117
ANEXO II: API del decodificador DF Raptor.....120

1.- Introducción

1.1.- Visión general

Con los nuevos avances es dispositivos de comunicación, la transmisión de datos de manera inalámbrica se ha convertido en algo cotidiano y personal. Desde cualquier parte del mundo, a cualquier hora y en cualquier circunstancia las personas quieren tener acceso a la red de redes. De esta manera, la sociedad entera está transformándose hacia el paradigma del usuario "siempre conectado".

Sin embargo, liberarnos del cable que nos anclaba a un puesto de trabajo estático no implica solo crear los aparatos necesarios que en lugar de conectarse al cable tal y como hacían antiguamente, se conecten a ese medio de transmisión que no nos coarta la libertad de movimiento, permitiéndonos tomar decisiones sobre dónde y cuándo hacer uso de la informática y no al revés: el aire.

Si examinamos los retos que se proponen en la nueva comunicación ubicua en la que nos queremos mover en un inminente futuro, los podemos agrupar quizá de una manera no muy precisa pero al menos de una manera elegante, en tres grandes grupos. Cada grupo se puede definir según las capacidades operativas que proporcionan, desde la posibilidad, pasando por una aceptable productividad, hasta la comodidad para el usuario.

- **Posibilidad:** El Hardware es el que hace posible la comunicación inalámbrica. Hay muchos sistemas desarrollados que actualmente ya se usan para llevar a cabo este tipo de comunicación. El máximo exponente de todos ellos es la familia del 802.11, de los cuales la mayoría de personas ya dispone uno o varios dispositivos en su hogar.
- **Productividad:** La única manera de sacar todo el partido posible al Hardware que hay ya desarrollado es poner sobre él una capa de software que consiga el máximo rendimiento al dispositivo sobre el que se despliegue. Dentro de este grupo entran los protocolos de transporte, encaminamiento o de distribución de contenidos.
- **Comodidad:** Una vez ya se ha conseguido una aceptable productividad es posible transmitir datos de manera inalámbrica y consiguiendo una productividad que compita de igual a igual con las soluciones cableadas, es necesario proveer una serie de servicios que acaben de definir el modelo de comunicación inalámbrica para que se adecue a la forma de comunicarse de las personas. Dentro de este grupo entran aspectos intangibles como la fiabilidad, la seguridad o la configuración automática.

Este trabajo se centra en la realización de un protocolo que permita la entrega y distribución de datos dentro de entorno inalámbrico consiguiendo unos niveles de productividad lo más próximo posible a las redes cableadas a las que estamos acostumbrados. Dado que el contenido de este trabajo cae dentro del segundo grupo vamos a detallar a continuación los problemas a los que se encuentran las redes a este nivel, y cómo se les ha intentado hacer frente.

En las redes cableadas, el protocolo más utilizado ha sido siempre el protocolo TCP, consiguiendo siempre unos muy buenos resultados. Por ello, y porque se basan también en tecnología IP, para las redes inalámbricas se decidió usar este protocolo igualmente. Sin embargo, en este tipo de redes la pérdida de paquetes es una realidad muy frecuente, por lo que TCP, que asocia la pérdida de un paquete con congestión en la red, infrautiliza un canal que ya de por si es

más limitado que el usado en redes cableadas. Por tanto, es recomendable organizar un sistema que nos otorgue las mismas características que TCP (servicio orientado a conexión, confiabilidad en la entrega, etc.) pero que no lastre la productividad de una manera tan decisiva.

Soluciones a este problema se han dado varias, las cuales se detallarán en puntos posteriores, pero dos son las tendencias principales que se han seguido a la hora de vencer esta limitación: las retransmisiones automáticas (ARQ) y las técnicas de corrección de errores, entre las que destacan las técnicas FEC.

La retransmisión automática se basa en la detección de los paquetes perdidos o erróneos y la petición de retransmisión. La más conocida de estas técnicas es la retransmisión selectiva que se ha implementado comúnmente como mejora a TCP. En la otra cara de la moneda están las técnicas de corrección de errores, como las técnicas FEC, en las que si un paquete se pierde o llega dañado no tenemos que pedir una retransmisión de ese paquete, sino que otro paquete que será enviado posteriormente (si no lo ha sido ya) nos podrá servir para recuperar la información que ha llegado en malas condiciones.

El tema sobre el que trata este documento es la implementación y desarrollo de RDT, un protocolo que hace uso de técnicas FEC para recuperar la información incluso en los casos en que parte de la información se ha perdido. Este es un concepto muy potente ya que evita tener que guardar un registro de los paquetes recibidos, paquetes por retransmitir, etc. y sólo tenemos que esperar a que vayan llegando los paquetes adicionales que necesitamos para recuperar la información que se nos quiere transmitir.

A primera vista los protocolos basados en técnicas FEC podrían parecer una solución ideal a los problemas de la comunicación en un canal con pérdidas, especialmente cuando el retardo extremo-a-extremo es elevado. Sin embargo, tienen una gran desventaja: el proceso de codificación y, en menor medida, el de decodificación, son muy costosos tanto en términos de tiempo de ejecución como en términos de memoria utilizada. Esto ha conducido tradicionalmente a desplegar este tipo de sistemas en redes de transmisión unidireccional desde grandes servidores en situaciones de *broadcast* y con sistemas de codificación poco robustos que solo resolvían en parte el problema.

Ahora, tanto los computadores como las técnicas de codificación han sido mejoradas. Sistemas de codificación como son los Raptor Codes nos permiten realizar el proceso en tiempo lineal, con lo que tareas como aumentar el tamaño de bloque pasan a ser un proceso asumible en los términos en los que ahora se mueve el típico usuario de computadoras. Eso nos permite utilizar estos sistemas en otro tipo de comunicaciones, como puede ser la difusión de contenidos mediante *unicast* sobre redes inalámbricas, así como en otro tipo de entornos software, como es a nivel de aplicación.

Visto esto, sería sencillo utilizar implementar haciendo uso de las ventajas que introduce este protocolo un sistema de distribución de contenidos que en entornos de alta pérdida de paquetes, limitado ancho de banda y altos retardos, como puede ser un entorno inalámbrico, mejorase las soluciones que hasta ahora se han propuesto.

1.2.- Objetivos del trabajo

Con este proyecto se pretende desarrollar una librería de comunicaciones que implemente RDT, un protocolo de transmisión de datos sobre canales con pérdidas que hace uso de codificación Raptor para transmitir información, evitando así la necesidad de retransmitir paquetes. Este protocolo se basa en el control de la tasa de transmisión con técnicas de medición extremo-a-extremo, haciendo innecesario el uso de técnicas basadas en la pérdida de paquetes o en el uso de ventanas.

Se buscará enfocar la solución desarrollada hacia la obtención de un mayor rendimiento de la propuesta de partida a todos los niveles. Para ello se presentarán un conjunto de mejoras que abarquen los planos de diseño de la librería, técnicas de interacción con el canal de comunicaciones, optimizaciones a nivel de codificación y evaluación de modificaciones a nivel de protocolo.

Dentro del grupo de mejoras de diseño se hará especial hincapié en el uso de técnicas multihilo que aprovechen la capacidad de los nuevos procesadores que están disponibles en el mercado. Para ello se evaluarán diferentes opciones que hay disponibles y cual es el impacto que cada una de ellas tiene sobre la productividad.

Se analizarán también las opciones que hay disponibles para realizar el envío de datos de la manera más fiel posible al protocolo, de forma que se pueda sacar el máximo partido posible a las técnicas extremo-a-extremo que se han implementado.

Por último se estudiará el impacto que tiene el protocolo sobre otros protocolos ampliamente utilizados, como puede ser el protocolo TCP, cuando ambos comparten el canal de comunicaciones. Así mismo también se estudiará el comportamiento del protocolo cuando tiene que competir contra otros flujos iguales que él sobre el mismo canal. Todo esto se realizará sobre varias configuraciones de la red en cuanto a niveles de pérdida de paquetes, retardo y ancho de banda.

1.3.- Estructura del documento

En la primera sección se hablará sobre los trabajos previos que han tocado la problemática de TCP sobre las redes inalámbricas y las soluciones implementadas, la importancia de la optimización en los protocolos de comunicación y una breve nota sobre algunos de los sistemas propuestos para la distribución de contenidos.

En la segunda sección se describirá el funcionamiento de los algoritmos de corrección de errores, entre las que se hará especial énfasis en las técnicas FEC y los Códigos Raptor (Raptor Codes).

En la tercera sección se comentará en profundidad la propuesta implementada en el presente proyecto, detallando su diseño y dejando de manifiesto el porqué de las decisiones tomadas.

En la cuarta sección se entrará en un nivel técnico dentro del proyecto comentando en más detalle la implementación concreta de cada uno de los elementos y las distintas alternativas de implementación que se experimentaron.

En la quinta sección se expondrán el conjunto de validación y pruebas a las que se sometió

la aplicación para verificar su correcto funcionamiento y, sobre todo, para poder determinar en base a resultados experimentales todos y cada uno de los incrementos cuantitativos a los que nos conducía cada propuesta evaluada, para poder tomar así las decisiones de manera informada y realista.

En la sexta y última sección se relatarán las posibles vías de desarrollo que pueden ser seguidas para mejorar el comportamiento de la aplicación en campos tales como la estabilidad, la eficiencia y la funcionalidad.

2.- Trabajos previos

La problemática de TCP sobre las redes inalámbricas ha sido estudiada desde muchos puntos de vista y han sido multitud las soluciones propuestas por los diferentes autores.

Los principales problemas que caracterizan las redes inalámbricas son los siguientes [1][2]:

- **Ancho de banda limitado:** El ancho de banda disponible en redes inalámbricas (11-54 Mbps) es muy inferior al que hay en las redes cableadas (típicamente 100 Mbps – 10 Gbps).
- **RTT grandes:** El tiempo de ida y vuelta de un paquete (RTT) es típicamente mayor en las redes inalámbricas que en las cableadas. Una estimación mayor del retardo de un ACK por parte del emisor hace que, en caso de su pérdida, el tiempo de expiración del temporizador correspondiente sea mayor.
- **Pérdidas aleatorias:** Mientras que en TCP las pérdidas se deben en más de un 99% a la congestión y en menos de un 1% a problemas en el enlace, en las redes inalámbricas gran cantidad de las pérdidas de paquetes producidas se deben a interferencias de la señal u otros factores momentáneos.
- **Usuarios móviles:** Un usuario puede pasar de una red inalámbrica a otra, produciéndose una pérdida de conexión en el proceso que también provoca pérdida de segmentos TCP.
- **Flujos cortos:** TCP es un protocolo orientado a conexión: Todo ese proceso de conexión lleva un tiempo que, en redes con alto retardo, degrada mucho el rendimiento, siendo este su efecto más perceptible en envíos de corta duración.
- **Consumo de energía:** Al estar la red inalámbrica inherentemente desprovista de cables, los dispositivos móviles dependen enormemente de la vida de sus baterías, por lo que esta característica gana importancia.

Como se puede ver, las diferencias entre las redes cableadas y las inalámbricas, tanto por las cualidades del propio medio como de los dispositivos que hacen uso de este medio, nos llevan a tener que crear un nuevo protocolo para aprovechar eficientemente sus recursos o a modificar el protocolo TCP con el que contamos actualmente.

Se han propuesto varias soluciones al problema, ya sea desde un punto de vista más particular centrándose solo en los enlaces inalámbricos, como para redes mixtas (parte cableada, parte inalámbrica) o para redes móviles. Estas son algunas de ellas [3]:

- **Soluciones a nivel de enlace:** Este tipo de soluciones se basan en la modificación del nivel de enlace del protocolo de red para incluir principalmente retransmisiones o sistemas FEC de recuperación ante errores. Al ser un nivel muy bajo, puede tratar los paquetes como si de entidades independientes se tratara.
  - **El protocolo AIRMAIL [4]:** Combinación de técnicas de retransmisión y corrección de errores, planteando un esquema asimétrico en el procesamiento que permita ahorrar energía en un dispositivo móvil.
  - **El protocolo Snoop:** Protocolo diseñado para grandes redes en la que se coloca un módulo agente encargado de conocer todos los paquetes y ACK que se transmiten entre emisor y receptor y en caso de detectar alguna pérdida, realizar la retransmisión.
  - **Tulip [5]:** Basado en la retransmisión automática de los paquetes al detectar una pérdida, introduce una aceleración MAC que le permite acelerar la recepción de los ACK sin tener que renegociar el acceso al medio.
  - **Retraso de los ACK duplicados:** Técnica que se basa en retransmisiones a nivel de enlace y en el retraso de los envíos de los ACK duplicados para intentar no interferir en

esas retransmisiones.
    - **Reliable TCP-Aware Link-Layer Retransmission for Wireless Networks:** Los paquetes son marcados con un número de secuencia a nivel de enlace y, cuando se detecta una pérdida, se retransmiten por la estación base. Esta estación base notifica al emisor este hecho para que no disminuya su tasa de transferencia.
- **Conexión dividida:** Divide una conexión TCP en dos conexiones distintas, una desde el emisor a la estación base, y la otra desde la estación base al receptor, usando TCP sobre la parte cableada y otro protocolo sobre la parte inalámbrica.
    - **Indirect TCP:** Usa TCP también sobre la parte inalámbrica.
    - **Mobile TCP:** Crea una estructura de tres capas que está diseñada para trabajar en entornos donde es frecuente la pérdida de conexión. Los elementos de la capa superior se encargan del encaminamiento, control de flujo, reconexión con los dispositivos móviles, etc.
    - **Wireless-TCP:** Diseñado para trabajar sobre WWAN, especifica un nuevo protocolo (WTCP) que cambia el control de flujo en base a ventanas por un control de flujo basado en la tasa de transferencia.
    - **TCP over Wireless Networks using Multiple Acknowledgments:** Se coloca un agente entre el emisor y el receptor que monitoriza el tráfico entre los dos y que cuando almacena un paquete envía un reconocimiento parcial al emisor.
    - **PTCP** [15]: Intenta reducir la degradación de prestaciones en TCP debidas a la entrega desordenada de paquetes.
- **Modificaciones de TCP:** Se modifica el protocolo TCP añadiendo elementos como los ACK selectivos o las notificaciones de pérdidas explícitas.
    - **TCP SACK [6]:** Se informa al emisor de cuales son exactamente los paquetes que han llegado al receptor, en lugar de enviar sólo el ACK para el último paquete que se ha recibido correctamente cuando no hay pérdida de ningún paquete intermedio, permitiendo así que no se hagan retransmisiones innecesarias, ni que se produzca una degradación del rendimiento por culpa de los temporizadores de ACK.
    - **TCP FACK:** Complementa al anterior, manteniendo información sobre el último paquete que ha llegado con éxito y la cantidad de información transmitida para estimar la ventana de congestión en base a esos parámetros.
    - **SMART [7]:** Estrategia de retransmisión que combina el uso de GBN (Go-Back-N) y los ACK selectivos. Incluye en los ACK el número del paquete que se está reconociendo y el del último paquete que se recibió, lo que permite al receptor suponer que la diferencia entre los dos son los paquetes que se han perdido.
    - **Fast Retransmission:** En la tecnología móvil, usa el evento de handoff para iniciar una retransmisión sin necesidad de esperar a que expiren los temporizadores.
    - **Explicit Congestion Notification:** Se basa en el uso del bit CE (Congestion Experienced) de la cabecera IP. Cuando el receptor recibe paquetes a los que los routers intermedios les han activado el bit CE (tienen los buffers muy llenos), responde con ACK con el bit CE activado. En caso contrario no activa el bit CE en sus ack. Cuando el emisor recibe tres o más ACK duplicados, si tienen el bit CE activo pasa al modo de control de congestión, y si no lo tienen activo solo se reduce la ventana de congestión posibilitando una recuperación rápida.
    - **TCP Santa-Cruz:** Se almacenan los tiempos de envío y recepción de todos los paquetes para poder calcular el tiempo entre llegadas y estimar en base a este dato la congestión del canal.
    - **Improving Performance of TCP over Wireless using Additional Message Types:** Se añade un nuevo tipo de mensaje ICMP, ICMP-DEFER, que insta al emisor a reiniciar su

temporizador de retransmisión. También se añade el paquete ICMP-RETRANSMIT para obligar al emisor a retransmitir un paquete concreto y no tener que esperar a que expire el temporizador.

- **Redes Ad Hoc:** En las redes ad-hoc se producen fallas en el rendimiento producidos por el retardo derivado del tiempo de computación de reencaminamiento o del particionamiento de la red.
  - **Feedback-Based Scheme for Improving TCP Performance (TCP-F):** Se envía un paquete RFN cuando se produce un fallo en un punto de la ruta de encaminamiento del paquete, y un paquete RRN cuando ésta se restablece. Esto permite al emisor detener el envío de paquetes y los temporizadores cuando la ruta se rompe y reactivarlos cuando se recupera.
  - **Ad Hoc TCP:** Varía el estado del emisor entre “retransmitir” y “persistir” en función de si se detecta una ruta de comunicación válida o no.

También han sido muchos los trabajos orientados a mejorar la productividad modificando un protocolo concreto, así como otros orientados a ofrecer protocolos que no solo van enfocados a obtener una mayor productividad sino también a favorecer otros aspectos de la comunicación.

Claros ejemplos de estos protocolos son mTCP [16], que usa rutas diferentes para mejorar la robustez además de la productividad en el envío de los datos; un protocolo orientado al transporte de imágenes [17] que es capaz de entregar bloques de las mismas fuera da orden para mejorar el nivel de PSNR; o un protocolo de transporte para redes de alta velocidad [18] que realiza mediciones de, entre otras cosas, escalabilidad y reparto justo entre flujos.

También es una práctica común la evaluación de los protocolos para obtener el consumo de los mismos en términos de uso de memoria o de gasto energético [19], para poder así clasificar una serie de propuestas presentadas. El tema del uso de recursos, y cómo éstos están ligados a la implementación, ha sido también estudiado previamente [20], ofreciendo la experiencia recogida como base para futuras investigaciones.

En cuanto a la transmisión de ficheros grandes y a la distribución de contenidos, varias técnicas se han propuesto, como pueden ser el protocolo de transporte Cabernet [21], que consigue duplicar las prestaciones de TCP sobre una red con con nodos en movimiento; FastReplica [22], que transmite el fichero por partes distribuidas en servidores distintos; se han propuesto diseños que se basan en mezclar las capas [23], para ofrecer calidad de servicio sobre redes inalámbricas; o la más cercana a nuestra propuesta, MBMS [24], que usa códigos FEC para distribuir ficheros sobre redes móviles usando broadcast. Dentro de las redes vehiculares, se ha propuesto un esquema [25] para la diseminación de datos para realizar comunicaciones, por ejemplo, en tiempo real. Además se han hecho mediciones de productividad sobre el impacto que tiene sobre las prestaciones la distribución de ficheros entre iguales, en lugar de hacerlo por el paradigma cliente-servidor [26].

3.- Codificación FEC (Forward Error Correction)

3.1.- La corrección de errores en la transmisión de información

Todos los sistemas reales están sometidos, en mayor o menor medida, al error. En el mundo de las comunicaciones todos los sistemas han sido diseñados y construidos de manera que los errores fueran minimizados, ya sea en número o en la severidad de sus efectos. Sin embargo, por mucho empeño que se ponga esta tarea, los errores siempre aparecen y, por tanto, hay que ofrecer alguna solución que permita eliminarlos del resultado.

Cuando intentamos transmitir una información, un error es normalmente sinónimo de corrupción o pérdida de una porción de esa información o, usualmente, un paquete de datos. La solución tradicional a este problema era sencilla: si un paquete enviado no llegaba en condiciones al destino, se reenviaba una y otra vez hasta que este problema era solucionado. Sin embargo, esta solución acaba desembocando en problemas que no pueden ser satisfactoriamente resolubles.

La otra opción que nos deja esta aproximación es la corrección de errores. Una vez que tenemos los datos en destino y hemos encontrado un error, sería deseable tener la suficiente información como para poder deducir a partir de ella cual era la información correcta que debíamos haber recibido.

Desde que esta idea se propuso, muchas han sido las propuestas que han aparecido, empezando por el código Hamming, que inserta bits extra en la información transmitida para poder recuperar la información en el caso de que algunos bits que componen el mensaje se corrompa. A partir de este concepto inicial la idea ha ido evolucionando de manera que no solo trabajara a nivel de bit, sino que también se pudiera ser usada para la transmisión de grandes cantidades de información por los canales de comunicación que ahora tomamos como cotidianos de manera sistemática. Así nacen las técnicas FEC.

La idea que subyace a los códigos FEC es la siguiente: a toda la información enviada se le añade una cantidad variable de información redundante. Esa información será suficiente como para que, aunque se pierda cualquier cantidad de la información enviada, el receptor sea capaz de, en base a una serie de operaciones matemáticas, recuperar unilateralmente la información recibida y utilizarla como la original. Esto permite que el emisor no tenga que preocuparse por los paquetes perdidos, y se evitan las ocasiones en las que tiene que volver atrás y reenviar información que ya ha sido enviada anteriormente.

Existen dos grandes tipos de códigos FEC:

- Aquellos que trabajan sobre bloques de información de tamaño fijo (códigos aritméticos o de bloque), que son codificados y divididos en símbolos para su posterior envío. En base a este bloque fuente son capaces de generar una serie de símbolos de recuperación que irán enviando al cliente y que le permitirán hacer su recuperación.
- Aquellos que trabajan sobre flujos de información, de tamaño variable (códigos convolucionales) que, haciendo una aproximación estadística gracias al algoritmo de Viterbi, pueden recuperar la información que el emisor quería enviar.

Estudios realizados sobre los códigos FEC han detectado que los códigos que trabajan sobre bloques presentan una mayor inmunidad a los errores producidos en ráfagas, mientras que los algoritmos que trabajan sobre flujos son más robustos frente a ruido blanco y errores aislados. De

esta observación nace una tercera aproximación a la codificación FEC, que es la concatenación de códigos. En esta teoría se plantea concatenar un codificador del primer tipo con un codificador del segundo de manera que los errores que uno no pueda abordar sean resueltos por el siguiente.

3.2.- Sistemas de codificación FEC

Sobre estas bases han sido muchas las técnicas, algoritmos y sistemas que se han construido con el objetivo mejorar el planteamiento inicial, sobre todo a nivel de rendimiento y prestaciones.

3.2.1.- Familia Reed-Solomon

La historia de este tipo de códigos se remonta a 1959, cuando A. Hocquenghem, junto con el trabajo posterior de R. C. Bose y D. K. Ray-Chaudhuri, inventan los códigos BCH. Estos códigos nacen como una generalización del código Hamming. En 1960, I. S. Reed y G. Solomon introducen sus códigos polinomiales, considerados unos códigos BCH cíclicos.

Estudios realizados sobre estos códigos han visto que el alfabeto de símbolos que utilizan es muy pequeño, lo que limita el número de símbolos a generar. Además, el tiempo de cómputo necesario para realizar la codificación y la decodificación es cuadrático, lo que vuelve su uso prohibitivo en muchos entornos. Además usa factores de estiramiento siempre constantes, lo que produce que la secuencia de símbolos enviados se repita y, en ciertos entornos de ejecución, se reciban muchos paquetes duplicados y, por tanto, inútiles.

Pese a esos inconvenientes, su uso se ha extendido en muchos otros entornos, mucho más adecuados a las características que poseen, como pueden ser los reproductores de CD, DVD, etc. Su alta longevidad ha hecho también que el estudio realizado sobre este tipo de códigos haya sido minucioso y proliferen diferentes versiones de este método. Así han aparecido los códigos Hadamard, Wash-Hadamard, Reed-Muller, etc. siendo algunos códigos casos específicos o casos generales de los otros.

3.2.2.- Turbo codes

Los Turbo codes fueron publicados en 1993 por Claude Berrou. Estos códigos, a diferencia de los anteriores, pertenecen a la rama de los códigos convolucionales. Estos códigos se basan en crear palabras de código (símbolos) formadas por: a) parte de la información a transmitir tal cual y b) información redundante. La información redundante se forma de dos fuentes distintas: para obtener la primera se le aplica a la información original un proceso de codificación y, para obtener la segunda, se le aplica a la información original un barajado y posteriormente otro codificado que no tiene por que ser igual que el de la fuente anterior. Un esquema sencillo de estos códigos usa símbolos constituidos en 1/3 por información original, 1/3 de información redundante de la fuente 1 y 1/3 de información redundante de la fuente 2.

Este tipo de códigos cuentan con la ventaja de ser mucho más eficientes que los anteriores, dado que es posible acercarse al límite de Shannon, el límite máximo de aprovechamiento del canal cuando se usan técnicas de codificación, que con propuestas anteriores.

3.2.3.- Tornado codes y el paradigma de las fuentes digitales

Los Tornado codes fueron creados en 1998 por Michael Luby como un tipo de LDPC o códigos de baja densidad. Este tipo de códigos hace que un símbolo de información redundante usado para recuperar información no dependa de todos los símbolos originales, como pasaba con los códigos Reed-Solomon, sino que solo dependa de algunos, lo que reduce el coste de codificación y decodificación.

Este fue el primer paso hacia un paradigma de codificación llamado “fuentes digitales” (digital fountains). Se consideran fuentes digitales aquellos códigos que, siendo capaces de recuperar la información incluso en caso de pérdida de parte de la misma (erasure codes), no tienen una tasa fija de información redundante a enviar junto con la información original (rateless). La primera solución que asimila completamente este paradigma son los códigos LT, de Michael Luby, que nacen como una evolución de los Tornado codes. Estos códigos establecen un grado para cada uno de los símbolos de recuperación en función del número de símbolos fuente usados para crearlos. Mención especial merecen los online codes de P. Maymounkov. Estos códigos también pertenecen al grupo de las fuentes digitales, y también están inspirados en los códigos Tornado.

Una evolución de los códigos LT son los códigos Rapid Tornado (Raptor). Estos códigos son una optimización de los códigos LT en cuanto a tiempo de codificación y decodificación. Como éstos son los códigos que se han usado en el proyecto que se presenta, están explicados en detalle en el apartado siguiente.

3.3.- Funcionamiento y características de los Raptor Codes

3.3.1.- Base teórica

Los códigos Raptor, o Raptor Codes, son una serie de códigos redundantes de tipo FEC (corrección de errores hacia delante) que permiten el envío de información de manera tolerante a fallos, pudiendo recuperar la información en virtualmente cualquier caso. Estos códigos trabajan sobre bloques fijos de datos, por lo que se engloban dentro del primer tipo de códigos FEC de los dos mencionados anteriormente.

El hecho de que se haga necesario el envío de datos adicionales a los datos originales a transmitir es el principal inconveniente de esta tecnología. Para que los datos redundantes enviados no produjeran una sobrecarga innecesaria en la red, es decir, para que todos los datos enviados por la red se perdieran o tuvieran que ser usados indiscutiblemente para recuperar el mensaje en el cliente, sería necesario conocer *a priori* cuales van a ser los paquetes que se van a perder. Obviamente eso es imposible. La tecnología de Raptor Codes viene a limitar ese problema, dada la siguiente restricción que cumplen dichos códigos: “Para un entero k dado, con un $\varepsilon > 0$, Raptor Codes pueden producir un flujo potencialmente infinito de símbolos tales que cualquier subconjunto de dichos símbolos de tamaño $k(1 + \varepsilon)$ es suficiente para recuperar los k símbolos originales con una alta probabilidad”. Con esto conseguimos que, independientemente de los símbolos que se pierdan, podamos recuperar los símbolos originales.

El problema con el que nos encontramos ahora es conocer cuantos paquetes se perderán y cuantos paquetes llegarán al cliente de manera efectiva. Si pudiésemos conocer a priori este valor sabríamos cuantos paquetes tenemos que enviar al cliente y, por lo tanto, cuando tenemos que

terminar de enviar símbolos redundantes. Sin embargo, de ese valor solo podemos hacer una estimación en base a la tasa de error, por lo que la única manera de conocer efectivamente cuando se ha decodificado un bloque es ser avisado por el cliente en ese momento.

El otro gran inconveniente de la tecnología es el coste computacional de la creación de símbolos de recuperación y también el coste de la recuperación de la información en el cliente. Encontrar una solución a este problema ha llevado a los investigadores a crear versiones óptimas de los códigos FEC, entre las que destaca la tecnología de Raptor Codes.

Los códigos Raptor fueron inventados por Amin Shokrollahi [8] y fueron presentados como una versión más eficiente de los códigos LT inventados por Michael Luby [9]. Estos códigos se caracterizan por un tiempo de codificación y decodificación lineal respecto al tamaño del mensaje. En el presente proyecto se ha usado la implementación de la empresa Digital Fountain, DF Raptor, en su versión número 11.

La mejor manera de entender el funcionamiento de los códigos FEC, y en especial de los códigos Raptor, es usando la metáfora con la que la misma empresa Digital Fountain acompaña sus productos. Los códigos Raptor son como una fuente digital que, al igual que una fuente de agua produce un flujo constante e infinito del líquido elemento, ésta produce una serie potencialmente infinita de símbolos; y, al igual que cualquier grupo de gotas de agua de una fuente real puede llenar un vaso, también cualquier conjunto de símbolos, independientemente de cuales sean y sabiendo sólo su número de orden, pueden ser usados para recuperar la información original del emisor.

3.3.2.- La implementación práctica: DF Raptor de Digital Fountain

DF Raptor es un código de corrección de errores que permite al receptor recuperar datos que han sido perdidos en su envío a través de la red, o que han sido descartados si habían sido recibidos en mal estado. Esto le convierte en un sistema de corrección de errores de la familia de los códigos correctores de errores de borrado. Esto quiere decir que, si de un pedazo de información codificado con este sistema borramos una serie de partes (símbolos), somos capaces de recuperar de todas formas la información original que teníamos antes de que fuera codificada.

Digital Fountain [10] nos ofrece un producto con tres características básicas:

- La capacidad de generar flujos potencialmente infinitos de información codificada a partir de un bloque original de información finito, permitiendo así recuperar la información incluso en situaciones con un nivel de error extremo.
- La capacidad de recuperar los datos partiendo de cualquier subconjunto de símbolos con la única restricción que su cantidad sea ligeramente mayor a la original.
- Una codificación y decodificación excepcionalmente rápida de coste lineal con la cantidad de datos originales

El proceso de codificación parte de una información que es dividida en primera instancia en bloques, constituyendo un bloque la unidad de información que será objeto de una codificación en el origen y una decodificación en el destino.

Para un uso más eficiente, flexible, óptimo y potente de las ventajas de una corrección de errores de borrado, el bloque anteriormente citado es dividido en símbolos, a los que se les llamará símbolos fuente. Cada símbolo suele ser identificado con un paquete en lo que posteriormente será la transmisión de la información a través de la red.

Este conjunto de símbolos es procesado para formar un conjunto de símbolos mayor, formado por los símbolos fuente originales y un conjunto de símbolos de recuperación o reparación del mismo tamaño que los anteriores. El número de símbolos de recuperación que puede ser generado es, como ya se ha dicho, infinito.

Tras esto, todos los símbolos fuente son enviados al receptor junto con los símbolos de recuperación que sean necesarios. De todos ellos, el receptor recibirá un subconjunto lo suficientemente grande como para poder recuperar completamente la información original. Para ello se generarán y se enviarán tantos símbolos de recuperación como sea necesario.

Una vez que el cliente ha recibido todos los símbolos necesarios iniciará el proceso de decodificación y obtendrá un conjunto de símbolos exactamente iguales a los símbolos fuente que tenía el receptor, es decir, obtendrá el bloque original que el emisor quería transmitirle. La única información extra que necesitará el emisor aparte de los propios símbolos es el tamaño del bloque, el tamaño de los símbolos, y la posición original que tenían cada uno de los símbolos en la estructura generada por el codificador tras procesar el bloque original.

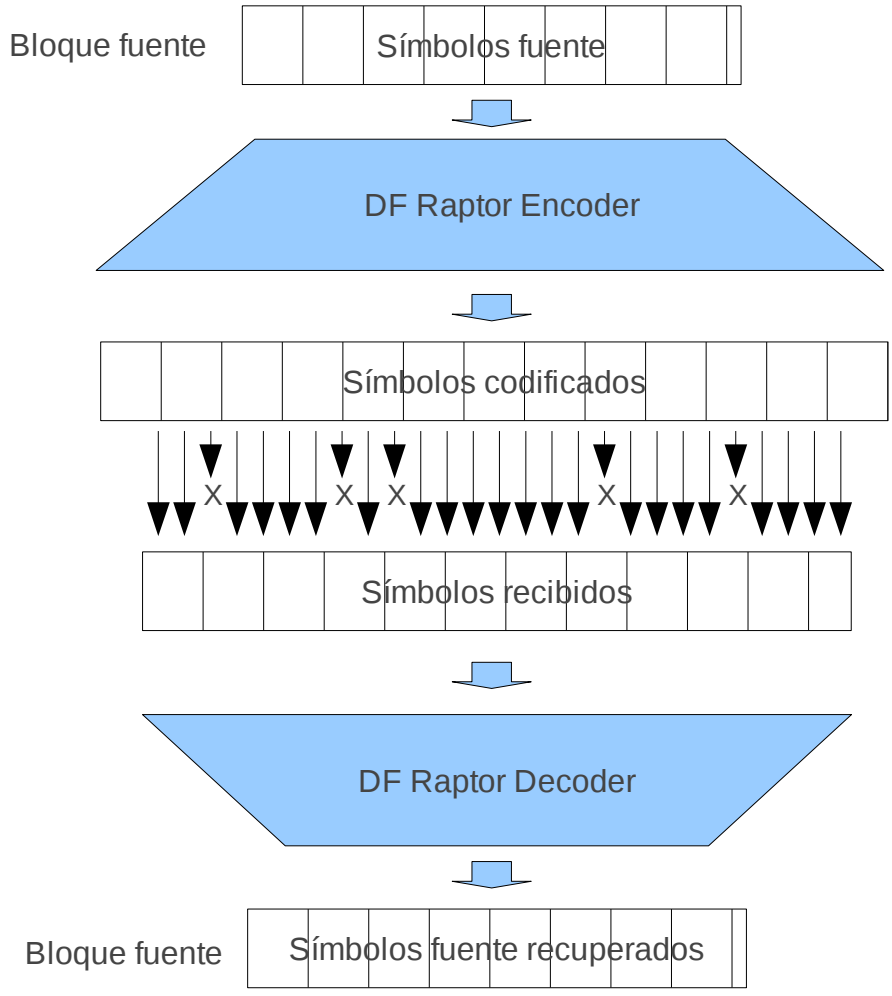

Figura 1: Esquema simplificado de codificación y decodificación de información usando Raptor Codes

El proceso de codificación del bloque de información está orientado a generar una sucesión infinita de símbolos de recuperación. Este proceso se basa en la realización de XORs sobre conjuntos de símbolos fuente para obtener así los nuevos símbolos de recuperación. En función del número de símbolos fuente que entran en contacto para generar el símbolo de recuperación se establecerá el grado de dicho símbolo. Así pues, un símbolo de recuperación para el que solo se ha usado un símbolo fuente será de grado uno, si se han usado dos símbolos fuente será de grado dos, de grado tres si se usaron tres símbolos, etc.

El grado de cada símbolo de recuperación es decidido por el codificador LT. Este componente del codificador Raptor decide, en base a una distribución de probabilidad, cuál debe ser el grado del símbolo de recuperación que está generando. Así mismo, también usa una distribución de probabilidad para la selección de cuales serán los símbolos fuente que entren a formar parte de esta operación concreta. Esto hace que sea necesario incluir en el envío al receptor la información relativa al grado del símbolo y a los bloques que entran en juego a la hora de realizar la decodificación. Por ello, esta información va incluida en el símbolo generado por la librería DF Raptor.

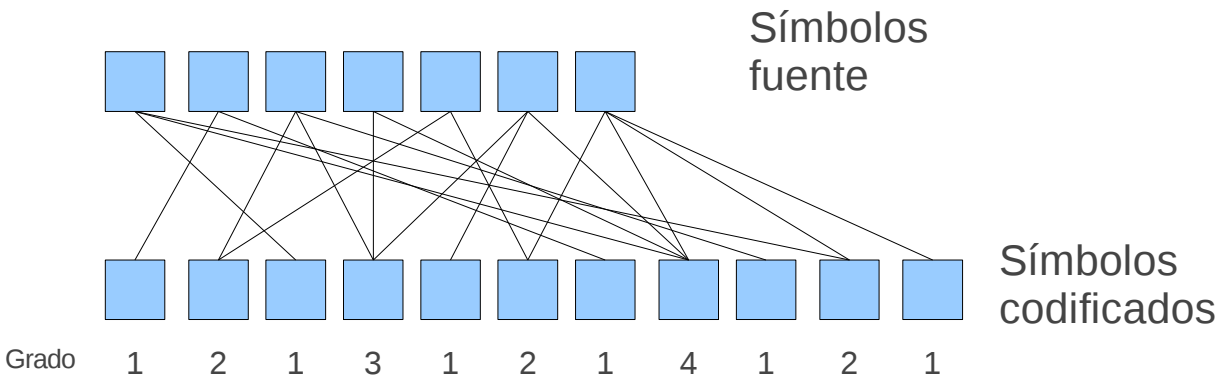

Figura 2: Creación de símbolos codificados en una sola fase

El coste temporal de codificación y decodificación viene determinado por el número de XOR que hay que realizar, y este número viene determinado a su vez por el grado de un bloque, que viene determinado por las distribuciones de probabilidad que usa el codificador LT para su generación. Sin embargo, la complejidad obtenida por las mejores distribuciones de grados no es lineal. Para obtener una librería con un coste lineal de codificación y decodificación, es necesario contar con un diseño de 2 etapas.

Las dos etapas necesarias para conseguir una complejidad lineal son las siguientes:

- **Etapa de pre-codificación:** A los símbolos fuente que debemos codificar se les aplica un algoritmo de preparación de bajo coste. El resultado de este algoritmo es un bloque pre-codificado de símbolos.
- **Etapa de codificación LT:** Al bloque de símbolos pre-codificados, le aplicamos el algoritmo de codificación LT, obteniendo así ya un número infinito de símbolos de recuperación, tal y como era nuestra nuestra intención inicial.

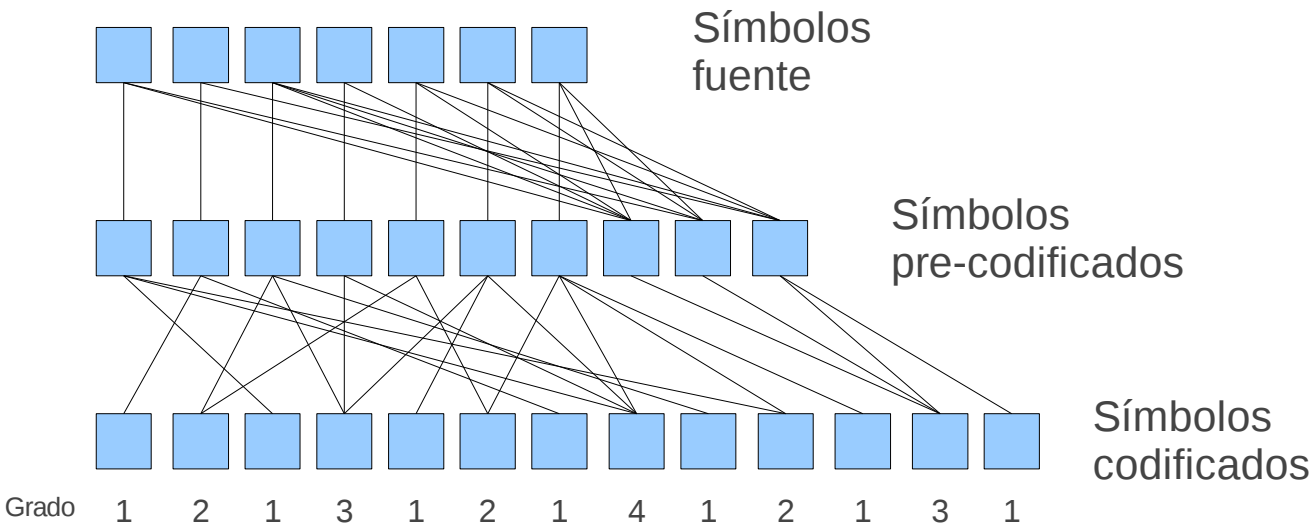

Figura 3: Creación de símbolos codificados en dos fases

Cuando el receptor ha recibido el número suficiente de símbolos puede proceder a la decodificación del bloque para obtener la información original. Esta decodificación, al igual de la codificación, también se basa en la realización de operaciones XOR. Esta decodificación se lleva a cabo aplicando también dos etapas para recuperar la información fuente, realizándose primero la decodificación LT para obtener los símbolos pre-codificados que, en la siguiente fase, se usarán para recuperar los símbolos fuente.

El codificador LT usado en la librería DF raptor intenta conseguir la menor complejidad posible, teniendo en cuenta que incluso si el decodificador LT no fuera capaz de recuperar todos los símbolos pre-codificados, el algoritmo de pre-codificación puede ser capaz de rellenar esos huecos correctamente.

3.3.2.1- El codificador DF Raptor R11

Según la compañía Digital Fountain, las características de su producto son las siguientes [11]:

- Permite una recuperación perfecta, robusta, flexible y escalable de los datos enviados por una red con errores.
- Uso eficiente de los recursos de red, con un 0% de sobrecarga en situaciones de bajas pérdidas, y una sobrecarga típica de menos del 2% del bloque original. La probabilidad de un fallo de decodificación es de 10^{-11} con un 3% de sobrecarga.
- Coste computacional lineal con respecto al tamaño del bloque.
- Al no modificar los símbolos fuente iniciales, permite ahorrar tiempo y evitar la etapa de decodificación FEC si no se perdiera ninguno de estos símbolos.
- Permite 56.405 símbolos fuente, con un tamaño que va desde 1 byte a 65.535 bytes, y permite la generación de hasta 2.147.478.648 símbolos de recuperación distintos.

Modos de operación:

La librería cuenta con tres modos de operación:

- **Modo Estándar:** Codifica con la menor cantidad de memoria posible. La conversión de símbolo fuente a símbolo intermedio se hace en el mismo espacio de memoria.

