etsinf
Escola Tècnica
Superior d'Enginyeria
Informàtica

UNIVERSIDAD POLITÉCNICA DE VALENCIA
ESCUELA TÉCNICA SUPERIOR DE INFORMÁTICA APLICADA

RDT: Un protocolo de transporte para redes inalámbricas basado en Raptor Codes

PROYECTO FIN DE CARRERA

Miguel Báguena Albaladejo

Director
Carlos Tavares Calafate

1 de Septiembre de 2010

Índice

1.- Introducción

1.1.- Visión general

TCP se ha mostrado siempre un protocolo robusto en cuanto a la comunicación entre ordenadores y otros dispositivos. Es capaz de proporcionar un envío de información al destino de manera confiable y eficiente. Eso ha hecho de este protocolo de transporte el líder en la comunicación mundial a nivel de datos, y ha posibilitado que se construyan sobre él grandes estructuras de las que dependen económicamente grandes empresas, en parte o en su totalidad.

Los principales elementos que han hecho a TCP el líder de las comunicaciones a nivel mundial son los siguientes:

- Proporciona un envío confiable mediante el uso de ACK.
- Evita la recepción de duplicados y permite la recepción ordenada de datos en paquetes gracias al uso de números de secuencia.
- Permite la comunicación orientada a conexión.
- Permite la detección de errores.
- Permite la multiplexación de varios flujos mediante el uso de puertos.
- Proporciona un control de flujo que evita la congestión que se pueda producir en la red.

Se han producido muchos avances en el mundo de las redes, en especial en las redes inalámbricas. Estos avances han permitido crear redes de alta velocidad sin cables para la interconexión de equipos. Se ha decidido implementar sobre este nuevo sistema el protocolo TCP que tan buenos resultados nos ha dado en la tecnología cableada, tanto para obtener un buen rendimiento como para conseguir compatibilidad con el resto de estándares de comunicación. Sin embargo, este nuevo medio trae consigo nuevas características que hacen que TCP no alcance el excelente rendimiento que nos ha proporcionado hasta ahora.

Esta pérdida de prestaciones es debida a una razón muy sencilla: en las redes cableadas la principal causa de pérdida de paquetes es la congestión de la red, por tanto TCP identifica esta pérdida de paquetes con un caso de congestión e intenta evitar esa pérdida reduciendo el número de paquetes por segundo enviados. En el caso de las redes inalámbricas, la pérdida y corrupción de paquetes puede ser debida no solo a la congestión de la red, sino también a factores ambientales que limitan la calidad de la recepción de la señal. En este último caso, la decisión del protocolo TCP de reducir la velocidad de transmisión lo único que realmente produce es una infrautilización del ancho de banda del canal.

Vista esta peculiaridad del comportamiento del protocolo TCP sobre las redes inalámbricas, se convierte en un deseo el encontrar un protocolo que nos de un mejor rendimiento en estos casos, y poder así sacar el mayor rendimiento posible a este sector, tan extendido hoy día.

Para solventar este problema se han hecho varios intentos por parte de varios grupos de investigación, proponiendo diversos métodos de conseguir las mismas ventajas que se tienen con TCP, pero evitando los problemas de rendimiento por los que se ve limitado en las redes inalámbricas.

1.2.- Objetivos del trabajo

Con este proyecto se pretende desarrollar una librería de comunicaciones que obtenga un mejor rendimiento que TCP en situaciones similares a las que se producen en una red inalámbrica atendiendo a las variables de pérdida de paquetes y retardo.

La creación de esta librería se enfocará de la siguiente manera: dado que el principal problema de infrautilización del canal por parte de TCP en redes inalámbricas es producido por el uso de un método de control de flujo que identifica las pérdidas de paquetes como un síntoma de congestión, se va a implementar una librería que sea capaz de ignorar los paquetes perdidos o corruptos y pueda estimar el ancho de banda del canal (y por tanto la congestión del mismo) de una manera distinta.

La manera de ignorar la pérdida y corrupción de paquetes es usar un sistema de FEC o corrección de errores hacia delante. Esto nos permite ser capaces de recuperar la información recibida aunque perdamos un conjunto de mensajes, dado que se envía un código de corrección de errores anexo al mensaje enviado. En este caso se ha usado el sistema de Raptor Codes que permite, en un coste lineal al tamaño del envío, generar los códigos redundantes para su envío.

La estimación de la capacidad del canal se hará en base a la recepción de los paquetes, emitiéndolos a ráfagas y calculando la diferencia entre el primero y el último.

Además, para obtener el resto de atributos que caracterizan TCP, como es el hecho de estar orientado a conexión, se usará como envoltura para todo esto la librería UDT, a la que se le harán una serie de modificaciones que permitan integrar todos los elementos de los que hemos hablado.

1.3.- Estructura del documento

En la primera sección se hablará sobre los trabajos previos que han tocado tanto la problemática de TCP sobre las redes inalámbricas como la solución de este problema con el uso de sistemas FEC. En la segunda sección se describirá el funcionamiento de los sistemas de corrección de errores hacia delante, haciendo especial hincapié en los Códigos Raptor (Raptor Codes) y se detallarán también sus características. En la tercera sección se expondrá la estructura de la librería UDT, así como su filosofía y funcionamiento. En la cuarta 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 quinta sección se entrará en un nivel técnico dentro del proyecto comentando en más detalle la implementación concreta de cada uno de los elementos, las distintas alternativas de implementación que se experimentaron, y el conjunto de problemas que surgen en todo proyecto; cabe destacar que en él se tocan temas tan delicados como el uso de tecnologías punteras y desconocidas por el gran público y la modificación de código ajeno partiendo de una documentación muy básica, detallando asimismo como fueron solventados. En la sexta sección se expondrán el conjunto de validación y pruebas a las que se sometió la aplicación para verificar su correcto funcionamiento, detallando la problemática surgida en el momento de la validación de un sistema tan especial y dependiente de tantas variables como son los sistemas de comunicación en red. En la séptima 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 (100 Mbps – 10 Gbps).
- **RTT grandes:** El tiempo de ida y vuelta de un paquete o 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.
- **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 ser la red inalámbrica un modo de conexión sin cables es la opción más usada en dispositivos móviles que dependen de la vida de sus baterías, por lo que esta característica gana en importancia.

Como se puede ver, las diferencias entre las redes cableadas y las inalábricas, 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, desde un punto de vista más particular centrándose solo en los enlaces inalámbricos, como para redes mixtas (parte cableada, parte inalámbrica) o como 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 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 enrutamiento, control de flujo, reconexión con los dispositivos móviles, etc.
    - **Wireless-TCP:** Protocolo diseñado para trabajar sobre WWAN, que 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.
- **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 resetear 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 reenrutamiento 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 reestablece. 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.

3.- Funcionamiento y características de Raptor Codes

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 virtualmente en cualquier caso la información.

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, 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, que haciendo una aproximación estadística gracias al algoritmo de Viterbi pueden recuperar la información que el emisor quería enviar.

En este proyecto se usan los Raptor Codes, 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 cualquier de 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 del emisor.

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 les 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 para poder recuperar la información. 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.