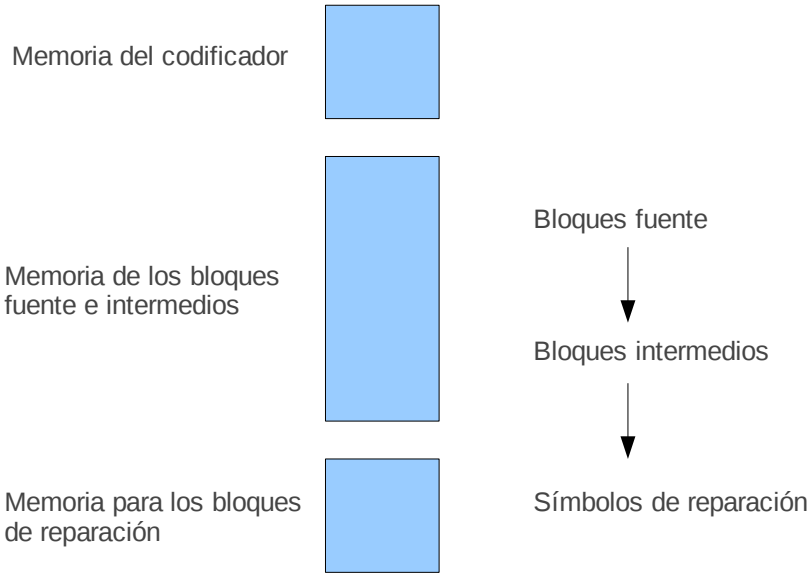

Figura 4: Modo estándar

- **Modo DMA:** Usa un dispositivo DMA para el intercambio de memoria.

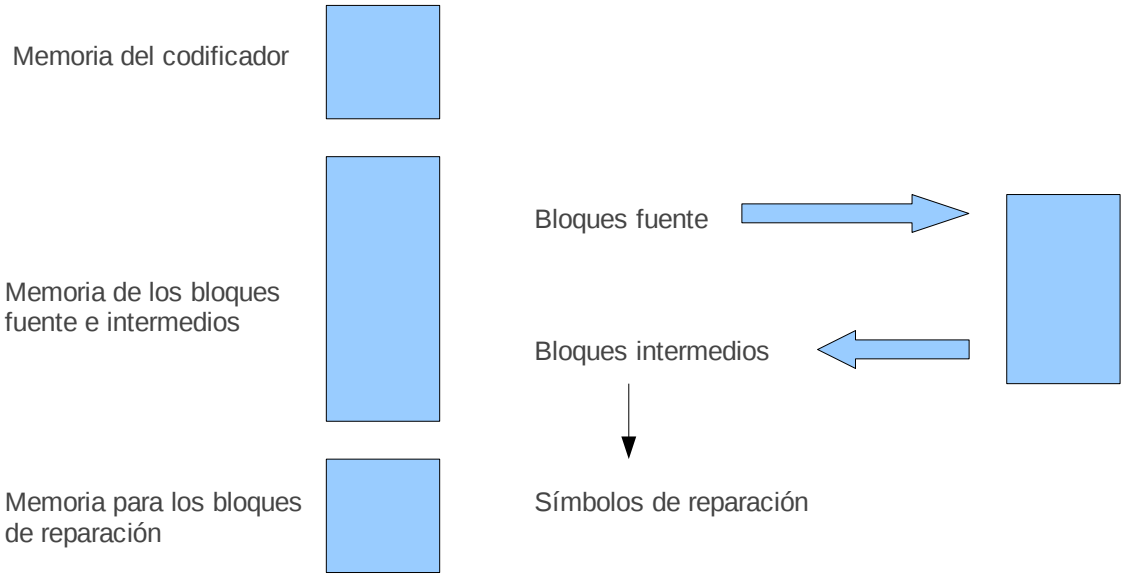

Figura 5: Modo DMA

- **Modo de alta velocidad:** Incrementa el uso de memoria para ganar velocidad en la codificación. Guarda en espacios de memoria distintos los símbolos fuente y los símbolos intermedios.

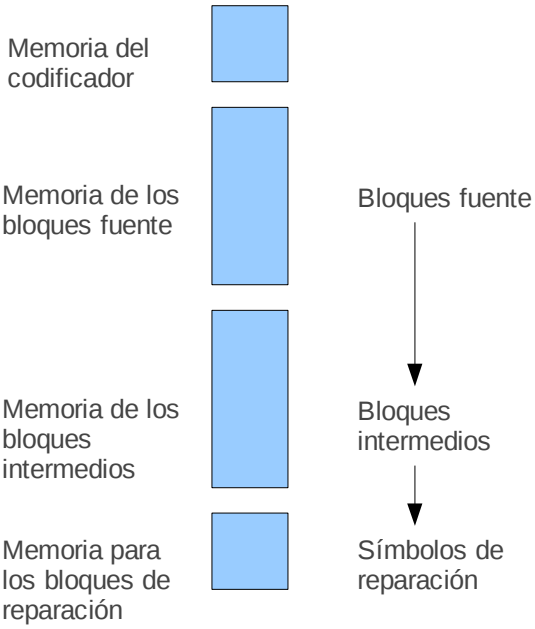

Figura 6: Modo de alta velocidad

Flujo de operación:

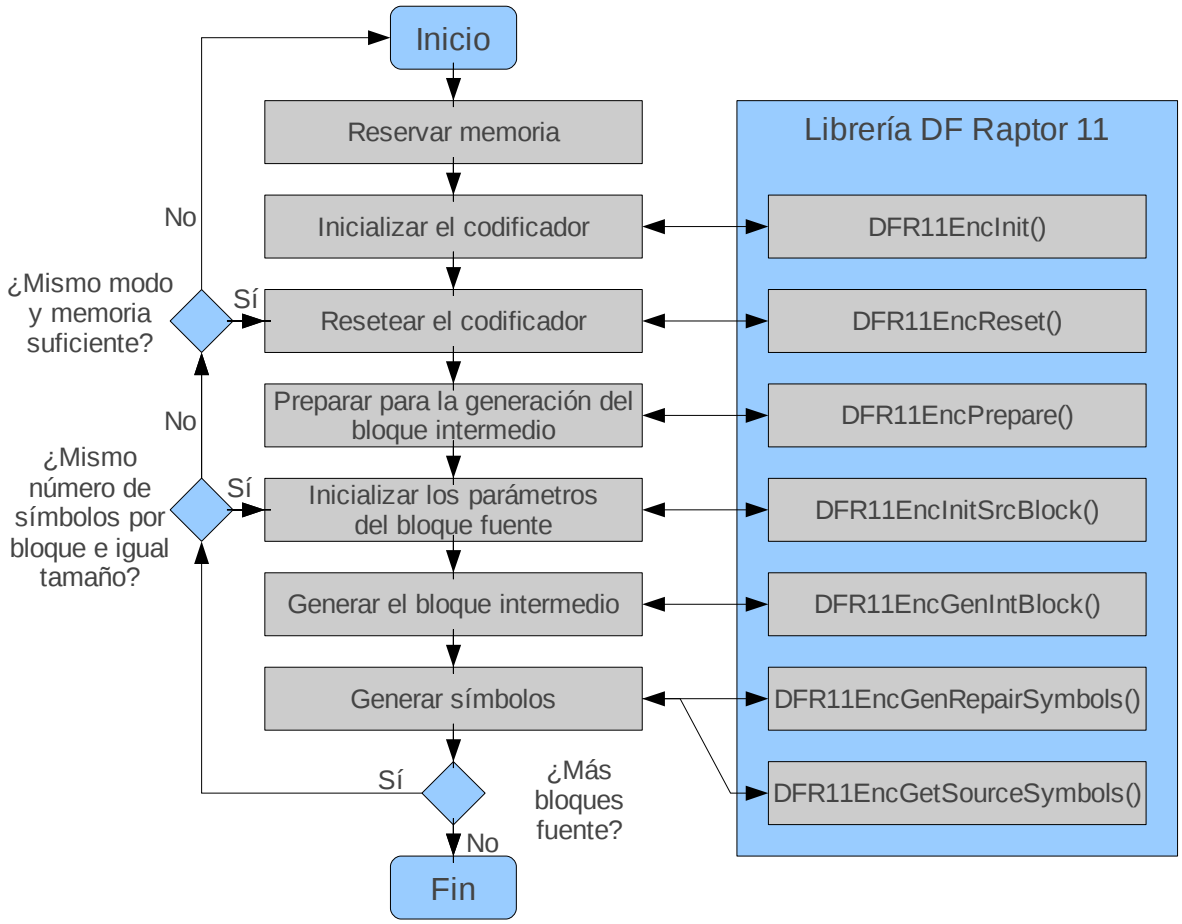

Figura 7: Flujo de operación de codificación

3.3.2.2- El decodificador DF Raptor R11

Según la compañía Digital Fountain, las características de su producto son las siguientes [12]:

- Permite una recuperación perfecta, robusta, flexible y escalable de los datos enviados por una red con errores.
- Uso eficiente de los recursos de red, con un 0% de sobrecarga en situaciones de bajas pérdidas, y una sobrecarga típica de menos del 2% del bloque original. La probabilidad de un fallo de decodificación es de 10^{-11} con un 3% de sobrecarga.
- Coste computacional lineal con respecto al tamaño del bloque.
- Al no modificar los símbolos fuente iniciales, permite ahorrar tiempo y evitar la etapa de decodificación FEC si no se perdiera ninguno de estos símbolos.
- Permite 56.405 símbolos fuente, con un tamaño que va desde 1 byte a 65.535 bytes y permite la generación de hasta 2.147.478.648 símbolos de recuperación distintos.

Modos de operación:

La librería cuenta con cuatro modos de operación:

- **Modo de poca memoria:** Al igual que el modo estándar, la conversión de símbolo recuperado a símbolo fuente se hace en el mismo espacio de memoria y además se compromete más la velocidad para intentar ahorrar la máxima cantidad de memoria.

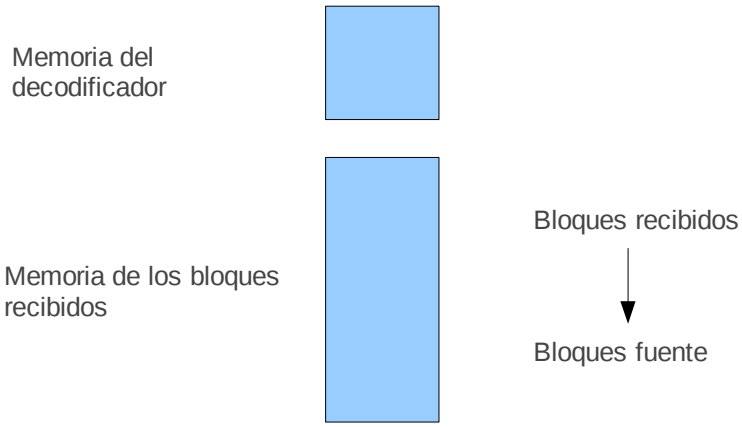

Figura 8: Modo de poca memoria

- **Modo Estándar:** La conversión de símbolo recuperado a símbolo fuente se hace en el mismo espacio de memoria.

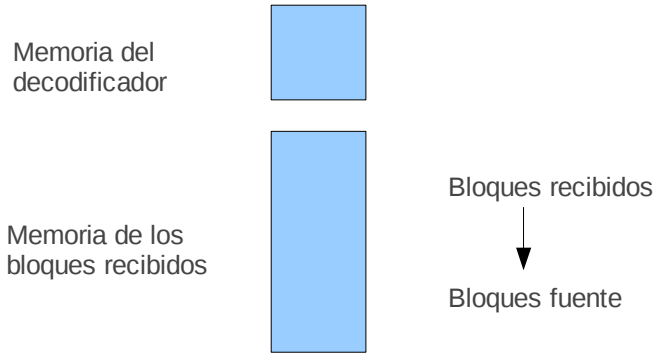

Figura 9: Modo Estándar

- **Modo de alta velocidad:** Incrementa el uso de memoria para ganar velocidad en la decodificación. Guarda en espacios de memoria distintos los símbolos fuente y los símbolos intermedios.

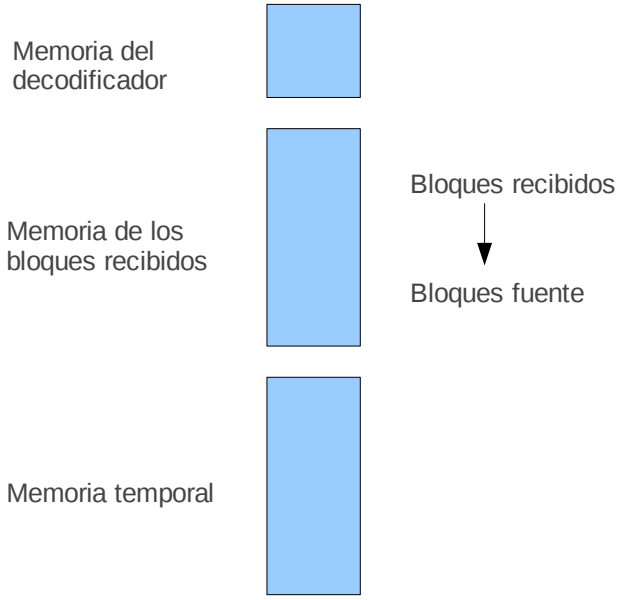

Figura 10: Modo de alta velocidad

- **Modo DMA:** Usa un dispositivo DMA para el intercambio de memoria.

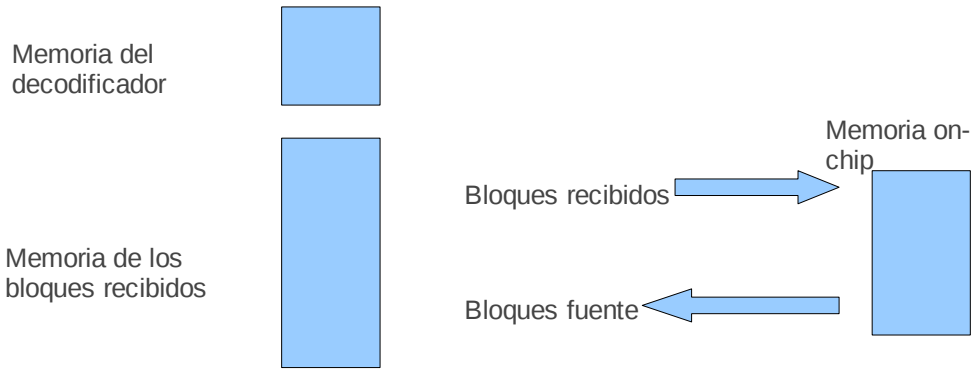

Figura 11: Modo DMA

Flujo de operación:

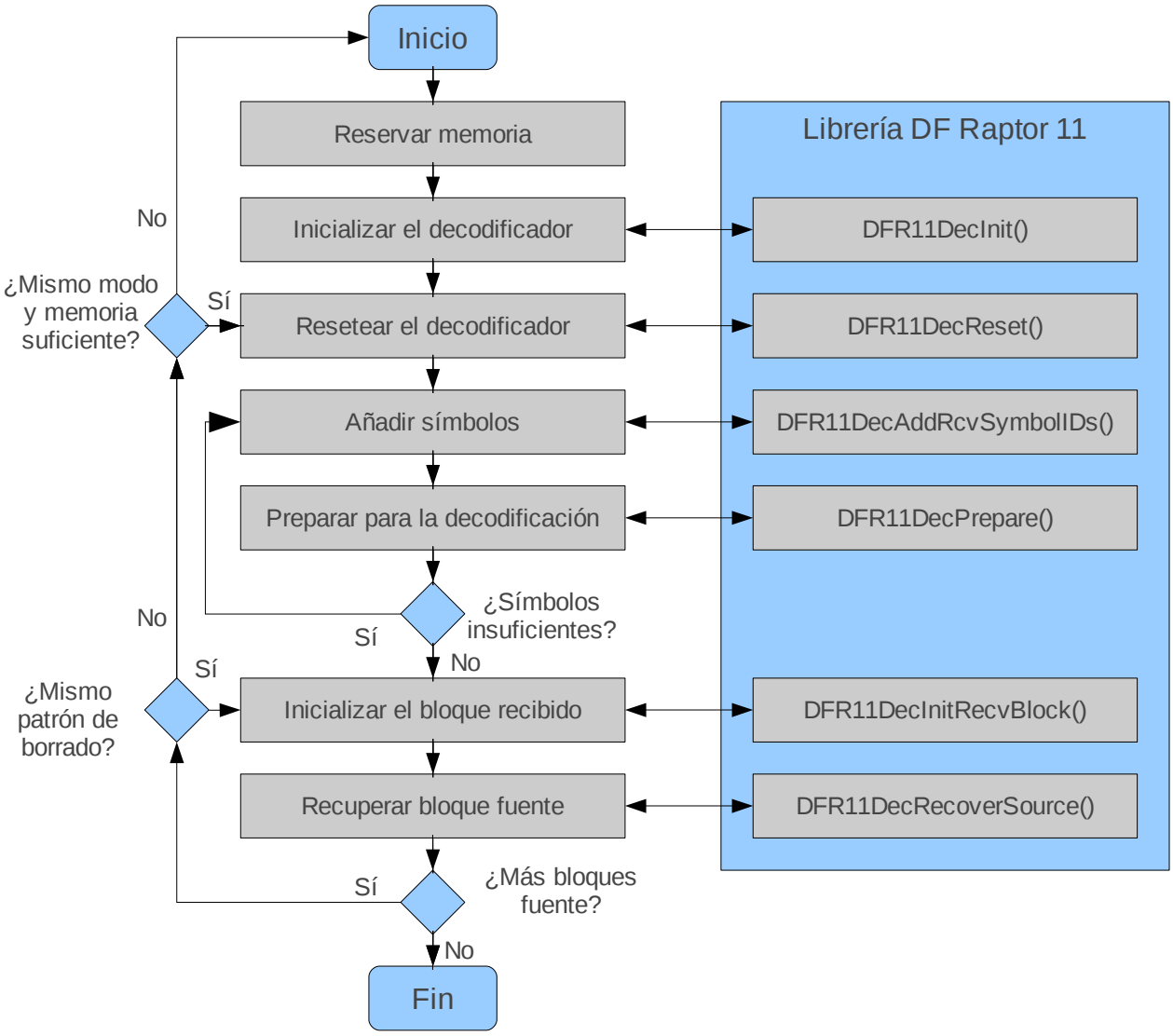

Figura 12: Flujo de operación de decodificación

4.- RDT: protocolo para la entrega fiable de datos en redes inalámbricas basado en Raptor Codes

Con el presente proyecto se pretende presentar un protocolo optimizado de transporte para redes inalámbricas basado en Raptor Codes. El principal objetivo es que este protocolo no sólo supere en rendimiento al protocolo TCP en condiciones similares a las típicas de una red inalámbrica, sino que también obtenga el mejor rendimiento posible a nivel de consumo de recursos internos de los dispositivos transmisor y receptor (CPU, memoria) y recursos externos (capacidad de red, congestión).

Para conseguir esto se explorarán todas las posibles mejoras implementables, así como se realizará un ajuste de todos los parámetros que pueden ser ajustados dentro de la estructura que implementa el protocolo, incluyendo un posterior estudio de los resultados obtenidos para cada configuración en cada uno de los entornos seleccionados como representativos para las pruebas.

4.1.- Descripción del protocolo

El protocolo RDT está diseñado para evitar los problemas que surgen en protocolos como TCP a la hora de transmitir datos en canales con alta pérdida de paquetes, como puede ser una red inalámbrica. RDT usa como esqueleto la librería UDT, cuya implementación será descrita más adelante, integrando a su estructura una nueva fase de codificación en el envío y decodificación en la recepción usando códigos Raptor. Con esta estrategia podremos olvidarnos de si los paquetes llegan íntegramente al destino, ya que podremos recuperar el contenido enviado usando las propiedades de los códigos Raptor.

La idea básica del protocolo queda culminada por la inclusión de un control de la tasa de transmisión al sistema basado en la frecuencia de recepción de paquetes por parte del cliente. Así, la estimación del ancho de banda del canal se podrá realizar sin tener en cuenta los paquetes perdidos o los reconocimientos de los paquetes enviados, y centrarnos verdaderamente en el ancho de banda disponible.

4.2.- Diseño

El diseño de este protocolo viene marcado por la estructura de la librería UDT. Así se consigue aprovechar, entre otros, su sistema de establecimiento de una conexión, así como la gestión interna de los sockets y demás estructuras de comunicación. Con esta estrategia se conseguirá evitar todo el trabajo no relacionado con el proyecto a realizar, pudiendo así destinar íntegramente los esfuerzos al objetivo que se pretende.

4.2.1.- El esqueleto: la librería UDT

La librería UDT [13] es una librería de comunicaciones que implementa a nivel de usuario el protocolo UDT, diseñado para sacar el máximo rendimiento en las redes WAN de alta velocidad.

La librería ofrece una estructura sencilla organizada en capas. La primera establece un interfaz de sockets con la aplicación desarrollada sobre ella. La segunda conforma el núcleo de la librería y es la encargada de controlar todo lo relacionado con el protocolo que implementa, el protocolo UDT. La tercera es una abstracción del canal UDP que usa para encapsular todo el sistema de comunicación dependiente del sistema operativo.

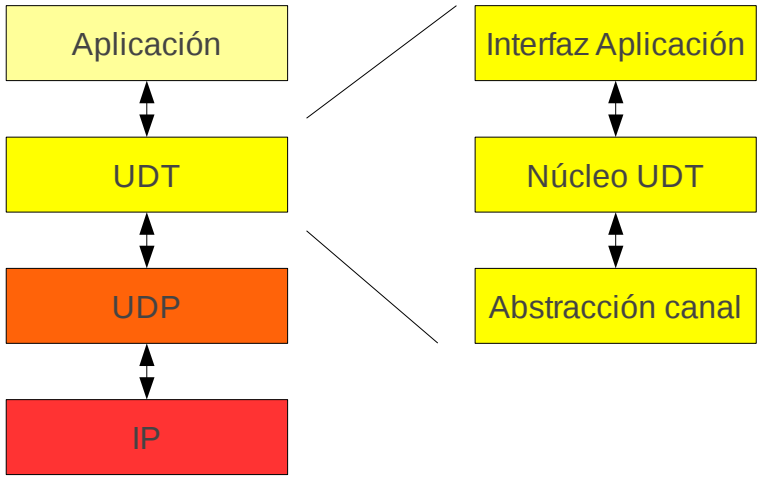

Figura 13: Estructura en capas de la librería UDT, relacionada con el resto de protocolos del sistema

La librería establece una encapsulación de la información a este nivel que hace posible una comunicación orientada a conexión sobre el protocolo UDP. Para conseguir esto se definen dos tipos de mensaje, de control y de datos, que difieren en el valor del primer bit. Los campos de la cabecera de los paquetes son los siguientes:

- **Paquete de datos:**
  - **Flag bit:** primer bit de la cabecera distingue si se trata de un paquete de datos (valor 0) o un paquete de control (valor 1).
  - **Número de secuencia:** Tiene la misma función que el número de secuencia de TCP, por lo que nos sirve para evitar duplicados y ordenar los mensajes cuando llegan al receptor.
  - **Número de mensaje:** Usado para el envío de mensajes por parte de la aplicación. Un mensaje se trata como un bloque de información con un sentido particular enviado por la aplicación y puede ocupar más de un paquete.
  - **Campo FF:** Dentro de un mensaje, indica de que tipo de paquete se trata. Se usa el valor 10 para el primer paquete de un mensaje, 00 para un paquete intermedio, 01 para el último paquete de un mensaje y 11 en caso de que el mensaje ocupe solo un paquete.
  - **Marca temporal (timestamp):** Marca temporal relativa del paquete, iniciada con el inicio de conexión, al nivel de microsegundos.
  - **Campo O (orden):** Indica si el paquete debe entregarse en orden, es decir, si debe de retrasarse la entrega de este paquete hasta que todos los paquetes anteriores se hayan entregado.

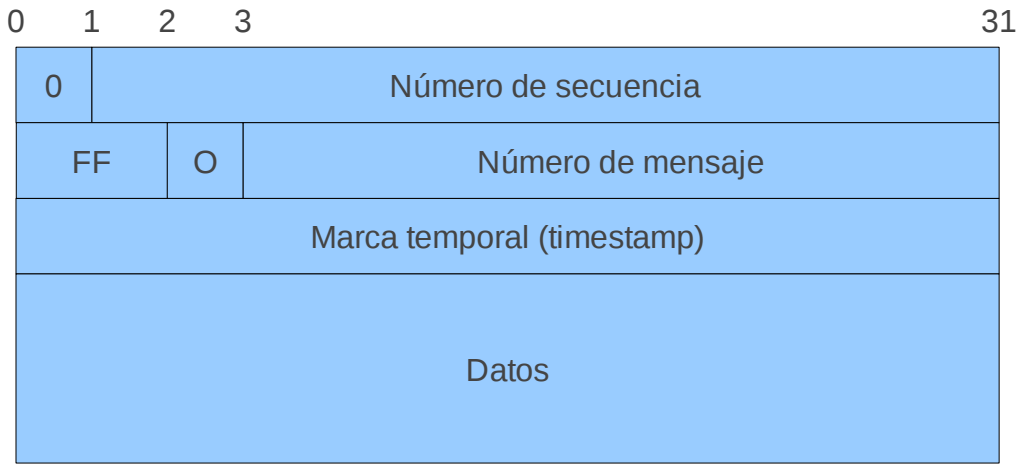

Figura 14: Paquete de datos

- **Paquete de control:**
  - **Flag bit:** primer bit de la cabecera distingue si se trata de un paquete de datos (valor 0) o un paquete de control (valor 1).
  - **Tipo de paquete de control:** Hay siete tipos de paquetes de control para la gestión del protocolo, un octavo tipo para los paquetes personalizados por la aplicación o definidos por el usuario, y un noveno tipo que no se usa.
    - **Handshake:** Usado para el inicio de una conexión nueva y la negociación de las características de esa conexión.
    - **ACK:** Mensaje de reconocimiento de un paquete recibido.
    - **ACK2:** Mensaje de reconocimiento de un mensaje de reconocimiento.
    - **Informe de pérdida:** Paquete que informa al emisor de que un paquete que envió se ha perdido.
    - **Keep-alive:** Paquetes que se envían periódicamente entre el emisor y el receptor para informarse mutuamente de que siguen vivos y no se debe cerrar la conexión.
    - **Shutdown:** Paquete de cierre de conexión.
    - **Paquete descartado:** El emisor pide al receptor que elimine un paquete que previamente le ha enviado
    - **Paquete definido por el usuario:** Paquete de control definido por la aplicación que se desarrolla sobre la librería. La librería permite al desarrollador definir paquetes de control y el código para tratar dichos paquetes.
    - **Advertencia de congestión:** Informa al emisor de que existe una congestión en la red para que reduzca su tasa de envío. No se usa.
  - **Tipo extendido del paquete de control:** Campo usado en los paquetes de control definidos por el usuario, que permite al desarrollador refinar más el tipo del paquete.
  - **Número de secuencia de los ACK:** A cada ACK se le incorpora un número de secuencia independiente del número de secuencia del paquete que reconocen.
  - **Marca temporal (timestamp):** Marca temporal relativa del paquete, iniciada con el inicio de conexión, al nivel de microsegundos.
  - **Información de control:** Información propia del paquete de control.

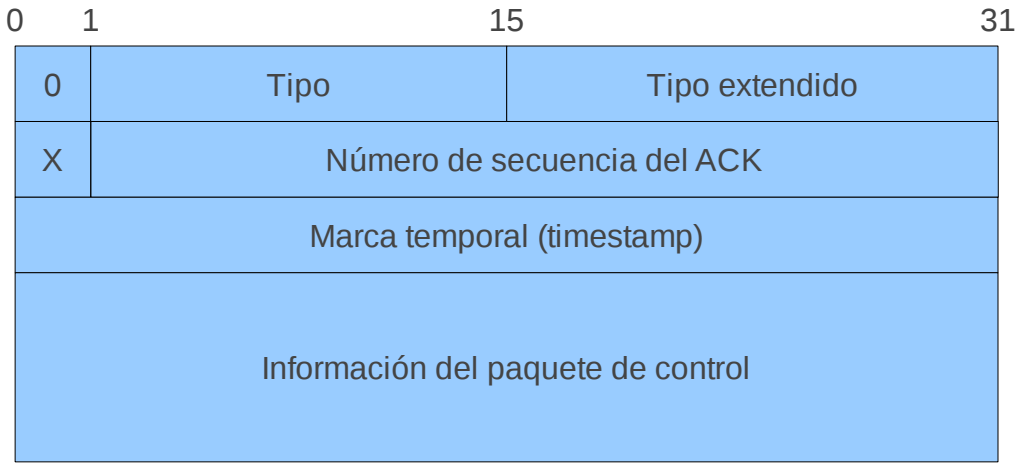

Figura 15: Paquete de control

4.2.1.1.- Ofreciendo un servicio orientado a conexión

Con esta estructura de empaquetado la librería es capaz de ofrecer dos procedimientos para el inicio de una conexión: el modo cliente-servidor y el modo *rendezvous*.

En el modo cliente-servidor el cliente envía un paquete de *handsake* al servidor. En él indica la versión de la librería UDT que está usando, el tipo de *socket* (orientado a conexión o no), el número de secuencia inicial (aleatorio), el tamaño máximo de paquete y el tamaño máximo de la ventana de congestión.

El servidor comprueba la versión de la librería, establece el tamaño de paquete al mínimo entre el suyo y el recibido y devuelve este valor, junto con su número de secuencia y el tamaño máximo de su ventana de congestión. Tras el intercambio de estos dos mensajes, tanto el cliente como el servidor están preparados para comunicarse.

En el modo *rendezvous* ambos nodos intentan establecer la conexión simultáneamente y el que reciba el mensaje enviará la respuesta e iniciará la fase de set-up.

Dado que en el envío de estos paquetes de inicio de conexión alguno se puede perder, la librería soluciona este problema con el reintento del establecimiento de conexión entre los extremos. Este detalle es especialmente importante en canales con pérdidas frecuentes como son las redes inalámbricas, medio en el que se centra el trabajo que estamos presentando.

Para detectar cuando la conexión sigue activa, la librería se basa en la recepción de paquetes del otro extremo. Cuando la transmisión se está produciendo no existe ningún problema en determinar si la conexión está disponible. Sin embargo, cuando hay periodos de espera en los que no se transmite nada, podría darse el caso que una conexión que está en perfectas condiciones fuera detectada como una conexión rota. Para evitar eso la librería periódicamente envía mensajes sin contenido útil para indicar que el extremo está vivo y ,por tanto, también lo está la conexión.

4.2.1.2.- Herramientas de soporte al desarrollo sobre UDT

La librería, por defecto, ofrece el control de congestión del protocolo UDT. Este sistema se ha definido como un algoritmo DAIMD (Decreasing Aditive Increase, Multiplicative Decrease / Incremento aditivo decreciente, decremento multiplicativo).

En el algoritmo de control de congestión, para cada intervalo, cuando no hay reconocimientos negativos (NACK) pero si los hay positivos (ACK), la tasa de envío se incrementa de una manera aditiva, es decir, la tasa de envío de paquetes (x) es igual a la tasa anterior más un valor dependiente de x. Así pues, $x = x + a(x)$. $a(x)$ es una función "no creciente" y que tiende a 0 cuando x tiende a infinito.

Cuando se reciben reconocimientos negativos (NACK / informes de pérdidas de paquetes) la tasa de envío se reduce de manera multiplicativa, es decir, la tasa de envío de paquetes (x) se multiplica por un valor constante (b) comprendido entre 0 y 1 ($0 < b < 1$).

Variando la función a(x) es como se consigue el efecto decreciente en el incremento aditivo del algoritmo. Este sistema se ha demostrado como estable asíncronamente y converge a un estado de equilibrio. Además, debe ofrecer un valor grande para puntos cercanos a a(0) y de un rápido descenso para reducir las oscilaciones.

Sin embargo, la librería no se cierra a ese control de congestión, sino que permite la definición de otros nuevos sin necesidad de modificar la implementación. A esta característica se la ha llamado *composable UDT* [14]. Esta característica está enfocada para:

- Implementar y desarrollar nuevos algoritmos de control.
- Configuración dinámica y soporte a la aplicación.
- Evaluación de nuevos algoritmos.

Composable UDT ofrece un interfaz para que la aplicación pueda interactuar con la librería. La librería UDT proporciona la clase CCC, una clase extensible que permite al desarrollador definir un comportamiento personalizado para algunos eventos que se produzcan en la transmisión. Las funciones que se pueden definir son las siguientes:

- **init:** Este método es llamado cuando se establece una nueva comunicación. Suele ser usado para inicializar estructuras de datos.
- **close:** Este método es llamado cuando se cierra una conexión. Suele ser usado para destruir las estructuras de datos inicializadas en el método init.
- **onACK:** Este método es invocado cada vez que se recibe un ACK por parte del emisor. Se puede conocer el número de secuencia del ACK ya que es pasado como parámetro al método.
- **onLoss:** Este método es invocado cada vez que se detecta un evento de pérdida de paquete. La información del paquete que se ha detectado como perdido es suministrada en los parámetros del método.
- **onTimeout:** Se puede definir un temporizador que ejecutará el código implementado en este método en el momento deseado por el programador.
- **onPktSent:** Este método es invocado cada vez que se envía un paquete de datos por parte del emisor. Toda la información del paquete es pasada como parámetro y está disponible dentro del método.
- **onPktReceived:** Este método es invocado cada vez que se recibe un paquete de datos por parte del receptor. Toda la información del paquete es pasada como parámetro y está

disponible dentro del método.

- **processCustomMsg:** Este método es invocado cada vez que se recibe un paquete de control personalizado (definido por el programador en la aplicación).

Para utilizar correctamente estos métodos y adaptar el protocolo a las características de un determinado entorno se ofrecen ciertos métodos a la aplicación para configurar el comportamiento de la librería:

- **setACKTimer:** Establece cada cuanto tiempo se envía un ACK para reconocer los paquetes del servidor.
- **setACKInterval:** Establece cada cuantos paquetes se envía un ACK para reconocer los paquetes del servidor.
- **sendCustomMsg:** Envía un mensaje de control personalizado usando el núcleo de la librería para hacerlo llegar al destino. Este mensaje puede ser procesado en el destino gracias al método processCustomMsg.
- **setRTT:** Establece el valor de RTT (round trip time) que usará la librería.
- **setRTO:** Establece el valor de RTO (request timed out) que usará la librería.

La librería, además, permite monitorizar el rendimiento de la transferencia de datos entre el cliente y el servidor, información que puede ser usada para conocer la eficiencia de la emisión o para definir el comportamiento de algoritmos que cambian de comportamiento a lo largo del tiempo. Las variables que se almacenan son las siguientes:

- Duración desde el inicio de la conexión
- RTT
- Tasa de envío
- Tasa de recepción
- Tasa de pérdidas
- Periodo de envío de paquetes
- Tamaño de la ventana de congestión
- Tamaño de la ventana de flujo
- Número de ACKs
- Número de NACKs (informe de pérdida de paquete)

Además, se pueden recuperar estas estadísticas de tres maneras distintas: histórico de valores desde que la conexión se inició, histórico de valores desde la última vez que se consultó la información, y el valor de las variables en el momento en el que se realiza la consulta. Esto puede servir como retroalimentación a los sistemas de control de congestión desarrollados utilizando la librería.

Usando este entorno, los propios desarrolladores de la librería ofrecen otras implementaciones de control de congestión, como es por ejemplo el control de congestión que usa TCP. Hay que tener en cuenta que no se puede definir directamente un control de la tasa de transmisión basado en técnicas extremo-a-extremo sobre este framework, pero sí es cierto que sirve de gran ayuda para el desarrollo.

4.2.2.- Diseño básico

La selección del diseño básico para el proyecto parte de dos pilares básicos: la estructura de la librería UDT que usará como esqueleto y la implementación existente de codificador y decodificador de Raptor Codes de Digital Fountain, que se verá integrada en este esqueleto.

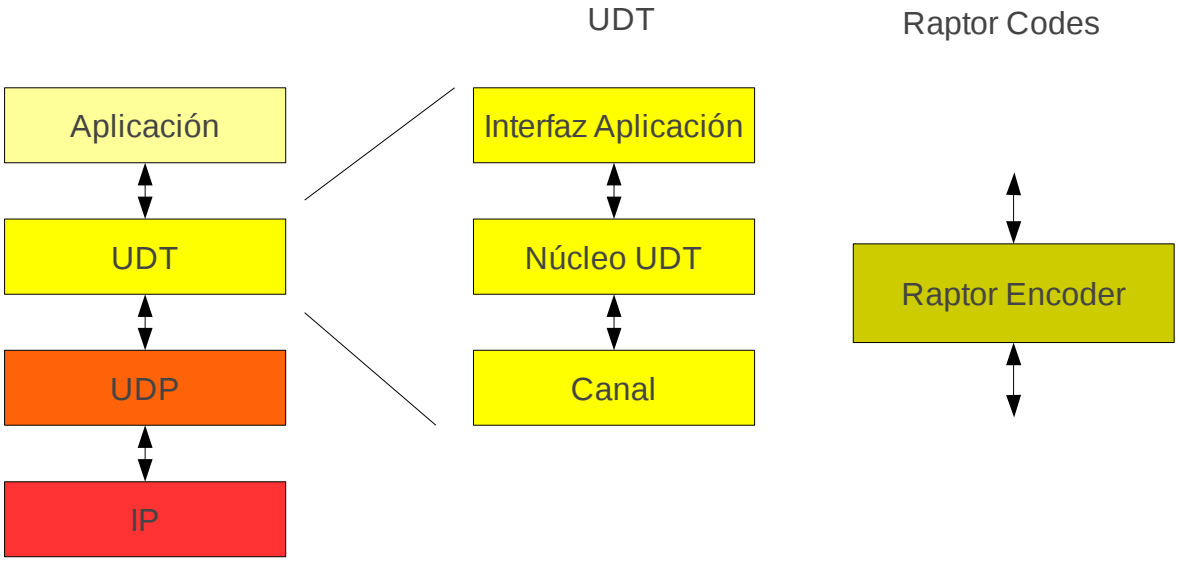

Figura 16: Punto de partida para la implementación de la librería

Esta integración supone incluir una nueva capa en la estructura de la librería UDT. De manera simplificada, se puede ver la librería UDT como un sistema de tres capas. La primera capa es una capa sencilla de interfaz con la aplicación. Esta capa le proporciona a la aplicación las funciones de sockets típicas de toda librería de comunicaciones. Además le permite a la aplicación hacer uso de la funcionalidad descrita anteriormente como *Composable CC*, para el control de congestión.

La segunda capa comprende el núcleo de la librería UDT, que gestiona todo el comportamiento interno de la librería. Esta capa se encarga de proporcionar a la aplicación la funcionalidad que promete la librería.

La tercera capa es una abstracción del canal que usa la librería para la comunicación entre el origen y el destino mediante el uso de UDT, que es el elemento que se encuentra inmediatamente por debajo de esta capa.

Esta nueva capa de codificación Raptor se sitúa entre la primera y la segunda capas. En la parte del emisor se incorporará la etapa de codificación Raptor, mientras que en la parte del receptor se incorporará la etapa de decodificación Raptor. La creación de esta nueva etapa de codificación y decodificación conlleva la creación de nuevas estructuras de datos y control, así como la modificación de las existentes, que se realizará de la manera más coherente posible con las estructuras de datos ya incluidas en la librería UDT.

Para que estas dos nuevas capas se coordinen es necesario que se intercambie información adicional entre los nodos emisor y receptor. Esto conlleva la incorporación de datos no propios del código Raptor enviado, pero necesarios para la recuperación de la información. Además, dado que se pueden perder paquetes en la transmisión, hay información que solo debería enviarse una vez,

pero que, para aumentar la robustez del sistema, debe incluirse en todos los paquetes enviados, provocando redundancia de información. Ejemplos de este tipo de información son el identificador del bloque o el identificador de símbolo.

Otro elemento de coordinación entre las capas es el cambio de bloque para codificar y decodificar. Para que ambas capas se encuentren trabajando sobre el mismo bloque es necesario que, cada vez que el receptor consiga decodificar un bloque y cambie de bloque, informe al emisor de que también debe cambiar de bloque. Para ello debe añadirse a la librería un nuevo paquete de control que cumpla esta función: que informe en cada momento de cual es el último bloque que consiguió decodificar con éxito. Si el emisor todavía está generando símbolos de ese bloque deberá avanzar al siguiente.

También es necesario mantener unas estadísticas sobre el ancho de banda del canal y los paquetes recibidos por el receptor. Esto conlleva el incluir un nuevo tipo de paquetes de control, el segundo si recordamos el que se incluyó para el cambio de bloque, y que se enviará de forma periódica informando, en este caso, del ancho de banda del canal, el número de paquetes que se han recibido desde el envío del último paquete de control, y el último bloque que se decodificó con éxito. La importancia de este paquete se verá reflejada en el apartado en el que se introducirá exhaustivamente el control de tasa de transmisión de este proyecto, destacando cómo encaja este paquete exactamente con la manera en que esta librería envía los paquetes de datos.

Como el control de flujo de la librería UDT es sustituido completamente por el control de tasa de transmisión de la librería RDT, algunos de los paquetes de control que antes se utilizaban en la librería UDT deben ser eliminados en la nueva librería RDT. Así evitaremos que, con el envío de estos paquetes, se consuma ancho de banda en la red que realmente no se necesita para nada, con la consiguiente degradación de prestaciones que esto pueda ocasionar.

Los paquetes de control que quedan sin utilidad son: (i) el ACK, ya que ahora no es necesario que todos los paquetes lleguen al destino, y, por lo tanto, ya no son necesarios reconocimientos ni retransmisiones, sino que se recuperará la información gracias a los paquetes extra enviados y a los códigos Raptor; (ii) los ACK2, que son ACK usados para reconocer los ACK y que, por tanto, al eliminar los paquetes de ACK ya no serán necesarios; y (iii) los informes de pérdidas o NACK que, por el mismo motivo que los ACK, tampoco serán necesarios.

Por último, para implementar el sistema de control de la tasa de transmisión por parte de la librería RDT es imprescindible crear una nueva clase heredada de la clase CC de la librería UDT original. Así se consigue mantener lo más posible el diseño original de la librería UDT. Sin embargo, no será suficiente la creación de esta nueva clase. También habrá que modificar otros elementos internos de la librería UDT para integrar el nuevo control de flujo de la librería RDT dentro de ella.

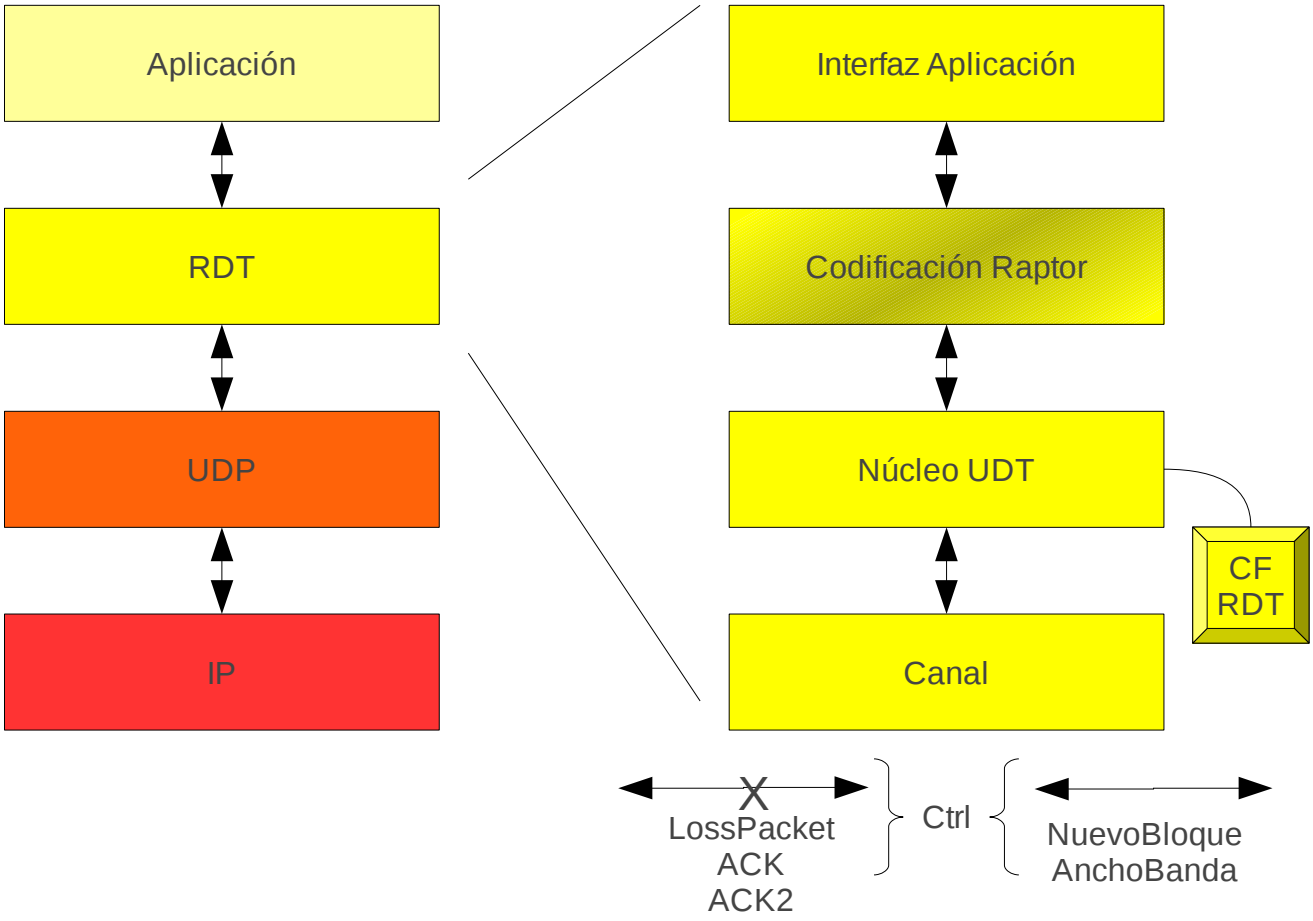

Figura 17: Diseño final de la librería RDT

4.2.3.- Diseño avanzado

Sobre este diseño inicial se han realizado una serie de mejoras básicas que optimizan el procesamiento de la información a todos los niveles. Además, se pretende aprovechar las capacidades multiprocesador de los sistemas de uso común en un proceso tan intensivo en cómputo y en uso de memoria como es un proceso de codificación, por lo que se ha realizado una evolución del diseño básico inicial hacia un sistema multihilo.

El proceso de codificación Raptor es muy costoso, y hace necesario un procesado previo de la información, al que hemos hecho mención bajo el nombre de primera etapa, antes de poder obtener la primera serie de símbolos de recuperación. Si el proceso de envío de datos se detiene durante ese momento, estaríamos introduciendo un retardo en el envío de la información innecesario.

Para evitar este inconveniente se ha optado por el rediseño de la capa de codificación Raptor de manera que la generación de códigos de recuperación de un bloque quede solapada con la inicialización de los datos del bloque siguiente. Así, durante el proceso de codificación, en lugar de detener el envío de paquetes por la red, se puede realizar esta tarea en un hilo independiente y tener un bloque preparado para iniciar su codificación en cuanto el bloque anterior sea reconocido.

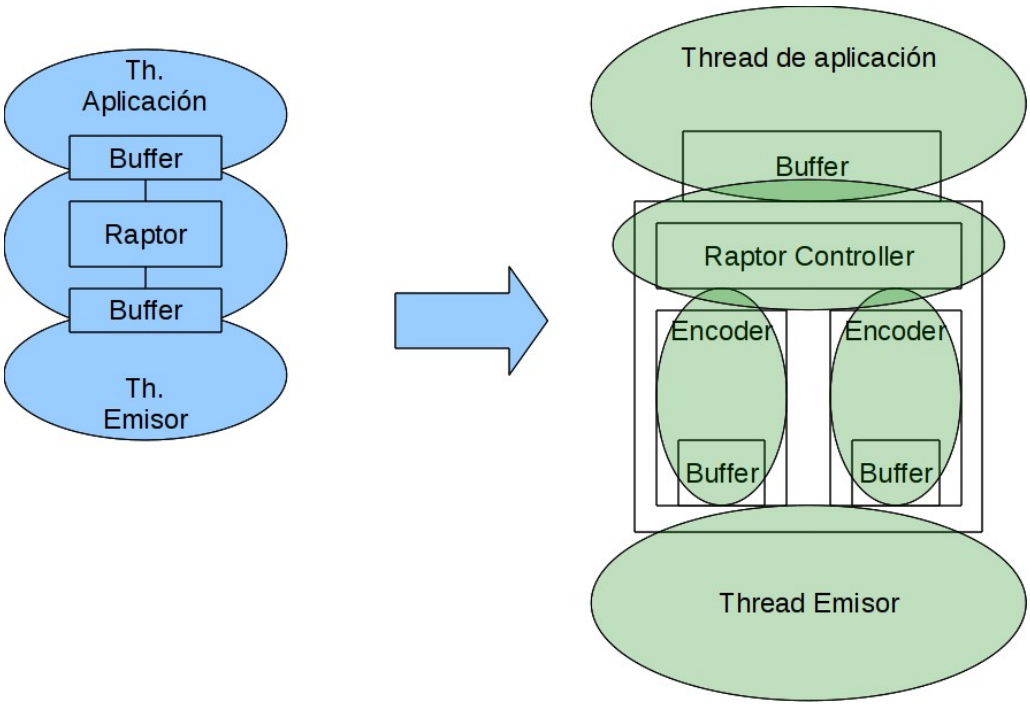

Figura 18: Cambio de diseño realizado sobre la capa de codificación Raptor

Como guía para la comprensión de la figura, señalar que las elipses representan hilos (threads), mientras que las cajas son estructuras estáticas que se corresponden con objetos independientes en la implementación.

El nuevo diseño de la capa permite mantener varios procesos de codificación en paralelo, en la figura se señalan como encoders, y que el controlador Raptor es capaz de renovar conforme los bloques fuente se van reconociendo por el receptor. Además, el controlador Raptor es capaz de decidir de qué proceso de codificación debe seleccionar el símbolo a enviar, proporcionando una respuesta inmediata al evento de reconocimiento del bloque y una mayor flexibilidad para la implementación de algoritmos de envío de símbolos más elaborados.

Dada la necesaria realización de esta modificación, se aprovechó para cambiar la integración que se había hecho de la capa dentro del esqueleto UDT. Así se consiguió una mejor simbiosis de la nueva estructura dentro de la antigua, eliminando la gestión centralizada que se hacía de los recursos por un mejor encapsulamiento de los procesos que se llevan a cabo en la capa de codificación.

A partir de ahí, se experimentó con una serie de mejoras en diversos puntos de la aplicación, intentando comprobar si el incremento de productividad era significativo en cuanto a las versiones previas. En la figura se presenta el grafo de mejoras introducidas durante esta fase de experimentación. En él se toma como prototipo 0 el diseño básico que se ha descrito en el apartado anterior y se van numerando los prototipos según se van introduciendo mejoras. Con los arcos se representan los cambios aplicados sobre cada prototipo y aquellas rutas que no conducen a ninguna parte se desestimaron por revelarse inútiles, insuficientes o insatisfactorias.

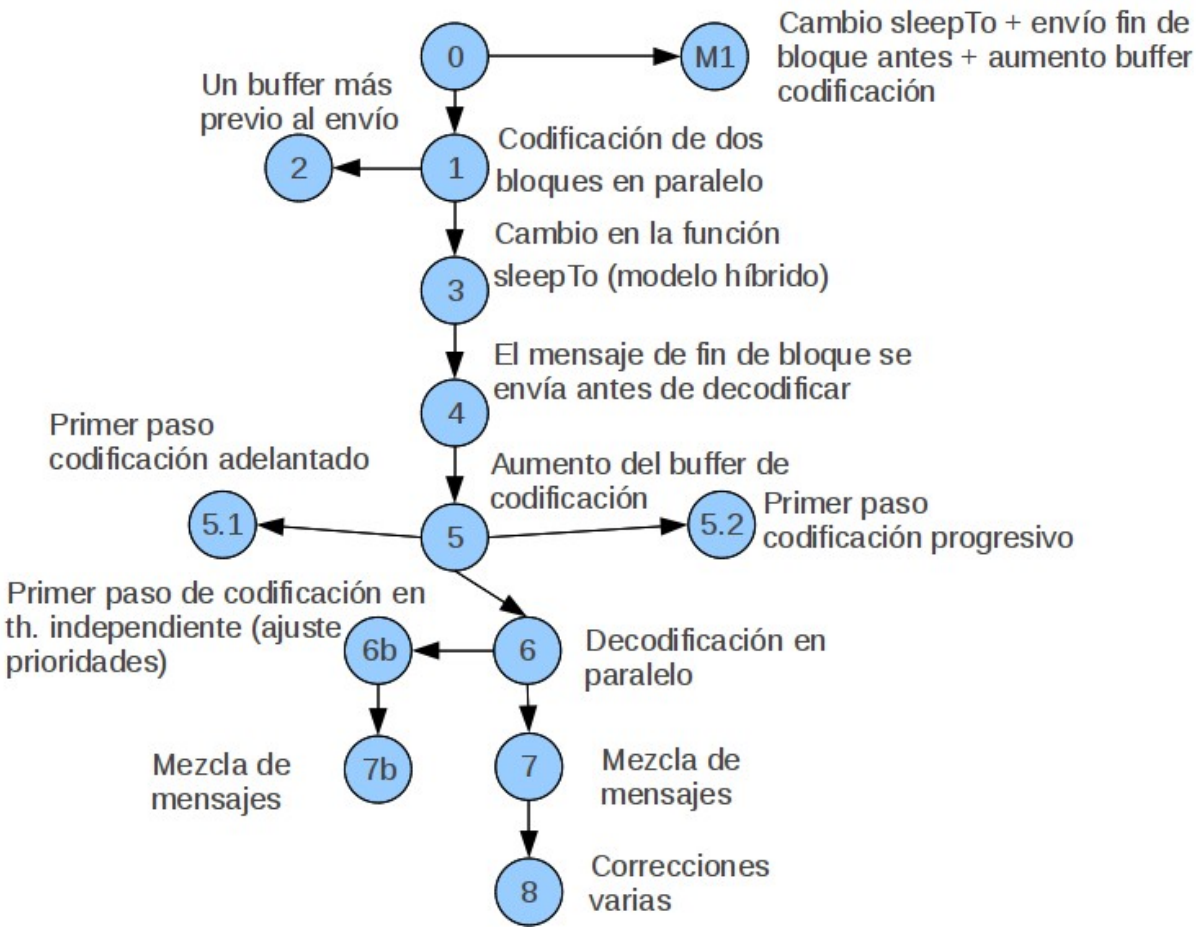

Figura 19: Grafo de mejoras

De todo el árbol de modificaciones se van a comentar en más detalle las modificaciones las más importantes. El prototipo 1 es el comentado en párrafos anteriores, que incluye la codificación en paralelo de dos elementos de codificación. El prototipo 2 incluye un buffer adicional a la estructura, como se puede ver en la siguiente mejora, para intentar regularizar lo más posible la generación de símbolos de los bloques.

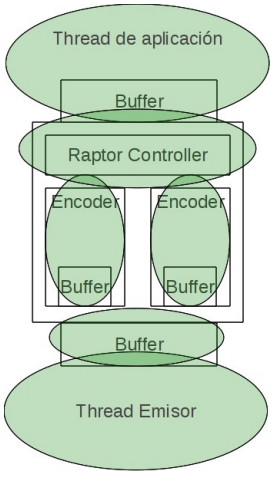

Figura 20: Diseño del prototipo 2

El prototipo 3 cambia la función de espera entre paquetes de un modelo de espera activa por un modelo híbrido que realiza la primera parte de la espera en un semáforo y la segunda parte como espera activa. Así se puede reaccionar de una manera más inmediata cuando pasa el tiempo en el que se está esperando, pudiéndonos evitar la sobrecarga impuesta por la función de espera en semáforo y evitando también un consumo excesivo de CPU.

El prototipo 4 hace un cambio crucial en el decodificador. En versiones previas la confirmación de que el bloque se había recuperado correctamente se enviaba una vez había terminado el proceso de decodificación completamente. Ahora se hace uso de una utilidad de la librería que permite identificar cuando se han recuperado suficientes símbolos para proceder a la decodificación. Esto permite enviar el paquete de reconocimiento antes, permitiendo una mayor eficiencia en el envío.

El conjunto de prototipos 5.X se implementó con el objetivo de evaluar el impacto del proceso de codificación sobre la productividad total de la aplicación y estudiar las diferentes posibilidades para enmascarar el proceso completo de codificación. El prototipo 5 (o 5.0) simplemente gestionaba la codificación aumentando el *buffer* interno de cada ítem de codificación, de manera que durante la primera pasada de la codificación hubiera suficientes paquetes almacenados como para no interrumpir el flujo de paquetes. El prototipo 5.1 mueve la primera fase de codificación al periodo inicial, junto a la inicialización, para aprovechar las ventajas desarrolladas para el prototipo 1. El prototipo 5.2 hace uso de la codificación progresiva que ofrece la librería, intercalando el envío de símbolos fuente con pequeños pasos en la codificación, de manera que, cuando se acaban los símbolos fuente, los símbolos de recuperación ya están listos para su generación.

A partir del prototipo 5.0 pasamos al 6, que incluye un nuevo diseño en la decodificación. Ahora se permite la decodificación de varios bloques en paralelo. Esto abre la puerta al envío de varios símbolos de diferentes bloques de manera entrelazada, característica que se estudiará más adelante.

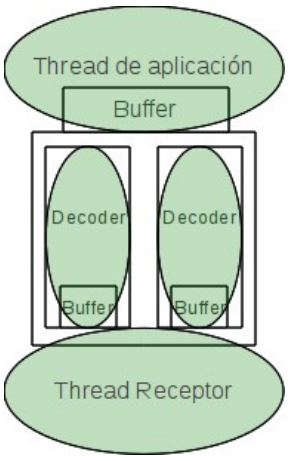

Figura 21: Diseño del prototipo 6

Desde el prototipo 6 las vías de acción se dividen en dos corrientes diferentes, ambas con las mismas modificaciones, pero con una diferencia. El proceso de codificación que queríamos enmascarar en el prototipo 5 se ejecutará en una de las vías como un hilo independiente, mientras que en la otra se ejecutará tal y como se definió para el prototipo 5.0.

En el prototipo 7 se implementará una variación del protocolo RDT. Hasta ahora no comenzaba el envío de los símbolos perteneciente a un bloque hasta que no se recibía confirmación de que el bloque anterior habría sido correctamente decodificado. Ahora se podrán enviar varios símbolos de bloques diferentes simultáneamente.

El prototipo 8 es igual al anterior, pero tras un exhaustivo proceso de testing que ha llevado a eliminar una serie de bugs, que si bien no afectaban al rendimiento ni al funcionamiento normal de la aplicación, en condiciones especiales podrían provocar un comportamiento inadecuado.

Por último, el prototipo M1 es un prototipo que implementa el subconjunto de mejoras que no introducen hilos adicionales. Esta versión se implementó como referencia para saber las ventajas que incluían estas mejoras sin hacer uso del nuevo diseño en paralelo y evitando, por tanto, la complejidad que esto supone.

También se propone otro conjunto de mejoras para mejorar la detección del ancho de banda disponible en el canal. Esto conllevó, por un lado, hacer uso de las utilidades proporcionadas por el sistema operativo y, por otro, la modificación de la gestión del temporizador.

Como la detección del ancho de banda disponible en el canal se realiza en base al tiempo transcurrido entre llegadas de los paquetes, es necesario conocer de la forma más precisa posible el momento exacto de la llegada de un paquete. Para obtener una mayor precisión se puede hacer uso de la opción del sistema *sockets SO_TIMESTAMP*, que indica el momento de llegada del paquete al *host* en cuanto es detectado por el sistema operativo.

El temporizador cumple la función de avisar lo antes posible del ancho de banda del canal estimado en base a la ráfaga. Como durante el proceso de envío de puede perder algún paquete, puede darse el caso de que no podamos detectar la pérdida hasta que no llegue el paquete de la siguiente ráfaga. Para evitar esto, el temporizador detecta el periodo sin envío de paquetes entre ráfagas, y envía en ese momento el paquete de control correspondiente.

Para conseguir una mejor detección de ese periodo entre ráfagas, se modificó la manera en que se calculaba el tiempo del temporizador. En versiones previas, el temporizador se calculaba como dos veces el tiempo medio entre paquetes detectado hasta el momento. El problema de esta aproximación es que podría establecerse un periodo muy pequeño para el temporizador cuando se realizaba el cálculo para una cantidad muy pequeña de paquetes recibidos. Por eso se modificó el cálculo a dos veces el tiempo medio entre paquetes detectado a partir de que ha transcurrido la primera mitad de ráfaga.

Por último, también con el objetivo de mejorar la precisión en la detección del ancho de banda del canal, se ha implementado una serie de filtros que han sido optimizados para conseguir mejorar el rendimiento global. El objetivo de estos filtros es, en base a la información que se está recibiendo del receptor sobre el ancho de banda, deducir cual es el ancho de banda real del canal. Para lograrlo se suaviza la señal recibida eliminando la máxima cantidad de ruido que haya podido haber en las mediciones. Los filtros con los que se ha experimentado pertenecen a dos familias: filtros de mediana y filtro exponencial, también conocido como pasa-bajo.

Los filtros de mediana calculan el ancho de banda del canal (C_n) como la mediana de las k estimaciones previas (E_n) recibidas del cliente. En síntesis, $C_n(k) = mediana(E_n, E_{(n-1)}, \dots, E_{(n-k-1)})$. Este filtro intenta eliminar del cálculo del ancho de

banda las mediciones incorrectas que pueden surgir de manera puntual durante el proceso de estimación.

Los filtros exponenciales calculan el ancho de banda del canal (C_n) como una combinación aritmética de la estimación recibida actualmente (E_n) en el canal y el valor previo de ancho de banda del canal que se había calculado para la estimación anterior, en base a un parámetro γ que puede tomar un valor entre 0 y 1. En síntesis, $C_n(\gamma)=\gamma \cdot E_n+(1-\gamma)\cdot C_{(n-1)}$. Este filtro está orientado a suavizar progresivamente el valor estimado para el canal, tomando en cuenta no solo la estimación actual sino estimaciones previas.

Tanto la implementación de todos estos prototipos como la pruebas que sobre ellos se han realizado y el rendimiento que se ha obtenido, será objeto de análisis en puntos sucesivos.

4.3.- El control de la tasa de transmisión

La librería RDT implementa un control de la tasa de transmisión basado en el ancho de banda disponible del canal. Se comienza por realizar una estimación en base a la recepción de paquetes, enviando los paquetes en ráfagas a una mayor velocidad de transmisión que la estimada en la recepción, pero comenzando la siguiente ráfaga cuando debería empezarse según la estimación realizada, o sea, respectando esa estimación.

El emisor recibe informes periódicos del receptor con el ancho de banda que ha estimado según la frecuencia de llegada de paquetes. Con este ancho de banda el emisor calcula la tasa de transmisión objetivo aplicándole a este valor un coeficiente beta que varía entre 0 y 1.

Una vez ha estimada la tasa de transmisión objetivo, se calcula una tasa de transmisión real más optimista a la que se enviará la siguiente ráfaga de paquetes. El cálculo de esta nueva tasa se calcula dividiendo la tasa objetivo por otro coeficiente beta que también puede adquirir un valor entre 0 y 1.

Una vez se ha enviado la ráfaga de paquetes, la siguiente no comenzará hasta el momento en el que debería de haber empezado si se hubiera usado la tasa de transferencia objetivo.

En síntesis, en un instante t_i:

- C_i: Capacidad del canal estimada en la recepción
- R_i: Tasa de datos estimada para la ráfaga
- Ω_i: Tasa de datos optimista de la ráfaga

  - $R_i = \beta \times C_i$, $0 < \beta < 1$
  - $\Omega_i = 1/\alpha \times R_i$, $0 < \alpha < 1$

Siendo Ω_i la tasa de transmisión de los paquetes y cumpliendo para cada instante que:

- $t_{i+1} = t_i + \text{tamaño\_rafaga} \times 1/R_i$

tendremos, para el envío:

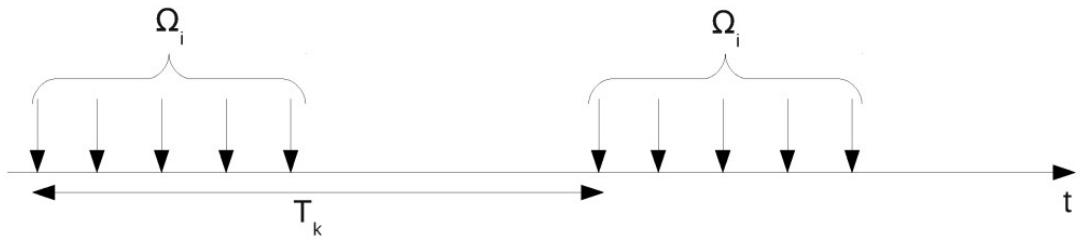

Figura 22: Envío

y para la recepción básicamente dos posibilidades:

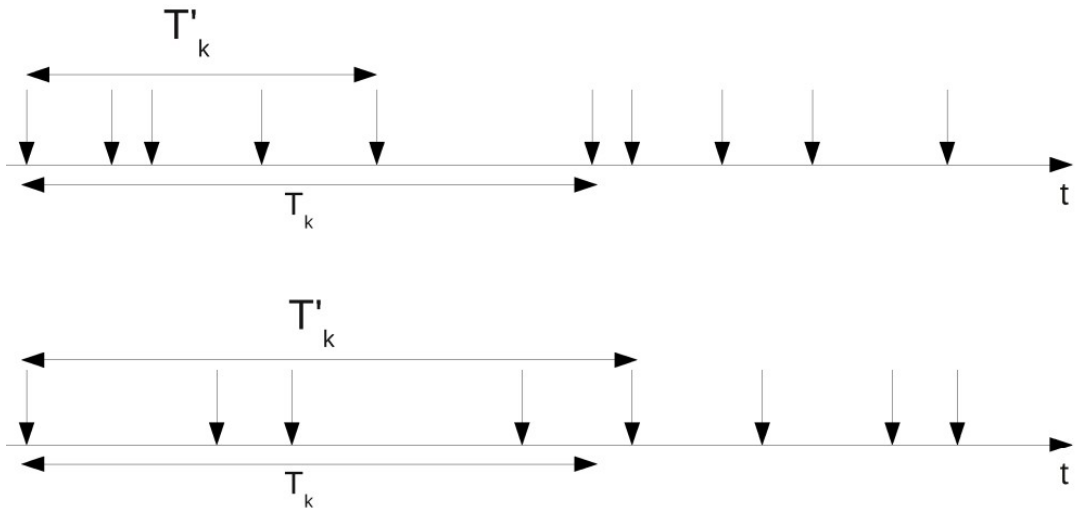

Figura 23: Recepción

En la primer ejemplo de la figura se verifica que el tiempo necesario para recibir toda la ráfaga (T'_k) es inferior al previsto inicialmente (T_k). Esto significa que las condiciones del canal han mejorado respecto a lo previsto anteriormente.

En el segundo ejemplo de la figura se verifica que el tiempo necesario para recibir toda la ráfaga (T'_k) es superior al previsto inicialmente (T_k). Esto significa que las condiciones del canal han empeorado respecto a lo previsto anteriormente.

En ambos casos el transmisor procederá a actualizar su tasa de transmisión a un valor más cercano a la situación real experimentada.

4.4.- Funcionamiento

Hasta ahora se han descrito las modificaciones a realizar sobre la librería UDT y el sistema de control de la tasa de transmisión que implementa la librería RDT. Ahora se pasará a describir el funcionamiento de la librería.

El funcionamiento del proceso **emisor** es el siguiente:

1) Recibe la información a enviar y la divide en bloques
2) Envía paquetes que contienen un símbolo del bloque. Primero envía los símbolos fuente que conforman el bloque y luego envía los símbolos de recuperación.
   a) Si quedan datos que enviar, se prepara y se envía una ráfaga de paquetes a la velocidad optimista de acorde al ancho de banda estimado por el receptor y enviado al emisor en ráfagas anteriores.
   b) Se va actualizando el ancho de banda con la información del receptor recibida por el emisor mediante los paquetes de control apropiados
   c) Se vuelve al inicio del paso 2.
3) Cuando se recibe el paquete de control de cambio de bloque se pasa al bloque siguiente y se va nuevamente al paso 2.
4) Si no queda información a enviar se detiene el proceso.

En el funcionamiento del proceso **receptor** es el siguiente:

1) Recibe paquetes con símbolos para la recuperación de un bloque y la información que contiene.
   - Estos símbolos pueden ser originales o de recuperación
   - Los símbolos se almacenan temporalmente en memoria
2) Cada vez que recibe todos los paquetes de una ráfaga, crea un paquete de control con el ancho de banda que ha estimado.
   - La estimación se realiza en base a los paquetes recibidos de la misma ráfaga.
   - El paquete se envía al emisor.
3) Cuando tiene suficientes paquetes con símbolos se recupera el bloque recibido
   - Pasa el bloque recibido a capas superiores.
   - Crea un paquete de cambio de bloque y lo envía al emisor para que avance al siguiente bloque.
4) Pasa a esperar más símbolos de otros bloques (paso 1).

4.5.- Funcionamiento alternativo

Se propone también un funcionamiento alternativo al protocolo para las versiones que utilizan un codificador y un decodificador en paralelo. La motivación de esta variante es, en canales con un retardo muy alto pero con una tasa de error baja, intentar reducir una posible infrautilización efectiva del canal en el periodo entre que se han enviado todos los símbolos de un bloque y se recibe el reconocimiento por parte del receptor.

El funcionamiento de esta nueva versión es muy similar a la anterior. Básicamente se sigue el mismo algoritmo que en el caso previo, pero para la conformación de las ráfagas y para conocer el número de bloques que se están enviando simultáneamente se sigue el siguiente diagrama de estados en el emisor:

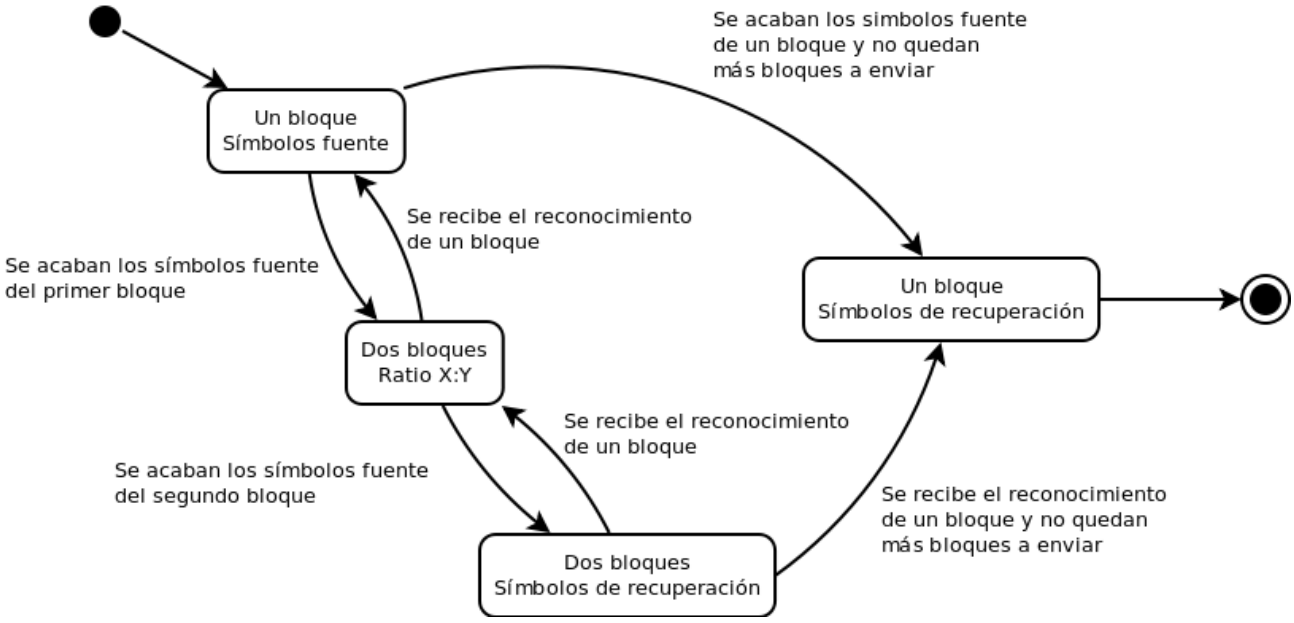

Figura 24: Diagrama de estados del emisor en el proceso de envío

Como leyenda a la figura, señalar que, en cada estado, la primera línea indica el número de bloques de los cuales se están enviando símbolos, y la segunda la conformación de la ráfaga. Esta puede ser: a) símbolos fuente de un solo bloque, b) símbolos de recuperación del primero de los bloques (o del único que haya), o c) ráfagas formadas conforme al ratio X:Y, que incluyen X símbolos de recuperación de un bloque e Y-X símbolos fuente del otro bloque por cada Y paquetes.

5.- Detalles de implementación

La implementación de esta librería se ha realizado en C++, ya que se iba a partir de herramientas y librerías escritas en C y C++. El desarrollo se ha realizado sobre un sistema operativo GNU/Linux Xubuntu 10.04 de 64 bits con el apoyo del entorno gráfico de desarrollo Eclipse IDE for C/C++ developers en su versión Galileo. Se ha compilado usando el compilador gcc de GNU a una librería de 32 bits.

5.1.- Codificación y decodificación Raptor

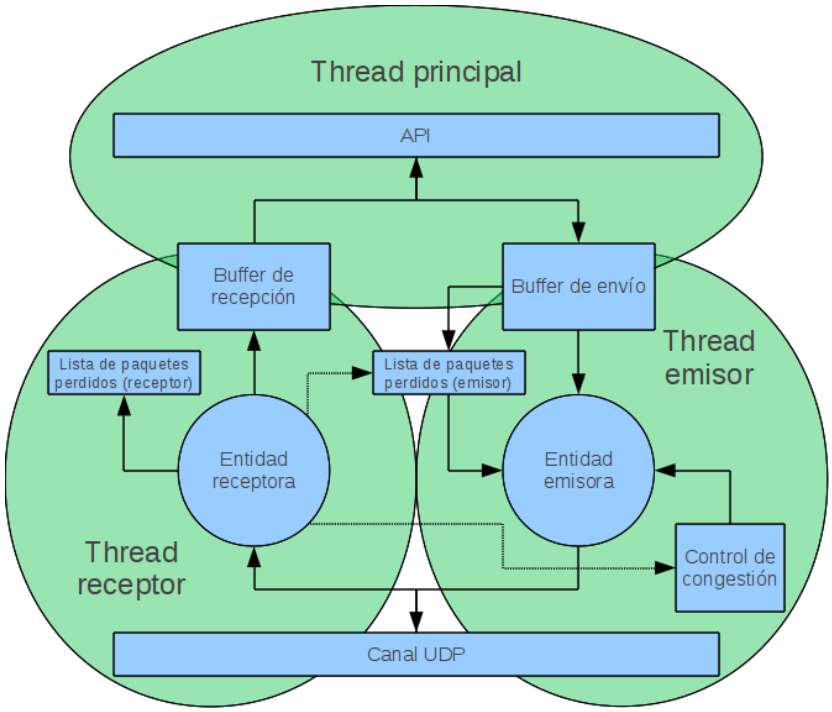

Figura 25: Diagrama de bloques de la librería UDT, incluyendo hilos

Volviendo al esquema donde se mostraban las entidades que conforman la librería UDT, se comentará que hay principalmente tres hilos que entran en juego a la hora de enviar un mensaje. El primero de ellos es el hilo principal de la aplicación construida sobre la librería, que invoca a las funciones que conforman el API UDT y RDT. El segundo y el tercero son el hilo de emisión y el hilo de recepción de información. Cada uno de esos hilos se encarga de recoger la información que le proporciona el primer hilo y enviarla, así como de leer la información del canal y pasarla al primer hilo, respectivamente.

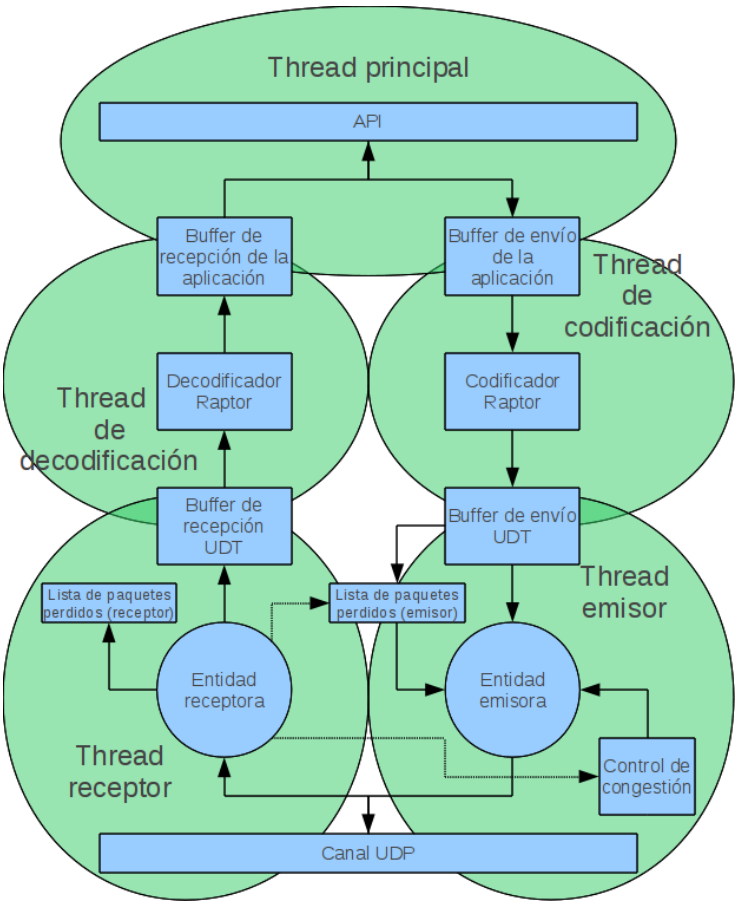

Figura 26: Diagrama de bloques de la librería RDT

Para incluir la nueva etapa de codificación y de codificación deberán de iniciarse dos nuevos hilos en la librería, un hilo de codificación Raptor y otro hilo de decodificación Raptor. En el caso del hilo de codificación, éste se encargará de leer del *buffer* donde el hilo de la aplicación escribe los datos a transmitir, y que en la librería UDT original leía el hilo de emisión. Este hilo generará los símbolos con el tamaño y la cabecera adecuados para su envío y los deposita en otro buffer para que el hilo de emisión los envíe al destino.

La implementación de este hilo se ha realizado a imagen y semejanza de los hilos de Java, creándose una clase que también es un hilo y que cuenta con un método run y otro método start: el primero contiene el código que ejecutará el hilo independiente, y el segundo es el método encargado de lanzar el hilo a ejecución.

```
class RaptorEncoderThread {
private:
	CSndBuffer* envioAplicacion;
	CSndBuffer* envioUDT;
	CUDT * udt;
	int tCarga;
	int symbolSize;
	int idBloque;
	bool finEnvio;
	bool closing;
	bool bypass;
```

```
public:
      RaptorEncoderThread(CSndBuffer* envioAplicacion,
                          CSndBuffer* envioUDT,
                          CUDT * libreria,
                          int tCarga,
                          bool usaRaptor);
      virtual ~RaptorEncoderThread();
      virtual void start();
      virtual void run();
      virtual void nextBlock(int id);
      virtual void nil();
      virtual void close();
};
```

Con el uso de este sistema se obtienen todas las ventajas de diseñol de hilos de java, lográndose un gran nivel de encapsulación. Esto permite una depuración mucho más exhaustiva de las funciones de codificación y decodificación Raptor, logrando que estas funciones sean mucho más robustas.

Esta clase cuenta también con un método, el método *nil*, que simplemente lee la información contenida en el *buffer* de aplicación y la copia al *buffer* UDT. Este método es el que se ejecuta en caso de que se elija el modo de operación UDT para la librería RDT. Este modo, como ya se ha comentado, hace que la librería RDT se comporte como la librería UDT y, por tanto, no es necesario realizar ningún procesado en esta etapa.

El inicio del hilo se lleva a cabo con el método start, que invoca a la librería *pthread* para crear el nuevo hilo.

```
pthread_create(&thread1, NULL, (void*(*)(void*)) lanzadorEncoder, this);
```

El propósito de la función auxiliar *lanzadorEncoder* es simplemente invocar el método *run* del objeto *RaptorEncoderThread* que se le pasa como parámetro.

```
void lanzadorEncoder(RaptorEncoderThread * param){
      RaptorEncoderThread* re = param;
      re->run();
}
```

Este método de lanzamiento del hilo difiere del utilizado por la librería UDT. En la librería UDT se usa un método estático con el código a ejecutar por el hilo, en lugar de usar un método del objeto que posee el nuevo hilo. Esto le ocasiona el tener que acceder a los atributos y métodos del objeto usando un puntero que le pasa como parámetro al método y que almacena en la variable *self*. Con el método que se implementa en la librería UDT podemos acceder a los atributos del objeto directamente sin tener que cambiar los modificadores a los atributos o crear métodos *get* y *set* específicos.