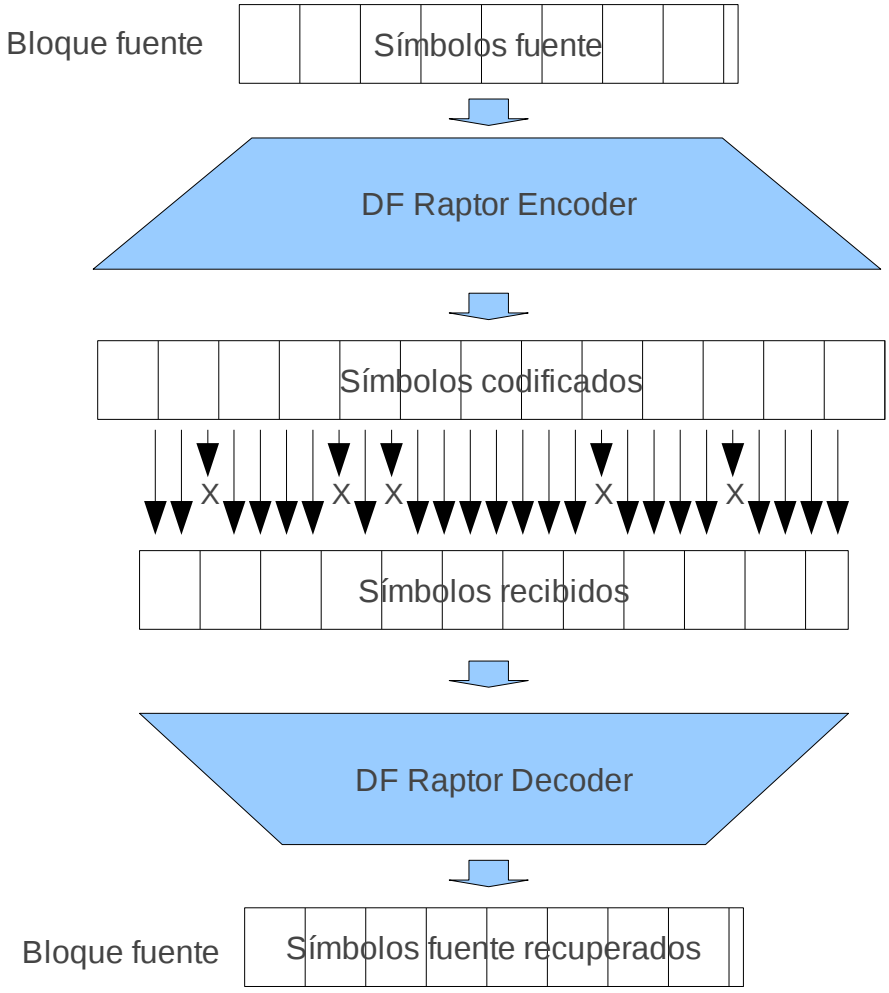

Figura 1: Esquema simplificado de codificación y decodificación de información usando Raptor Codes

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.

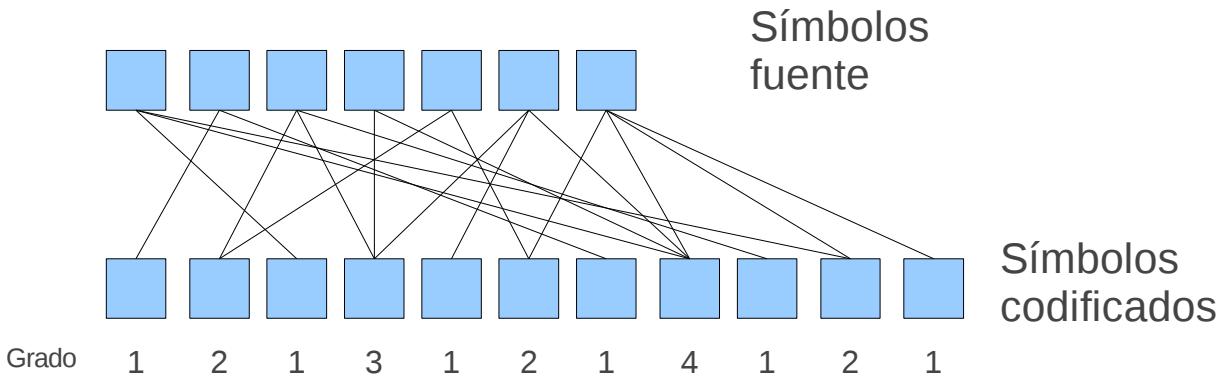

Figura 2: Creación de símbolos codificados en una sola fase

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.

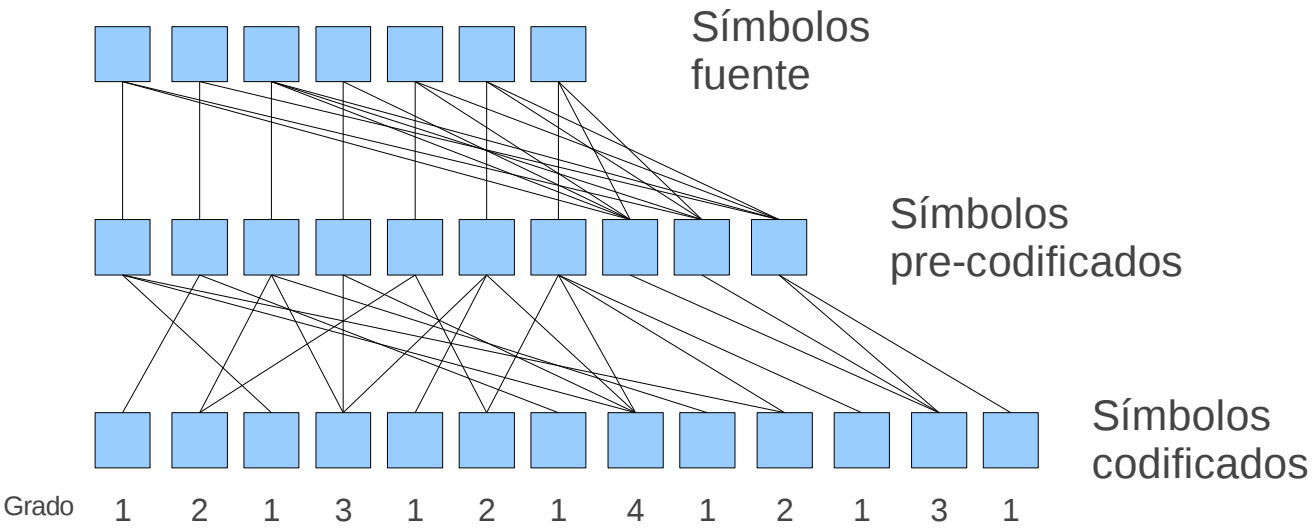

Figura 3: Creación de símbolos codificados en dos fases

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.

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.

3.2.1.- 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.

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

3.2.1.1.- 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

3.2.1.2.- 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

3.2.1.3.- 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

3.2.1.4.- 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.

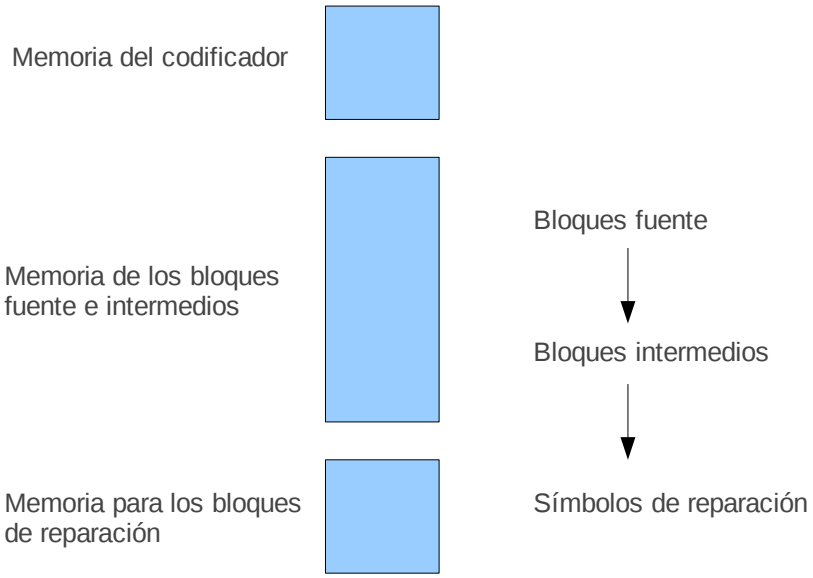

Figura 4: Modo estándar

- **Modo 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.

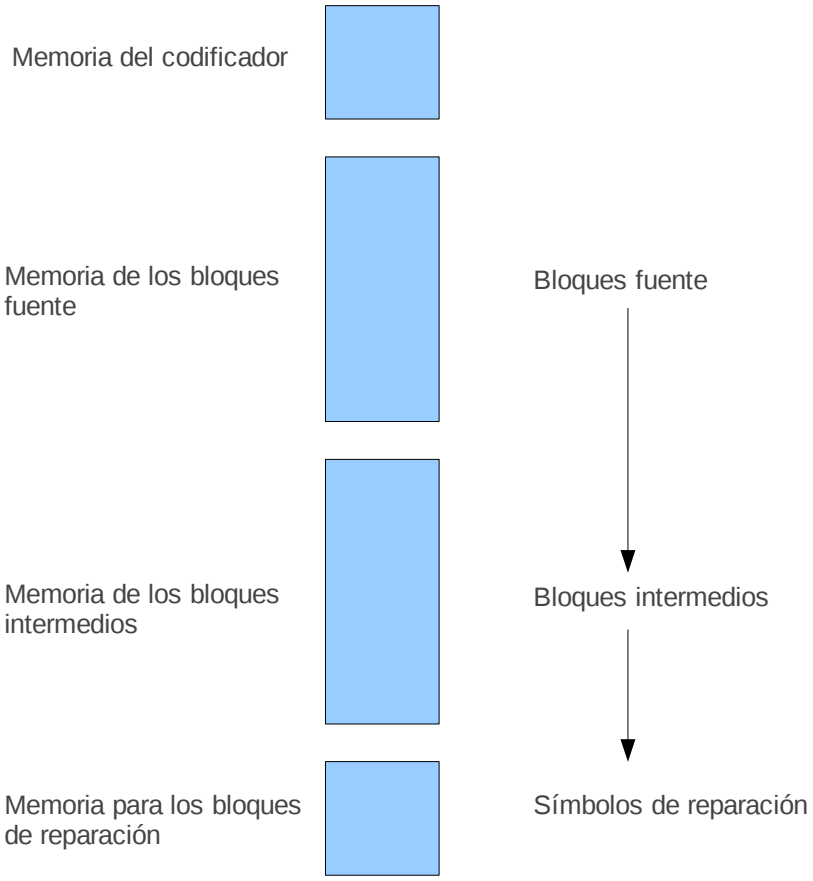

Figura 5: Modo de alta velocidad

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

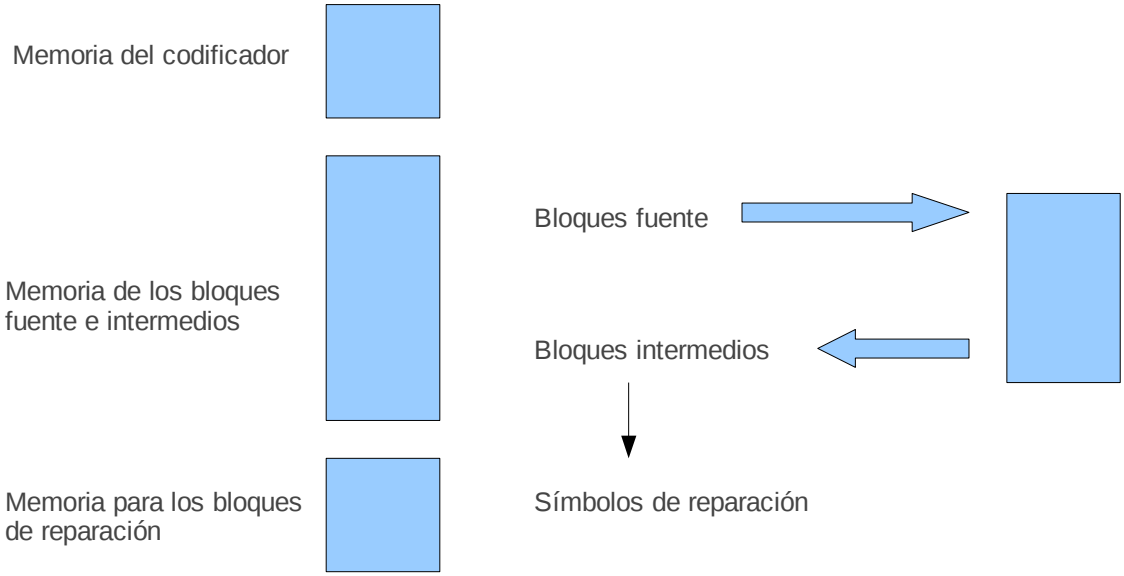

Figura 6: Modo DMA

3.2.1.5.- Flujo de operación:

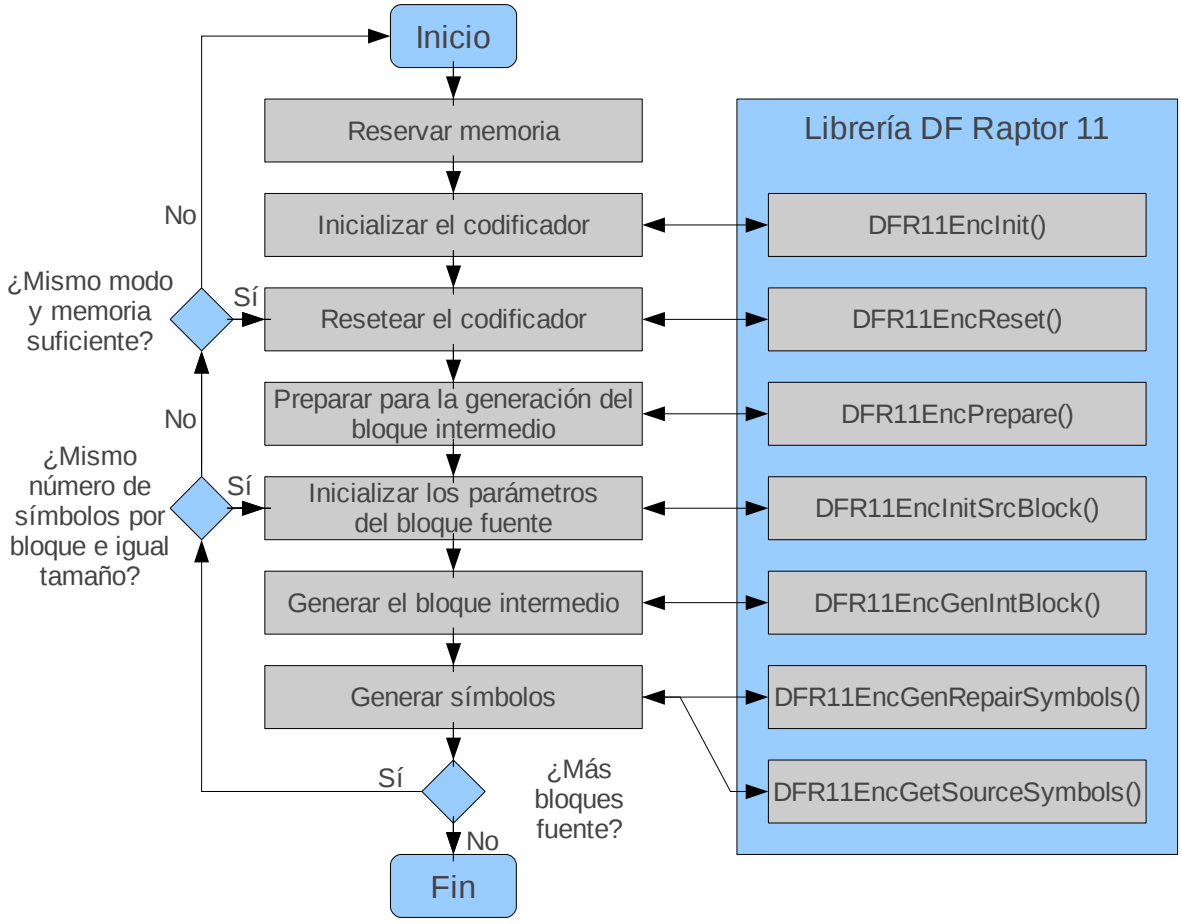

Figura 7: Flujo de operación de codificación

3.2.2.- El decodificador DF Raptor R11

Según las 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.

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

3.2.2.1.- 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	SymbolsPerBl	

	ock + 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

3.2.2.2.- 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

3.2.2.3.- 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

3.2.2.4.- 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.