El método *run* de la clase contiene un ejemplo clásico de codificación de información usando códigos Raptor. Si bien no se ha seguido al pie de la letra el guión propuesto por la documentación de la librería, se ha implementado una versión muy clara de codificación que se comentará de aquí en adelante.

```
void RaptorEncoderThread::run(){

	//Resultado de las operaciones de RAPTOR
	DFR11EncError result;
	//Estructura para manejar la memoria de RAPTOR
	DFR11EncMemRequirement memReq;
	//"Objeto" codificador raptor
	RaptorEncoder *encoder;

	char ** c = new char*[1];
	int32_t aux = 0;
```

Hasta ahora solo se han declarado parte de las variables que serán usadas durante todo el proceso de codificación. Ahora se entrará en un bucle infinito que leerá la información a enviar y escribirá la información codificada en el buffer correspondiente. De este buffer solo se saldrá cuando se inicie el proceso de cerrado, gestionado por el atributo booleano closing.

```
	while(true){
```

En este primer paso se procede a la lectura de los datos proporcionados por la capa superior. Dado que el *buffer* puede estar vacío, se debe entrar en un periodo de espera hasta que se introduzca información en el *buffer*. Este método de espera se ha implementado siguiendo las líneas marcadas en la librería UDT. Esto ha implicado que los semáforos de espera se almacenen en el núcleo de la librería. La librería proporciona con su método *waitAppBuffer* un servicio para permitir al hilo esperar a que el *buffer* se llene, y poder así continuar la codificación.

```
int longEnvio = envioAplicacion->readData((char**) c, aux);
while(longEnvio == 0){
	udt->waitAppBuffer();
	if(closing) break;
	longEnvio = envioAplicacion->readData((char**) c, aux);
}
if(closing)break;
unsigned char* datos = (unsigned char*)(*c);
```

En el segundo paso se procede a calcular el número de símbolos fuente de los que disponemos. Dado que conocemos el tamaño de símbolo, introducido como parámetro en el constructor del hilo, así como la longitud del envío, que acabamos de obtener al leer la información del *buffer*, solo tenemos que realizar el módulo entre estos valores para conocer el número de símbolos que tenemos que enviar.

```
int nSourceSymbols = 0;
if(longEnvio % symbolSize == 0){
	nSourceSymbols = longEnvio / symbolSize;
} else {
	nSourceSymbols = (longEnvio / symbolSize) + 1;
}
```

En el tercer paso, utilizando el número de símbolos que hemos calculado en el paso anterior, consultamos a la librería cuales serán los requisitos de memoria que necesitaremos para llevar a cabo la codificación. Esta información queda almacenada en la estructura de tipo *DFR11EncMemRequirement*, cuyos campos nos revelan valores como el tamaño en bytes necesario para el bloque fuente, para el bloque intermedio, para la memoria de trabajo del codificador y el valor necesario para alinear los bloques en memoria. Además, la memoria utilizada en el proceso cambia según el modo de ejecución utilizado por la librería de codificación. El modo de operación

usado en este caso el el modo estándar y queda encapsulado por la constante CONFIGURACION_ENC_RAPTOR.

```
result = DFR11EncMemRequest(nSourceSymbols,
                            symbolSize,
                            CONFIGURACION_ENC_RAPTOR,
                            &memReq);
```

En el cuarto paso se reserva memoria para el codificador, se inicializa, se resetea y se prepara para codificar símbolos.

```
encoder = (RaptorEncoder *) malloc (memReq.encoderSize);
result = DFR11EncInit(encoder, CONFIGURACION_ENC_RAPTOR);
result = DFR11EncReset (encoder);
result = (DFR11EncError) DFR11EncPrepare (encoder,
                                          nSourceSymbols,
                                          symbolSize,
                                          TAREA_ENC_RAPTOR);
```

En el quinto paso se reserva memoria para los bloques fuente e intermedio. Punteros a esta memoria son pasados al codificador, dejándolo preparado para la codificación.

```
void * srcBlock = NULL;
posix_memalign(&srcBlock, memReq.alignment,
                          memReq.srcBlockSize);
void * intermBlock = NULL;
posix_memalign(&intermBlock, memReq.alignment,
                             memReq.intermBlockSize);
result = DFR11EncInitSrcBlock(encoder,srcBlock,intermBlock);
```

Dado que los primeros símbolos son los símbolos fuente que conforman el bloque, deben pasarse tal cual al núcleo de la librería. Para ello creamos los paquetes formados por la cabecera Raptor y la cantidad suficiente de información para rellenar el paquete. La cabecera Raptor es una estructura formada por cuatro enteros seguidos que contienen el número de símbolos fuente que componen el bloque de información, el identificador del símbolo enviado, el identificador del bloque enviado y el tamaño exacto del bloque enviado. Tras crear los paquetes con los símbolos fuente, se copia la información al bloque fuente para su codificación. Esto constituye el sexto paso.

```
char * auxiliar = (char*) datos;
for(int id = 0; id < nSourceSymbols;id++){
      char * paquete = new char[tCarga];
      memcpy(paquete,
             generarCabecera(nSourceSymbols,
                             id,
                             idBloque,
                             longEnvio),
             CABECERA);
      paquete = paquete + CABECERA;
      memcpy(paquete, auxiliar, symbolSize);
      envioUDT->addBuffer(paquete - CABECERA, tCarga);
      delete [] (paquete - CABECERA);
      auxiliar = auxiliar + symbolSize;
}
memcpy(srcBlock, datos, nSourceSymbols * symbolSize);
```

En el séptimo paso generamos el bloque intermedio. Esta fase coincide con la fase de preprocesado de los bloques fuente en la codificación Raptor. La librería permite la realización de la

tarea completamente por partes. En este caso con la constante TAREA_ENC_RAPTOR se indica que se realizará completamente en un solo paso.

```
result = (DFR11EncError) DFR11EncGenIntermediateBlock
                                    (encoder, TAREA_ENC_RAPTOR);
```

En el octavo paso, tras reservar espacio para los símbolos de recuperación, se entra en un bucle en el que se van generando los símbolos intermedios y copiando al buffer de salida. Este bucle solo se detendrá cuando el núcleo de la librería informe al objeto de que debe pasar al siguiente bloque. Para ello se usará el método nextBlock del objeto que se detallará más adelante.

```
int id = nSourceSymbols;
void * repairSymbol = NULL;
posix_memalign(&repairSymbol, memReq.alignment, symbolSize);
finEnvio = false;
while (!finEnvio){
            result = DFR11EncGenRepairSymbols(encoder,
                                              id,
                                              1,
                                              repairSymbol);
```

El símbolo de recuperación se empaqueta siguiendo el mismo método utilizado para la creación de los paquetes de los símbolos fuente.

```
char * paquete = new char[tCarga];
memcpy(paquete,
       generarCabecera(nSourceSymbols,
                       id,
                       idBloque,
                       longEnvio),
       CABECERA);
paquete = paquete + CABECERA;
memcpy(paquete, repairSymbol, symbolSize);
envioUDT->addBuffer(paquete - CABECERA, tCarga);
delete [] (paquete – CABECERA);
```

Dado que la generación de símbolos es un proceso infinito que saturaría el procesador y la memoria del emisor, no se puede dejar que se generen todos los símbolos posibles seguidos. Se debe introducir un retardo entre la generación de los símbolos de recuperación. Para ello hacemos uso de un método de la librería encargado de proporcionarnos ese retardo.

```
udt->waitPacketGeneration();
```

Si hubiera una gran cantidad de problemas en el canal que produjera la pérdida de una gran cantidad de símbolos de recuperación, y teniendo en cuenta que de forma práctica no se pueden generar símbolos de recuperación infinitos, para bloques muy grandes podría darse el caso de que los bloques de recuperación se acabaran. Para evitar esa eventualidad, el incremento del identificador de bloques fuente se hace a imagen de los apuntadores circulares, para retomar la cuenta desde el principio si fuera necesario. Este es un caso extremo con una probabilidad de aparición muy baja, pero aún así se ha contemplado.

```
            id = (id + 1) % DFR11ENC_MAXIMUM_REPAIR_ID;
}
```

Una vez terminado con el bloque fuente que estábamos codificando, pasamos a informar de este hecho al *buffer* de la aplicación para que borre la información del bloque fuente que

almacenaba. Esta memoria no se puede liberar antes debido a que la librería usa el tamaño de los *buffers* para saber si se ha enviado toda la información y ya puede cerrar la conexión. Si el espacio se liberara antes podría darse el caso de que, mientras trabajamos en la codificación, la librería detectase que se han vaciado los *buffers* y procediera a cerrar la conexión antes de tiempo.

```
                envioAplicacion->ackData(1);
```

Como ya se ha avanzado al siguiente bloque, los símbolos de recuperación que se encuentran en el buffer de envío UDT ya no son necesarios para nada en el receptor. Por eso, se vacía el buffer.

```
                envioUDT->clear();
```

Además se libera la memoria que se había reservado para el codificador, el bloque fuente, los símbolos de recuperación y el bloque intermedio.

```
                free(encoder);
                free(srcBlock);
                free(repairSymbol);
                free(intermBlock);
```

Se avanza al siguiente bloque y se informa al núcleo de la librería de este hecho. La librería tendrá en cuenta esta información para la generación de las ráfagas de paquetes del control de flujo.

```
                idBloque++;
                udt->nextGroup();
        }
}
```

Gracias a este algoritmo conseguimos rellenar el *buffer* de envío UDT con los paquetes listos y preparados para su envío.

La generación de la cabecera de los paquetes se realiza reservando memoria y escribiendo en ella, como si de un vector de enteros se tratara, los cuatro enteros que la conforman.

```
char * generarCabecera(int nSourceSymbols,
                       int idSimbolo,
                       int idBloque,
                       int tBloque){
        char * cabecera = new char[CABECERA];
        int * auxiliar = (int *) cabecera;
        *auxiliar = nSourceSymbols;
        auxiliar++;
        *auxiliar = idSimbolo;
        auxiliar++;
        *auxiliar = idBloque;
        auxiliar++;
        *auxiliar = tBloque;
        return cabecera;
}
```

Una mejora obvia de este código sería liberar la memoria que se reserva para el vector de cabecera, pero como no se produce un impacto sobre el rendimiento de la librería, siguiendo la filosofía del segundo párrafo de este punto no se implementó esta mejora.

Otro punto importante de la clase RaptorEncoderThread es el método *nextBlock*. Hay que destacar de este método que no será ejecutado por el hilo que lanza la propia clase, sino por otro distinto. Este hecho ha sido tenido en cuenta en la implementación de este método.

```
void RaptorEncoderThread::nextBlock(int id){
      if(idBloque == id){
            finEnvio = true;
      }
}
```

De esta manera, el hilo que tiene que modificar la variable *finEnvio* sólo accederá a ella en caso exclusivo que lo necesite y sólo para modificarla. El acceso a la variable *finEnvio* podría haberse hecho en exclusión mutua. Sin embargo, dada la estructura del algoritmo, esta sobreprotección es innecesaria.

Otro aspecto importante es el método close. Este método se invoca cuando la librería se está cerrando, y permite detener de manera segura la ejecución de los hilos que se crean para la codificación y la decodificación.

```
void RaptorEncoderThread::close(){
      this->closing = true;
}
```

Como se puede ver, el método es muy sencillo: lo único que hace es modificar el valor de una variable booleana. Sin embargo, la importancia de tener este método de cerrado seguro de los hilos es crucial, ya que al cerrar la librería se destruyen estructuras de datos como los *buffers* que son utilizadas por el hilo. Si no se estableciera este método de finalización, al cerrar la librería el hilo intentaría acceder a posiciones de memoria que ya han sido liberadas, lo que se traduciría en una finalización inmediata del programa.

Ahora se pasará a la clase RaptorDecoderThread, que es la que implementa el hilo decodificador.

```
class RaptorDecoderThread {
private:
      SimplerBuffer* recepcionAplicacion;
      CRcvBuffer* recepcionUDT;
      CUDT * udt;
      int tCarga;
      int symbolSize;
      int idBloqueEsperado;
      bool closing;
      bool bypass;

public:
      RaptorDecoderThread(SimplerBuffer* recepcionAplicacion,
                          CRcvBuffer* recepcionUDT,
                          CUDT* libreria,
                          int tCarga, bool usaRaptor);
      virtual ~RaptorDecoderThread();
      virtual void start();
      virtual void run();
      virtual void nil();
      virtual void close();
};
```

La declaración de la clase es similar a la del codificador. La funcionalidad de los métodos son similares. También, al igual que en el hilo codificador, el método donde se implementa todo el proceso de decodificación es el método *start* que se va a comentar en detalle a continuación. Este algoritmo tampoco sigue el esquema definido en la documentación de Digital Fountain, pero realiza su cometido a la perfección.

```
void RaptorDecoderThread::run(){

      //Resultado de las operaciones de RAPTOR
      DFR11DecError result;
      //Estructura para manejar la memoria de RAPTOR
      DFR11DecMemRequirement memReq;
      //"Objeto" codificador raptor
      RaptorDecoder *decoder;

      //Variable para la lectura de Buffer
      char * c = new char[symbolSize];
```

Hasta ahora se han definido las variables de las estructuras que se van a usar en todo el proceso. Este parte ha sido similar a la homóloga del proceso de codificación. Ahora se procederá a entrar en un bucle infinito, también similar al del proceso de codificación.

```
      while(true){
```

Este bucle leerá del *buffer* de recepción UDT e irá guardando en memoria los paquetes recibidos con los símbolos. Cuando tenga suficientes decodificará el bloque y volverá una vez más al principio. Solo se terminará este bucle cuando el booleano *closing* pase a valer 1, es decir, cuando el núcleo de la librería invoque el método *close* del objeto codificador.

En el primer paso, tras inicializar las variables correspondientes, el hilo intenta leer del *buffer* de recepción. Se van leyendo por orden los valores que conforman la cabecera. Después lee el resto del paquete. El contenido de ese paquete, una vez retiradas las cabeceras, será el símbolo recibido del emisor. Tras leer el paquete, la condición de guarda del bucle comprueba dos cosas: la primera es que verdaderamente se haya conseguido leer algo del *buffer*, o sea, que no estuviera vacío. La segunda es que el símbolo recibido sirva para decodificar el bloque actual. Esta comprobación debe hacerse ya que el paquete de cambio de bloque puede tardar en llegar al emisor o incluso perderse. Si esto sucede, símbolos de recuperación para un bloque antiguo pueden estar llegando y por tanto deben ser descartados.

```
            int restante = symbolSize;
            int nSrcSymbols = 0;
            int id = 0;
            int idBloqueFuente = idBloqueEsperado;
            int longEnvio = 0;
            int aux = 0;
            while(aux == 0 || idBloqueFuente != idBloqueEsperado){
                  udt->waitUDTBuffer();
                  if(closing)break;
                  aux = recepcionUDT->readBuffer((char*) &nSrcSymbols,
                                                 sizeof(int));
                  aux = recepcionUDT->readBuffer((char*) &id,
                                                 sizeof(int));
                  aux = recepcionUDT->readBuffer((char*) &idBloqueFuente,
                                                 sizeof(int));
```

```
            aux = recepcionUDT->readBuffer((char*) &longEnvio,
                                           sizeof(int));
            aux = recepcionUDT->readBuffer((char*) c, restante);
        }
        if(closing)break;
```

En el segundo paso procedemos a la obtención de la información de la memoria necesaria para la decodificación. Los argumentos de la función son también iguales a los de la función de codificación, y sirven para el mismo propósito. Al igual que en el caso del proceso de codificación la llamada revela los valores del tamaño (en bytes) necesario para el bloque fuente, para el bloque intermedio, para la memoria de trabajo del codificador y el valor necesario para alinear los bloques en memoria. Además nos indica también el número máximo de símbolos que se pueden incorporar al proceso de decodificación. Para mejorar el entendimiento del código expuesto, destacar que muchas funciones y estructuras de la parte de codificación se repiten con una funcionalidad similar en la parte de decodificación. Su nombre solo difiere en que la sílaba “Enc” es sustituida por la sílaba “Dec”.

```
result = DFR11DecMemRequest(nSrcSymbols,
                            symbolSize,
                            DFR11_DEFAULT_TOTAL_SYMBOLS,
                            CONFIGURACION_DEC_RAPTOR,
                            &memReq);
```

En el paso número tres se reserva la memoria que usará internamente el decodificador. Después se inicializa y se hace un *reset*.

```
decoder = (RaptorDecoder *) malloc(memReq.decoderSize);
result = DFR11DecInit(decoder,
                      CONFIGURACION_DEC_RAPTOR,
                      memReq.nMaxTotalSymbols);
result = DFR11DecReset(decoder);
```

En el cuarto paso se inicializan las variables reservando la memoria para el bloque temporal y el bloque recibido. Después se entra en un bucle que irá leyendo los símbolos del *buffer* mientras no tenga suficientes para recuperar el bloque.

```
bool faltanSimbolos = true;
void * rcvBlock = NULL;
posix_memalign (&rcvBlock,
                memReq.alignment,
                memReq.rcvBlockSize);
void * tempBlock = NULL;
posix_memalign(&tempBlock,
               memReq.alignment,
               memReq.tempBlockSize);
char* auxptr = (char*) rcvBlock;
while(faltanSimbolos){
```

En el quinto paso, tras añadir el símbolo que se leyó al comienzo del proceso relativo al decodificador y actualizar los punteros y las posiciones de memoria adecuadamente, se intenta preparar el codificador para que inicie su tarea. Si esta operación devuelve un error, significa que el número de símbolos que se ha recibido hasta el momento es insuficiente y se debe esperar hasta recibir alguno más. Esta espera y posterior lectura se hace igual que la lectura inicial. Si, por el contrario, no se recibe ningún error en la llamada a la función, es que ya se han recibido símbolos suficientes y se saldrá del bucle.

```
                result = DFR11DecAddRcvSymbolIDs(decoder, id, 1);
                result = (DFR11DecError)DFR11DecPrepare(decoder,
                                                        nSrcSymbols,
                                                        symbolSize,
                                                        TAREA_DEC_RAPTOR);
                memcpy(auxptr, c, symbolSize);
                auxptr = auxptr + symbolSize;
                if(result < 0) {
                      int restante = symbolSize;
                      while(restante > 0 ||
                            idBloqueFuente != idBloqueEsperado){
                            udt->waitUDTBuffer();
                            recepcionUDT->readBuffer((char*)
                                              &nSrcSymbols, sizeof(int));
                            recepcionUDT->readBuffer((char*) &id,
                                                     sizeof(int));
                            recepcionUDT->readBuffer((char*)
                                           &idBloqueFuente, sizeof(int));
                            recepcionUDT->readBuffer((char*)
                                             &longEnvio, sizeof(int));
                            int aux = recepcionUDT->readBuffer(c,
                                                      symbolSize);
                            restante = restante - aux;
                      }
                } else {
                      faltanSimbolos = false;
                      idBloqueEsperado++;
                }
          }
```

En este sexto paso se inicializa el bloque recibido. La realización de esta invocación con éxito ya es posible debido a que ya se han recibido todos los símbolos necesarios en el paso anterior.

```
result = DFR11DecInitRcvBlock (decoder, rcvBlock, tempBlock);
```

En el séptimo paso se ejecuta la rutina de recuperación de la librería. Es ahora donde verdaderamente se recupera el bloque fuente, y es por ello el paso más costoso de realizar en cuanto a tiempo de ejecución.

```
result = (DFR11DecError)DFR11DecRecoverSource(decoder,
                                              TAREA_DEC_RAPTOR);
```

Una vez realizada la decodificación, el bloque decodificado se pasa a la capa de aplicación mediante el *buffer* de recepción de aplicación, quedando así a disposición del programa del nivel superior.

```
this->recepcionAplicacion->addData((char *)rcvBlock,
                                                longEnvio);
```

Después se manda al núcleo de la librería la orden de enviar el paquete de control que informará al emisor de que se debe proceder a codificar el siguiente bloque de información, en caso de que haya alguno.

```
udt->sendNextBlock(idBloqueFuente);
```

Por último, liberamos toda la memoria que hemos reservado durante el proceso para evitar

dejar porciones de memoria inaccesibles que degraden el rendimiento del sistema. Como esta memoria se usará el posteriores iteraciones del bucle no sería necesario reservarla y liberarla en cada iteración del hilo. Sin embargo, como el impacto sobre el rendimiento es mínimo, se ha dejado esta versión de la aplicación.

```
                free(decoder);
                free(rcvBlock);
                free(tempBlock);
        }
}
```

Gracias a este algoritmo, la información contenida en el buffer de recepción UDT es decodificada y entregada a la capa de aplicación.

El resto de los métodos de la clase son similares en estructura y funcionamiento a los de la clase de codificación. Por eso, y para evitar repetir información, no se van a comentar nuevamente en el presente documento.

5.2.- Integración con la librería UDT

Una vez que se han expuesto las clases de codificación y decodificación Raptor, ahora queda integrarlas en la estructura UDT para formar la librería RDT.

El primer paso es duplicar los *buffers* de envío y recepción, simplificando aquellos cuya funcionalidad sea un inconveniente para el desarrollo. Así pues, se crea una clase que será usada para hacer de *buffer* de recepción de la capa de aplicación. Esta clase ofrece un subconjunto de los métodos de los *buffers* originales, siendo este subconjunto suficiente para realizar todas las operaciones necesarias.

```
class SimplerBuffer {
 public:
    SimplerBuffer();
    ~SimplerBuffer();
    int addData(char* data, int len);
    int readBuffer(char* data, const int& len);
    int readBufferToFile(std::fstream& ofs, const int& len);
    int readMsg(char* data, const int& len);
    int getRcvDataSize() const;

 //[...]
 private:
    pthread_mutex_t m_BufLock;

    struct Buffer {
        char* m_pcData;
        char* posicionActual;
        int m_iSize;
        Buffer* m_pNext;
    } *actual, *ultimo;

 //[...]
 };
```

SimplerBuffer implementa una cola de tiras de bytes, estructura que es lo suficientemente flexible como para que la aplicación pueda acceder secuencialmente a los datos recibidos del emisor.

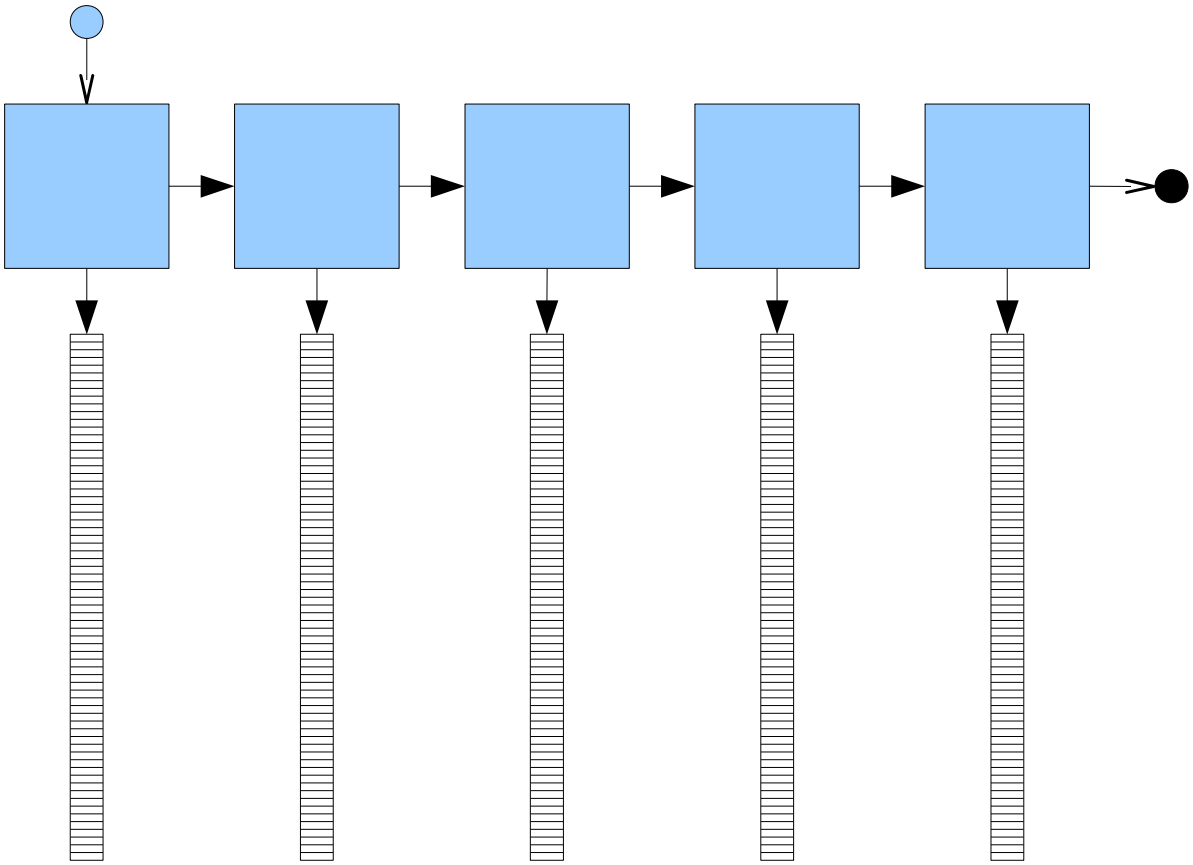

Figura 27: Diagrama de la estructura de SimplerBuffer

La declaración tanto de los *buffers* como de los hilos se realiza en el núcleo de la librería, siendo ésta la encargada de inicializarlos cuando se establezca una conexión.

```
private:
   CSndBuffer* m_pAppSndBuffer;
   CSndBuffer* m_pUDTSndBuffer;
   RaptorEncoderThread* raptorEncoder;

private:
   SimplerBuffer* m_pAppRcvBuffer;
   CRcvBuffer* m_pUDTRcvBuffer;
   RaptorDecoderThread* raptorDecoder;


m_pAppSndBuffer = new CSndBuffer(32, BLOCK_SIZE);
m_pUDTSndBuffer = new CSndBuffer(32, m_iPayloadSize);
raptorEncoder = new RaptorEncoderThread(m_pAppSndBuffer,
                                        m_pUDTSndBuffer, this,
                                        m_iPayloadSize, useRaptor);
raptorEncoder->start();

m_pAppRcvBuffer = new SimplerBuffer();
```

```
m_pUDTRcvBuffer = new CRcvBuffer(m_iRcvBufSize,
                                 &(m_pRcvQueue->m_UnitQueue));
raptorDecoder = new RaptorDecoderThread(m_pAppRcvBuffer,
                                        m_pUDTRcvBuffer, this,
                                        m_iPayloadSize, useRaptor);
raptorDecoder->start();
```

La constante BLOCK_SIZE es el tamaño del bloque a codificar y decodificar usando Raptor codes. En esta implementación el valor de esta constante es 10.000.000 bytes.

Para acceder en exclusión mutua a los *buffers*, y no hacer una espera activa en la lectura de los datos, es necesario duplicar también el sistema de semáforos de la librería. Esta duplicación se hace siguiendo las pautas de la implementación previa de la librería UDT. Así pues, se definen los nuevos semáforos y variables de condición, se incluyen los métodos de inicialización y destrucción y se crean los métodos para acceder a esos semáforos.

```
pthread_cond_t m_AppSendBlockCond;
pthread_mutex_t m_AppSendBlockLock;
pthread_cond_t m_RaptorSendBlockCond;
pthread_mutex_t m_RaptorSendBlockLock;

pthread_cond_t m_UDTRecvDataCond;
pthread_mutex_t m_UDTRecvDataLock;
pthread_cond_t m_AppRecvDataCond;
pthread_mutex_t m_AppRecvDataLock;

void CUDT::initSynch(){
   //[...]
      pthread_mutex_init(&m_AppSendBlockLock, NULL);
      pthread_cond_init(&m_AppSendBlockCond, NULL);
      pthread_mutex_init(&m_RaptorSendBlockLock, NULL);
      pthread_cond_init(&m_RaptorSendBlockCond, NULL);
      pthread_mutex_init(&m_AppRecvDataLock, NULL);
      pthread_cond_init(&m_AppRecvDataCond, NULL);
      pthread_mutex_init(&m_UDTRecvDataLock, NULL);
      pthread_cond_init(&m_UDTRecvDataCond, NULL);
   //[...]
}

void CUDT::destroySynch(){
   //[...]
      pthread_mutex_destroy(&m_AppSendBlockLock);
      pthread_cond_destroy(&m_AppSendBlockCond);
      pthread_mutex_destroy(&m_RaptorSendBlockLock);
      pthread_cond_destroy(&m_RaptorSendBlockCond);
      pthread_mutex_destroy(&m_AppRecvDataLock);
      pthread_cond_destroy(&m_AppRecvDataCond);
      pthread_mutex_destroy(&m_UDTRecvDataLock);
      pthread_cond_destroy(&m_UDTRecvDataCond);
    //[...]
}

void CUDT::releaseSynch(){
    //[...]
      pthread_mutex_lock(&m_AppSendBlockLock);
      pthread_cond_signal(&m_AppSendBlockCond);
      pthread_mutex_unlock(&m_AppSendBlockLock);
      pthread_mutex_lock(&m_RaptorSendBlockLock);
```

```
        pthread_cond_signal(&m_RaptorSendBlockCond);
        pthread_mutex_unlock(&m_RaptorSendBlockLock);
    //[...]
        pthread_mutex_lock(&m_AppRecvDataLock);
        pthread_cond_signal(&m_AppRecvDataCond);
        pthread_mutex_unlock(&m_AppRecvDataLock);
        pthread_mutex_lock(&m_UDTRecvDataLock);
        pthread_cond_signal(&m_UDTRecvDataCond);
        pthread_mutex_unlock(&m_UDTRecvDataLock);
    //[...]
}
```

Una vez duplicadas las estructuras, el núcleo de la librería debe ofrecer acceso a ese método de sincronización a los objetos codificador y decodificador. Esto lo realizará mediante ciertos métodos. Los métodos de espera que ofrece el núcleo de la librería, como pueden ser *waitAppBuffer* o *waitUDTBuffer*, no son más que una espera condicional en un semáforo.

```
void CUDT::waitAppBuffer(){
    //[...]
    if (0 == m_pAppSndBuffer->getCurrBufSize())
    {
       //[...]
       pthread_mutex_lock(&m_RaptorSendBlockLock);
       if (m_iSndTimeOut < 0){
          while (!m_bBroken && m_bConnected && !m_bClosing
                 && (0 == m_pAppSndBuffer->getCurrBufSize()))
             pthread_cond_wait(&m_RaptorSendBlockCond,
                               &m_RaptorSendBlockLock);
       }
       //[...]
       pthread_mutex_unlock(&m_RaptorSendBlockLock);
    }
 }
```

Estos métodos están hechos también siguiendo la pauta marcada por la librería UDT en su implementación.

Una vez definidos los semáforos y sus accesos, y haciendo posible que un hilo se suspenda indefinidamente, queda seleccionar el momento en el que dicho hilo se despertará y continuará con su cometido. Así pues, cada vez que se envía un paquete nuevo, el hilo de escritura se despertará; de igual manera, cada vez que se recibe un paquete se despertará el hilo de lectura.

Una vez integrada estructuralmente la clase en el núcleo de la librería, queda integrarla funcionalmente. Para ello hay que definir el paquete de control de cambio de bloque, estableciendo en qué momento se envía, cómo y por qué campos está compuesto.

El nuevo paquete que se llamará paquete de cambio de bloque, o *next block packet*, y llevará el identificador 32766. El paquete contendrá el número del bloque que se acaba de decodificar. Se enviará mediante el método *sendNextBlock* del núcleo de la librería.

```
void CUDT::sendNextBlock(int id){
    this->m_iLastBlock = id;
    int * aux = new int[1];
    *aux = id;
    this->sendCtrl(32766, NULL,(void *) aux, 4);
```

```
}
```

El resto de modificaciones se han hecho siguiendo el diseño de la librería, por lo que ha habido que modificar para ello el método *pack* de la clase paquete, ya que éste es invocado desde el método *sendCtrl* del núcleo de la librería y usa el identificador para empaquetar la información suministrada al paquete.

```
void CPacket::pack(const int& pkttype,
                   void* lparam, void* rparam, const int& size){
   //[...]
   case 32766:
         m_PacketVector[1].iov_base = (char *)rparam;
         m_PacketVector[1].iov_len = size;
         break;

   //[...]
}
```

En la recepción de dicho paquete de control solo queda que se informe al hilo de codificación, para lo que se ha creado el método *nextBlock* en el codificador.

```
void CUDT::processCtrl(CPacket& ctrlpkt){
   //[...]
   case 32766:
         raptorEncoder->nextBlock(*((int*)ctrlpkt.m_pcData));
         break;
   //[...]
}
```

Con esto se incluye la capa de codificación y decodificación Raptor en la librería UDT, con lo que una aplicación podría funcionar normalmente usando esta librería y usando las capas de codificación y decodificación Raptor, pero no aprovecharía ninguna de sus ventajas. Se debe continuar con las modificaciones para conseguir una librería RDT completa.

5.3.- Eliminación del sistema de envío UDT

La librería UDT proporciona un servicio de comunicación orientado a conexión que se basa en el envío de mensajes ACK, ACK2 y control de los paquetes perdidos a través de listas. Toda esa estructura es inservible para la librería RDT y, por tanto, se eliminará.

Como ya no será necesario mantener una lista de paquetes perdidos, sería conveniente eliminar las estructuras de datos de almacenaje de dichos paquetes. Sin embargo, como la librería quiere ofrecer la posibilidad de ser ejecutada como UDT, solo se creará un bypass en función de un booleano que especificará el modo de operación.

```
   // Loss detection.
   if (CSeqNo::seqcmp(packet.m_iSeqNo,CSeqNo::incseq(m_iRcvCurrSeqNo))>0
       && !useRaptor) {
      // If loss found, insert them to the receiver loss list
      m_pRcvLossList->insert(CSeqNo::incseq(m_iRcvCurrSeqNo),
                             CSeqNo::decseq(packet.m_iSeqNo));

   if(!useRaptor)
```

```
m_pRcvLossList->remove(packet.m_iSeqNo);
```

Estos son solo algunos ejemplos de los bypasses creados para evitar el uso de la lista de paquetes perdidos. Evitando el uso de esta lista se indica al núcleo de la librería que nunca tiene retransmisiones pendientes, cosa que en el sistema de envío que estamos implementando, que no usa retransmisiones de paquetes sino que recupera los datos perdidos mediante códigos Raptor, es lo que se persigue.

Cabe comentar también el uso del booleano *useRaptor*. En la versión actual de la librería el valor de este booleano se define en tiempo de compilación de la librería. Una ampliación de esta funcionalidad sería la posibilidad de incluirlo como una opción de la librería usando el sistema de configuración que proporciona la librería UDT. Esta posibilidad se comentará en el apartado de ampliaciones y mejoras.

Una vez eliminado el uso de las listas de paquetes perdidos, debe de eliminarse el envío de paquetes de informes de pérdidas o NACK. Esta eliminación se realiza de la misma manera que se realizó anteriormente con las listas, incluyendo el bypass con el booleano *useRaptor*.

```
if (CSeqNo::seqcmp(packet.m_iSeqNo, CSeqNo::incseq(m_iRcvCurrSeqNo)) > 0
    && !useRaptor) {
      // [...]
      sendCtrl(3, NULL, lossdata, (CSeqNo::incseq(m_iRcvCurrSeqNo) ==
                                   CSeqNo::decseq(packet.m_iSeqNo)) ?
                                                                  1 : 2);

case 3: //011 - Loss Report
 if(useRaptor)
    break;
```

El código 3 es el que corresponde al paquete de control de NACK, y solo se enviará si no se está usando la codificación Raptor. Como no se enviará ningún paquete de NACK, el código de la recepción de paquetes no se ejecutará nunca y, por tanto, no será necesario añadir ningún bypass como el empleado en estos ejemplos en ese código.

Una vez eliminado el control de paquetes perdidos se pasará a eliminar el envío de ACK y ACK2. Existen los ACK normales y los lite ACK. Dado que queremos eliminar todos los tipos de ACK habrá que hacer dos modificaciones similares a la siguiente en dos puntos distintos.

```
if(!useRaptor)
    m_pSndQueue->sendto(m_pPeerAddr, ctrlpkt);
```

Con los ACK2 pasa lo mismo que con el código de la recepción de NACK. Como los ACK2 solo se envían en respuesta a los ACK, al no enviarse ningún ACK tampoco se enviará ningún ACK2, por lo que no habrá que hacer ninguna modificación especial para evitar esas transmisiones.

Otro detalle a tener en cuenta es que el control por ventanas que se usaba en UDT tampoco va a ser necesario en la librería RDT. Por tanto, a la hora de enviar, también se a quitado la restricción extra que imponían las ventanas de control de control de congestión y de flujo en el antiguo sistema. La evasión de esta restricción se implementará, una vez más, con el mismo bypass por booleano utilizado en las anteriores ocasiones.

```
// check congestion/flow window limit
int cwnd = (m_iFlowWindowSize < (int)m_dCongestionWindow) ?
```

```
                                m_iFlowWindowSize :
                                (int)m_dCongestionWindow;
if (cwnd >= CseqNo::seqlen(const_cast<int32_t&>(m_iSndLastAck),
                                  CseqNo::incseq(m_iSndCurrSeqNo))
    || useRaptor){
    //[...]
```

Con todo esto, se ha eliminado todo el antiguo control de flujo del sistema de envío de paquetes, eliminando también el consumo de ancho de banda innecesario para la implementación del presente proyecto. Ahora se introducirá el nuevo sistema de control de flujo en base a ráfagas que hemos introducido en los apartados anteriores.

5.4.- Inclusión del sistema de control de flujo RDT

El control de congestión se lleva a cabo en esta estructura en una clase independiente que hereda de la clase CCC. En la librería UDT esta clase era CUDTCC, y se ha implementado una clase CRDTCC para sustituir a la clase anterior

```
class CRDTCC: public CCC {
 public:
    CRDTCC();

 public:
    virtual void init();
    virtual void onTimeout();
    virtual void onPktSent(const CPacket*);

//[...]
 private:
    int m_iGrupoActual;
    int m_iNumeroPaquetesFormanGrupo;
    int m_iPaquetesDeGrupoEnviados;
    int m_iTasaObjetivo;
    double m_dTiempoParaCadaGrupo;
    double m_dTiempoEstimadoParaCadaPaquete;
    double m_dTiempoOptimistaParaCadaPaquete;
    uint64_t m_iTimestampPrimerPaqueteDeGrupo;
    double alfa;
    double beta;
 };
```

Es en el método *onPktSent* que se realiza el control de congestión. Este método se ejecuta cada vez que se va a enviar el paquete, y su cometido es calcular el tiempo que se tendrá que esperar hasta enviar el siguiente paquete. Este valor se almacena en el atributo de la clase *m_dPktSndPeriod*.

```
void CRDTCC::onPktSent(const CPacket* pkt){
   if(pkt->m_iMsgNo > m_iGrupoActual){
      m_iPaquetesDeGrupoEnviados = 0;
      m_iGrupoActual = pkt->m_iMsgNo;
      m_iTasaObjetivo = beta * m_iBandwidth;
   }
   m_iPaquetesDeGrupoEnviados++;
   if(m_iPaquetesDeGrupoEnviados == 1){
```

```
            m_iTimestampPrimerPaqueteDeGrupo = CTimer::getTime();
            m_dTiempoEstimadoParaCadaPaquete = (1 /
                                               ((double) m_iTasaObjetivo));
            m_dTiempoOptimistaParaCadaPaquete = alfa *
                                        m_dTiempoEstimadoParaCadaPaquete;
            m_dTiempoParaCadaGrupo = m_dTiempoEstimadoParaCadaPaquete *
                                        m_iNumeroPaquetesFormanGrupo;
        }
        m_dPktSndPeriod = m_dTiempoOptimistaParaCadaPaquete * 1000000ULL;
        if(m_iPaquetesDeGrupoEnviados == m_iNumeroPaquetesFormanGrupo){
            m_dPktSndPeriod = (m_dTiempoParaCadaGrupo -
                               m_dTiempoOptimistaParaCadaPaquete *
                               m_iNumeroPaquetesFormanGrupo) * 1000000ULL;
        }
    }
```

Se comienza comprobando si el paquete que se va a enviar está dentro del grupo actual, es decir, si pertenece a la ráfaga para la que hemos hecho todos los cálculos. Esto es comprobado para saber si el núcleo de la librería ha decidido empezar una nueva ráfaga dejando la anterior incompleta, o si ya a terminado la ráfaga actual. Las razones que podría tener el núcleo para cambiar de ráfaga serán examinadas más adelante. Si la ráfaga es nueva, se reinician los valores de la ráfaga, o, lo que es lo mismo, el número de paquetes del grupo se pone a cero para volver a empezar la cuenta. El número de grupo actual se actualiza con el valor del paquete que se va a enviar, y se calcula la tasa objetivo tal y como se ha introducido en capítulos previos.

Después de esto se incrementa el contador de paquetes de grupo enviados, ya que este paquete a enviar forma parte del grupo actual. Una vez hecho esto se comprueba si este paquete es el primero del grupo. Si es así se siguen inicializando variables para la ráfaga que comienza. Esta inicialización es la siguiente: (i) se lee el instante de tiempo actual, valor que de momento no se usa pero que más adelante se verá su posible funcionalidad; (ii) se calcula el tiempo estimado entre paquetes, que no es más que el inverso de la tasa objetivo; (iii) se calcula el tiempo entre paquetes optimista, valor que de acuerdo a lo expuesto en puntos anteriores, se calcula aplicando un valor de corrección alfa al tiempo estimado entre paquetes; y (iv) se calcula el tiempo que deberá tardar cada grupo de paquetes según la tasa objetivo. Este conjunto de inicializaciones podría haberse incluido junto con las inicializaciones del párrafo anterior, pero dadas las constantes modificaciones a las que se ha tenido que someter el método se ha optado por separarlas, dejando así el código más cómodo de cara al programador.

Una vez hechas todas la actualizaciones necesarias se procede a hacer el cálculo del tiempo de espera entre paquetes, que es el principal cometido del método en cuestión. Realmente, como el valor ya lo teníamos calculado en las inicializaciones anteriores, solo queda tomar el valor correcto y pasarlo a microsegundos, valor que utiliza el núcleo de la librería para sus cálculos.

Una excepción a este cálculo del tiempo entre paquetes es el que corresponde al último paquete de la ráfaga. El paquete posterior a éste deberá hacer una espera extra, ya que esa es la filosofía de la librería. Este cálculo se realiza restando al tiempo total que hay para cada grupo el tiempo consumido por la ráfaga de paquetes. Otra manera sería calcularlo en base al *timestamp* de inicio y el *timestamp* actual para tener un valor práctico en lugar del valor teórico calculado hasta ahora.

```
    m_dPktSndPeriod = m_dTiempoParaCadaGrupo -
                      (CTimer::getTime() - m_iTimestampPrimerPaqueteDeGrupo)
                      * 1000000ULL;
```

Una vez creada la clase de control de congestión, queda integrarla en la estructura completa de la librería. El primer paso para conseguirlo es reemplazar el control de congestión CUDTCC actual por este nuevo en la inicialización de la librería. Como todas las modificaciones que se hacen en el núcleo de la librería, se parametrizará en base al booleano *useRaptor*.

```
if(useRaptor){
      m_pCCFactory = new CCCFactory<CRDTCC>;
} else {
      m_pCCFactory = new CCCFactory<CUDTCC>;
}
```

Ahora queda formar las ráfagas o grupos de paquetes para enviarlos, que en esta implementación estarán formadas por 20 paquetes. Un elemento importante de las ráfagas es proporcionar una forma de que sean reconocidas en el receptor, pues si el receptor no puede reconocer las ráfagas no podrá hacer los cálculos pertinentes para el ancho de banda. Una manera de implementar esto sería mediante el número de secuencia, valor consecutivo que marca los paquetes, y realizar una división entera de este valor entre el tamaño de la ráfaga para saber a que ráfaga pertenece el paquete. Para implementar este método no es necesario incluir ningún elemento extra en el envío de los paquetes, pero si habrá que modificar la recepción.

```
if(m_iPaquetesDeGrupoRecibidos == 0){
      m_iPrimerMsgNoDelGrupo = packet.m_iSeqNo;
      m_iTimestampPrimerPaqueteDeGrupo = CTimer::getTime();
}  else if(packet.m_iSeqNo - m_iPrimerMsgNoDelGrupo >
           m_iNumeroPaquetesFormanGrupo){
      informePeriodicoTerminado();
      m_iPrimerMsgNoDelGrupo = packet.m_iSeqNo;
      m_iTimestampPrimerPaqueteDeGrupo = CTimer::getTime();
}
m_iPaquetesDeGrupoRecibidos++;
m_iTimestampUltimoPaqueteDeGrupo = CTimer::getTime();
```

Este sistema tiene varios inconvenientes. El primero es que el número de secuencia empieza en un valor aleatorio, y por tanto no podemos simplemente hacer la división, sino que se tiene que tener en cuenta el primer mensaje enviado para saber cual es el inicio de la secuencia. Sin embargo, dado que el primer mensaje enviado no tiene por qué ser el primer mensaje recibido, el cálculo estará siempre en riesgo de estar sometido a una falta de sincronización con el emisor, lo cual, en este sistema de envío a ráfagas, provoca problemas en el sistema de control y estimación de la tasa de ytansmisión.

Debido a que el sistema anterior es muy deficiente se optó por usar otro más robusto. Para ello se marcarán todos los paquetes con el número de ráfaga al que pertenecen. Para no tener que incluir un nuevo campo en la cabecera, se reutilizará el campo del número de mensaje, campo que en la librería RDT no tiene ningún uso.

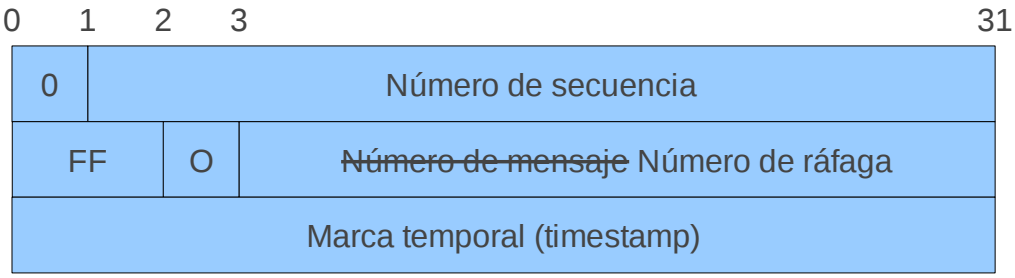

Figura 28: Modificación en la cabecera de los mensajes de datos

Después se modificará el sistema de envío para marcar los paquetes con el número de ráfaga a la que pertenecen, teniendo que mantener para ello los contadores pertinentes.

```
if(useRaptor){
	if(m_bNextGroup){
		m_iGrupoActualEnvio++;
		m_iPaquetesDeGrupoEnviados = 0;
		m_bNextGroup = false;
	}
	m_iPaquetesDeGrupoEnviados++;
	packet.m_iMsgNo = m_iGrupoActualEnvio;
	if(m_iPaquetesDeGrupoEnviados == m_iNumeroPaquetesFormanGrupo){
		m_iGrupoActualEnvio++;
		m_iPaquetesDeGrupoEnviados = 0;
	}
}

m_pCC->onPktSent(&packet);
```

En primer lugar se comprueba si hay que pasar al siguiente grupo de paquetes por alguna razón distinta a la de finalización de trama. Esto ocurre cuando se pasa a codificar un nuevo bloque de información, ya que en la codificación Raptor se invoca el método *nextGroup* que actualiza esa variable y la pone a cierto.

```
void CUDT::nextGroup(){
	m_bNextGroup = true;
}
```

Después se incrementa el contador de paquetes de grupo enviados y se hace la marca con el número de ráfaga en la cabecera del mensaje, sobre el campo del número de mensaje. Así, una vez que el paquete llegue al destino, es trivial clasificarlo dentro de una ráfaga u otra. Además, si por alguna razón, como el cambio de bloque, el servidor tiene que reiniciar la cuenta, el cliente se entera inmediatamente y no se produce ningún error en el cálculo del ancho de banda.