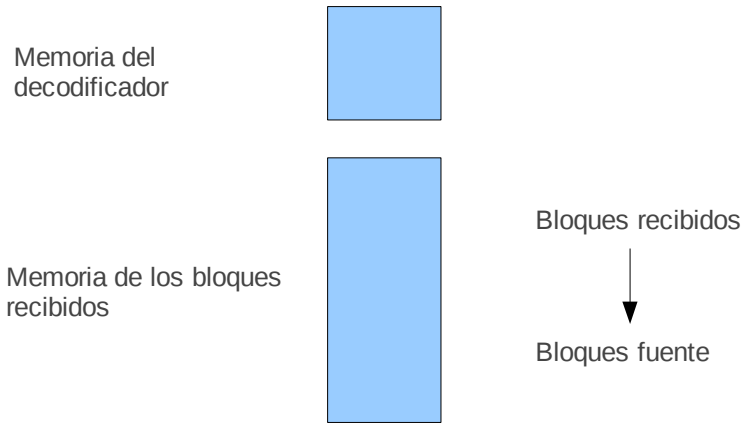

Figura 8: Modo de poca memoria

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

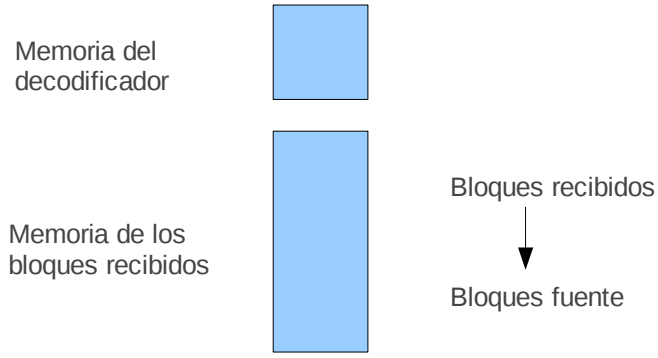

Figura 9: Modo Estándar

- **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.

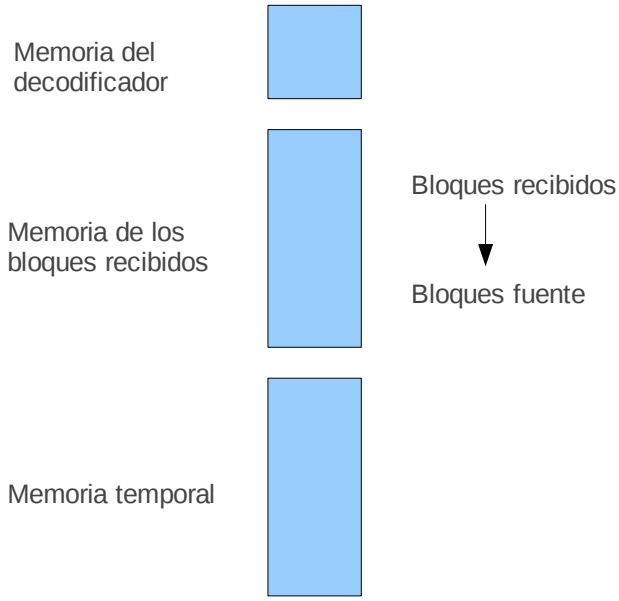

Figura 10: Modo de alta velocidad

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

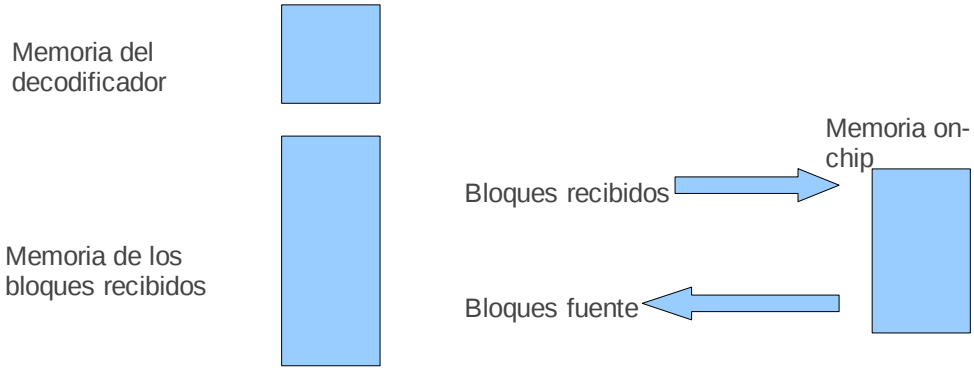

Figura 11: Modo DMA

3.2.2.5.- Flujo de operación:

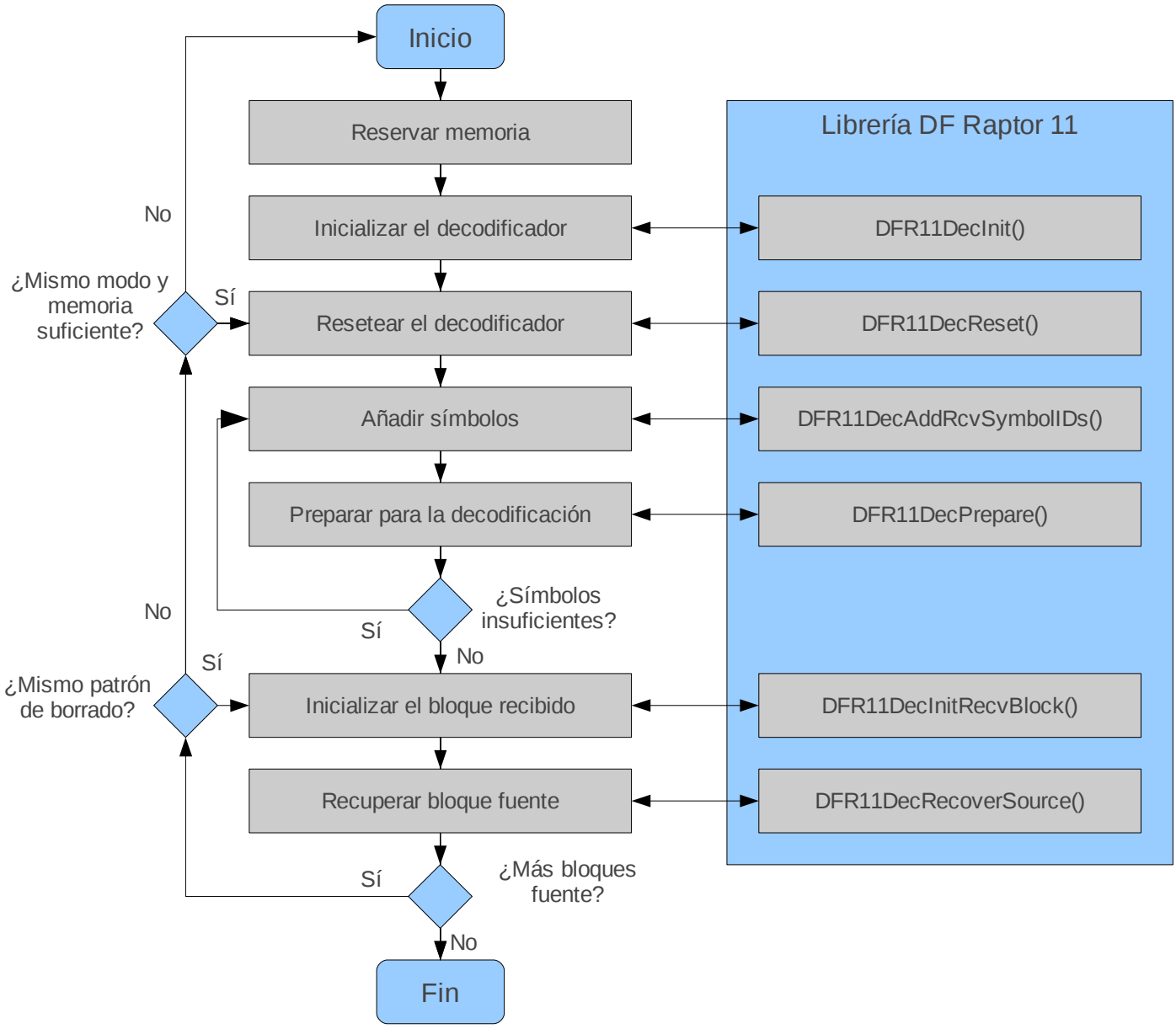

Figura 12: Flujo de operación de decodificación

4.- 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.