Tras esto se comprueba si este es el último paquete de grupo. Si es así, hay que hacer las mismas operaciones que si el cambio de grupo hubiera sido forzado por un cambio de bloque de codificación.

Además, se puede ver en el código que, justo después de este conjunto de operaciones, se llama a la función *onPktSent* de la librería de control de congestión. Esto es así ya que ambos

grupos de operaciones son complementarias.

En el lado del receptor hay que preparar el código necesario para tratar apropiadamente las ráfagas de paquetes que se vayan recibiendo.

```
if(useRaptor){
   if(packet.m_iMsgNo > m_iGrupoActualRecepcion){
        if(m_iPaquetesDeGrupoRecibidos > 1)
             informePeriodicoTerminado();
        m_iGrupoActualRecepcion = packet.m_iMsgNo;
        m_iPaquetesDeGrupoRecibidos = 0;
        m_iExpiraTimerControlFlujoRDT = 0;
   }
   if(packet.m_iMsgNo == m_iGrupoActualRecepcion){
        m_iPaquetesDeGrupoRecibidos++;
        if(m_iPaquetesDeGrupoRecibidos == 1){
             m_iTimestampPrimerPaqueteDeGrupo = CTimer::getTime();
        }
        m_iTimestampUltimoPaqueteDeGrupo = CTimer::getTime();
        if(m_iPaquetesDeGrupoRecibidos > 1){
             m_iExpiraTimerControlFlujoRDT =
                              m_iTimestampUltimoPaqueteDeGrupo +
                              2 * ((m_iTimestampUltimoPaqueteDeGrupo -
                                    m_iTimestampPrimerPaqueteDeGrupo) /
                                    (m_iPaquetesDeGrupoRecibidos - 1));
        }
        if(m_iPaquetesDeGrupoRecibidos == m_iNumeroPaquetesFormanGrupo){
             informePeriodicoTerminado();
        }
    }
}
```

Si el paquete que se ha recibido es de una ráfaga posterior a la que estamos esperando significa que se ha cambiado de ráfaga y, por tanto, debemos reinicializar los contadores. Si además tenemos información suficiente para estimar el ancho de banda del canal, creamos y enviamos el informe con la información que tenemos hasta el momento.

Una vez hecho esto, y si no se trata un paquete rezagado de una ráfaga anterior, lo anotamos como recibido incrementando el contador correspondiente. Además, si es el primero, se guarda el instante en el que lo hemos recibido para calcular más tarde el ancho de banda. Después se actualiza también el valor del instante temporal del último paquete de grupo recibido al instante actual.

Después actualizamos el valor del temporizador. Este temporizador se usa en aquellas ocasiones en las que se pierden paquetes y no podemos obtener la ráfaga completa, evitando así tener que esperar al primer paquete de la siguiente ráfaga para enviar el informe con los datos de recepción. La comprobación del estado del temporizador se realiza, junto con el resto de temporizadores de la librería, en el método *checkTimers*.

```
if(useRaptor){
      if(currtime > m_iExpiraTimerControlFlujoRDT &&
         m_iExpiraTimerControlFlujoRDT != 0){
            informePeriodicoTerminado();
      }
}
```

Por último se comprueba si es el último paquete de la ráfaga. Si es así se envía el informe usando el método *informePeriodicoTerminado*.

```
void CUDT::informePeriodicoTerminado(){
      char * aux = new char[sizeof(int) + sizeof(int) + sizeof(int)];
      char * auxptr = aux;
      int anchoBandaPaquetesSegundo = MAX_ANCHO_BANDA;
      if(m_iTimestampPrimerPaqueteDeGrupo !=
         m_iTimestampUltimoPaqueteDeGrupo){
            uint64_t incrementoTiempo = (m_iTimestampUltimoPaqueteDeGrupo
                                        - m_iTimestampPrimerPaqueteDeGrupo);
            anchoBandaPaquetesSegundo =
                     ((m_iPaquetesDeGrupoRecibidos - 1) * 1000000ULL)
                     / (incrementoTiempo);
      }
      memcpy(auxptr, &anchoBandaPaquetesSegundo, sizeof(int));
      auxptr = auxptr + sizeof(int);
      memcpy(auxptr, &m_iLastBlock, sizeof(int));
      auxptr = auxptr + sizeof(int);
      memcpy(auxptr, &m_iPaquetesDeGrupoRecibidos, sizeof(int));
      sendCtrl(32765, NULL, (void *) aux,
               sizeof(int) + sizeof(int) + sizeof(int));
      m_iPrimerMsgNoDelGrupo = -1;
      m_iPaquetesDeGrupoRecibidos = 0;
      m_iExpiraTimerControlFlujoRDT = 0;
      m_iGrupoActualRecepcion++;
}
```

En este método se calcula el ancho de banda de la siguiente manera. Si los tiempos del primer y último paquete son diferentes, es decir, si se ha recibido más de un paquete, se calcula el incremento de tiempo entre esos dos valores. Por tanto, el ancho de banda es igual al número de paquetes recibidos menos uno dividido entre el incremento temporal. Además se pasa el valor de paquetes por microsegundo a paquetes por segundo para cumplir con las especificaciones de la librería.

Una vez calculado el ancho de banda, se copia la información en la estructura que se enviará con el paquete y se invocará la función que envíe dicho paquete. Por último se inicializarán las variables que se usan en el proceso para comenzar con la siguiente estimación del ancho de banda del canal.

Como este es un nuevo paquete de control, de número de identificación 32765 y cuyo contenido son tres enteros (ancho de banda estimado, último bloque recuperado con éxito y número de paquetes de la ráfaga recibidos, por ese orden), también se han realizado modificaciones en la función de envío y en el método *pack* de la clase paquete.

```
void CUDT::sendCtrl(const int& pkttype,
                    void* lparam, void* rparam, const int& size){
   //[...]
   case 32765:
         ctrlpkt.pack(32765, lparam, rparam, size);
         ctrlpkt.m_iID = m_PeerID;
         m_pSndQueue->sendto(m_pPeerAddr, ctrlpkt);
         break;
   //[...]
}
```

```
void CPacket::pack(const int& pkttype,
                   void* lparam, void* rparam, const int& size){
   //[...]
   case 32765:
   case 32766:
         m_PacketVector[1].iov_base = (char *)rparam;
         m_PacketVector[1].iov_len = size;
         break;

   //[...]
}
```

Y, como es lógico, también es necesario incluir código que trate el paquete de control cuando llegue al lado del emisor.

```
void CUDT::processCtrl(CPacket& ctrlpkt){
   //[...]
   case 32765:
         m_pCC->setBandwidth(*((int*)ctrlpkt.m_pcData));
         raptorEncoder->nextBlock(*((int*)(ctrlpkt.m_pcData +
                                                   sizeof(int))));
         break;
   //[...]
}
```

En este momento se actualiza el valor del ancho de banda contenido en la clase CRDTCC, que es la encargada de calcular el tiempo entre paquetes, cerrando así el círculo y proporcionando la retroalimentación de la que se lleva hablando en todo el proceso.

5.5.- Codificación en paralelo

Recordando el diseño presentado en la figura 18 se puede hacer un mapeo directo de las estructuras a las clases implementadas en C++. Así, el *Raptor Controller*, la estructura contenedora y el hilo asociado se implementó en la clase *EncoderBuffer*, la estructura etiquetada como *Encoder* se implementó mediante la clase *RaptorEncoderWorker*, y el *buffer* interno en la clase codificador se implementó como la clase *WorkerBuffer*. Estas clases sustituirán a la ya presentada como *RaptorEncoderThread* en previos puntos.

Ahora pasaremos a mostrar la clase *EncoderBuffer*. Las partes más importantes de la declaración de esta clase es la siguiente:

```
class EncoderBuffer {

public:
      EncoderBuffer(CSndBuffer* envioAplicacion, CUDT* libreria, int tCarga);
      EncoderBuffer(CSndBuffer* envioAplicacion, CUDT* libreria, int tCarga,
                                                                 int nBloques);
      virtual ~EncoderBuffer();

      int readData(char** data, int32_t& msgno);
      int getCurrBufSize();
      void clear();

      void monitoriza();
```

```
        void awake();
        void close();

        void nextBlock(int id);

        void printStatus();
private:
        void inicializarSemaforos();
        void inicializar(CSndBuffer* aE, CUDT* l, int tC, int nB);

private:
        int tCarga;
        CSndBuffer* envioAplicacion;
        CUDT* libreria;

        bool closing;

        //Gestión de la lista de Workers
        int nBloques;

        RaptorEncoderWorker** bloques;
        int slotsOcupados;
        int libre;
        int cabeza;
        int siguienteBloque;
        int primerBloque;

        // Gestión de semáforos
        pthread_mutex_t m_BufLock;
        pthread_mutex_t m_StreamLock;
        pthread_mutex_t m_WaitInfoLock;
        pthread_cond_t m_StreamCond;
        pthread_cond_t m_WaitInfoCond;

        pthread_t monitorizaWorkers;

        //[...]
};
```

Para la creación de la estructura completa se siguió el paradigma de caja negra. Se buscaba ocultar todo el procesamiento interno de la información al esqueleto UDT, y que el nuevo diseño diera la imagen de ser simplemente un *buffer* en el que UDT metiera la información en claro y pudiera sacar la información codificada poco a poco, sin que tuviera que preocuparse por ningún detalle más. Eso nos permitía, además, que el sistema de capas quedase mucho más marcado, así como limitar interacciones entre los componentes de la librería. Esto supuso una ventaja muy importante a la hora de construir el protocolo más elaborado del que se ha hablado en el apartado de diseño.

Dado que se quería mostrar un aspecto de *buffer* a la capa, la clase muestra las típicas funciones de escritura y lectura en *buffer*, así como un función que informa del tamaño actual del *buffer* y otra que permite vaciarlo. Junto con estas funciones aparecen otras necesarias para la gestión del hilo interno encargado de controlar la estructura, como son *monitoriza()*, que inicia el hilo; *awake()*, que lo despierta cuando tiene que realizar tareas de mantenimiento y *close()* que realiza una finalización segura y ordenada de toda la estructura de hilos interna. Por último señalar que también se incluye la función de *nextBlock()*, que permite notificar al buffer cuando se ha recibido el reconocimiento de uno de los bloques que hay que enviar; y una función *printStatus()*,

creada para debug.

A las variables que la clase homóloga tenía anteriormente se le añaden una serie de variables que implementan una lista que permite gestionar rápidamente los bloques en proceso de codificación. El otro grupo de variables guarda la información necesaria para la sincronización interna de los hilos, incluyendo la apariencia externa de este *buffer* como un flujo o *stream*.

Ahora pasaremos a describir la implementación de las funciones más relevantes de las que se han descrito anteriormente. La primera es la función *monitoriza()*, que es la que ejecuta el hilo de monitorización y se encarga de comprobar los *RaptorEncoderWorkers*, sí como de renovarlos cuando ya se han reconocido en destino.

```
void EncoderBuffer::monitoriza() {
```

Se entra en un bucle que monitoriza el estado de los *RaptorEncoderWorker.* Solo terminará este bucle cuando se reciba la orden de cerrar, invocándose el método *close()*.

```
    while (!closing) {
```

Después, se pasa a esperar a que un evento requiera realizar el proceso de actualización de los *RaptorEncoderWorker*. Estos eventos pueden ser: a) un bloque se ha decodificado en destino y debe reemplazarse un *RaptorEncoderWorker* de un slot por otro que codifique el bloque siguiente, b) la capa de aplicación tiene nueva información para enviar o c) la librería se está cerrando.

```
        pthread_mutex_lock(&m_WaitInfoLock);
        while (!closing &&
               (nBloques <= slotsOcupados ||
               (envioAplicacion->getCurrBufSize() - slotsOcupados <= 0))){
              pthread_cond_wait(&m_WaitInfoCond, &m_WaitInfoLock);
        }
        pthread_mutex_unlock(&m_WaitInfoLock);
```

Después, comprueba que tareas puede hacer. Como a esta estructura pueden acceder múltiples hilos, por mayor seguridad se realizan todas las actualizaciones en exclusión mutua.

```
        CGuard::enterCS(m_BufLock);
        if (nBloques > slotsOcupados &&
            (envioAplicacion->getCurrBufSize() - slotsOcupados) > 0 &&
            !closing) {
```

En este paso se procede a leer información del *buffer* que comunica la capa de aplicación con la de codificación y se crea una nueva estructura encargada de codificar esa nueva porción de información.

```
            int longEnvio = 0;
            int32_t aux = 0;
            char ** c = new char*[1];
            while (longEnvio == 0) {
                  longEnvio = envioAplicacion->readData((char**) c, aux);
            }

            bloques[libre] = new RaptorEncoderWorker(siguienteBloque,
                                                     longEnvio,
                                                     (unsigned char*)(*c),
                                                     tCarga);
```

Después, se llevan a cabo todas las actualizaciones de gestión de la lista de RaptorEncoderWorkers.

```
                bloques[libre]->start();
                if (primerBloque == -1) {
                        primerBloque = siguienteBloque;
                }
                siguienteBloque++;
                if (cabeza == -1) {
                        cabeza = libre;
                }
                libre = (libre + 1) % nBloques;
                slotsOcupados++;
        }
        CGuard::leaveCS(m_BufLock);
    }
}
```

Otra función interesante es la función *nextBlock()*, encargada de actualizar la lista eliminando los bloques que van siendo reconocidos por el destino.

```
void EncoderBuffer::nextBlock(int id) {
```

Al igual que en la versión anterior, lo primero que hace es comprobar si el bloque que se está reconociendo es el que toca, o si ese ya ha sido renovado.

```
    if (primerBloque == id) {
        Cguard::enterCS(m_BufLock);
```

Si tiene que eliminar un bloque, lo primero que hace es finalizarlo, con lo que detiene los hilos que se estén ejecutando en él. Después notifica a la capa superior que los datos ya han sido enviados con éxito al destino, con lo que se pueden eliminar del *buffer* o sustituir por nueva información a enviar.

```
            bloques[cabeza]->finalizar(id);
            envioAplicacion->ackData(1);
            libreria->awakeSend();
```

Después actualiza la información correspondiente a la gestión de la lista de *RaptorEncoderWorkers*.

```
            slotsOcupados--;
            if (slotsOcupados == 0) {
                cabeza = -1;
            } else {
                cabeza = (cabeza + 1) % nBloques;
            }
            primerBloque++;
```

Por último, despierta al hilo monitor para que realice sus tareas de mantenimiento y comprueba si debe instanciarse un nuevo ítem de la estructura *RaptorEncoderWorker*.

```
            pthread_mutex_lock(&m_WaitInfoLock);
               pthread_cond_signal(&m_WaitInfoCond);
            pthread_mutex_unlock(&m_WaitInfoLock);
            CGuard::leaveCS(m_BufLock);
```

```
	}
}
```

El último método verdaderamente relevante de esta clase es el llamado *readData()*. En esta versión consta de una implementación muy sencilla, pero es importante que tengamos claro cómo está implementado para comparar con la evolución del mismo que se verá más adelante.

```
int EncoderBuffer::readData(char** data, int32_t& msgno) {

	CGuard::enterCS(m_BufLock);
	int leido = 0;
	if(cabeza != -1)
		leido = bloques[cabeza]->readData(data);

	CGuard::leaveCS(m_BufLock);

	return leido;
}
```

Como se puede ver, simplemente se accede al primer *RaptorEncoderWorker* y se lee el primer símbolo que haya disponible.

La siguiente clase que se va a analizar es la clase *RaptorEncoderWorker*. Como ya se ha dicho, cada una de estas clases está encargada de codificar de manera independiente un bloque e ir suministrando tanto los bloques fuente como los bloques de recuperación. Los objetos de esta clase son gestionados por la clase que se ha descrito anteriormente.

```
class RaptorEncoderWorker {
public:
	RaptorEncoderWorker();
	RaptorEncoderWorker(int bloque,
			    int dataLength,
			    unsigned char* datos,
			    int tCarga);
	virtual ~RaptorEncoderWorker();

	void start();
	void run();
	void finalizar(int id);

	int readData(char** data);

	int getSize();

private:
	unsigned char* datos;		//Datos de entrada
	WorkerBuffer* workerBuffer;	//Buffer de salida

	//Atributos para la codificación
	int tCarga;
	int dataLength;
	int symbolSize;
	int nBloque;
	int nSourceSymbols;

	RaptorEncoder *encoder;
	DFR11EncMemRequirement memReq;
```

```
        int id;
        void * repairSymbol;

        //Finalización
        bool closing;
        bool finEnvio;

        pthread_t worker;            //Thread interno

};
```

Esta clase contiene los métodos necesarios para la inicialización y lanzamiento del hilo que realiza la codificación. Además, dado que el *buffer* donde se almacenan los símbolos se gestiona internamente, también ofrece un método para recuperar los símbolos.

El único método relevante de esa clase es el de codificación, implementado en el método *run()*, que es el encargado de ir generando los símbolos. Este método es similar al presentado en puntos anteriores, pero adaptado a las nuevas estructuras, por lo que no se va a comentar en detalle.

La última clase a introducir en este punto es la que representa el *buffer* interno de cada *RaptorEncoderWorker*: *WorkerBuffer*.

```
class WorkerBuffer {
public:
        WorkerBuffer(int packetSize, int bufferSize);
        virtual ~WorkerBuffer();

        int readData(char** data);
        void addData(char* datos);

        void closeUp();

        int getSize();

private:

        // Atributos para la gestión de la cola
        struct Buffer {
            char* m_pcData;
            Buffer* m_pNext;
        } *actual, *ultimo;

        int packetSize;
        int bufferSize;
        int actualSize;

        bool closingUp;
        bool closingDown;

        // Atributos para la gestión del acceso a los datos de la cola
        pthread_mutex_t m_BufLock;
        pthread_mutex_t m_StreamInLock;
        pthread_mutex_t m_StreamOutLock;
        pthread_cond_t m_StreamInCond;
        pthread_cond_t m_StreamOutCond;

};
```

Esta clase implementa una estructura de cola clásica haciendo uso de memoria dinámica, que gestiona la exclusión mutua y el acceso al *buffer* como si de un flujo o *stream* se tratara.

5.6.- Gestión del tiempo de espera entre paquetes

Para conseguir un rendimiento lo más ajustado posible y dado que la implementación se hace a nivel de capa de aplicación, se ha cambiado la forma en la que la librería UDT gestionaba el tiempo de espera entre paquetes. Para ello ha sido necesario cambiar la implementación de la función *sleepTo()* por otra más acorde a nuestras necesidades, a la que se ha llamado *accurateSleepTo()*.

La idea básica de esta función es implementar un sistema híbrido de espera entre paquetes, que realice una espera temporizada para la primera parte, y que en la segunda se realice una espera activa. La implementación, no obstante, sigue el esquema marcado por las directrices de la librería UDT, haciendo una espera en semáforo fraccionada.

```
void CTimer::accurateSleepto(const uint64_t& nexttime){
	// [...]
```

En primer lugar, se comprueba que el momento actual sea inferior al momento en el que se ha planificado que el hilo se despierte. Mientras que no sea así, el hilo permanecerá detenido en la función.

```
	m_ullSchedTime = nexttime;

	uint64_t t;
	rdtsc(t);
	while (t < m_ullSchedTime){
		// [...]
		timeval now;
		timespec timeout;
		gettimeofday(&now, 0);
		int paso = 100;
```

Después se comprueba la cantidad de tiempo que queda para despertar el hilo. Si se decide que el hilo tiene que esperar en un semáforo, esperará la mitad del tiempo que le queda. Si se decide lo contrario hará espera activa. En este caso, la decisión se basa en si esa mitad que tendría que esperar en el semáforo es mayor que cierto umbral o no. Para hacer la espera activa simplemente se calcula un tiempo de espera negativo, lo que hará que la ejecución no se detenga en el semáforo.

```
		if((m_ullSchedTime - t) / 2 > 50){
			paso = (m_ullSchedTime - t) / 2;
		} else {
			paso = (t - m_ullSchedTime);
		}
```

Tras esta decisión, se programa el tiempo de espera y se duerme al hilo usando una función de espera condicional temporizada en un semáforo.

```
		if (now.tv_usec + paso < 1000000) {
			timeout.tv_sec = now.tv_sec;
			timeout.tv_nsec = (now.tv_usec + paso) * 1000;
		} else {
```

```
                timeout.tv_sec = now.tv_sec + 1;
                timeout.tv_nsec = (now.tv_usec + paso - 1000000) * 1000;
            }
            pthread_mutex_lock(&m_TickLock);
            pthread_cond_timedwait(&m_TickCond, &m_TickLock, &timeout);
            pthread_mutex_unlock(&m_TickLock);
            // [...]
            rdtsc(t);
    }
}
```

5.7.- Cambio en la estrategia de reconocimiento de bloques

Esta optimización, pequeña en su magnitud pero grande en su alcance, ha permitido mejorar la productividad global del sistema. La implementación de esta mejora, dado el diseño modular con el que cuenta la librería, ha sido muy sencilla de implementar y se va a comentar a continuación.

```
void RaptorDecoderThread::run(){
      //[...]
      while(faltanSimbolos){
            result = DFR11DecAddRcvSymbolIDs(decoder, id, 1);
            result = (DFR11DecError)DFR11DecPrepare(decoder,
                                                    nSrcSymbols,
                                                    symbolSize,
                                                    TAREA_DEC_RAPTOR);
            //[...]
            if(result < 0) {
                  //[...]
                  int aux = recepcionUDT->readBuffer(c, symbolSize);
                  //[...]
            } else {
                  faltanSimbolos = false;
                  idBloqueEsperado++;
            }
      }
      result = DFR11DecInitRcvBlock (decoder, rcvBlock, tempBlock);
      result = (DFR11DecError) DFR11DecRecoverSource(decoder, TAREA_DEC_RAPTOR);
      udt->sendNextBlock(idBloqueFuente);
      //[...]
}
```

Este es el aspecto que tenía antes esa porción de código. Como se puede ver en este fragmento resumido de código, el codificador va recibiendo los símbolos y almacenándolos. Para saber si puede decodificar el bloque o necesita más símbolos intenta preparar el bloque para su decodificación. Si no puede, espera más símbolos. Si puede, realiza el proceso de decodificación y posteriormente envía el reconocimiento.

```
void RaptorDecoderThread::run(){
      //[...]
      while(faltanSimbolos){
            result = DFR11DecAddRcvSymbolIDs(decoder, id, 1);
            result = (DFR11DecError)DFR11DecPrepare(decoder,
                                                    nSrcSymbols,
                                                    symbolSize,
                                                    TAREA_DEC_RAPTOR);
```

```
            //[...]
            if(result < 0) {
                  //[...]
                  int aux = recepcionUDT->readBuffer(c, symbolSize);
                  //[...]
            } else {
                  faltanSimbolos = false;
                  idBloqueEsperado++;
            }
      }
      udt->sendNextBlock(idBloqueFuente);
      result = DFR11DecInitRcvBlock (decoder, rcvBlock, tempBlock);
      result = (DFR11DecError) DFR11DecRecoverSource(decoder, TAREA_DEC_RAPTOR);
      //[...]
}
```

Arriba se muestra el código resultante de la modificación. En este caso simplemente se ha invertido el orden de los dos últimos pasos descritos anteriormente. Así el paquete de decodificación se envía antes que en el caso anterior y por tanto se va agilizando el proceso de cambio de bloque.

5.8.- Optimizaciones a la codificación: desarrollos evaluados

Se han implementado una serie de variaciones del proceso de codificación con el objetivo de evaluar el impacto que tenían en el rendimiento cada una de ellas. La motivación y las características de cada una de estas variantes ya han sido descritas en el punto previo de diseño avanzado, por lo que se procederá directamente a presentar el código relacionado con la implementación.

La primera de estas mejoras es sencilla, porque solo suponía variar el tamaño de un buffer. En este caso, se cambió el tamaño del buffer interno *WorkerBuffer* para que su capacidad fuera mayor. De esta forma, el proceso de codificación se amortigua mejor y se consigue mantener un flujo constante de símbolos para enviar. Dado que esta modificación es muy sencilla, no se va a presentar código.

La segunda modificación cambia el proceso de codificación de sitio para intentar solapar su efecto con el envío de símbolos del bloque anterior.

```
void RaptorEncoderWorker::run() {
      //[...]

      encoder = (RaptorEncoder *) malloc(memReq.encoderSize);
      result = DFR11EncInit(encoder, CONFIGURACION_ENC_RAPTOR);
      result = DFR11EncReset(encoder);
      result = (DFR11EncError) DFR11EncPrepare(encoder,
                                               nSourceSymbols,
                                               symbolSize,
                                               TAREA_ENC_RAPTOR);

      //[...]
      result = DFR11EncInitSrcBlock(encoder, srcBlock, intermBlock);

      //[...]
      for (int id = 0; id < nSourceSymbols; id++) {
```

```
            //[...]
            workerBuffer->addData(paquete - CABECERA);
            //[...]
      }

      result = (DFR11EncError) DFR11EncGenIntermediateBlock(encoder,
                  TAREA_ENC_RAPTOR);
      //[...]
}
```

Aquí se presenta de manera muy esquemática el proceso de codificación llevado a cabo en el emisor. Como se puede ver, la primera pasada de codificación se realiza después de que los símbolos fuente se hayan enviado o al menos cargado en el *buffer*.

```
void RaptorEncoderWorker::run() {
      //[...]

      encoder = (RaptorEncoder *) malloc(memReq.encoderSize);
      result = DFR11EncInit(encoder, CONFIGURACION_ENC_RAPTOR);
      result = DFR11EncReset(encoder);
      result = (DFR11EncError) DFR11EncPrepare(encoder,
                                               nSourceSymbols,
                                               symbolSize,
                                               TAREA_ENC_RAPTOR);

      //[...]
      result = DFR11EncInitSrcBlock(encoder, srcBlock, intermBlock);

      //[...]
      result = (DFR11EncError) DFR11EncGenIntermediateBlock(encoder,
                  TAREA_ENC_RAPTOR);

      for (int id = 0; id < nSourceSymbols; id++) {
            //[...]
            workerBuffer->addData(paquete - CABECERA);
            //[...]
      }

      //[...]
}
```

En esta segunda porción se código se puede ver cómo se ha alterado el procedimiento de codificación para que la primera pasada se realice antes del envío de los símbolos fuente. Aunque aquí se ha omitido para mejorar la claridad del código, también se a trasladado el proceso de reserva de memoria para intentar aprovechar la misma ventaja.

La tercera modificación hace uso de la codificación progresiva que implementa la librería de codificación. En este caso, como se puede ver en el código, la tarea de codificación se realiza progresivamente durante el envío de los símbolos fuente.

```
void RaptorEncoderWorker::run() {
      //[...]

      encoder = (RaptorEncoder *) malloc(memReq.encoderSize);
      result = DFR11EncInit(encoder, CONFIGURACION_ENC_RAPTOR);
      result = DFR11EncReset(encoder);
      result = (DFR11EncError) DFR11EncPrepare(encoder,
```

```
                                                nSourceSymbols,
                                                symbolSize,
                                                TAREA_ENC_RAPTOR);

        //[...]
        result = DFR11EncInitSrcBlock(encoder, srcBlock, intermBlock);

        //[...]
        int mod = nSourceSymbols / 1000;
        int paso = 1;
        if(nSourceSymbols <= 1000){
              mod = 1;
              paso = 1000 / nSourceSymbols;
        }
        for (int id = 0; id < nSourceSymbols; id++) {
              //[...]
              workerBuffer->addData(paquete - CABECERA);
              //[...]
              if(id % mod == 0){
                    result = (DFR11EncError) DFR11EncGenIntermediateBlock(encoder,
                                                                          paso);
              }
        }
        //[...]
}
```

Para a cuarta modificación se realiza la codificación en un hilo independiente. Para ello, obviando las modificaciones necesarias en la declaración de la clase, se muestran las modificaciones realizadas al método de codificación.

```
void RaptorEncoderWorker::run() {
        //[...]

        encoder = (RaptorEncoder *) malloc(memReq.encoderSize);
        result = DFR11EncInit(encoder, CONFIGURACION_ENC_RAPTOR);
        result = DFR11EncReset(encoder);
        result = (DFR11EncError) DFR11EncPrepare(encoder,
                                                nSourceSymbols,
                                                symbolSize,
                                                TAREA_ENC_RAPTOR);
        //[...]
        result = DFR11EncInitSrcBlock(encoder, srcBlock, intermBlock);

        //[...]
        pthread_create(&helper, NULL, (void* (*)(void*)) lanzadorAuxiliar, this);
        for (int id = 0; id < nSourceSymbols; id++) {
              //[...]
              workerBuffer->addData(paquete - CABECERA);
              //[...]
        }
        pthread_join(helper, &status);
        //[...]
}
```

La función encargada de realizar esta tarea simplemente tenía que hacer la llamada a la función de codificación encargada de hacer el trabajo.

```
void RaptorEncoderWorker::help(){
	//[...]
	DFR11EncGenIntermediateBlock(encoder, TAREA_ENC_RAPTOR);
	//[...]
}
```

5.9.- Decodificación en paralelo

Otra de las implementaciones realizadas sobre el diseño básico fue la inclusión de la posibilidad de decodificar varios bloques en paralelo. Es decir, se pueden recibir y alojar símbolos de varios bloques diferentes para su posterior decodificación. En este caso, las clases se mapean con el diseño del prototipo 6 de la siguiente manera: *DecoderBuffer* se corresponde con el envoltorio general de la capa, la estructura etiquetada como *Decoder* se implementó mediante la clase *RaptorDecoderWorker*, y el *Buffer* interno en la clase decodificadora se implementó como la clase *BlockingWorkerBuffer*.

Ahora pasamos a presentar la primera de las tres clases: *DecoderBuffer.* Al igual que la clase codificadora, también se creó para que diera la imagen de un *buffer* al resto de la librería. Por un método entran símbolos y por el otro salen bloques de datos decodificados.

```
class DecoderBuffer {

public:
	DecoderBuffer(SimplerBuffer* recepcionAplicacion,
			CUDT* libreria,
			int tCarga);
	DecoderBuffer(SimplerBuffer* recepcionAplicacion,
			CUDT* libreria,
			int tCarga,
			int nBloques);
	virtual ~DecoderBuffer();

	int writeData(char* data);

	void close();
	void remove(int bloque);
	int getRcvDataSize();
private:
	void inicializarSemaforos();
	void inicializar(SimplerBuffer* aE, CUDT* l, int tC, int nB);

private:
	//Atributos para la gestión de los items de decodificación
	int nBloques;

	RaptorDecoderWorker** bloques;
	int slotsOcupados;
	int libre;
	int cabeza;
	int siguienteBloque;
	int primerBloque;
	bool closing;

	int tCarga;
```

```
	//Atributos para la sincronización
	pthread_mutex_t m_BufLock;
	pthread_mutex_t m_StreamLock;
	pthread_mutex_t m_WaitInfoLock;
	pthread_cond_t m_StreamCond;
	pthread_cond_t m_WaitInfoCond;

	//Conexión con el resto de la librería
	SimplerBuffer* recepcionAplicacion;
	CUDT* libreria;

	//[...]
};
```

Para dar la imagen de un *buffer*, la clase presenta las funciones de escritura y consulta de tamaño. La lectura, tal y como se ha diseñado la aplicación, no es necesario que se implemente en este caso. Los atributos que contiene son los necesarios para gestionar la lista de decodificadores para los bloques cuya decodificación está en curso y para gestionar la concurrencia de acceso al *buffer*.

La función para escribir los datos es muy sencilla, simplemente se debe comprobar en el mensaje para qué bloque es y redirigirlo hacia él.

```
int DecoderBuffer::writeData(char* data){
	for(int i = 0; i < nBloques; i++){
		if(this->bloques[i]->getBlockNumber() == *(((int*)data)+2) ){
			this->bloques[i]->writeData(data);
		}
	}
	return 0;
}
```

También es muy sencilla la función de eliminar un bloque una vez que se ha decodificado correctamente. Cada bloque decodificado será el encargado de eliminarse a sí mismo y cargar otra estructura par decodificar el siguiente bloque. Para ello invocará este método.

```
void DecoderBuffer::remove(int bloque){
	for(int i = 0; i < nBloques; i++){
		if(this->bloques[i]->getBlockNumber() == bloque ){
			bloques[i] = new RaptorDecoderWorker(recepcionAplicacion,
                                                             this,
                                                             libreria,
                                                             tCarga,
                                                             siguienteBloque);
			bloques[i]->start();
			siguienteBloque++;
		}
	}
}
```

La siguiente clase a comentar es *RaptorDecoderWorker*. Esta clase realiza la decodificación de un solo bloque de datos.

```
class RaptorDecoderWorker {
public:
	RaptorDecoderWorker();
	RaptorDecoderWorker(SimplerBuffer* aplicacion, DecoderBuffer* decoder,
CUDT* l, int tCarga, int bloque);
	virtual ~RaptorDecoderWorker();

	void start();
	void run();

	void close();

	void writeData(char* data);
	int getBlockNumber();
private:
	unsigned char* datos;
	BlockingWorkerBuffer* workerBuffer;

	int dataLength;
	int symbolSize;
	int nBloque;
	int tCarga;

	SimplerBuffer* recepcionAplicacion;
	bool closing;

	pthread_t worker;
	DecoderBuffer* decoder;
	CUDT* udt;
};
```

Esta clase contiene los métodos necesarios para la inicialización y lanzamiento del hilo que realiza la decodificación. Además, dado que el *buffer* donde se almacenan los símbolos se gestiona internamente, también ofrece un método para escribir los símbolos.

El único método relevante de esa clase es el de decodificación, implementado en el método *run()*, que es el encargado de ir generando los símbolos. Este método es similar al presentado en puntos anteriores, adaptado a las nuevas estructuras, por lo que no se va a comentar en detalle.

La última clase a introducir en este punto es la que representa el *buffer* interno de cada *RaptorDecoderWorker*: *BlockingWorkerBuffer*.

```
class DecoderBuffer {

public:
	DecoderBuffer(SimplerBuffer* recepcionAplicacion,
			CUDT* libreria,
			int tCarga);
	DecoderBuffer(SimplerBuffer* recepcionAplicacion,
			CUDT* libreria,
			int tCarga,
			int nBloques);
	virtual ~DecoderBuffer();

	int writeData(char* data);

	void close();
```

```
	void remove(int bloque);
	int getRcvDataSize();
private:
	void inicializarSemaforos();
	void inicializar(SimplerBuffer* aE, CUDT* l, int tC, int nB);

private:
	int nBloques;

	RaptorDecoderWorker** bloques;
	int slotsOcupados;
	int libre;
	int cabeza;
	int siguienteBloque;
	int primerBloque;
	bool closing;

	int tCarga;

	pthread_mutex_t m_BufLock;
	pthread_mutex_t m_StreamLock;
	pthread_mutex_t m_WaitInfoLock;
	pthread_cond_t m_StreamCond;
	pthread_cond_t m_WaitInfoCond;

	SimplerBuffer* recepcionAplicacion;

	CUDT* libreria;

	int msgNo;
};
```

Esta clase es similar a la ya descrita *WorkerBuffer*. Sin embargo, ambos hilos de lectura y escritura se detienen en este *buffer* si está vacío o lleno respectivamente mientras que en el *buffer* anterior solo lo hacía el de escritura.

5.10.- Implementación del protocolo avanzado

El protocolo avanzado permite enviar símbolos de varios bloques de acuerdo a lo descrito en puntos anteriores. Para conseguir esto hay que hacer una serie de modificaciones en el codificador que serán descritas a continuación.

Para almacenar la información de estado que no puede ser comprobada directamente se ha creado el atributo *status* que va variando en función de los eventos que se producen. Algunos eventos pueden cambiar *status* de una manera directa, como puede ser el reconocimiento de un bloque. Para conseguir esto solo hay que incluir, como se muestra a continuación una línea más en la función *nextBlock()*.

```
void EncoderBuffer::nextBlock(int id) {
	//[...]
	primerBloque++;
	status--;   //Línea añadida
```

```
	pthread_mutex_lock(&m_WaitInfoLock);
	//[...]
	}
}
```

Sin embargo, hay otros casos en los que no es posible detectar el evento inmediatamente. Este ocurre para la detección de la condición “se acaban los símbolos fuente”. Esto no se puede detectar cuando se están generando los símbolos, ya que hay un *buffer* de por medio que supondría una imprecisión en la información. Por tanto,una manera muy sencilla de hacerlo es incluir en el código un símbolo con un identificador, como si una violación de código se tratara, para detectar cuándo pasa por el *buffer* y poder actualizar el estado correctamente. A este símbolo especial se le llamará burbuja. Este símbolo se generará en el codificador una vez que se hayan metido en el *buffer* todos los símbolos fuente, y se usará como separador entre los dos tipos de símbolos.

```
void RaptorEncoderWorker::run() {
	//[...]

	for (int id = 0; id < nSourceSymbols; id++) {
		//[...]
		workerBuffer->addData(paquete - CABECERA);
		//[...]
	}

	/********* Bloque de líneas de código añadidas **********/
	char * paquete = new char[tCarga];
	int * burbuja = (int *)paquete;
	*burbuja = -1;
	workerBuffer->addData(paquete);
	delete[] paquete;
	/*******************************************************/

	result = (DFR11EncError)
			DFR11EncGenIntermediateBlock(encoder,TAREA_ENC_RAPTOR);

	//[...]
}
```

En este punto, ya solo queda modificar el método que suministra los métodos para que, teniendo en cuenta el estado actual de la aplicación, decida de qué bloque tiene que sacar el símbolo a enviar.

```
int EncoderBuffer::readData(char** data, int32_t& msgno) {
	CGuard::enterCS(m_BufLock);

	bool bubble = true;
	int leido = 0;
```

Cuando se procede a sacar un símbolo del *buffer*, esta operación se realizará hasta que tengamos un símbolo válido, es decir, no sea una burbuja.

```
	while (bubble){
		leido = 0;
		if(cabeza != -1){
```

Si algún bloque se está codificando (`cabeza != -1`) comprobamos si aún no se han acabado

los datos fuente del primer bloque (`status == 0`), o ya se han acabado los datos fuente del segundo bloque (`status == 2`). Si es así, seleccionamos un símbolo del primero (o único) de los bloques para enviarlo.

```
                if(status != 1){
                        leido = bloques[cabeza]->readData(data);
```

Si no es así, es porque ya se han acabado los símbolos fuente del primer bloque y no del segundo, o bien, no hay segundo bloque. Si estamos en el primer caso, se seleccionará el bloque de acuerdo al ratio definido para el nuevo protocolo. Para gestionar esta información se usa la variable *packetCounter*, que cuando sea 0 implicará el envío de un paquete del primer bloque, y se enviarán paquetes del segundo bloque en caso contrario. Si estamos en el segundo caso, simplemente enviamos paquetes del bloque que tenemos disponible.

```
                } else if(status == 1){
                        if(packetCounter != 0 && slotsOcupados >= 2){
                           leido =
                               bloques[(cabeza + 1) % nBloques]->readData(data);
                        } else {
                           leido = bloques[cabeza]->readData(data);
                        }
                        packetCounter = (packetCounter + 1) % newBlockRate;
                }
        }

        bubble = false;
```

Si el paquete que acabamos de leer es una burbuja, entonces no debemos enviarlo, sino que tenemos que actualizar la variable de estado e impedir que la ejecución salga del bucle. Para ello comprobamos si el el identificador del símbolo es -1.

```
        int aux = 0;
        if(leido > 0){
                aux = *((int*)*data);
                if(aux == -1){
                        status++;
                        bubble = true;
                }
        }
    }

    CGuard::leaveCS(m_BufLock);
    return leido;
}
```

5.11.- Filtrado

Para conseguir tener información lo más precisa posible del ancho de banda disponible en el canal, se han realizado una serie de modificaciones tanto en el receptor, que es el encargado de calcular el ancho de banda en base a los paquetes recibidos, y en el emisor, encargado de decidir en cada momento la tasa de transmisión.

En el receptor se hicieron dos modificaciones principales. La primera de ellas fue cambiar el

momento en el que se leían los tiempos de llegada de paquetes. En versiones previas, esta operación se llevaba a cabo cuando se trataban los paquetes dentro de la librería, con lo que había una falta de precisión en los cálculos al no tener en cuenta el tiempo que los paquetes llevaban en el *buffer* interno del *kernel*. Para solucionar esta pérdida de precisión, se hizo uso de la característica de los sockets en linux que permite ser informado del momento en el que el paquete llegó al sistema operativo. El código que se incluyó para hacer uso de esta característica es el siguiente:

```
void CChannel::setUDPSockOpt()
{
      //[...]
      //timestamping
      int timestampOn = 1;
      if(0 != setsockopt(m_iSocket, SOL_SOCKET, SO_TIMESTAMP,
                         (int *) &timestampOn, sizeof(timestampOn)))
        perror("ERROR ");
      //[...]
}

int CChannel::recvfrom(sockaddr* addr, CPacket& packet, struct timeval *
time_kernel) const
{
      //[...]
      msghdr mh;

      //[...]

      int res = recvmsg(m_iSocket, &mh, 0);

      if (cmsg->cmsg_level == SOL_SOCKET &&
          cmsg->cmsg_type  == SCM_TIMESTAMP &&
          cmsg->cmsg_len   == CMSG_LEN(sizeof(struct timeval))){

              memcpy(time_kernel, CMSG_DATA(cmsg), sizeof(struct timeval));
        }

      //[...]
}
```

La segunda modificación implementada a este nivel ha sido el cambio en el temporizador descrito anteriormente. La librería comprueba los temporizadores cada vez que se recibe un paquete, y si uno de ellos ha expirado, realiza la acción correspondiente. La modificación implementada inhibe a la librería de dar la ráfaga como terminada si no se ha recibido al menos la mitad de los paquetes de la ráfaga.

```
void CUDT::checkTimers()
{
   //[...]

   if(useRaptor){

         if(currtime > m_iExpiraTimerControlFlujoRDT &&
            m_iExpiraTimerControlFlujoRDT != 0 &&
            m_iNumeroPaquetesFormanGrupo / 2 < m_iPaquetesDeGrupoRecibidos){
               informePeriodicoTerminado();
         }
   }
   //[...]
```

```
}
```

Del lado del emisor, se han implementado una serie de filtros que eliminen las imprecisiones en las muestras recogidas por el receptor. Se han definido dos tipos de filtros: los filtros exponenciales y los filtros de mediana.