4.1.- El protocolo UDT

El protocolo UDT es un protocolo dúplex orientado a conexión que permite el envío confiable de información. Usa un algoritmo de control de congestión en base a la tasa de transferencia y un control de flujo basado en ventanas. El control de congestión actualiza el periodo entre paquetes a intervalos constantes, y el control de flujo actualiza el tamaño de la ventana cada vez que un ACK es recibido.

4.2.- Tipos de paquetes

En este protocolo se definen dos tipos de mensajes, 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 o un paquete de control. En este caso su valor es 0.
  - **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.

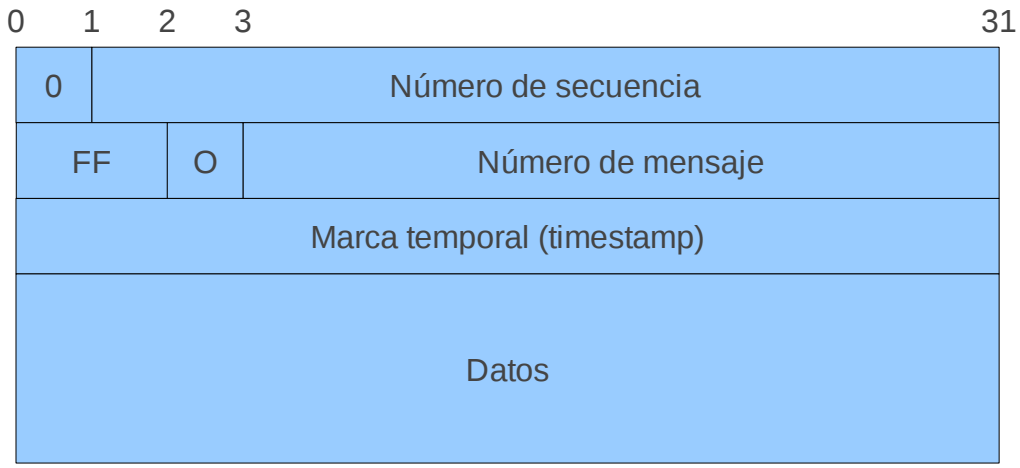

Figura 13: Paquete de datos

- **Paquete de control:**
  - **Flag bit:** primer bit de la cabecera, distingue si se trata de un paquete de datos o un paquete de control. En este caso su valor es 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.

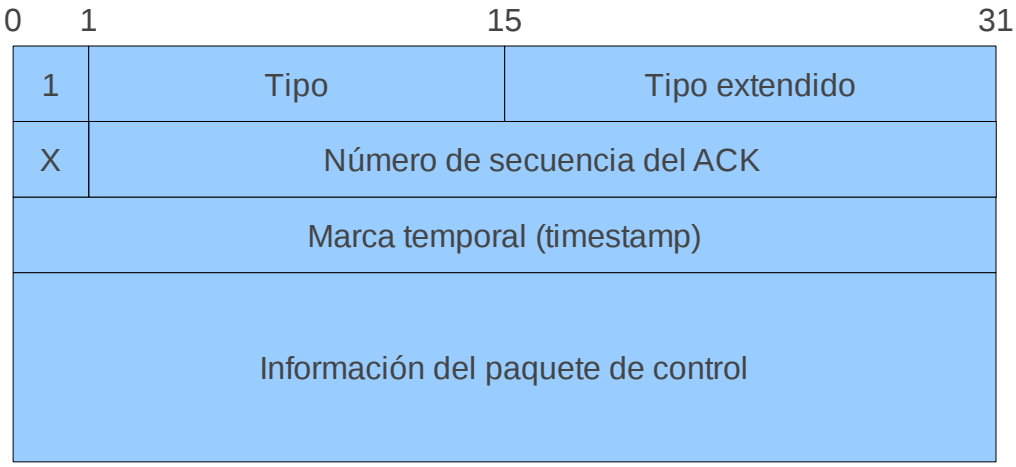

Figura 14: Paquete de control

4.3.- Inicio de conexión

UDT proporciona dos tipos de inicio de 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.

4.4.-Reconocimientos

El protocolo permite dos tipos de reconocimientos de los mensajes enviados: en base a tiempo, en el que se reconocen todos los paquetes que han sido recibidos en un intervalo de tiempo; y en base a número de paquetes recibidos, en el que se envía un paquete de reconocimiento cada n paquetes recibidos. El comportamiento por defecto del protocolo es el uso del reconocimiento temporizado en momentos de alta velocidad de recepción, y el reconocimiento en base a número de paquetes cuando el ancho de banda es pequeño.

Cada ACK es reconocido por el emisor con un ACK2 con lo que el receptor evita el envío de ACK duplicados.

El protocolo complementa este comportamiento con los NACK o informes de pérdidas. Cuando se recibe un mensaje con un número de secuencia superior al esperado, el receptor envía un informe de pérdida de los paquetes cuyos números de secuencia se encuentran entre el esperado y el recibido.

4.5.- Control de congestión

El protocolo UDT proporciona su propio control de congestión. 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.

Para cumplir con estos requisitos, el protocolo UDT establece el siguiente control de congestión en su protocolo:

$$a(x) = 10^{\lceil \log(L - C(x)) \rceil - \tau} \times \frac{1500}{S} \cdot \frac{1}{SYN}$$

Donde:

- x: Tasa de envío en paquetes por segundo
- L: Capacidad del canal (ancho de banda) en bits por segundo
- S: Tamaño del paquete en bytes
- C: Función de conversión de unidades (paquetes/segundo → bits/segundo)
- τ: Parámetro del protocolo (9)
- SYN: Intervalo de sincronización (0,01)

Este algoritmo se activa una vez recibido el primer NACK o cuando la ventana de flujo alcanza su tamaño máximo, siendo este periodo previo el periodo de contención. Este periodo solo se activa al principio de una conexión y una vez que se ha desactivado, no vuelve a activarse jamás. Durante este periodo el tiempo entre llegadas de paquetes es cero y la ventana de control de flujo se inicializa a 2, cambiando al número de paquetes reconocidos cada vez que se recibe un ACK.

El protocolo estima el ancho de banda del canal (L) enviando una pareja de paquetes (dos paquetes seguidos) cada 16 paquetes. Se almacenan los tiempos de llegada de ambos paquetes y se

estima la capacidad del canal como S/T, siendo S el tamaño medio de paquete y T el tiempo medio entre llegadas calculado a partir de las llegadas de la pareja de paquetes.

Así pues, el algoritmo de envío queda como sigue:

1) Si no hay datos de la aplicación a enviar, el proceso se suspende hasta que vuelva a ser activado por la aplicación
2) Envío del paquete:
   a) Si la lista de paquetes perdidos del emisor no está vacía y el número de paquetes sin reconocer no excede el tamaño de la ventana de congestión, elimina el primer número de secuencia de la lista y envía el correspondiente paquete
   b) Sino, si el número de paquetes no reconocidos no excede el tamaño de la ventana de congestión, se envía un nuevo paquete.
   c) Sino, se espera hasta que se recibe un ACK o un NACK, o hasta que expira el temporizador. Se va al paso 1.
3) Se invoca el método onPktSent().
4) Envía el paquete.
5) Espera el tiempo para enviar el siguiente paquete. Vuelve al paso 1.

Y el algoritmo de recepción es el siguiente:

1) Consulta los temporizadores
   a) Si el temporizador de ACK ha expirado y hay paquetes que reconocer, se envía un ACK; sino, si el intervalo de ACK definido por el usuario se ha alcanzado, se envía un ACK “ligero”.
   b) Si el temporizador de NACK ha expirado y la lista de paquetes perdidos del receptor no está vacía, se envía un NACK.
   c) Si el temporizador EXP ha expirado y hay paquetes enviados pero no reconocidos, se invoca onTimeOut() y se pone los números de secuencia de estos paquetes en la lista de paquetes perdidos.
   d) Reinicia los temporizadores que han expirado.
2) Inicia la recepción UDP. Si nada se recibe en un tiempo determinado se pasa al paso 1.
3) Si no hay paquetes no reconocidos, se reinicia el temporizador EXP.
4) Si el paquete recibido es un paquete de control, se procesa y se reinicia el temporizador EXP (si es un ACK o NACK) y se invoca la función correspondiente según el tipo de paquete (onACK(), onLoss() o processCustomMsg()). Después se pasa al paso 1.
5) Si no es un paquete de control, es un paquete de datos. Se procesa.
6) Se comprueba si se ha perdido algún paquete. Si es así, se insertan sus números de secuencia en la lista de paquetes perdidos del receptor y se genera el informe de pérdidas (NACK).
7) Se invoca onPktReceived(). Vuelve al paso 1.