El filtrado se realiza cada vez que se recibe una nueva notificación del ancho de banda detectado en el canal. Para eso, se ha modificado el método *setBandwidth()* en la clase ccc de manera que se realice el filtrado correctamente en cada clase. A continuación aparece el código para los dos tipos de filtro.

```
//Filtrado exponencial.
void CCC::setBandwidth(const int& bw){
   m_iBandwidth = (int) (m_dgamma * ((double) bw) +
                         (1 - m_dgamma) * ((double) m_iBandwidth));
}

//Filtrado de mediana
void CCC::setBandwidth(const int& bw){
   m_iBandwidth = filtro->filter(bw);
}
```

Como el filtrado de mediana requiere un procesado más completo, se creó la clase Filter, que encapsulaba toda la lógica necesaria para realizar el filtrado.

```
class Filter {
public:
      Filter();
      virtual ~Filter();

      int filter(int data);

private:
      int datos[N_DATOS];
      int nDatos;
      int last;

};
```

Esta clase proporciona un método (filter()) que realiza el filtrado de N_DATOS, usando el valor que se pasa como parámetro a la función y los N_DATOS - 1 valores recibidos anteriormente. Se va a mostrar ahora un par de porciones de código que muestran el filtado contenido en la función *filter*.

```
int Filter::filter(int data){
      //[...]
      if(datos[0] >= datos[1] && datos [0] >= datos[2]){
            if(datos[1] >= datos[2]){
                  return datos[1];
            } else {
                  return datos[2];
            }
      } else if (datos[0] <= datos[1] && datos[0] <= datos[2]){
            if(datos[1] <= datos[2]){
                  return datos[1];
```

```
            } else {
                  return datos[2];
            }
      } else {
            return datos[0];
      }
      //[...]
}
```

Esta primera porción representa un filtrado sencillo para tres elementos. Calcula la mediana sin ejecutar un algoritmo de ordenación ni búsqueda, haciendo uso de comparaciones sencillas.

```
int Filter::filter(int data){
      //[...]
      //Se crea la copia de trabajo
      int * copiaDatos = new int[N_DATOS];
      for (int i = 0; i < N_DATOS; i++){
            copiaDatos[i] = datos[i];
      }
      //Se recorren n medias veces el vector machacando
      //los n medios mayores valores
      for (int i = 0; i < N_DATOS/2; i++){
            int maxPos= -1;
            int maxVal = -1;
            for(int j = 0; j < N_DATOS; j++){
                  if(copiaDatos[j] > maxVal){
                        maxPos = j;
                        maxVal = copiaDatos[j];
                  }
            }
            copiaDatos[maxPos] = -1;
      }
      //Del resto de valores que quedan se busca el mayor
      //y ya tenemos la mediana
      int maxPos= -1;
      int maxVal = -1;
      for(int j = 0; j < N_DATOS; j++){
            if(copiaDatos[j] > maxVal){
                  maxPos = j;
                  maxVal = copiaDatos[j];
            }
      }
      delete [] copiaDatos;
      return maxVal;
      //[...]
}
```

En esta porción ya se calcula la mediana para cualquier número de elementos, pero para ahorrar la ordenación de todos los elementos, simplemente se realiza una serie de búsquedas sucesivas hasta encontrar el valor de la mediana.

Como este código va a trabajar con tallas del problema muy pequeñas, no se ha buscado la eficiencia computacional hasta el extremo sino que se ha implementado una solución simplemente efectiva.

6.- Validación y pruebas

6.1.- Validación

Para comprobar el buen funcionamiento de la librería y estimar la ventaja de este nuevo sistema de envío. se ha hecho el conjunto de pruebas que se detalla a continuación.

En una primera fase, se realizaron pruebas de funcionamiento cuyo único propósito eran comprobar que los datos enviados llegaban correctamente al destino. Para ello se creó un pequeño programa de pruebas, basado en los propios ejemplos suministrados en la documentación de la librería UDT.

```
/*
 * cliente.cpp
 *
 */
//[...]
int main() {
      UDTSOCKET client = UDT::socket(AF_INET, SOCK_STREAM, 0);
      //[...]
      if (UDT::ERROR ==
          UDT::connect(client,(sockaddr*)&serv_addr,sizeof(serv_addr))){
        cout << "connect: " << UDT::getlasterror().getErrorMessage();
        return 0;
      }
      char cadena[100000];
      scanf("%s", cadena);
      while(cadena[0] != '\\'){
            cout << "send: " << UDT::getlasterror().getErrorMessage();
            if (UDT::ERROR ==
                UDT::send(client, cadena, strlen(cadena) + 1, 0)){
              return 0;
            }
            scanf("%s", cadena);
      }
      UDT::close(client);
      return 1;
}

/*
 * servidor.cpp
 *
 */
//[...]
int main() {
      UDTSOCKET serv = UDT::socket(AF_INET, SOCK_STREAM, 0);
      //[...]
      if (UDT::ERROR ==
          UDT::bind(serv, (sockaddr*)&my_addr, sizeof(my_addr))) {
        cout << "bind: " << UDT::getlasterror().getErrorMessage();
        return 0;
      }
      UDT::listen(serv, 10);
      //[...]
      UDTSOCKET recver =
```

```
                        UDT::accept(serv, (sockaddr*)&their_addr, &namelen);
        //[...]
        while (UDT::ERROR != UDT::recv(recver, data, 200, 0)) {
          cout << data << endl;
        }
        UDT::close(recver);
        UDT::close(serv);

        return 1;
}
```

En este programa un servidor espera la conexión de un cliente. Una vez establecida esta conexión, el cliente le envía una cadena de caracteres al servidor y éste la imprime por pantalla. Con esta pareja de programas se pretendía comprobar el buen funcionamiento de la implementación en los casos más básicos.

Una vez comprobado el buen funcionamiento con estos programas y superadas todas las pruebas, se utilizó una pareja de programas de ejemplo con los que se contaba en la documentación de la que se disponía.

```
/*
 * recvfile.cpp
 *
 */
//[...]
int main(int argc, char* argv[])
{
   //[...]
   UDT::startup();
   UDTSOCKET fhandle = UDT::socket(AF_INET, SOCK_STREAM, 0);
   //[...]
   if (UDT::ERROR ==
       UDT::connect(fhandle, (sockaddr*)&serv_addr, sizeof(serv_addr))){
      cout << "connect: "
      cout << UDT::getlasterror().getErrorMessage() << endl;
      return 0;
   }

   // send name information of the requested file
   // Longitud del nombre
   int len = strlen(argv[3]);
   if (UDT::ERROR == UDT::send(fhandle, (char*)&len, sizeof(int), 0)){
      cout << "send: " << UDT::getlasterror().getErrorMessage() << endl;
      return 0;
   }
   // Nombre
   if (UDT::ERROR == UDT::send(fhandle, argv[3], len, 0)){
      cout << "send: " << UDT::getlasterror().getErrorMessage() << endl;
      return 0;
   }

   // get size information
   int64_t size;
   if (UDT::ERROR ==
       UDT::recv(fhandle, (char*)&size, sizeof(int64_t), 0)){
      cout << "recv: " << UDT::getlasterror().getErrorMessage() << endl;
      return 0;
   }
```

```
   // receive the file
   fstream ofs(argv[4], ios::out | ios::binary | ios::trunc);
   int64_t recvsize;
   if (UDT::ERROR == (recvsize = UDT::recvfile(fhandle, ofs, 0, size))){
      cout << "recvfile: "
      cout << UDT::getlasterror().getErrorMessage() << endl;
      return 0;
   }


   UDT::close(fhandle);
   ofs.close();
   UDT::cleanup();

   return 1;
}

/*
 * sendfile.cpp
 *
 */
//[...]
int main(int argc, char* argv[])
{
   //[...]
   UDT::startup();
   UDTSOCKET serv = UDT::socket(AF_INET, SOCK_STREAM, 0);
   //[...]
   if (UDT::ERROR ==
       UDT::bind(serv, (sockaddr*)&my_addr, sizeof(my_addr))){
      cout << "bind: " << UDT::getlasterror().getErrorMessage() << endl;
      return 0;
   }
   //[...]
   UDT::listen(serv, 1);
   //[...]
   if (UDT::INVALID_SOCK ==
      (fhandle = UDT::accept(serv, (sockaddr*)&their_addr, &namelen))){
      cout << "accept: "
      cout << UDT::getlasterror().getErrorMessage() << endl;
      return 0;
   }

   UDT::close(serv);

   // aquiring file name information from client
   char file[1024];
   int len;
   if (UDT::ERROR == UDT::recv(fhandle, (char*)&len, sizeof(int), 0))
   {
      cout << "recv: " << UDT::getlasterror().getErrorMessage() << endl;
      return 0;
   }
   if (UDT::ERROR == UDT::recv(fhandle, file, len, 0))
   {
      cout << "recv: " << UDT::getlasterror().getErrorMessage() << endl;
      return 0;
   }
```

```
        file[len] = '\0';

        // open the file
        fstream ifs(file, ios::in | ios::binary);
        //[...]
        // send file size information
        if (UDT::ERROR ==
            UDT::send(fhandle, (char*)&size, sizeof(int64_t), 0)){
           cout << "send: " << UDT::getlasterror().getErrorMessage() << endl;
           return 0;
        }
        //[...]
        // send the file
        if (UDT::ERROR == UDT::sendfile(fhandle, ifs, 0, size)){
           cout << "sendfile: "
           cout << UDT::getlasterror().getErrorMessage() << endl;
           return 0;
        }
        //[...]
        UDT::close(fhandle);
        ifs.close();
        UDT::cleanup();

        return 1;
    }
```

Esta pareja de programas nos permite realizar pruebas más complejas, ya que alterna el envío de valores grandes y pequeños. En este caso, se produce un esquema clásico de transferencia de ficheros, en el que un cliente se conecta a un servidor y le envía el nombre del fichero que desea recibir. Una vez que el servidor ha recibido esta información, envía al cliente el tamaño del fichero y comienza la transferencia.

La comprobación del correcto funcionamiento de la transferencia de ficheros se realizó calculando el *md5* del fichero fuente y el recibido, y la comparación de ambos valores arroja el resultado de que ambos ficheros eran idénticos.

Para comprobar en más detalle si el envío de las ráfagas se hacía correctamente o no se conectó el programa *wireshark* en la máquina servidora y se empezó a capturar los paquetes. Para clarificar las trazas, la tasa objetivo se dejó a un valor bajo y fijo. Luego, analizando la traza con el programa *tcpdump* y calculando el tiempo transcurrido entre paquetes, se pudieron identificar inequívocamente las ráfagas de los paquetes enviados.

```
>: tcpdump -tt -r $1 | grep "9000 >" | grep "length 1472" | awk
'BEGIN{var=0}{print (var - $1); var = $1}' | less
```

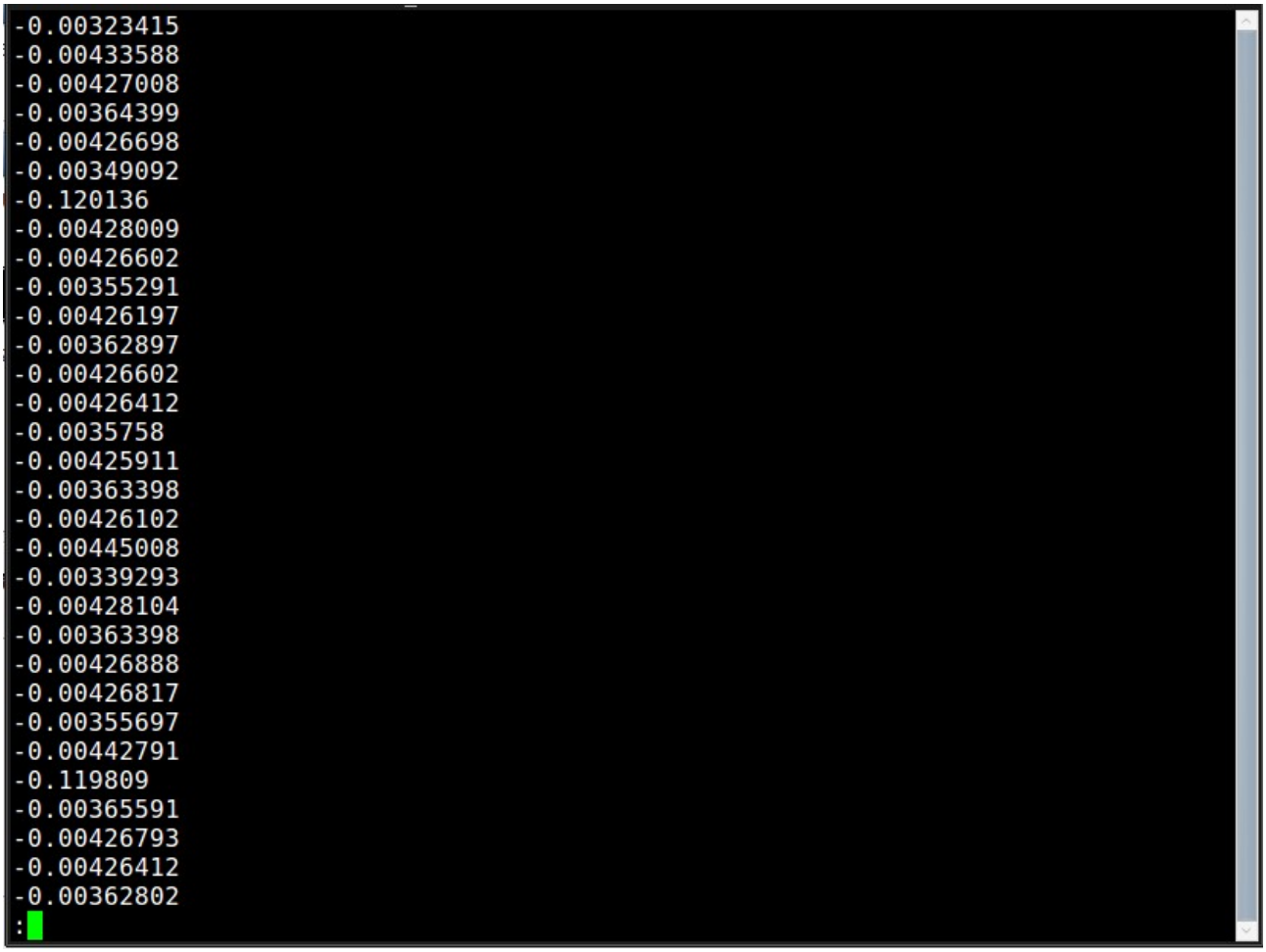

Figura 29: Diferencia temporal de emisión entre paquetes

Esta figura refleja la diferencia entre los tiempos de emisión de los paquetes de datos. Se puede ver que entre paquetes consecutivos hay una diferencia aproximada de 0,004 segundos, excepto cada veinte paquetes, que se muestra un retardo mayor. Este es el efecto que se produce en la emisión en ráfagas del sistema de control de flujo. Como se ha fijado la tasa objetivo a 100 paquetes por segundo, la diferencia temporal entre ráfagas será de 0.2 segundos y, si el alfa es de 0.4, la diferencia entre la emisión de los paquetes de la ráfaga será de 0.004. Se comprueba que se cumplen todas esas restricciones.

Así pues, verificado el buen funcionamiento de la librería y del control de flujo en todos los aspectos, se puede pasar a realizar una comparación de rendimiento entre TCP y RDT. Para ello se creó una pareja de programas de transmisión de ficheros idéntica a la anterior, salvo por el hecho de que trabajaba con TCP.

6.2.- Evaluación del rendimiento para el diseño básico

Para esta comparación de rendimiento se conectaron dos máquinas a un *switch* mediante FastEthernet y se simularon retardos y pérdidas de paquetes usando el comando *tc* de Linux, tanto en el cliente como en el servidor. Se tomaron cinco muestras por configuración. Se midieron los tiempos y se calculó el ancho de banda que se pudo obtener en el canal. Para las pruebas se usó un fichero con un tamaño de 5MB aproximadamente. Para la realización de todas esas pruebas se

escribió un *script* en lenguaje BASH que controlaba la correcta ejecución de las pruebas.

```
#!/bin/bash

tc qdisc del dev eth0 root netem
ssh root@vaio-grc-cable tc qdisc del dev eth0 root netem

for retardo in 1 2 5 10 20 50 100 200 500
do
	for error in 0 0.1 0.2 0.5 1 2 5 10 20 30
	do
		tc qdisc add dev eth0 root netem delay ${retardo}ms loss ${error}%
		ssh root@vaio-grc-cable tc qdisc add dev eth0 root netem delay \
		    ${retardo}ms loss ${error}%

		echo "TC: " $retardo ms $error % 1>&2
		for ((k=0; $k<5; k=$k+1))
		do
			echo "TCP" 1>&2
			/home/miguel/servidor_tcp 9000 &
			servidor=$!
			ssh root@vaio-grc-cable time /home/usuario-grc/RDT/cliente_tcp \
				192.168.2.100 9000 /home/miguel/UDTRaptor.zip bla.zip
			kill -9 $servidor 2>/dev/null

			echo "RDT" 1>&2
			/home/miguel/servidor &
			servidor=$!
			ssh root@vaio-grc-cable time /home/usuario-grc/RDT/cliente \
				 192.168.2.100 9000 /home/miguel/UDTRaptor.zip bla.zip
			kill -9 $servidor 2>/dev/null

		done
		tc qdisc del dev eth0 root netem
		ssh root@vaio-grc-cable tc qdisc del dev eth0 root netem
	done
done
```

Con esta metodología se han obtenido, entre otros, los resultados que se presentan a continuación de forma de gráfica. En cada una se ha mantenido fijo uno de los dos parámetros, el retardo o el porcentaje de pérdida de paquetes, y se ha variado el otro parámetro, representando en cada punto el ancho de banda que se obtuvo en el canal.

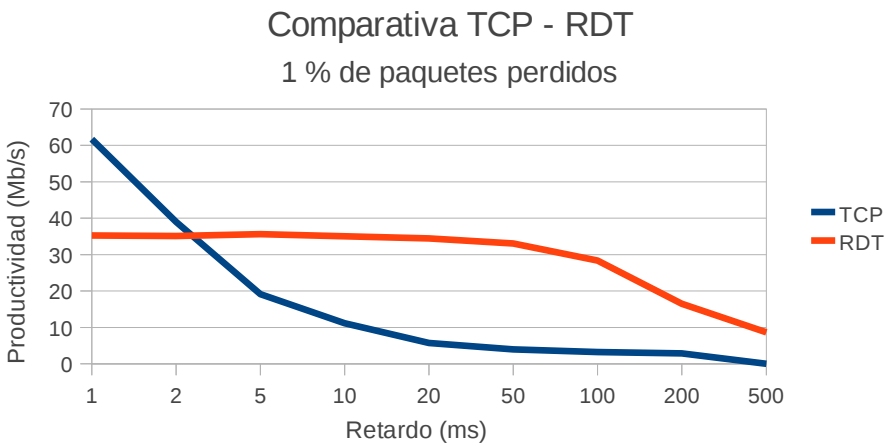

Figura 30: Comparativa TCP - RDT

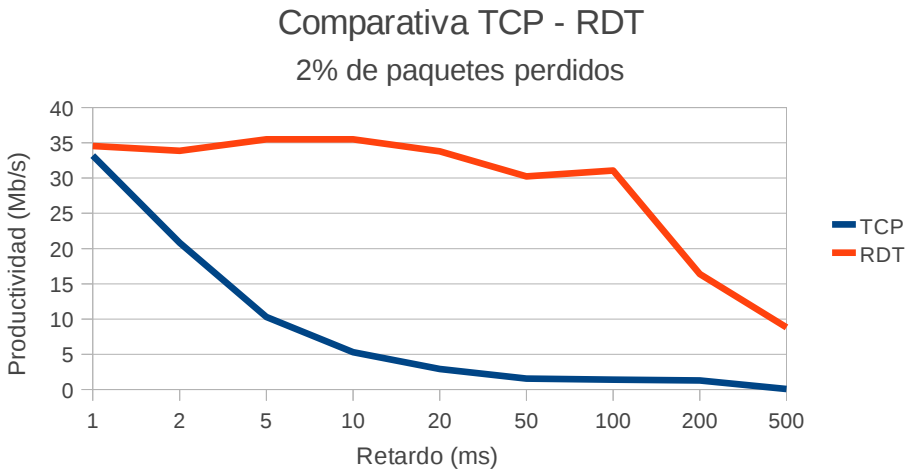

Figura 31: Comparativa TCP - RDT

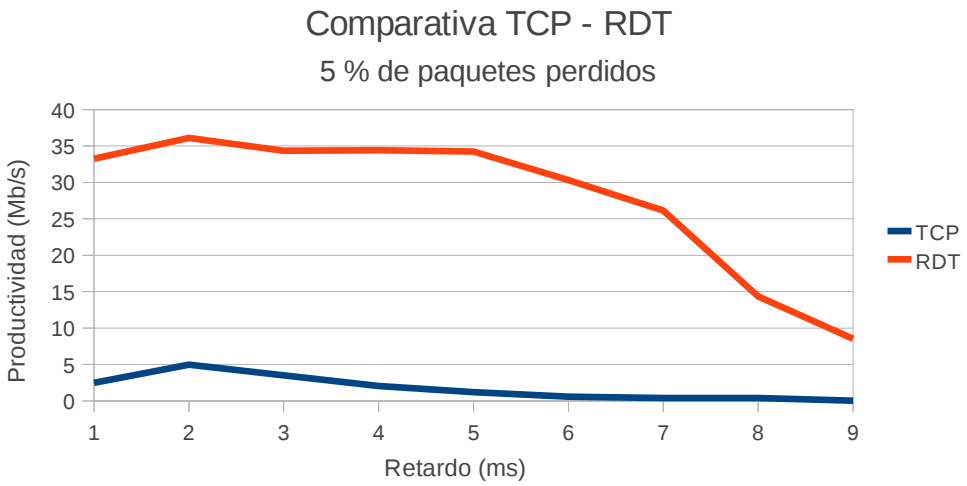

Figura 32: Comparativa TCP - RDT

En este primer trío de figuras, se ve como evoluciona la productividad de TCP y RDT conforme se varía el retardo entre fuente y destino. Se puede ver claramente que, aunque con un bajo valor de retardo y de paquetes perdidos TCP supera a RDT, RDT aguanta mucho mejor el impacto del aumento del retardo y del porcentaje de paquetes perdidos.

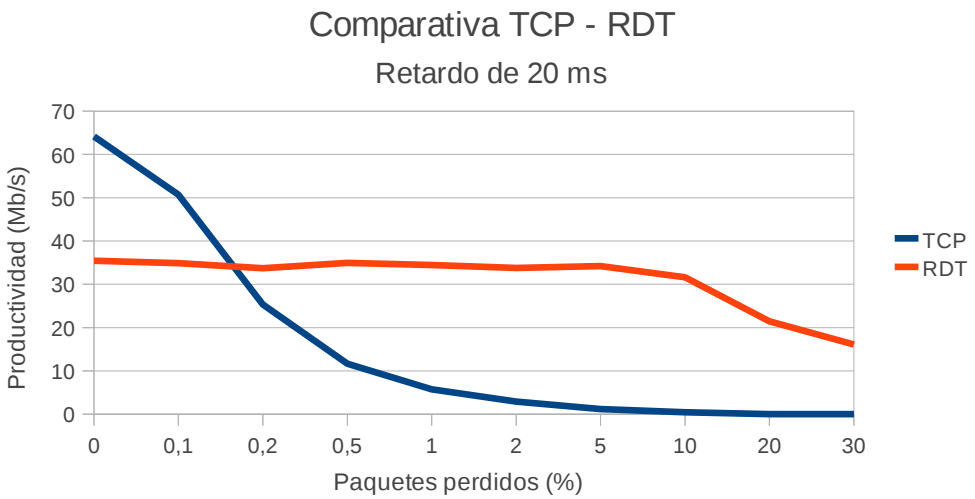

Figura 33: Comparativa TCP - RDT

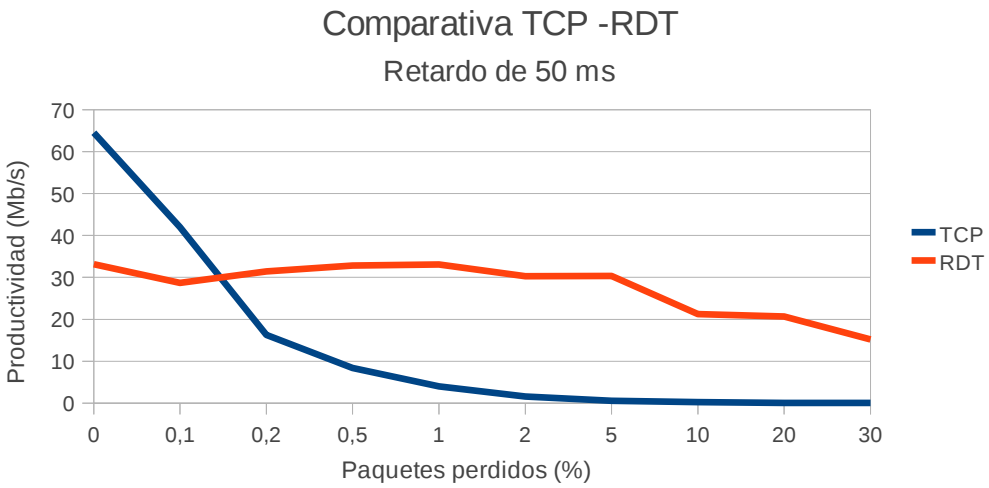

Figura 34: Comparativa TCP - RDT

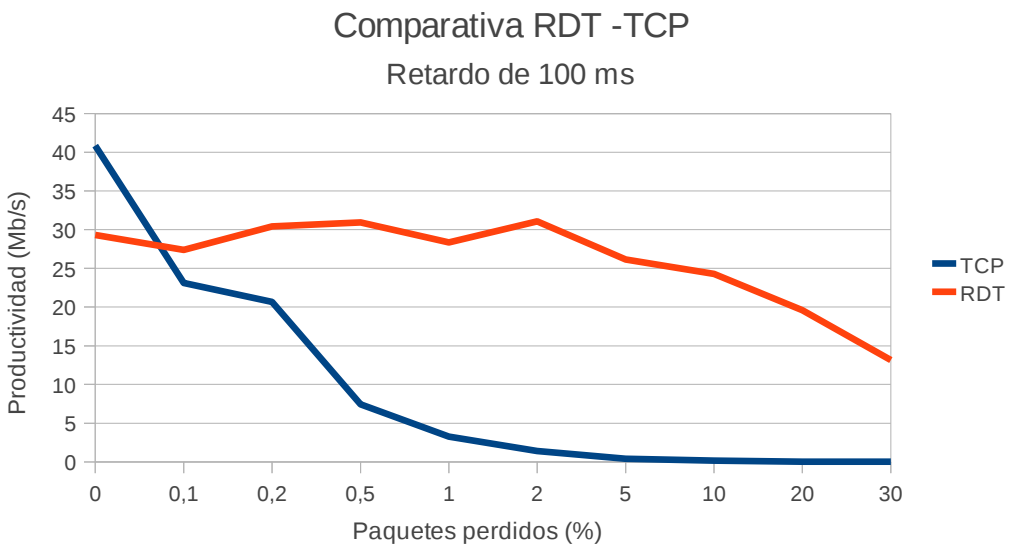

Figura 35: Comparativa TCP - RDT

En este segundo trío de imágenes se ve como evoluciona la productividad de TCP y RDT conforme se varía el porcentaje de paquetes perdidos (fijando el retardo). También se puede ver claramente que, aunque con un bajo valor de retardo y de paquetes perdidos TCP supera a RDT, RDT aguanta mucho mejor el impacto del aumento del retardo y del porcentaje de paquetes perdidos, evitando la inanición de la conexión.

6.3.- Evaluación comparativa de las mejoras propuestas

Se ha realizado un estudio de productividad de las diferentes versiones presentadas en la figura 19, obteniéndose para un fichero de aproximadamente 30 MB los siguientes resultados:

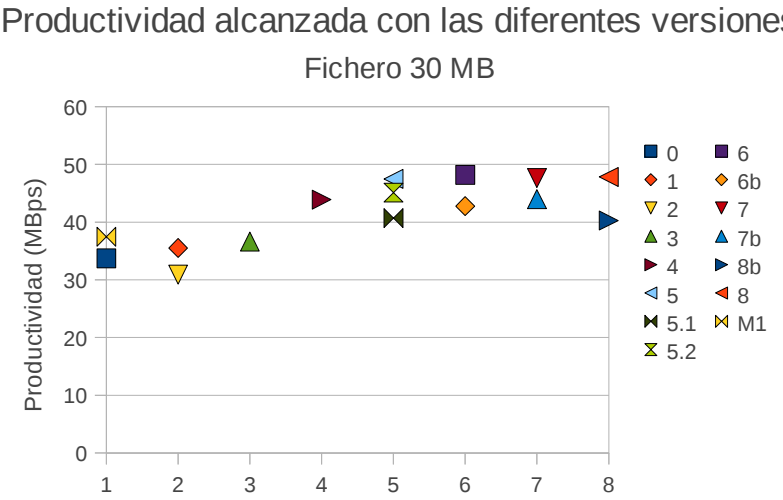

Figura 36: Productividad de las diferentes versiones presentadas

A la vista de la figura, se pueden elegir una serie de versiones como representativas para establecer la evolución que ha presentado la productividad de la librería. Las versiones clave que se han elegido, junto con el nombre que se ha seleccionado para cada una de ellas ha sido el siguiente:

Versión	**Nombre**	**Descripción**
0	RCDP	Implementación del diseño básico
M1	RCDP+BO	Implementación del diseño básico y de las mejoras que no implican añadir ningún hilo adicional.
1	RCDP+PE	Implementación del diseño básico y la codificación en paralelo.
5	RCDP+PE+BO	Implementación del diseño básico, de la codificación en paralelo y del conjunto de mejoras que no implican añadir ningún hilo adicional.
6	RCDP+PED	Implementación de RCDP+PE+BO y de la decodificación en paralelo.
6b	RCDP+PEIT	Implementación de RCDP+PED, codificando la información en un hilo independiente.
8	RCDP+PED+MB	Implementación de RCDP+PED con el funcionamiento alternativo (Sección 4.5)

Para evaluar su funcionamiento, al igual que en casos anteriores, se realizó la transmisión de un fichero de para diferentes condiciones de la red, variando un parámetro determinado en cada ocasión. La metodología de las pruebas fue la misma que en los casos anteriores.

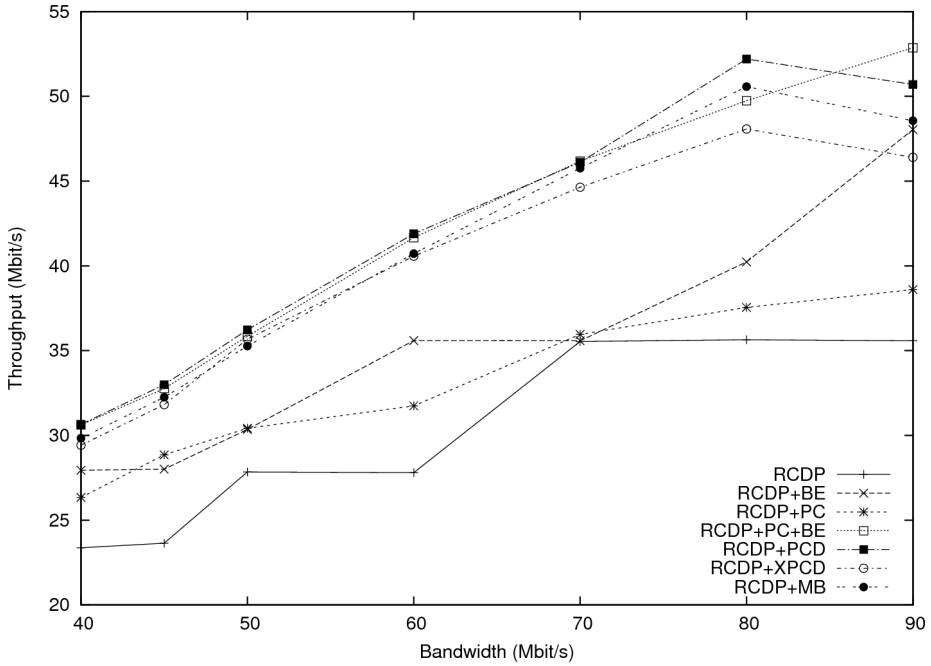

Figura 37: Ancho de banda vs. productividad en un canal con un retardo de 10 ms. (0 % de paquetes perdidos)

La figura anterior muestra la productividad lograda conforme se varía la capacidad del canal. Podemos ver que, en general, las diferentes versiones de la librería se comportan como se esperaba, mostrando un incremento lineal de la productividad conforme el ancho de banda del canal se incrementa. Se observan dos grupos claramente diferenciados: aquellos que combinan las mejoras de codificación en paralelo, y las optimizaciones base (cambio de la función *sleepTo*, el aumento del *buffer* de codificación y cambio en el envío del paquete de confirmación) y el resto. Se puede ver claramente como el primer grupo es capaz de lograr una productividad mayor que el segundo.

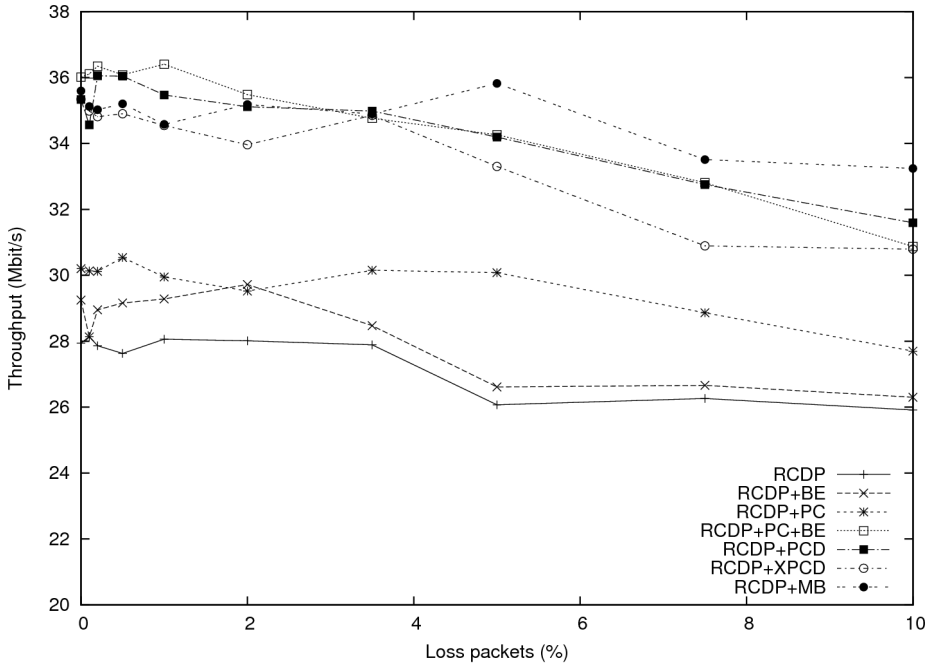

Figura 38: Tasa de pérdida de paquetes vs. productividad en un canal de 50 Mbps y 10 ms. de retardo

En esta figura podemos ver la productividad alcanzada cuando se varía la tasa de pérdidas en el canal. El comportamiento deseado sería que la productividad decreciera linealmente conforme la tasa de paquetes aumenta. Nos encontramos que, en general, todas las versiones siguen esta tendencia. Igual que en el caso anterior, también aparecen los dos grupos señalados. Además, podemos ver que la inmunidad a errores no se ve afectada por los cambios introducidos, mientras que los valores de productividad alcanzada sí.

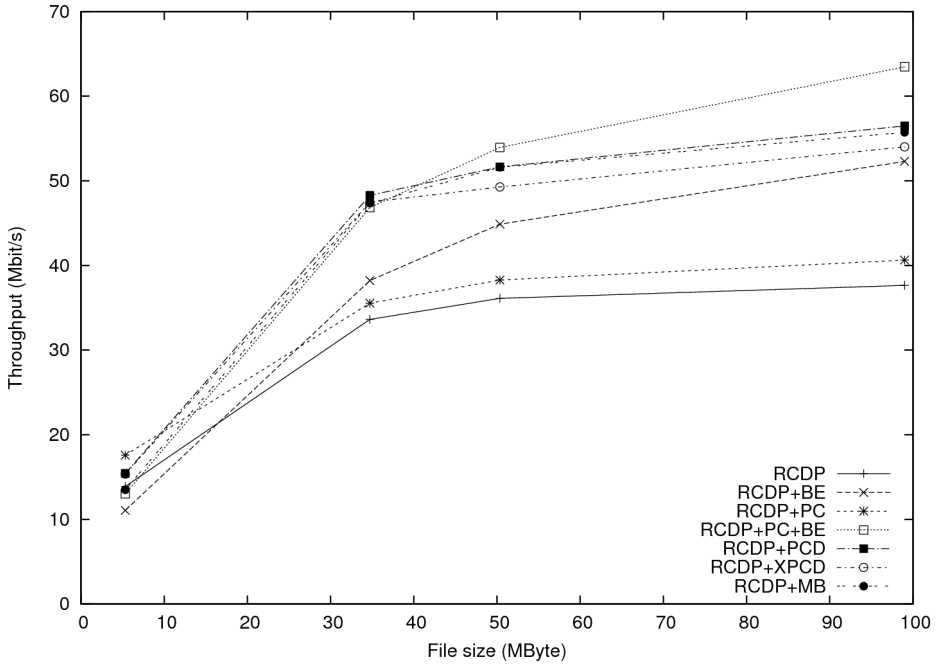

Figura 39: Tamaño de fichero vs. productividad en un canal con un retardo de 10 ms. y 0% de tasa de error

En esta figura podemos ver el nivel de productividad alcanzado cuando variamos el tamaño del fichero a enviar. Nótese que, cuando el tamaño de fichero es pequeño, la productividad media es baja debido a que la sobrecarga de inicialización es parecida al tiempo de transmisión. Para ficheros mayores, el impacto del tiempo de inicialización se reduce.

Una vez más se encuentran identificados claramente otros dos grupos. Sin embargo, las dos implementaciones más simples son más difíciles de clasificar. Esta diferencia es debida a la complejidad del software de las diferentes soluciones. En particular, la gestión de bloques de estas dos implementaciones es más ligera (menor número de hilos, menos uso de CPU, menos uso de memoria) que las otras implementaciones; característica que, en un proceso que necesita tanta CPU y memoria como es la codificación Raptor, tiene un impacto importante en términos de productividad alcanzada.

En esta figura podemos ver el número de hilos que tiene cada implementación, así como cuántos toman parte en el envío y cuántos en la recepción.

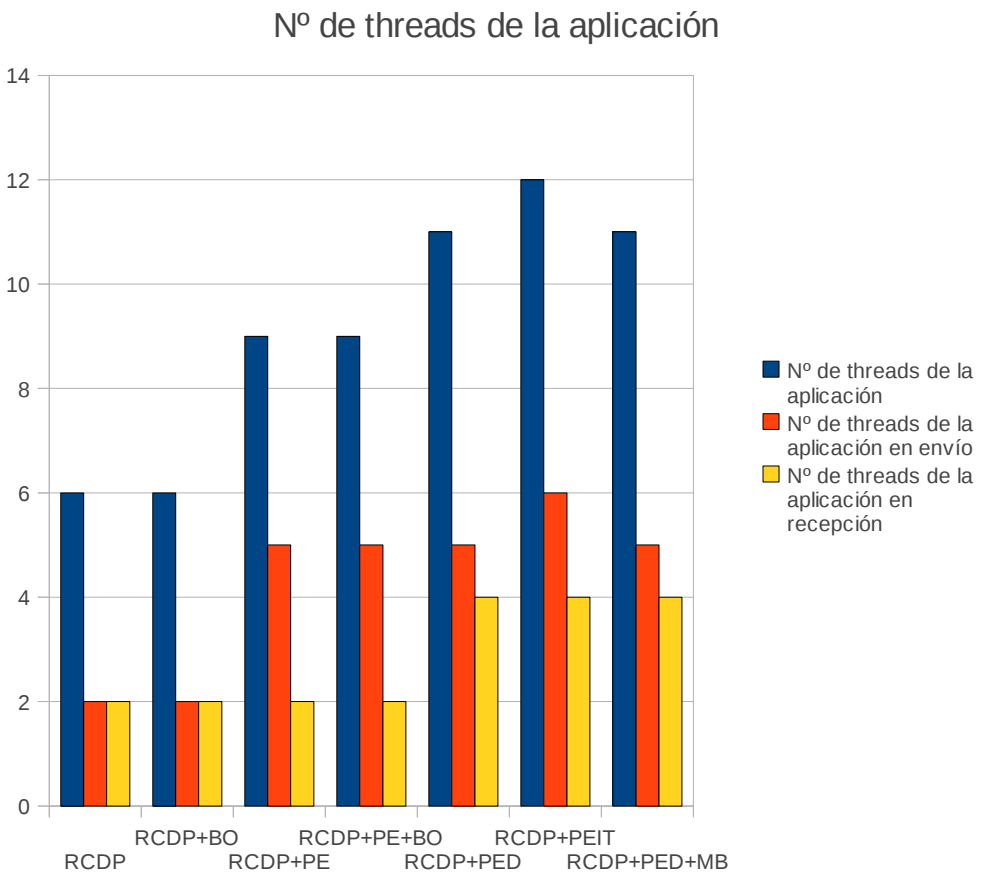

Figura 40: Número de hilos de cada versión

Un efecto claro del impacto de añadir nuevos hilos a la implementación se puede ver si se compara RCDP+PED y RCDP+PEIT. La única diferencia entre las dos implementaciones es un nuevo hilo añadido a la codificación. Este cambio reduce la productividad hasta un 5%. Además, esta aproximación necesita más RAM, lo que conlleva una pérdida de prestaciones en el acceso a memoria.