4.6.- Implementación

El protocolo UDT se ha implementado en la librería UDT. Cada nodo cuenta con una entidad emisora y una entidad receptora. En cada conexión, el emisor usa su entidad emisora para enviar un paquete de datos a la entidad receptora del receptor. Esta a su vez realiza un envío y recepción de paquetes de control con la entidad receptora del emisor.

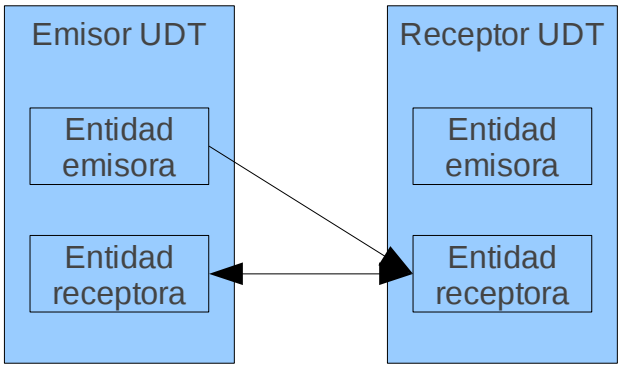

Figura 15: Esquema de entidades de la librería UDT

La librería cuenta con una primera capa de interfaz con la aplicación, la API, de la que penden dos buffers, uno para la recepción y otro para el envío, que sirven de comunicación con las entidades de emisión y recepción. La entidad emisora accede a este buffer para recoger los datos a enviar y los transmite al otro componente que hace la función de canal. Esta emisión se realizará en base a la información proporcionada por el módulo de control de congestión. Además cuenta con una lista de paquetes perdidos que usará para saber qué paquetes ha de retransmitir. La entidad receptora recoge del canal los mensajes y los coloca en el buffer de recepción. Si detecta la pérdida de un paquete la apunta también en su lista de paquetes perdidos.

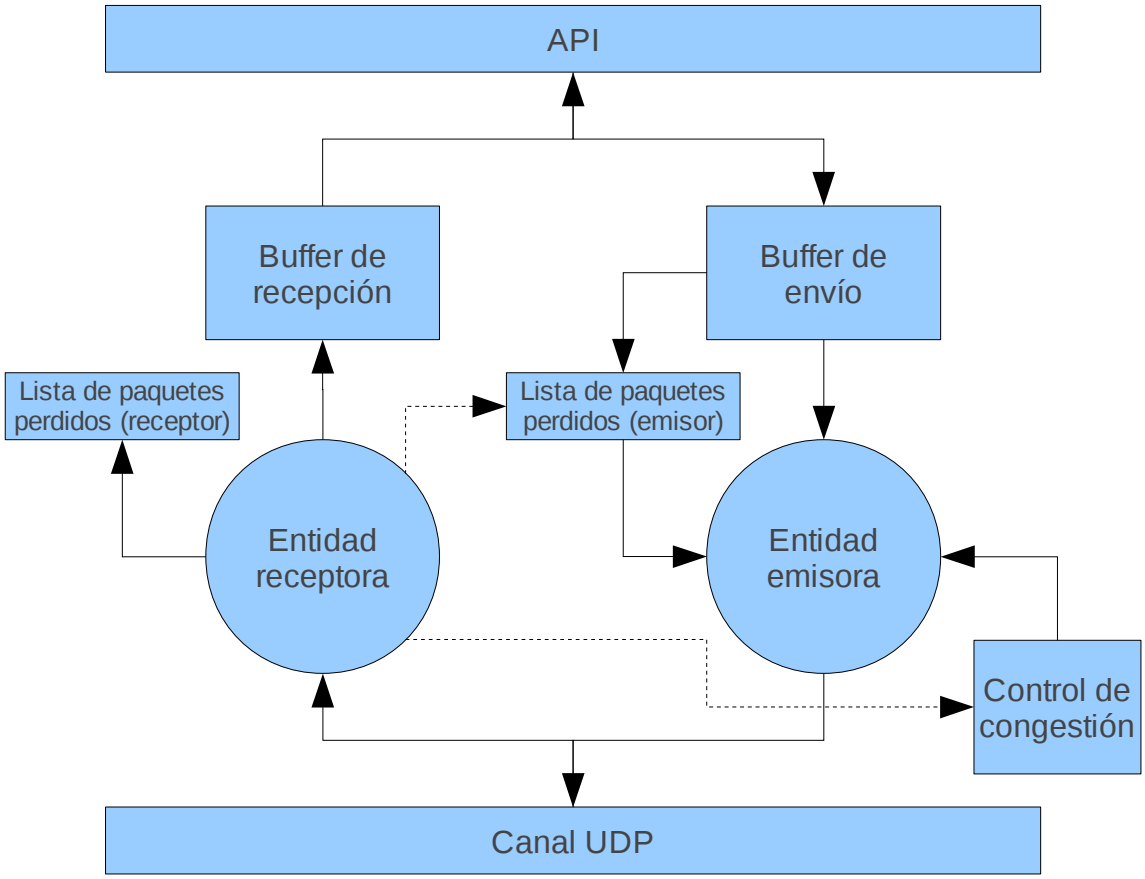

Figura 16: Diagrama simplificado de la estructura de la librería UDT

La librería UDT proporciona un entorno ideal para el desarrollo de protocolos de comunicaciones debido a que ofrece una estructura orientada a conexión sobre el protocolo UDP ya implementada. Por tanto, si se desea tener un entorno para desarrollar un nuevo protocolo de comunicación orientado a conexión evitando tareas innecesarias, como la gestión de las conexiones, y además poder trabajar a un nivel tan bajo como es el de UDP, esta librería es la elección ideal.

Además, la librería UDT proporciona una base para el desarrollo de sistemas para el control de congestión 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 a 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).

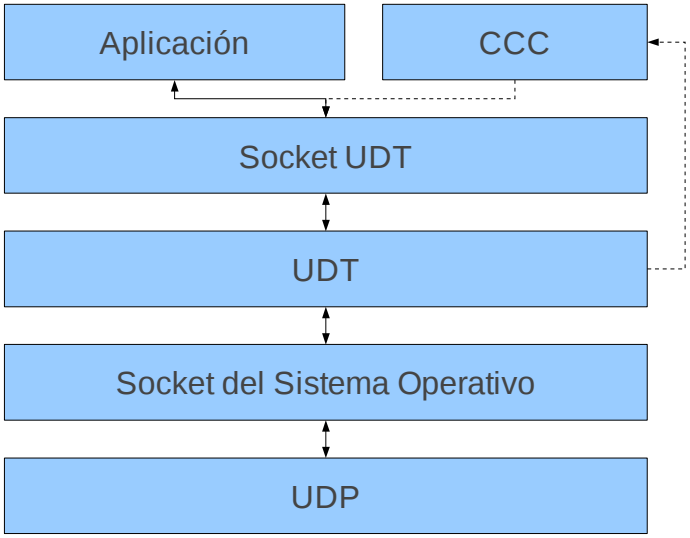

Figura 17: Diagrama de capas de la librería UDT

Para utilizar correctamente estos métodos y configurar el protocolo al gusto 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.

5.- RDT: un novedoso protocolo de transporte para redes inalámbricas basado en Raptor Codes

Con el presente proyecto, al que se le llamará librería RDT, se quiere presentar un nuevo protocolo de transporte para redes inalámbricas basado en Raptor Codes con el fin de superar en rendimiento al protocolo TCP en condiciones similares a las típicas de una red inalámbrica.