También se realizaron mediciones del consumo de CPU y memoria. Para obtener los resultados se realizaron varias mediciones usando el comando ps o usando el comando top. El script encargado de calcular esos datos era el siguiente:

```
#!/bin/bash

FICHERO=/home/miguel/UDTRaptor.zip

#CPU+RAM
top -b -d 0.5 > CPU_RAM.txt &
local_top=$!
foreign_top=`ssh root@vaio-grc-cable 'top -b -d 0.5 > CPU_RAM.txt & echo $!'`

echo "Midiendo CPU y RAM" 1>&2
for dir in pfc estable_doble_cod_1 \
mas_buff_5 doble_decod_6 doble_decod_6b final_8 \
monstruo1
do
        echo "TC: ejecutable -" $dir $FICHERO 1>&2
        for ((k=0; $k<10; k=$k+1))
        do
                echo "Inicio servidor: " `date +%s:%N` 1>&2
```

```
                ssh root@vaio-grc-cable 'echo "Inicio cliente:" `date +%s:%N`' 1>&2
                /home/miguel/RDT/pruebas/ejecutables/$dir/servidor 9002 &
                servidor=$!
                ssh root@vaio-grc-cable time /home/usuario-grc/RDT/ejecutables/$dir/cliente \
                                            192.168.2.100 9002 $FICHERO bla
                kill -9 $servidor 2>/dev/null
                echo "Fin servidor:" `date +%s:%N` 1>&2
                ssh root@vaio-grc-cable 'echo "Fin cliente:" `date +%s:%N`' 1>&2
        done
done
```

Para el procesado de los datos se usan las marcas temporales de las ejecuciones para comprobar qué muestras de *top* corresponden a qué ejecución, tanto en el cliente como en el servidor. Una vez determinado esto, se calcula la media de los valores de RAM y CPU calculados por *top* para cada ejecución.

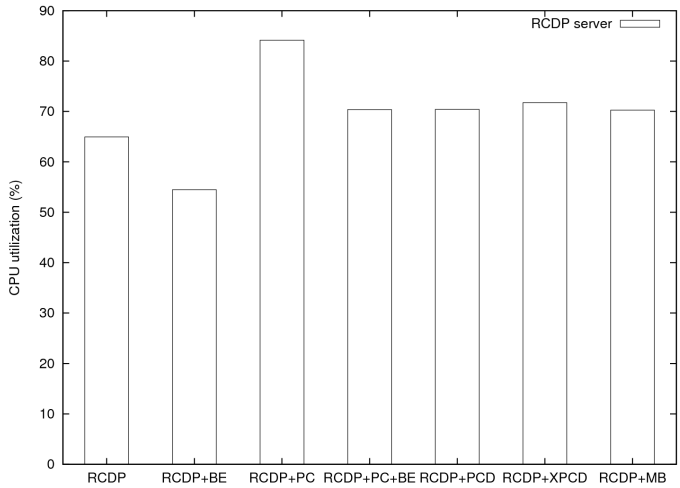

Figura 41: Consumo de CPU en un Intel Core 2 Duo a 1.80 Ghz en el servidor

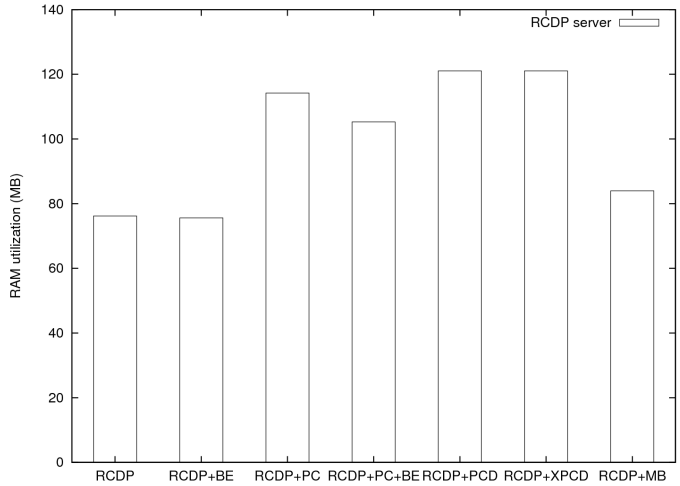

Figura 42: Uso de memoria RAM en el servidor

Las figuras anteriores muestran el consumo de recursos en el lado del servidor. Nótese que el mayor incremento de CPU ocurre cuando se adopta la codificación en paralelo, aunque es parcialmente mitigada con las posteriores modificaciones. Cuando se añaden las mejoras en la decodificación, el uso de CPU permanece estable. Si nos centramos en el uso de RAM,

encontramos que las estructuras requeridas para soportar la codificación o la decodificación en paralelo hacen que el consumo de memoria se incremente. En particular, RCDP+PED y RCDP+PEIT muestran el mayor consumo de memoria dado que tanto la codificación en paralelo como la decodificación en paralelo están implementadas en esta versión. No importa en qué extremo nos encontremos (cliente o servidor), ambos deben hacer tareas de codificación y decodificación para soportar la comunicación bidireccional.

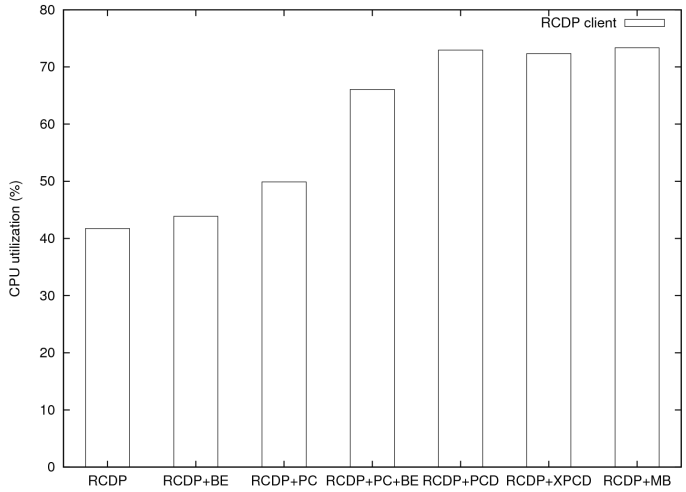

Figura 43: Consumo de CPU en un Pentium M a 1.20 GHz en el cliente

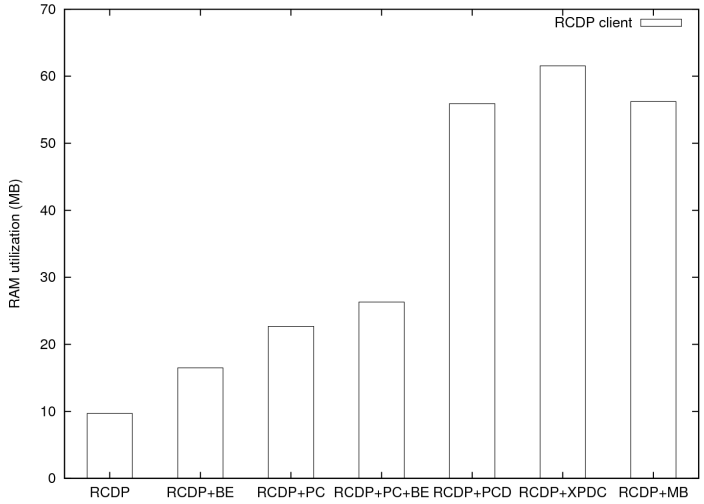

Figura 44: Uso de memoria RAM en el cliente

Las figuras anteriores muestran el consumo de recursos en el lado del cliente. Nótese que el consumo de CPU en el cliente es mucho menor comparado con la carga del servidor, aunque se incremente conforme se van introduciendo las mejoras de las que ya se ha hablado. Esta tendencia se manifiesta dado que las mejoras están enfocadas, en su mayoría, a mejorar la productividad y el uso de la CPU en el cliente. El máximo nivel de carga en la CPU es alcanzado por las soluciones que adoptan la decodificación en paralelo.

Si nos centramos en el consumo de RAM, las tras soluciones que adoptan la decodificación en paralelo tienen los requisitos de memoria más altos. En particular, RCDP+PEIT tiene los

mayores requisitos de RAM dado que introduce un hilo adicional para funciones de codificación cuyo comportamiento repercute el la RAM utilizada.

Aunque tanto cliente como servidor tengan la misma arquitectura, la asimetría asociada al proceso de codificación y decodificación permiten reducir los requerimientos del cliente comparado con el servidor, quien se ve más cargado dada la complejidad del proceso de codificación.

Si pasamos a realizar ahora la comparativa de la productividad con un proceso que se lleve a cabo la transmisión usando el protocolo TCP, se obtiene la siguiente gráfica.

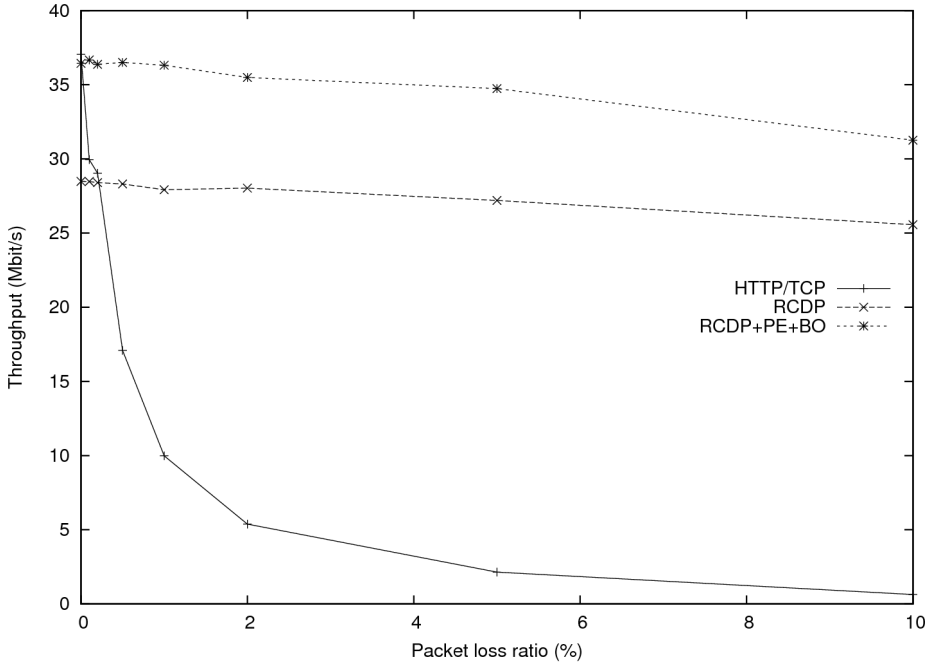

Figura 45: Comparativa con TCP

En esta figura se muestra una comparación entre un protocolo de comunicaciones que usa TCP (HTTP/TCP), el protocolo etiquetado como RCDP y el protocolo etiquetado como RCDP+PE+BO cuando variamos la tasa de pérdidas del canal. El canal se ha configurado con un ancho de banda de 50 Mbps y un retardo de 10 ms.

La implementación basada en TCP es muy sensible a la pérdida de paquetes. Conforme aumenta la pérdida de paquetes, la productividad de la solución basada en TCP cae muy rápidamente, mientras que para las versiones de nuestro protocolo el ancho de banda va permaneciendo estable (decrece como máximo un 15%). Las figuras también muestran que la optimización de nuestro protocolo logra un incremento de un 25% comparado con la implementación del diseño básico.

6.4.- Evaluación comparativa de los sistemas de filtrado

Como se ha detallado en el apartado de implementación, tres han sido las mejoras principales que se han implementado para conseguir una mayor eficiencia en el cálculo del ancho de banda disponible: uso del *timestamp* a nivel de sistema operativo, mejoras en el temporizador a introducción de filtros.

Para ver como afecta a la detección del ancho de banda del canal las diferentes técnicas implementadas, se van a mostrar gráficas en las que se muestra la evolución del ancho de banda en el canal.

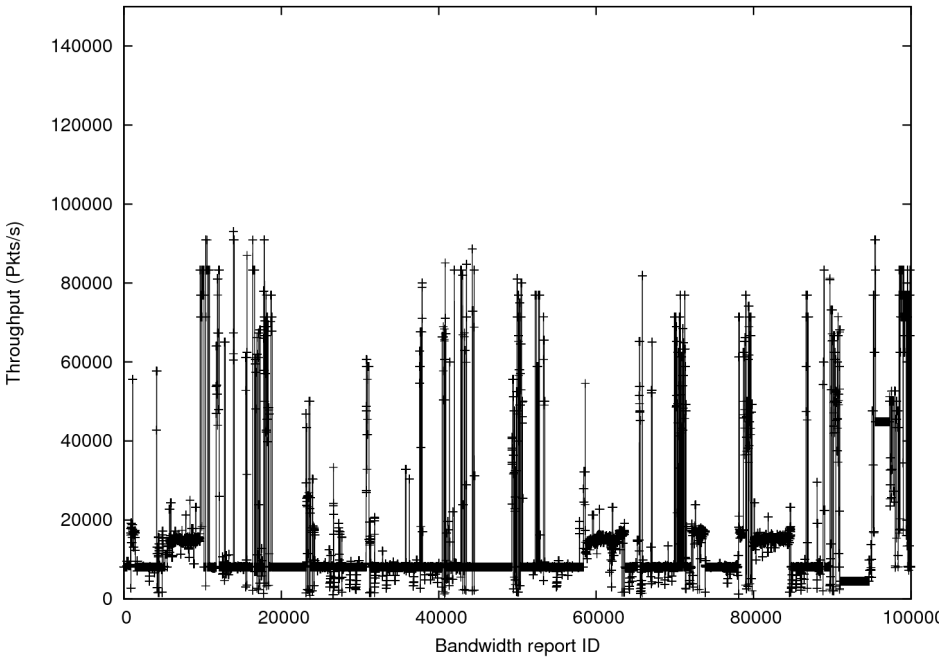

Figura 46: Ancho de banda detectado en el canal antes de aplicar ninguna técnica

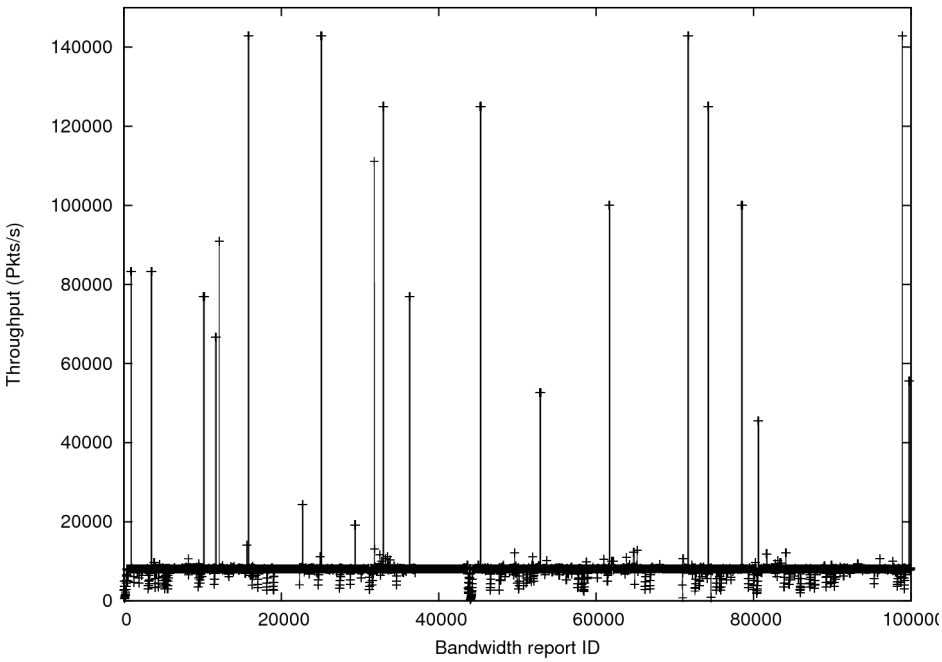

Figura 47: Ancho de banda detectado en el canal tras usar los tiempos de llegada de los paquetes al sistema operativo

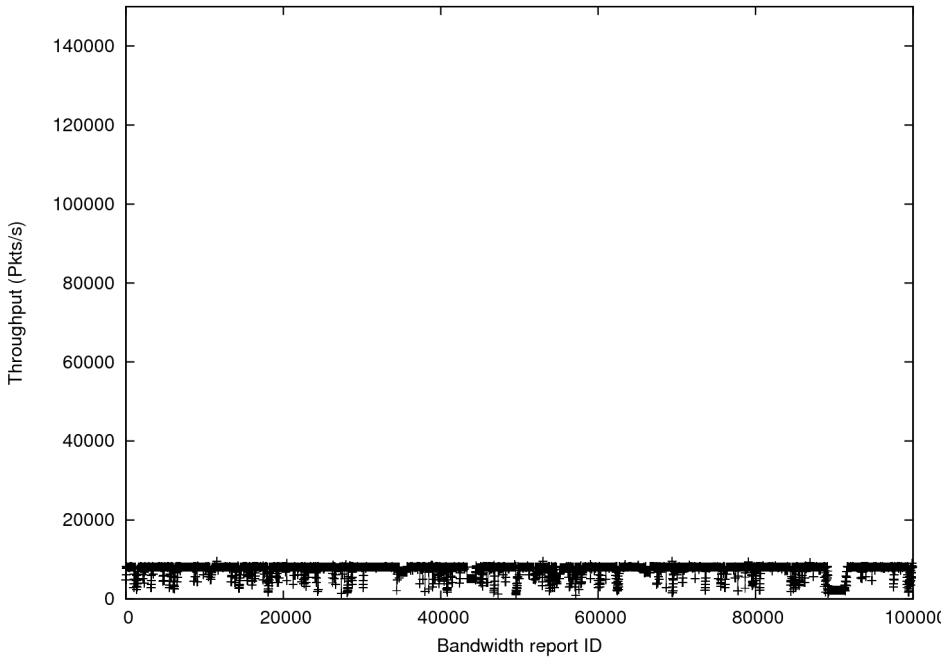

Figura 48: Ancho de banda detectado en el canal tras implementar el nuevo temporizador

Como se muestra en las imágenes, estas primeras dos variaciones mejoran la detección del ancho de banda para valores por encima del valor real del canal. Con el filtrado en el emisor se evitan problemas derivados de la estimación incorrecta de valores inferiores al nivel real del canal.

Como muestra gráfica, se aportan cuatro imágenes con las que hacer comparativas. La primera ilustra la estimación del ancho de banda sin aplicar ningún filtro de mediana y las otras tres la estimación tras la aplicación de un filtro de mediana con valor 3, 5 y 7 respectivamente.

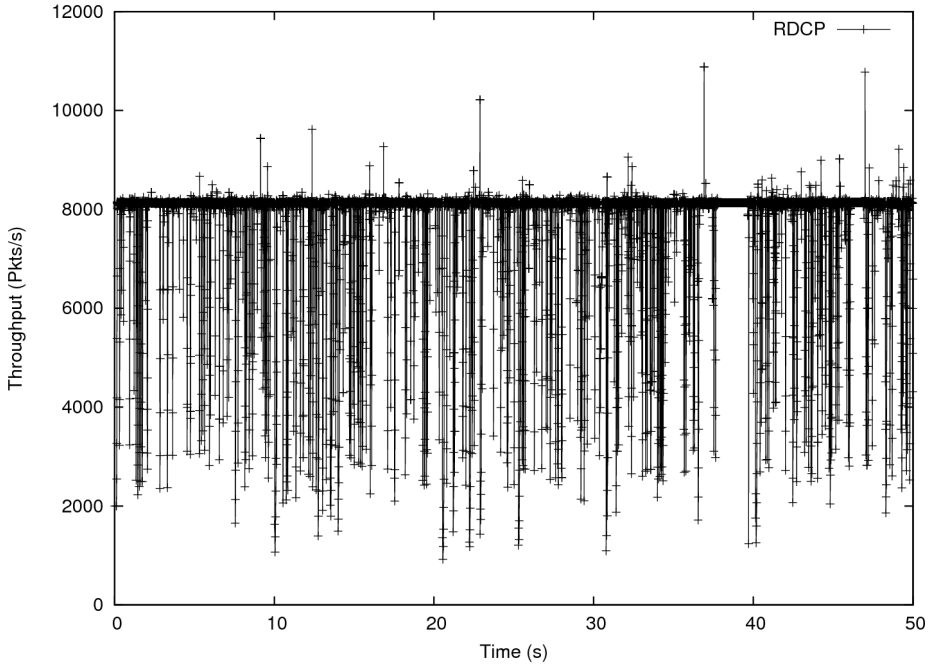

Figura 49: Ancho de banda detectado sin aplicar filtro de mediana

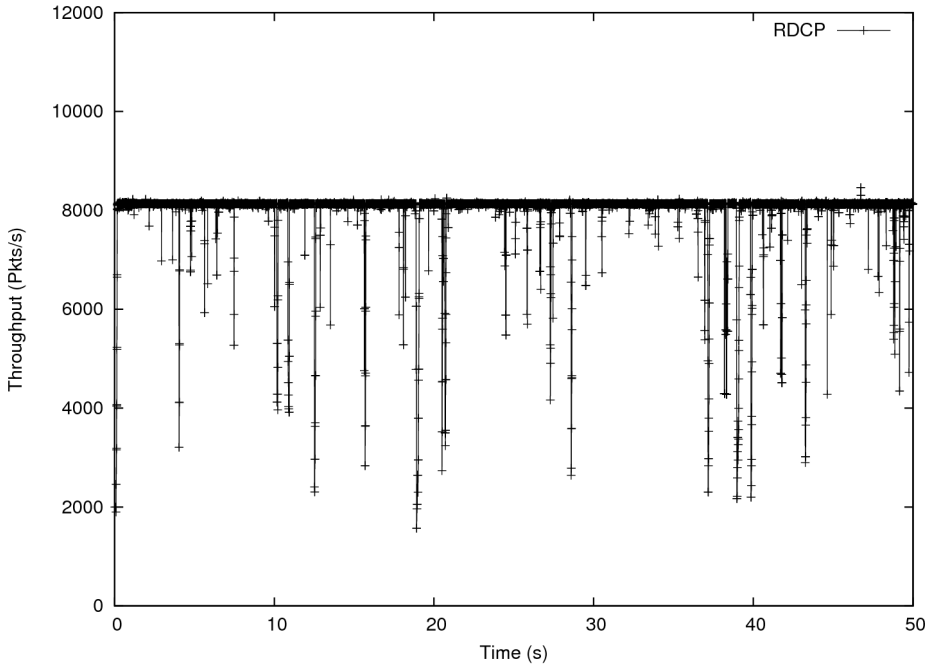

Figura 50: Ancho de banda detectado tras aplicar un filtro de mediana (3 valores)

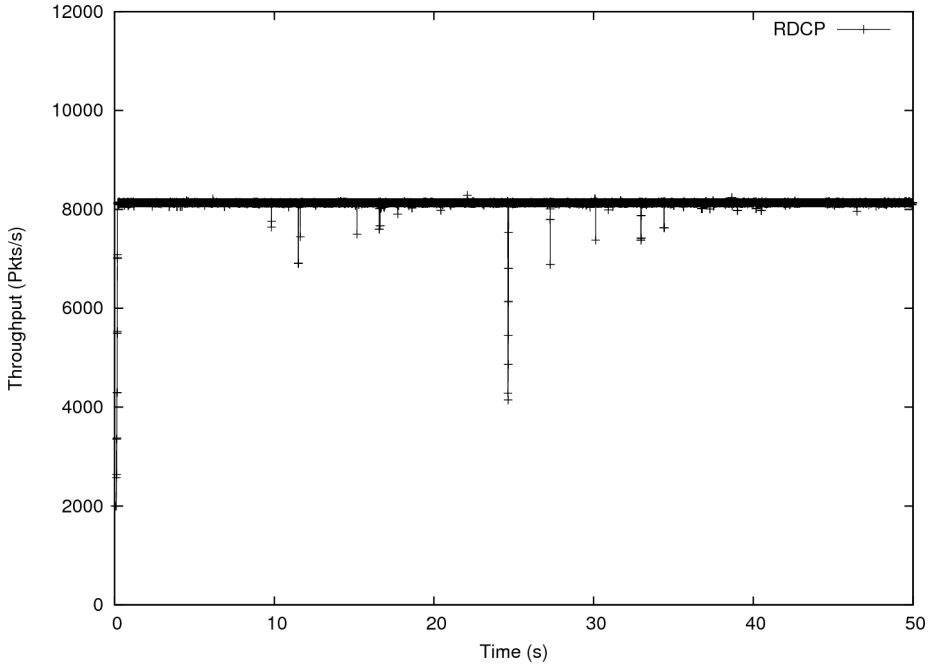

Figura 51: Ancho de banda detectado tras aplicar un filtro de mediana (5 valores)

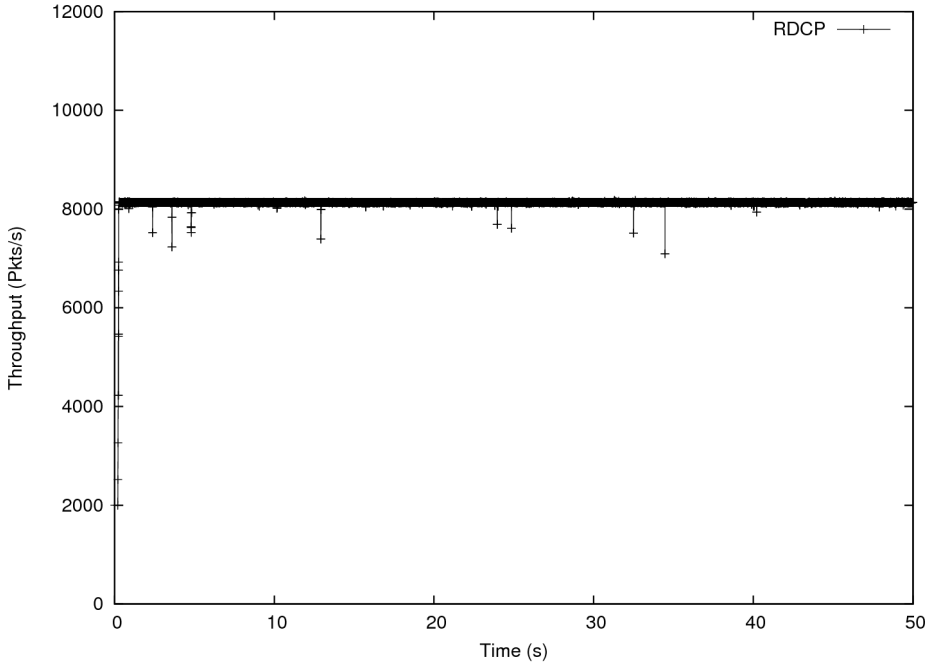

Figura 52: Ancho de banda detectado tras aplicar un filtro de mediana (7 valores)

Como se puede apreciar en las figuras, conforme se va aumentando el número de muestras utilizado para calcular la mediana, mayor es la precisión en la detección del ancho de banda del canal. Para poder cuantificar cuánto mejor era la estimación del ancho de banda del canal usando filtros, tanto de mediana como exponenciales, con respecto a la no utilización de los mismos, se realizaron una serie de pruebas que se describirán a continuación.

Se ha calculado el valor de dos parámetros: la varianza en las estimaciones del ancho de banda del canal y el tiempo de estabilización. El primer parámetro se ha calculado sobre un canal a 100Mbps sin ningún retardo o pérdida de paquetes añadidos midiendo para cada informe de ancho de banda cual es el valor que, tras la transformación del filtro, se usará como estimación del ancho de banda del canal. Después se calculará la varianza de todos estos valores.

Para la estimación del tiempo de estabilización, se establecerá un canal de 50Mbps que transcurridos 20 segundos se ampliará a 100Mbps. El tiempo de estabilización se medirá como el tiempo transcurrido entre que la capacidad del canal es duplicada y el momento en que el servidor estima el ancho de banda del canal como 100Mbps. Las estimaciones de ancho de banda se recogen igual que en caso anterior.

Los resultados que se han obtenido, tanto cuando no se usa filtro de mediana ni exponencial como para cuando si se usa alguno de ellos, aparecen a continuación. Se puede ver como ambos filtros van reduciendo la varianza de los valores conforme sus parámetros los hacen más restrictivos, obteniéndose mediciones más precisas con los filtros de mediana. En cuanto al retardo, se puede ver cómo la variación de los parámetros va aumentando el retardo para todos los filtros. Combinando ambos valores se puede ver que el filtro de mediana 5 es el más adecuado dado su bajo nivel de varianza y su moderado retardo.

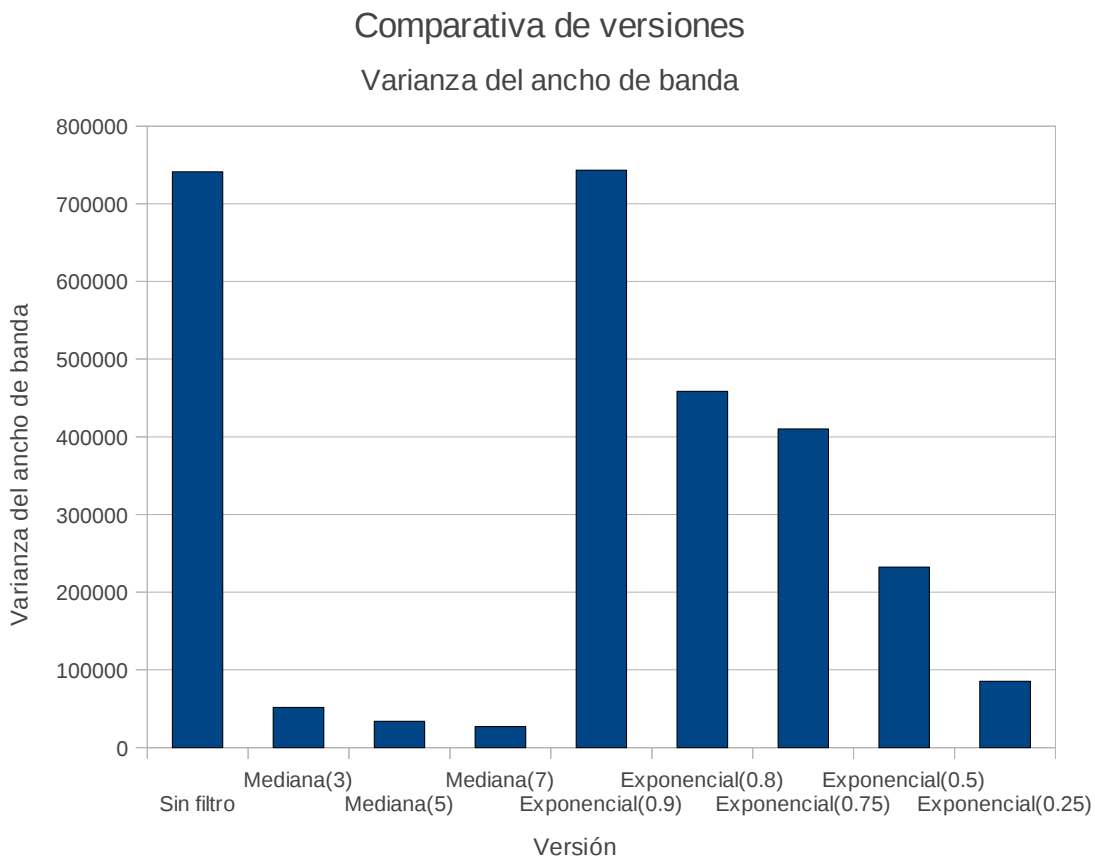

Figura 53: Comparativa de la varianza del ancho de banda entre versiones

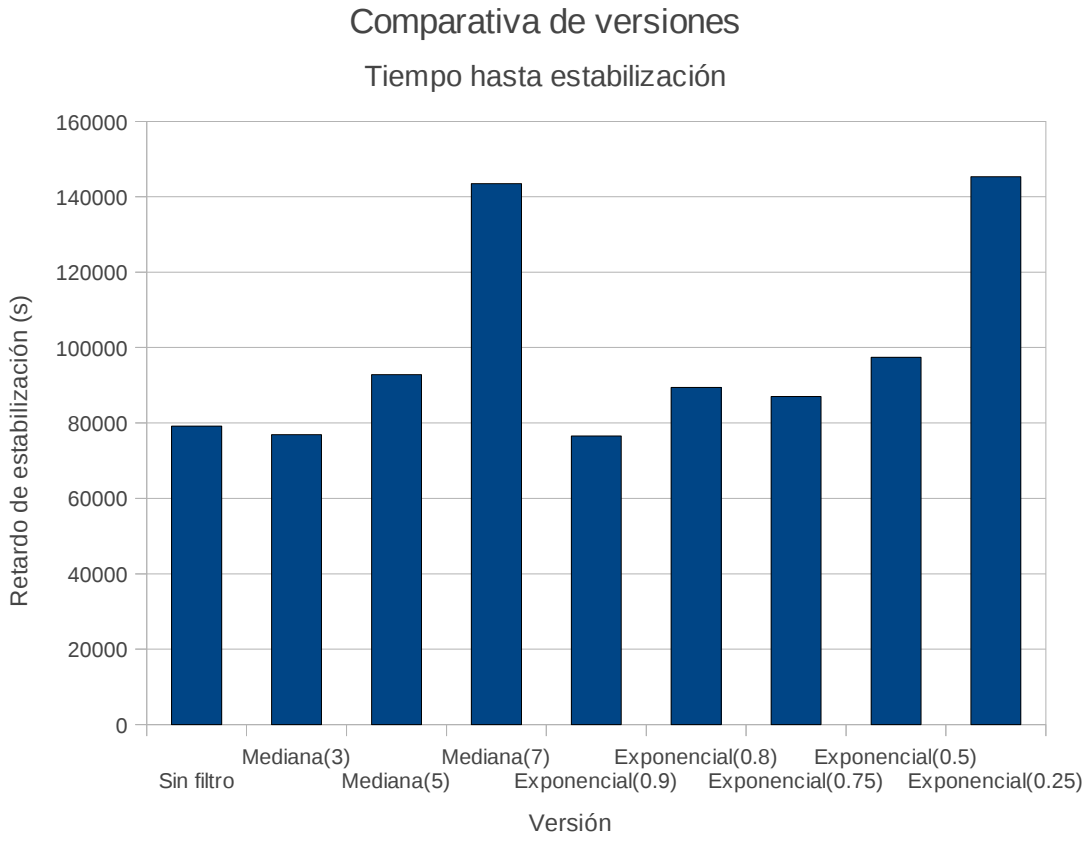

Figura 54: Comparativa del tiempo de estabilización entre versiones

6.5.- Evaluación del rendimiento para el diseño avanzado

Para comprobar que ninguno de los atributos de los obtenidos en el diseño básico se había perdido tras la incorporación de las mejoras del diseño avanzado y del filtrado, se volvió a repetir el conjunto de pruebas que se realizaron para el diseño básico. También se hicieron pruebas para comprobar la productividad en entornos multipath. Para ello se añadió a nuestro testbed la posibilidad de incluir variabilidad en el retardo o paquetes desordenados como parámetros a modificar.

Para mejorar el entorno de simulación se desarrolló un *script* que actuaría como interfaz con la herramienta tc y que permitiría interactuar con ella de una manera mucho más sencilla. El *script* en cuestión es el siguiente:

```
#!/bin/bash

#USO:
#  supra_tc.sh -d retardo:varianza:correlación
#              -l pérdidas:correlacion
#              -b límite ancho de banda
#              -r g:tamaño ó r:reordenados:correlacion
#              -h host remoto
#             Otros:
#                     solo borrar todo (-e)
#                     show qdisc (-s)
#                     permitir especificar el dispositivo (-D)

#INICIALIZACIONES
#[...]

TC=`echo "tc"`
DEV=`echo eth0`

while getopts d:l:b:r:esh:D opcion
do
        #Se recogen y almacenan en variables los parámetros de la línea de comandos
        #[...]
done


if [ $MOSTRAR -eq 1 ]
then
       echo $TC qdisc show
       $TC qdisc show
       echo $TC class show
       $TC class show
else
       ## Borramos las posibles rutas existentes
       if [ $BORRAR -eq 1 ]
       then
              echo $TC qdisc del dev $DEV root
       fi
       $TC qdisc del dev $DEV root

       if [ $BORRAR -eq 0 ]
       then
              PADRE=`echo "root"`

              if [ $DATOS_BW -gt 0 ]
              then
                     ## Creamos la Disciplina
                     echo $TC qdisc add dev $DEV root handle 1:0 htb default 12
                     $TC qdisc add dev $DEV root handle 1:0 htb default 12

                     ## Creamos la clase Root
                     echo $TC class add dev $DEV parent 1:0 classid 1:1 htb rate 100000kbps
```

```
        $TC class add dev $DEV parent 1:0 classid 1:1 htb rate 100000kbps

        echo $TC class add dev $DEV parent 1:1 classid 1:10 \
                                htb rate ${DATOS_BW}kbit ceil ${DATOS_BW}kbit prio 1
        $TC class add dev $DEV parent 1:1 classid 1:10 \
                                htb rate ${DATOS_BW}kbit ceil ${DATOS_BW}kbit prio 1
        PADRE=`echo "parent 1:10"`
fi

AUX=`expr ${DATOS_RETARDO[0]} + ${DATOS_PERDIDAS[0]}`
AUX=`expr $AUX + ${DATOS_DESORDEN[1]}`
if [  $AUX -gt 0  ]
then

        retardo=""
        error=""
        desorden=""
        gap=""

        if [ ${DATOS_RETARDO[0]} -gt 0 ]
        then
                retardo=`echo delay ${DATOS_RETARDO[0]}ms`

                if [ ${DATOS_RETARDO[1]} -gt 0 ]
                then
                        retardo=`echo $retardo ${DATOS_RETARDO[1]}ms`

                        if [ ${DATOS_RETARDO[2]} == N ]
                        then
                                retardo=`echo $retardo distribution normal`
                        elif [ ${DATOS_RETARDO[2]} == P ]
                        then
                                retardo=`echo $retardo distribution pareto`
                        elif [ ${DATOS_RETARDO[2]} == PN ]
                        then
                                retardo=`echo $retardo distribution paretonormal`
                        elif [ ${DATOS_RETARDO[2]} -gt 0 ]
                        then
                                retardo=`echo $retardo ${DATOS_RETARDO[2]}%`
                        fi
                fi
        fi

        if [ ${DATOS_PERDIDAS[0]} -gt 0 ]
        then
                AUX=`echo "scale=2;" ${DATOS_PERDIDAS[0]} "/100" | bc`
                error=`echo loss ${AUX}%`

                if [ ${DATOS_PERDIDAS[1]} -gt 0 ]
                then
                        error=`echo $error ${DATOS_PERDIDAS[1]}%`
                fi
        fi

        if [ ${DATOS_DESORDEN[0]} == r ]
        then
                desorden=`echo reorder ${DATOS_DESORDEN[1]}%`

                if [ ${DATOS_DESORDEN[2]} -gt 0 ]
                then
                        desorden=`echo $desorden ${DATOS_DESORDEN[2]}%`
                fi
        fi

        echo $TC qdisc add dev $DEV $PADRE handle 10: \
                                        netem $gap $retardo $error $desorden
        $TC qdisc add dev $DEV $PADRE handle 10: \
                                        netem $gap $retardo $error $desorden
fi

if [ $DATOS_BW -gt 0 ]
then
        #Esto solo si limitamos por BW
        echo $TC filter add dev $DEV parent 1: protocol ip prio 1 u32 \
                                                match ip dst 192.168.2.0/24 flowid 1:10
```

```
                $TC filter add dev $DEV parent 1: protocol ip prio 1 u32 \
                                                match ip dst 192.168.2.0/24 flowid 1:10
                echo $TC filter add dev $DEV parent 1: protocol ip prio 1 u32 \
                                                match ip src 192.168.2.0/24 flowid 1:10
                $TC filter add dev $DEV parent 1: protocol ip prio 1 u32 \
                                                match ip src 192.168.2.0/24 flowid 1:10
        fi
    fi
fi
```

Haciendo uso de este script se procedió a realizar el juego de pruebas siguiendo la misma metodología que en las pruebas anteriores, es decir, se fijó un par de parámetros y se fue variando un tercero para conocer como le afectaba ese parámetro a la implementación. Los resultados obtenidos fueron los siguientes:

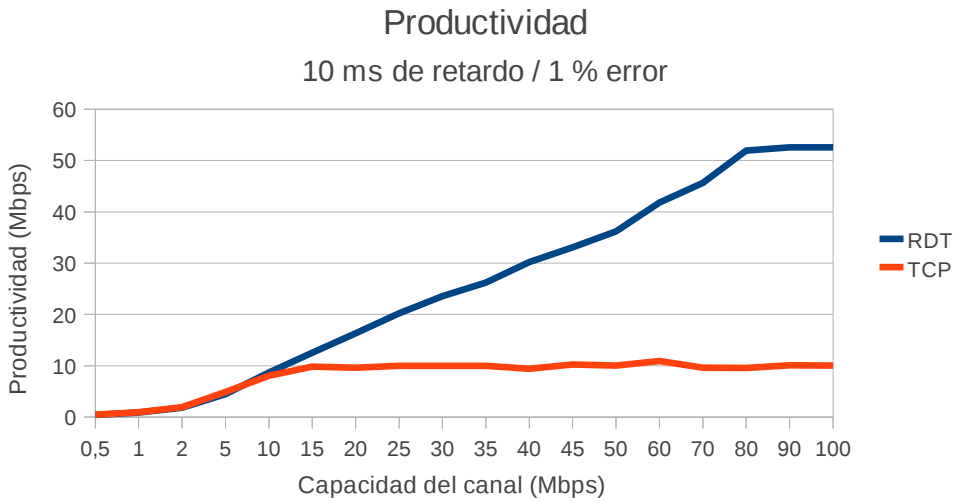

Figura 55: Comparativa TCP - RDT

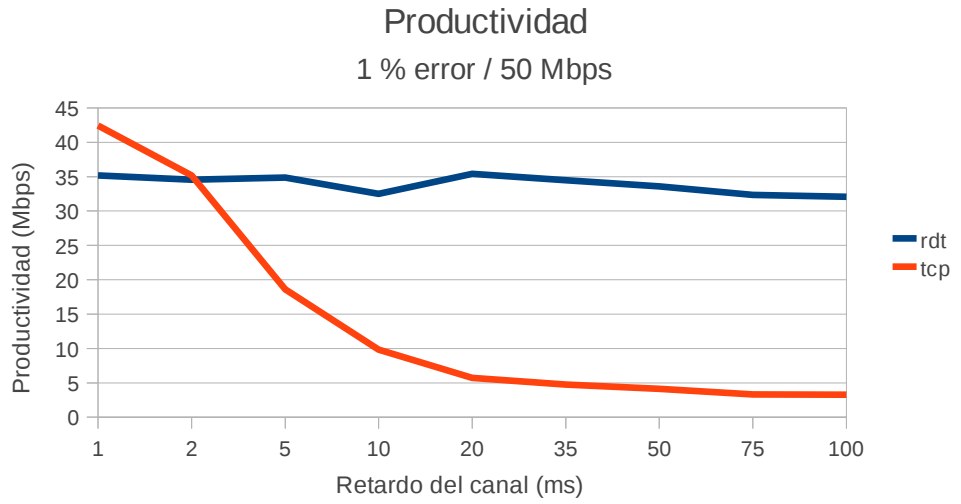

Figura 56: Comparativa TCP - RDT

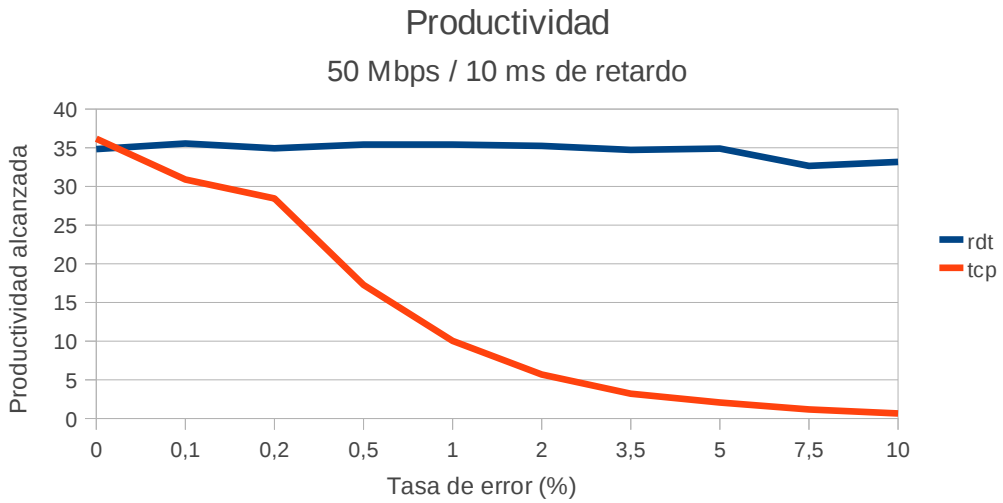

Figura 57: Comparativa TCP - RDT

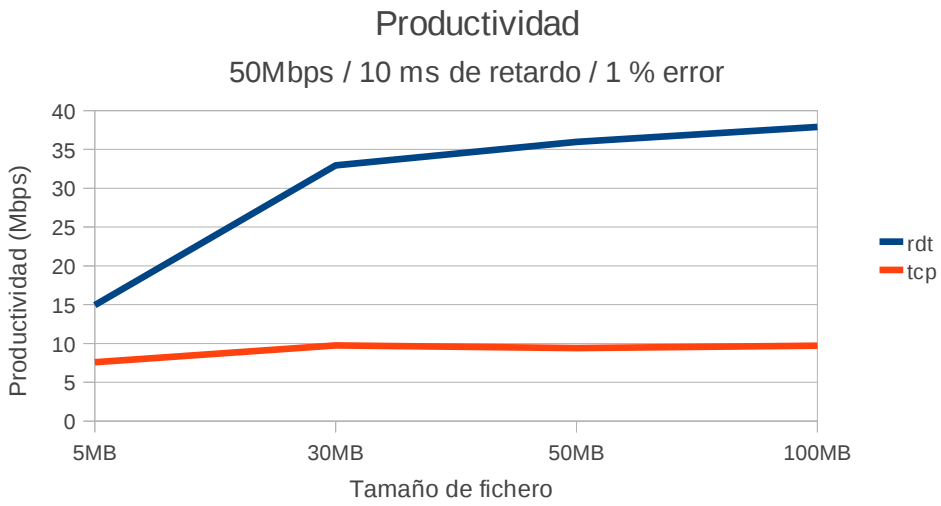

Figura 58: Comparativa TCP -RDT

Nótese que los valores a los que se fijaron los parámetros son diferentes que en ocasiones anteriores, pero la tendencia queda reflejeada de igual forma. Así podemos ver que en este caso, al igual que en el caso anterior, RDT aguanta el impacto del retardo y de la tasa de error mucho mejor que TCP.