5.1.- Descripción de la propuesta realizada

La idea básica del proyecto es la siguiente. Usando como esqueleto la librería UDT, se incorporará 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.

Una vez eliminado el problema de la pérdida de paquetes, se le incluirá un control de flujo 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.

5.2.- Diseño

Como ya se ha dicho, se partirá por un lado de la librería UDT de la que se aprovechará su estructura general y su sistema de establecimiento de una conexión, así como la gestión interna de los sockets y demás estructuras de comunicación. Así se conseguirá evitar todo el trabajo trabajo no relacionado con el proyecto a realizar, pudiendo así destinar íntegramente los esfuerzos al objetivo que se pretende.

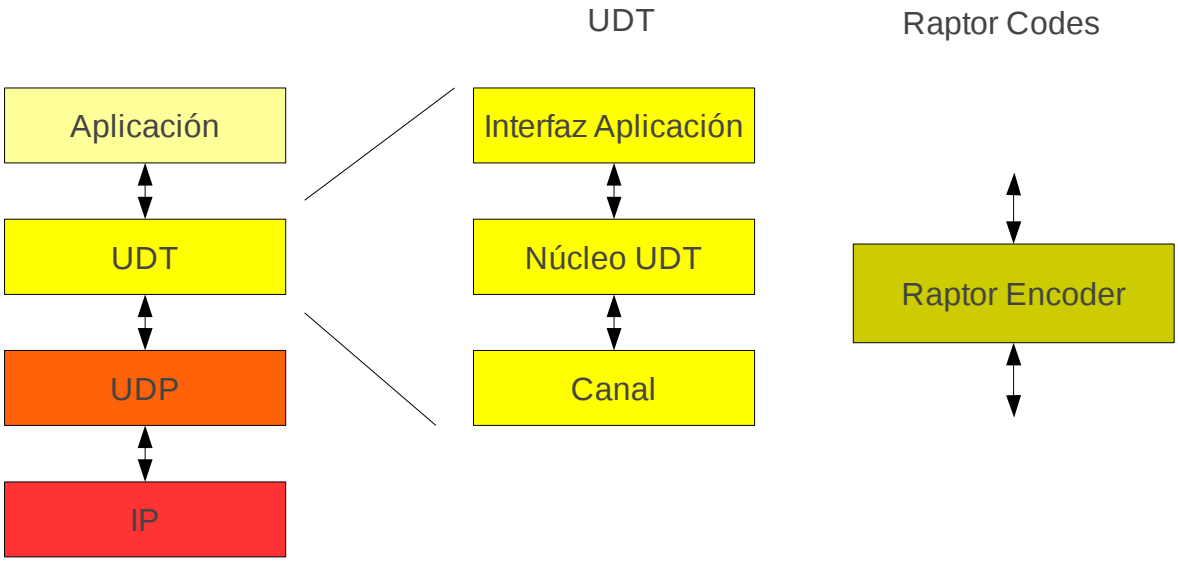

Figura 18: Punto de partida para la implementación de la librería

Por otro lado se partirá de la implementación de un codificador y decodificador de Raptor Codes de Digital Fountain. El uso de esta librería permitirá integrar fácilmente un sistema de generación de códigos Raptor en la estructura de la librería UDT.

Esta integración se realizará incluyendo 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 CCC, 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 integrará 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 conllevará la creación de nuevas estructuras de datos y control y la modificación de las existentes, que se realizarán 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 intercambien 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 nuestro propósito mantener unas estadísticas sobre el ancho de banda del canal y los paquetes recibidos por el emisor. Esto conlleva el incluir un nuevo tipo de paquetes de control, el segundo si recordamos el que se incluyó para el cambio de bloque, 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 flujo de este proyecto, destacando cómo encaja este paquete exactamente con la manera en que esta librería envía los paquetes de datos.

Como se va a cambiar totalmente el control de flujo de la librería UDT para sustituirlo por el control de flujo de la librería RDT, esto nos lleva a que algunos de los paquetes de control que antes se utilizaban en la librería UDT deban 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 se eliminarán 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 extras 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 ya serán necesarios.

Por último, para implementar este nuevo control de flujo de la librería RDT se creará una nueva clase heredada de la clase CC de la librería UDT original. Esto se hará así para seguir con 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.

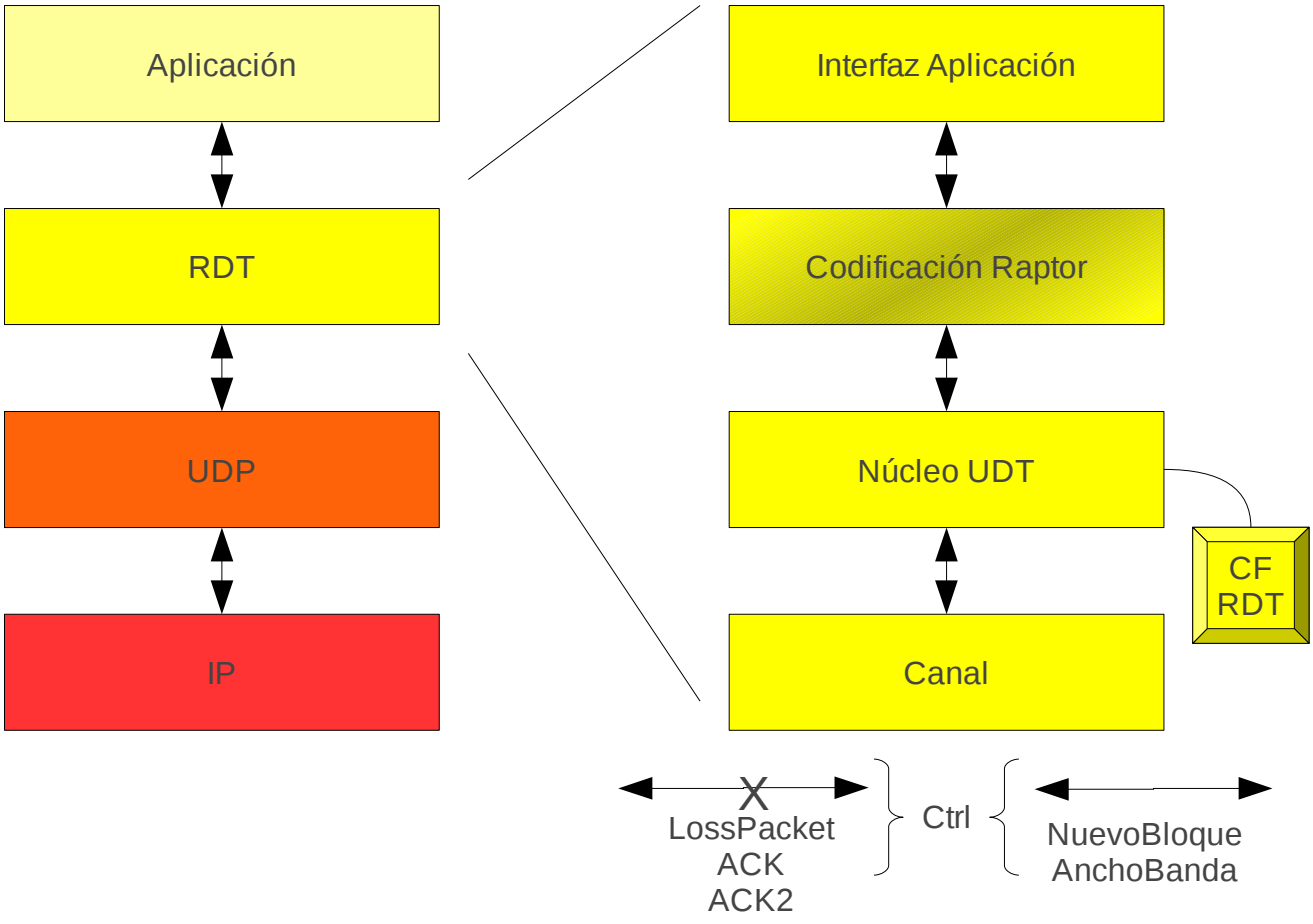

Figura 19: Diseño final de la librería RDT

Destacar que todas las modificaciones que se han descrito anteriormente se realizarán de manera que la librería RDT pueda ser usada tanto como la antigua librería UDT o como la novedosa librería RDT con las nuevas ventajas que esta ofrece.