Junto con esa serie de pruebas se hizo otra serie similar a las anteriores, pero en este caso se procedió a incluir variabilidad en el retardo y un porcentaje de paquetes desordenados. Los resultados obtenidos de estas pruebas son los siguientes:

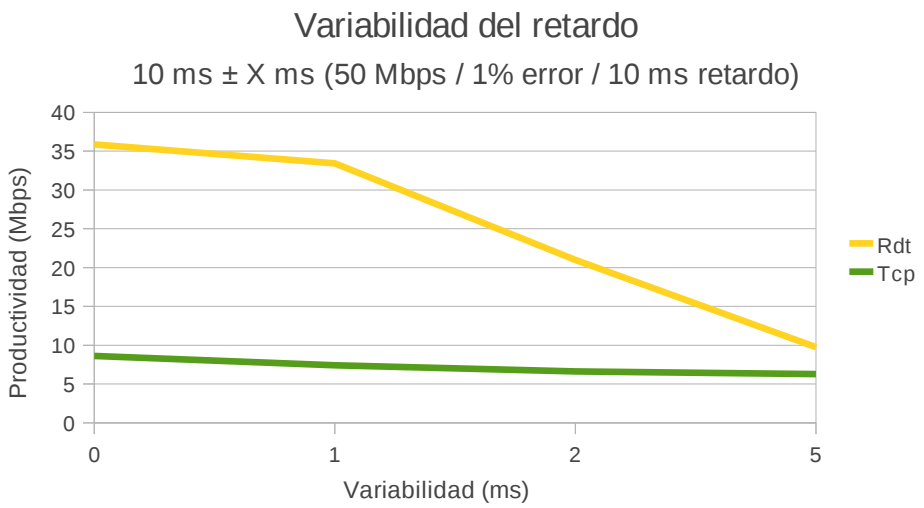

Figura 59: Comparativa TCP - RDT

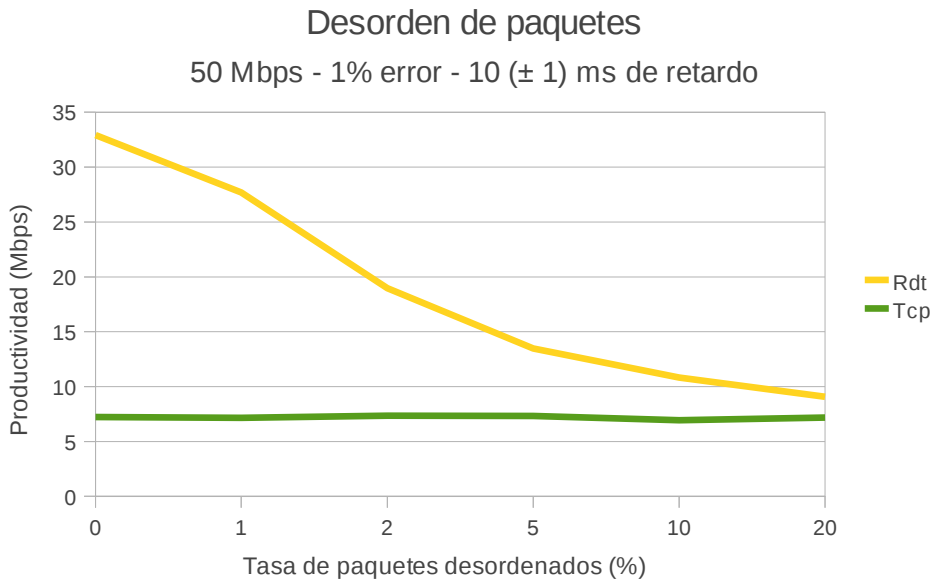

Figura 60: Comparativa TCP -RDT

En este caso observamos como la productividad de RDT decae con la variabilidad del retardo y con el número de paquetes desordenados. Esto es debido al sistema de detección de ancho de banda basado en ráfagas, que tiene una mayor dificultad para estimar el ancho de banda cuando los paquetes que las forman llegan desordenados o con un retardo muy variable.

6.6.- Pruebas de concurrencia

Por último se realizaron una serie de pruebas para observar como los flujos RDT se comportan en situaciones de concurrencia, ya sea con otros flujos RDT o con flujos TCP.

Para automatizar estas pruebas se creó un script similar a este:

```
#!/bin/bash

MUESTRAS=11
FLUJOS_TCP=1
PUERTO_TCP=9001
PUERTO_RDT=9001

ES_ROOT=`id | grep -c root`

if [ $ES_ROOT -eq 0 ]
then
       echo "Ejecutar como root"
else

FLUJOS_TCP=$1
ejecutable=$2

       for ((i=1;$i<$MUESTRAS;i=$i+1))
        do
               echo "TCP: Ejecutable " $ejecutable

               # Servidores
               /home/miguel/RDT/pruebas/bws/ejecutables/${ejecutable}/servidor $PUERTO_RDT \
                                                    > ${ejecutable}_tcp.${FLUJOS_TCP}.${i}.txt &
               servidor=$!
               /home/miguel/sendinfinitefile $PUERTO_TCP &
               servidor_tcp[1]=$!

               # Offset de Flujos
               for((j=2; $j <= $FLUJOS_TCP; j=$j+1))
               do
                   /usr/bin/ssh root@vaio-grc-cable "/home/miguel/RDT/recvinfinitefile \
                             192.168.2.100 $PUERTO_TCP /home/miguel/P_F_4_7.avi bla.${j}.avi" &
                   PUERTO_TCP=`/usr/bin/expr $PUERTO_TCP + 1`
                   /home/miguel/sendinfinitefile $PUERTO_TCP &
                   servidor_tcp[$j]=$!
               done

               # Timestamp inicio
               momento=`/bin/date +%s%N`
               echo $momento > ${ejecutable}_tcp_milestone_inicial.${FLUJOS_TCP}.${i}.txt

               # Clientes
               /usr/bin/ssh root@vaio-grc-cable "/home/miguel/RDT/recvinfinitefile 192.168.2.100 \
                      $PUERTO_TCP /home/miguel/P_F_4_7.avi bla.avi.tcp" \
                      2> ${ejecutable}_tcp_tamtcp.${FLUJOS_TCP}.${i}.txt &
               cliente=$!
               /usr/bin/ssh root@vaio-grc-cable "/home/miguel/RDT/cliente 192.168.2.100 \
                                          $PUERTO_RDT /home/miguel/P_F_4_7.avi bla.avi.rdt"

               # Finalizamos TCP
               kill -9 ${servidor_tcp[${FLUJOS_TCP}]}

               # Timestamp final
               momento=`/bin/date +%s%N`
               echo $momento > ${ejecutable}_tcp_milestone_final.${FLUJOS_TCP}.${i}.txt

               # CleanUp
               for ((j=1; $j < $FLUJOS_TCP; j=$j + 1))
               do
                   kill -9 ${servidor_tcp[$j]}
               done
               wait $cliente
               kill -9 $servidor
               PUERTO_RDT=`/usr/bin/expr $PUERTO_RDT + 1`
               PUERTO_TCP=`/usr/bin/expr $PUERTO_TCP + 1`
       done

fi
```

Los resultados obtenidos para la concurrencia entre un flujo RDT y varios flujos TCP en una red a 50Mbps con un 1% de pérdida de paquetes y 10 ms. de retardo han sido los siguientes:

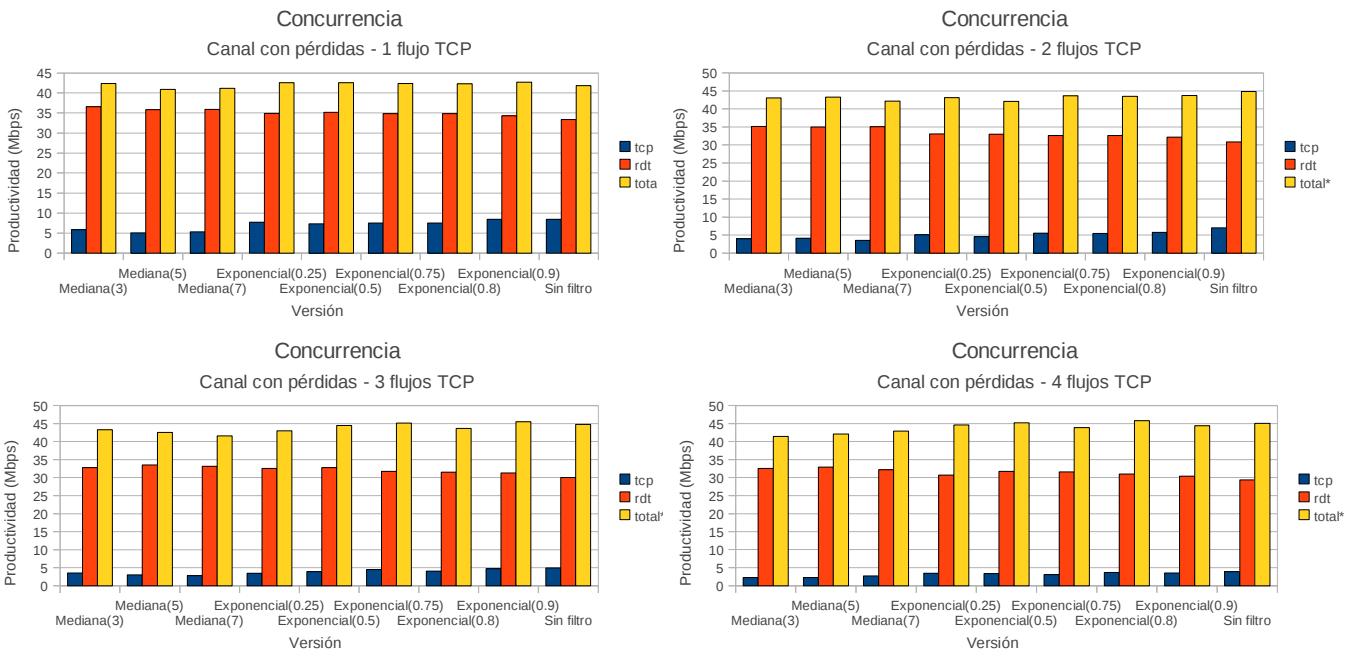

Figura 61: Concurrencia de TCP y RDT en una red limitada

En este caso se han recogido los valores de productividad de RDT y de uno de los flujos TCP y se han representado gráficamente. La línea *total* hace referencia al ancho de banda total (T) usado en el canal, y se ha calculado de la siguiente manera: $T = P_{RDT} + P_{TCP} * n$ siendo n el número de flujos TCP que se están ejecutando concurrentemente. Como se puede ver en las gráficas, RDT es capaz de aprovechar el ancho de banda que los flujos RDT no pueden usar debido a su bajo rendimiento en condiciones de pérdida de paquetes y retardo.

Con el fin de realizar el conjunto de pruebas que implicaban concurrencia RDT de manera que el uso de recursos (CPU, memoria) de los servidores RDT no se interfiriera entre varias instancias en ejecución, se implementó un *testbed* particular que se describirá a continuación. Se conectaron dos máquinas que contendrían los servidores RDT, un cliente y una cuarta máquina a un *switch* Ethernet. Esta cuarta máquina actuará de controlador y se incluye para limitar la red con los parámetros adecuados para las pruebas (50Mbps en lugar de 100Mbps, etc.). Se forzó que todo el tráfico pasara por esta cuarta máquina y se aplicó sobre ella el conjunto de restricciones en la red.

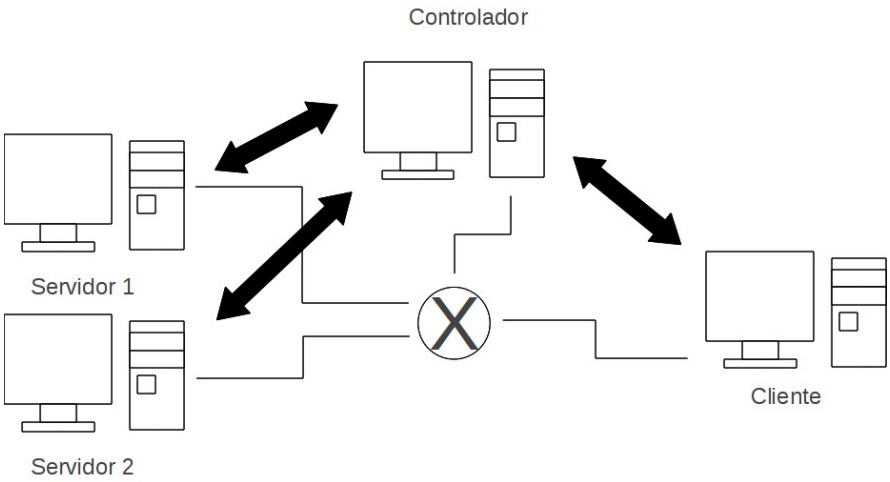

Figura 62: Testbed para las pruebas de concurrencia entre flujos RDT

También en este caso se implementaron *scripts*, los cuales eran similares a este:

```
#!/bin/bash

MUESTRAS=11
FLUJOS_TCP=1
SERVIDOR_RDT=9001
CLIENTE_RDT=9001
BASE_IP=200

#[...]

FLUJOS_TCP=$1
ejecutable=$2

retardo=10
error=1
bw=50000

/home/miguel/RDT/pruebas/bws/ejecutables/supra_tc.sh -d $retardo -l $error -b $bw

        for ((i=1;$i<$MUESTRAS;i=$i+1))
         do
                echo "TCP: Ejecutable " $ejecutable

                # Servidores
                for((j=1; $j <= $FLUJOS_TCP; j=$j+1))
                do
                    IP_SERVIDOR=`/usr/bin/expr $j - 1`
                    IP_SERVIDOR=`/usr/bin/expr $IP_SERVIDOR + $IP_SERVIDOR`
                    IP_SERVIDOR=`/usr/bin/expr $IP_SERVIDOR + 1`
                    IP_SERVIDOR=`/usr/bin/expr $IP_SERVIDOR + $BASE_IP`
                    echo $IP_SERVIDOR:$SERVIDOR_RDT
                    /usr/bin/ssh root@192.168.2.$IP_SERVIDOR "/root/RDT/${ejecutable}/servidor \
                      $SERVIDOR_RDT > /root/RDT/${ejecutable}_rdt.${j}.${FLUJOS_TCP}.${i}.ltd.txt" &
                    servidor_tcp[$j]=$!
                    SERVIDOR_RDT=`/usr/bin/expr $SERVIDOR_RDT + 1`
                done
                sleep 15

                # Timestamp inicio
                momento=`/bin/date +%s%N`
                echo $momento > /${ejecutable}_rdt_milestone_inicial.${FLUJOS_TCP}.${i}.ltd.txt

                # Clientes
                for((j=1; $j <= $FLUJOS_TCP; j=$j+1))
                do
                    IP_CLIENTE=`/usr/bin/expr $j - 1`
                    IP_CLIENTE=`/usr/bin/expr $IP_CLIENTE + $IP_CLIENTE`
                    IP_SERVIDOR=`/usr/bin/expr $IP_CLIENTE + 1`
                    IP_SERVIDOR=`/usr/bin/expr $IP_SERVIDOR + $BASE_IP`
                    IP_CLIENTE=`/usr/bin/expr $IP_CLIENTE + $BASE_IP`
                    echo $IP_SERVIDOR:$CLIENTE_RDT
                    /usr/bin/ssh root@192.168.2.200 "time /root/RDT/cliente 192.168.2.$IP_SERVIDOR \
                                        $CLIENTE_RDT /root/P_F_4_7.avi bla.${j}.avi" \
                                        2> ${ejecutable}_rdt_time.${j}.${FLUJOS_TCP}.${i}.ltd.txt &
                    cliente_rdt[$j]=$!
                    CLIENTE_RDT=`/usr/bin/expr $CLIENTE_RDT + 1`
                done
                wait ${cliente_RDT[$FLUJOS_TCP]}

                # Timestamp final
                momento=`/bin/date +%s%N`
                echo $momento > ${ejecutable}_rdt_milestone_final.${FLUJOS_TCP}.${i}.ltd.txt

                # CleanUp
                for ((j=1; $j < $FLUJOS_TCP; j=$j + 1))
                do
                    kill -9 ${servidor_tcp[$j]}
                    kill -9 ${cliente_rdt[$j]}
                done
        done

/home/miguel/RDT/pruebas/bws/ejecutables/supra_tc.sh -e
```

Los resultados obtenidos para la concurrencia entre varios flujos RDT en una red a 50Mbps con un 1% de pérdida de paquetes y 10 ms de retardo han sido los siguientes:

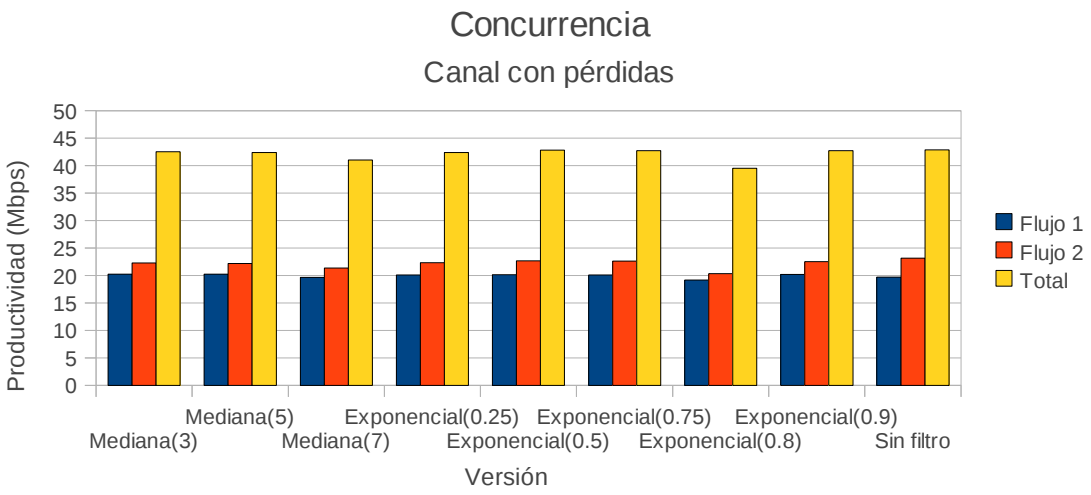

Figura 63: Productividad alcanzada por RDT con dos flujos concurrentes

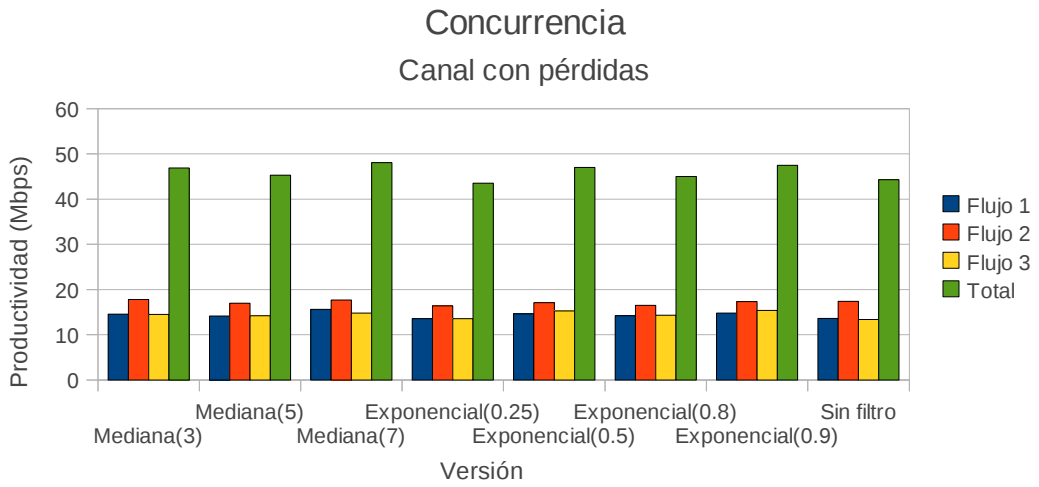

Figura 64: Productividad alcanzada por RDT con tres flujos concurrentes

En este caso se ha medido la productividad alcanzada por todos los flujos RDT. Como aparece en las gráficas, el ancho de banda se reparte a partes prácticamente iguales entre los tres flujos, independientemente de la versión utilizada, de manera que se consigue bastante ecuanimidad entre los diferentes flujos.

Ahora pasaremos a examinar la influencia de los parámetros alfa y beta en la productividad. Para ello, se realizarán pruebas fijando beta Si atendemos ahora a la variación del parámetro alfa

mientras fijamos el parámetro beta en primer lugar a 0.8 y después a 0.9 y se irán variando los valores de alfa (0.0, 0.1, 0.4 y 0.7). Se van a realizar las pruebas en dos escenarios: *red normal* y *red limitada*. El escenario *red normal* es el testbed utilizado para las pruebas tal cual (computadores conectados mediante Ethernet), mientras que red limitada es el mismo testbed al que se le ha aplicado un retardo de 10 ms, 1% de pérdida de paquetes y un límite al ancho de banda de 50Mbps. En las siguientes figuras se muestran los resultados obtenidos:

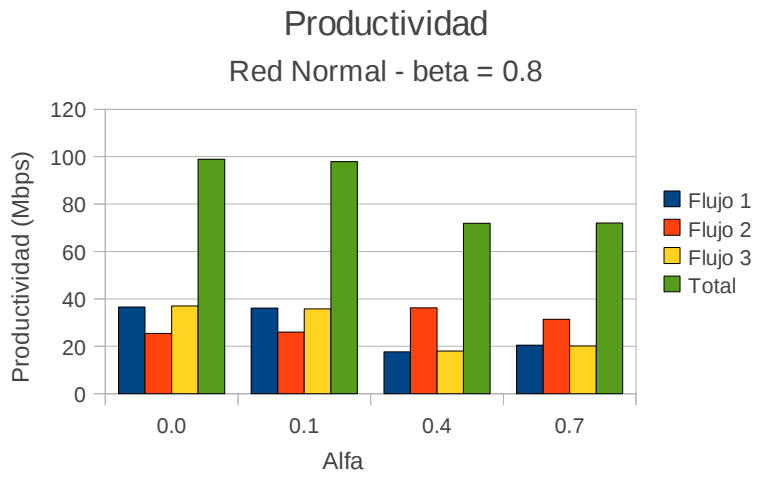

Figura 65: Productividad alcanzada por RDT con tres flujos concurrentes con beta igual a 0.8

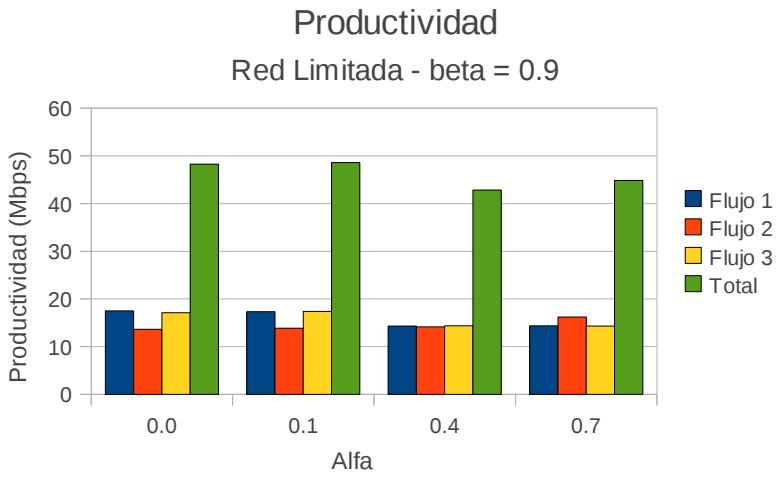

Figura 66: Productividad alcanzada por RDT con tres flujos concurrentes con beta igual a 0.9

En este caso se puede comprobar que el ancho de banda se reparte también de manera igualitaria. No obstante, si que se observa un curioso fenómeno que hace que con valores de alfa bajos se utilice más eficientemente el canal. Este puede ser debido a que con un alfa menor las ráfagas son más agresivas y pueden ajustarse al ancho de banda disponible más fácilmente.

7.- Conclusiones

La eficiencia del protocolo TCP se ha visto cuestionada en distintos ámbitos, especialmente en el ámbito de las redes inalámbricas. Muchas son las soluciones que se han propuesto a este problema, y con muy distintos enfoques. En el presente documento se a descrito una propuesta novedosa basada en códigos Raptor, así como su implementación. Se ha puesto de manifiesto como las barreras al rendimiento que impone el protocolo TCP cuando es implementado sobre las redes inalámbricas pueden ser burladas simplemente prescindiendo de él, y analizando con detenimiento el problema para lograr un nuevo paradigma de comunicación.

El presente proyecto presenta una librería de comunicaciones que permite realizar una entrega fiable de contenidos en canales con pérdida mediante el envío del contenido codificado usando Raptor Codes. Además, evita el sistema de detección de congestión de TCP basado en pérdidas de paquetes, reemplazándolo por técnicas de estimación del ancho de banda extremo-a-extremo.

Se han presentado un diseño básico y una serie de mejoras a ese diseño, junto con la implementación de cada una de ellas. También se presenta una serie de pruebas que evalúan independientemente, y comparativamente a TCP, el rendimiento de la librería en las condiciones típicas de una red inalámbrica (alta pérdida de paquetes, bajo ancho de banda, alto retardo, etc.). Como resultado de las pruebas se puede observar que el protocolo basado en codificación Raptor obtiene como resultado una mayor productividad que TCP en la mayoría de escenarios, especialmente cuando las condiciones del canal son pobres.

Gracias a las pruebas se ha comprobado también la equidad en el comportamiento de los flujos RDT entre sí y con flujos TCP, incluyendo un análisis preliminar de la influencia de los parámetros más representativos del protocolo (alfa y beta) en los envíos concurrentes.

Como conclusión, podemos decir que la solución basada en codificación Raptor que aquí presentamos muestra un protocolo a tener muy en cuenta en aquellas situaciones en las que nos encontramos con un canal propenso a perder paquetes y/o a introducir mucho retardo en su entrega. Además, dada la cómoda interfaz que ofrece al programador, el protocolo puede ser fácilmente incluido en aplicaciones en desarrollo o ya implementadas de una manera sencilla y rápida.

Respecto al trabajo futuro, se desarrollará un entorno de simulación que permita validar el protocolo propuesto en redes ad-hoc, y de forma más exhaustiva mediante competición entre los distintos flujos RDT o de diferente tipo.

8.- Bibliografía

[1] Ashish Natani, Jagannadha Jakilinki, Mansoor Mohsin, Vijay Sharma : TCP for Wireless Networks , University of Texas at Dallas , 2001

[2] Kostas Pentikousis : TCP in wired-cum-wireless environments , IEEE Communications Surveys, 2000

[3] Hari Balakrishnan : A Comparison of Mechanisms for Improving TCP Performance over Wireless Links , IEEE/ACM Transactions on Networking, vol. 5, p. 756-769 , IEEE Press Piscataway, NJ, USA, 1997

[4] Ender Ayanoglu , Sanjoy Paul , Thomas F. LaPorta , Krishan K. Sabnani , Richard D. Gitlin : AIRMAIL: A Link-Layer Protocol for Wireless Networks , Wireless Networks, vol. 1, p. 47-60 , Springer Netherlands, 1995

[5] Christina Parsa , J.J. Garcia-Luna-Aceves : TULIP: A Link-Level Protocol for Improving TCP over Wireless Links , University of California , 1999

[6] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow: TCP Selective Acknowledgment Options, IETF Request for Comments 2018, 1996

[7] S. Keshav , S. P. Morgan : SMART Retransmission: Performance with Overload and Random Losses, IEEE INFOCOM, 1997

[8] Amin Shokrollahi : Raptor Codes , IEEE Transactions on Information Theory, vol. 52, pp. 2551-2567, 2006

[9] Michael Luby : LT Codes , The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002

[10] DF Raptor Technology , Digital Fountain, Inc. , 2007

[11] DF Raptor R11 Encoder 2.2 , Digital Fountain, Inc. , 2008

[12] DF Raptor R11 Decoder 2.2 , Digital Fountain, Inc. , 2008

[13] Yunhong Gu, Robert L. Grossman : UDT: UDP-based Data Transfer for High-Speed Wide Area Networks , University of Illinois at Chicago, 2007

[14] Yunhong Gu, Robert L. Grossman : Supporting Configurable Congestion Control in Data Transport Services , University of Illinois at Chicago, 2005

[15] H.Y. Hsieh and R. Sivakumar. pTCP: An end-to-end transport layer protocol for striped connections. 2002

[16] M. Zhang, J. Lai, A. Krishnamurthy, L. Peterson, and R. Wang. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In Proceedings of the annual conference on USENIX Annual Technical Conference, page 8. USENIX Association, 2004.

[17] S. Raman, H. Balakrishnan, and M. Srinivasan. An image transport protocol for the Internet. In icnp, page 209. Published by the IEEE Computer Society, 2000.

[18] X. Wu, M.C. Chan, AL Ananda, and C. Ganjihal. Sync-TCP: A new approach to high speed congestion control. In Network Protocols, 2009. ICNP 2009. 17th IEEE International Conference on, pages 181-192. IEEE, 2009

[19] H. Zang, M. Ghaderi, and A. Sridharan. TCP-aware power control in wireless networks. In Network Protocols, 2009. ICNP 2009. 17th IEEE International Conference on, pages 334-343. IEEE, 2009

[20] Y. Gu and R.L. Grossman. Optimizing UDP-based protocol implementations. In Proceedings of the Third International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet 2005). Citeseer, 2005.

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

[22] Ludmila Cherkasova and Jangwon Lee. Fastreplica: eficient large file distribution within content delivery networks. In Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4, USITS'03, pages 7-7, Berkeley, CA, USA, 2003.

[23] Jie Chen, Tiejun Lv, and Haitao Zheng. Joint cross-layer design for wireless qos content delivery. EURASIP J. Appl. Signal Process., 2005:167-182, January 2005.

[24] M. Luby, M. Watson, T. Gasiba, T. Stockhammer, and Wen Xu. Raptor codes for reliable download delivery in wireless broadcast systems. In Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE, volume 1, pages 192 - 197, January 2006.

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

[26] D. Stutzbach, D. Zappala, and R. Rejaie. The scalability of swarming peer-to-peer content delivery. NETWORKING 2005, pages 15-26, 2005.

ANEXO I: API del codificador DF Raptor

El API que el codificador DF Raptor proporciona al programador es el siguiente:

Constantes:

Nombre	Valor	Descripción
DFR11ENC_MINIMUM_SYMBOL_SIZE	1	Valor mínimo de tamaño del símbolo (SymbolSize) en bytes
DFR11ENC_MAXIMUM_SYMBOL_SIZE	65535	Valor máximo de tamaño del símbolo (SymbolSize) en bytes
DFR11ENC_MINIMUM_NUM_SOURCE_SYMBOLS	1	Número mínimo de símbolos fuente por bloque (SymbolsPerBlock)
DFR11ENC_MAXIMUM_NUM_SOURCE_SYMBOLS	56405	Número máximo de símbolos fuente por bloque (SymbolsPerBlock)
DFR11ENC_MAXIMUM_REPAIR_ID	2147478648	Valor máximo del identificador de símbolo.
DFR11ENC_DO_ALL	1000	Valor en ‰ que hace que la librería realice todo el trabajo en una sola iteración

Códigos de error:

Código	Descripción
DFR11ENC_SUCCESS	Función invocada con éxito
DFR11ENC_INTERNAL_ERROR	Error en la memoria de trabajo del codificador
DFR11ENC_NULL_ENCODER	Puntero no válido al codificador
DFR11ENC_NULL_BLOCK	Puntero no válido al bloque fuente o al bloque de reparación
DFR11ENC_INVALID_NUMBER_OF_SYMBOLS	Número de símbolos fuente por bloque (SymbolsPerBlock) no válido
DFR11ENC_INVALID_SYMBOL_SIZE	Tamaño de símbolo (SymbolSize) no válido
DFR11ENC_INVALID_PERMILLE	Valor de ‰ no válido
DFR11ENC_MEM_NOT_ALIGNED	Alineamiento de memoria incorrecto
DFR11ENC_INVALID_SYMBOL_ID	ID de símbolo no válido
DFR11ENC_UNSUPPORTED_CONFIG	Modo de operación no soportado por la librería
DFR11ENC_INVALID_REPAIR_REQUEST	Petición de símbolo de reparación incorrecta. No se ha generado completamente el bloque intermedio.
DFR11ENC_INVALID_SOURCE_REQUEST	Petición de símbolo fuente incorrecta. Los símbolos fuente no están disponibles.
DFR11ENC_NOT_INITIALIZED	Codificador no inicializado.
DFR11ENC_NULL_MEMORY_REQUIREMENT	Estructura de requerimientos de memoria no válida.
DFR11ENC_INVALID_SYMBOL_COUNT	Contador de símbolos no válido
DFR11ENC_INVALID_BLOCK_SIZE	Tamaño de bloque no válido
DFR11ENC_DMA_NOT_ENABLED	DMA no habilitado correctamente
DFR11ENC_INVALID_DMA_PARAMS	Parámetros DMA no válidos
DFR11ENC_INCONSISTENT_DMA_PARAMS	Parámetros DMA inconsistentes
DFR11ENC_DMA_DEVICE_FAILURE	Fallo del dispositivo al intentar usar DMA
DFR11ENC_DMA_INSUFFICIEN_MEMORY	Insuficiente memoria DMA para la codificación
DFR11ENC_NOT_PREPARED	Codificador no preparado correctamente

Funciones disponibles:

Nombre	**Descripción**
DFR11EncMemRequest()	Indica los requisitos de memoria necesarios para la codificación
DFR11EncInit()	Inicializa el codificador.
DFR11EncReset()	Reinicia el codificador para un nuevo procesamiento
DFR11EncPrepare()	Prepara el codificador para empezar la codificación de los símbolos
DFR11EncInitSrcBlock()	Inicializa las variables de estado del codificador relativas al bloque fuente
DFR11EncRelocateIntermBlockAddr()	Establece la dirección del bloque intermedio
DFR11EncGenIntermediateBlock()	Genera un bloque intermedio
DFR11EncGenRepairSymbol()	Genera un bloque de reparación
DFR11EncGetSourceSymbols()	Devuelve un símbolo fuente
DFR11EncSetRepairRatioMode()	Establece la tasa de reparación del codificador
DFR11EncVersion()	Devuelve un string que identifica la versión
DFR11EncErrorString()	Devuelve un string que identifica el código de error
DFR11EncEnableDMA()	Habilita el dispositivo DMA
DFR11EncDisableDMA()	Deshabilita el dispositivo DMA

ANEXO II: API del decodificador DF Raptor

El API que el decodificador DF Raptor proporciona al programador es el siguiente:

Constantes:

Nombre	Valor	Descripción
DFR11DEC_MINIMUM_SYMBOL_SIZE	1	Valor mínimo de tamaño del símbolo (SymbolSize) en bytes
DFR11DEC_MAXIMUM_SYMBOL_SIZE	65535	Valor máximo de tamaño del símbolo (SymbolSize) en bytes
DFR11DEC_MINIMUM_NUM_SOURCE_SYMBOLS	1	Número mínimo de símbolos fuente por bloque (SymbolsPerBlock)
DFR11DEC_MAXIMUM_NUM_SOURCE_SYMBOLS	56405	Número máximo de símbolos fuente por bloque (SymbolsPerBlock)
DFR11DEC_MAXIMUM_NUM_RECEIVED_SYMBOLS	64000	Máximo número de símbolos recibidos
DFR11DEC_MAXIMUM_SYMBOL_ID	2147478648	Valor máximo del identificador de símbolo.
DFR11DEC_DEFAULT_TOTAL_SYMBOLS	SymbolsPerBlock + Max(30, 0.03* SymbolsPerBlock)	
DFR11DEC_DO_ALL	1000	Valor en ‰ que hace que la librería realice todo el trabajo en una sola iteración
DFR11DEC_INVALIDATION_ESI	-1	Símbolo no válido

Códigos de error:

Código	Descripción
DFR11DEC_SUCCESS	Función invocada con éxito
DFR11DEC_INTERNAL_ERROR	Error en la memoria de trabajo del codificador
DFR11DEC_NULL_DECODER	Puntero no válido al decodificador
DFR11DEC_NULL_BLOCK	Puntero no válido al bloque de datos
DFR11DEC_INVALID_NUMBER_OF_SYMBOLS	Número de símbolos fuente por bloque (SymbolsPerBlock) no válido
DFR11DEC_INVALID_SYMBOL_SIZE	Tamaño de símbolo (SymbolSize) no válido
DFR11DEC_INVALID_PERMILLE	Valor de ‰ no válido
DFR11DEC_INVALID_SYMBOL_ID	ID de símbolo no válido
DFR11DEC_TOO_MANY_SYMBOLS	Número de símbolos recibidos no válido
DFR11DEC_INTEGRITY_FAILURE	Fallo en la integridad del bloque fuente
DFR11DEC_MEM_NOT_ALIGNED	Alineamiento de memoria no válido
DFR11DEC_INSUFFICIENT_SYMBOLS	Número de símbolos recibidos insuficiente
DFR11DEC_UNSUPPORTED_CONFIG	Modo de operación no soportado por la librería
DFR11DEC_NOT_INITIALIZED	Decodificador no inicializado.
DFR11DEC_NULL_MEMORY_REQUIREMENT	Estructura de requerimientos de memoria no válida.
DFR11DEC_INVALID_SYMBOL_COUNT	Contador de símbolos no válido
DFR11DEC_INVALID_BLOCK_SIZE	Tamaño de bloque no válido
DFR11DEC_DMA_NOT_ENABLED	DMA no habilitado correctamente
DFR11DEC_INVALID_DMA_PARAMS	Parámetros DMA no válidos
DFR11DEC_INCONSISTENT_DMA_PARAMS	Parámetros DMA inconsistentes
DFR11DEC_DMA_DEVICE_FAILURE	Fallo del dispositivo al intentar usar DMA
DFR11DEC_DMA_INSUFFICIEN_MEMORY	Insuficiente memoria DMA para la decodificación
DFR11DEC_NOT_PREPARED	Decodificador no preparado correctamente

Funciones disponibles:

Nombre	**Descripción**
DFR11DecMemRequest()	Indica los requisitos de memoria necesarios para la decodificación
DFR11DecInit()	Inicializa el decodificador.
DFR11DecReset()	Reinicia el decodificador para un nuevo procesamiento
DFR11DecPrepare()	Prepara el decodificador para empezar la decodificación de los símbolos
DFR11DecInitRcvBlock()	Inicializa las variables de estado del decodificador relativas al bloque recibido
DFR11DecRecoverSource()	Recupera el bloque fuente
DFR11DecVersion()	Devuelve un string que identifica la versión
DFR11DecErrorString()	Devuelve un string que identifica el código de error
DFR11DecEnableDMA()	Habilita el dispositivo DMA
DFR11DecDisableDMA()	Deshabilita el dispositivo DMA