5.3.- El control de flujo

En la librería RDT se ha implementado un control de flujo basado en el ancho de banda del canal. Se comienza por realizar una estimación en base a la recepción de paquetes y enviar 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 estimado 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 $t_{i+1} = t_i$ + tamaño_rafaga x $1/R_i$, tendremos:

Envío:

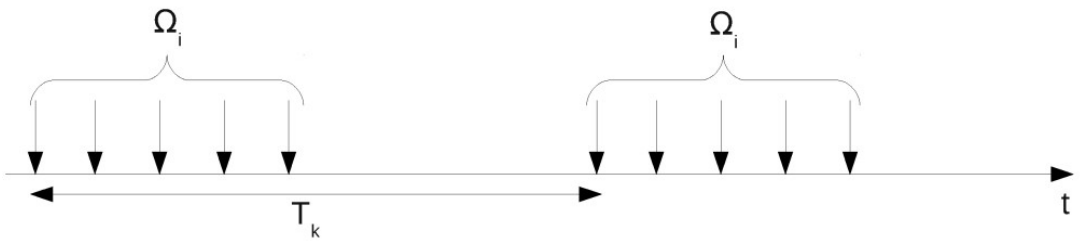

Figura 20: Envío

Recepción:

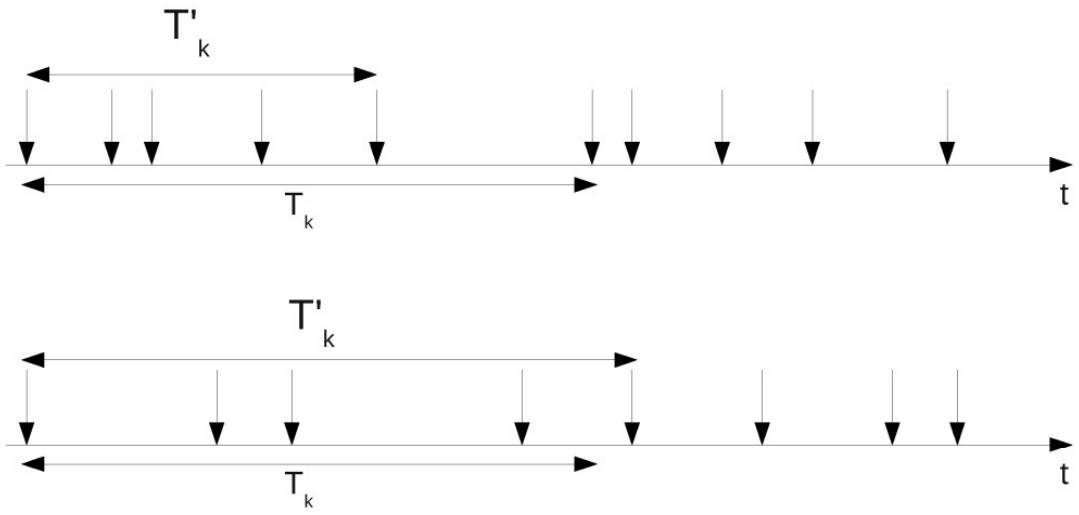

Figura 21: Recepción

5.4.- Funcionamiento

Hasta ahora se han descrito las modificaciones a realizar sobre la librería UDT y el sistema de control de flujo 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).

6.- 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.

Cabe destacar que el principal fin de esta implementación no es conseguir la máxima eficiencia en el procesamiento de la información a transmitir, sino que se quiere conseguir la implementación más apropiada para conseguir verificar con el mayor rigor y seguridad un nuevo sistema. Por tanto no se pretende con este proyecto conseguir una versión óptima a un nivel de comercialización de una librería, sino realizar la prueba de la bondad de un concepto.

6.1.- Codificación y decodificación Raptor

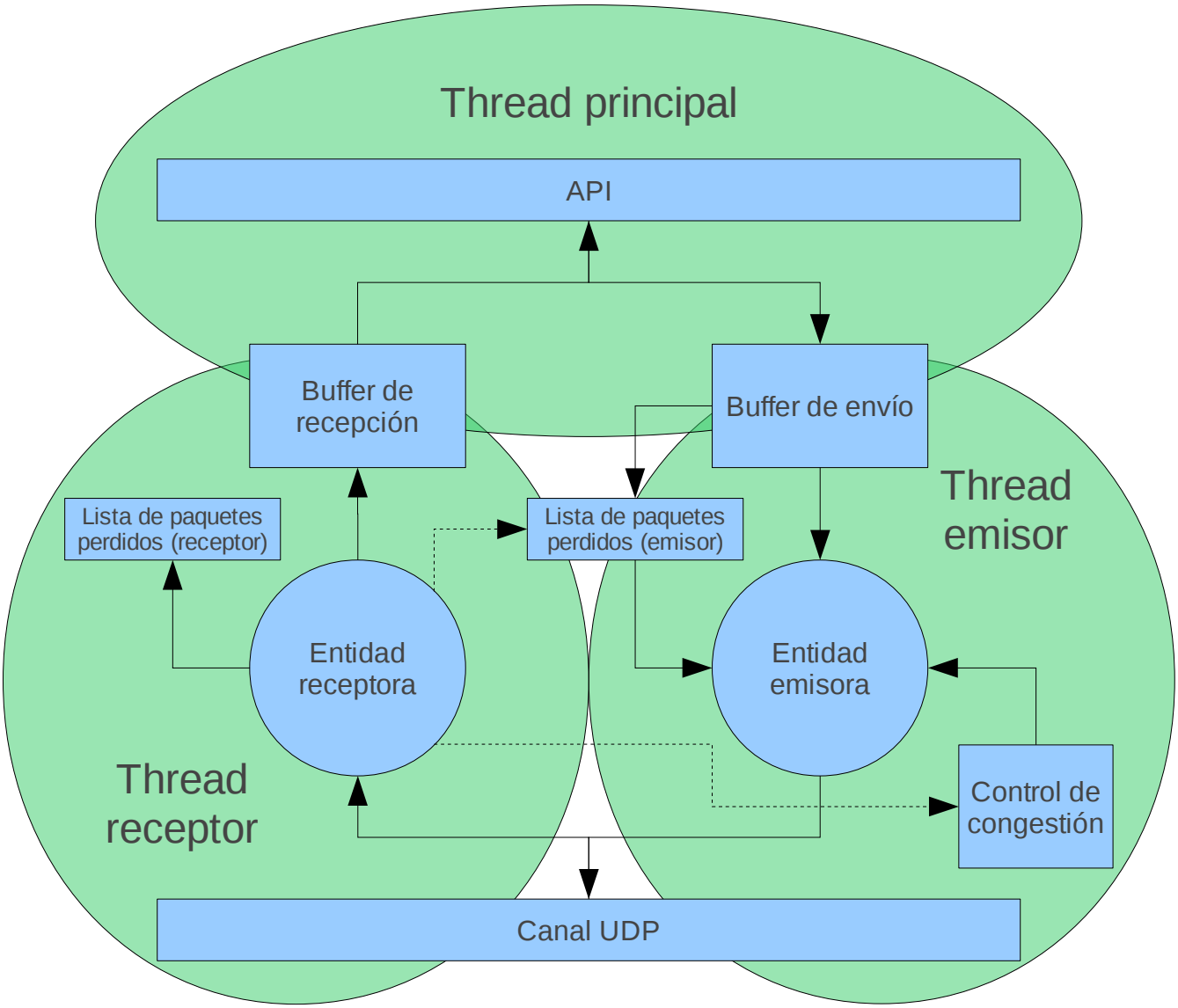

Figura 22: Diagrama de bloques de la librería UDT, incluyendo threads

Volviendo al esquema donde se mostraban las entidades que conforman la librería UDT, se comentará que hay principalmente tres threads que entran en juego a la hora de enviar un mensaje. El primero de ellos es el thread 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 thread de emisión y el thread de recepción de información. Cada uno de esos threads se encargan de recoger la información que le proporciona el primer thread y enviarla, así como de leer la información del canal y pasarla al primer thread, respectivamente.

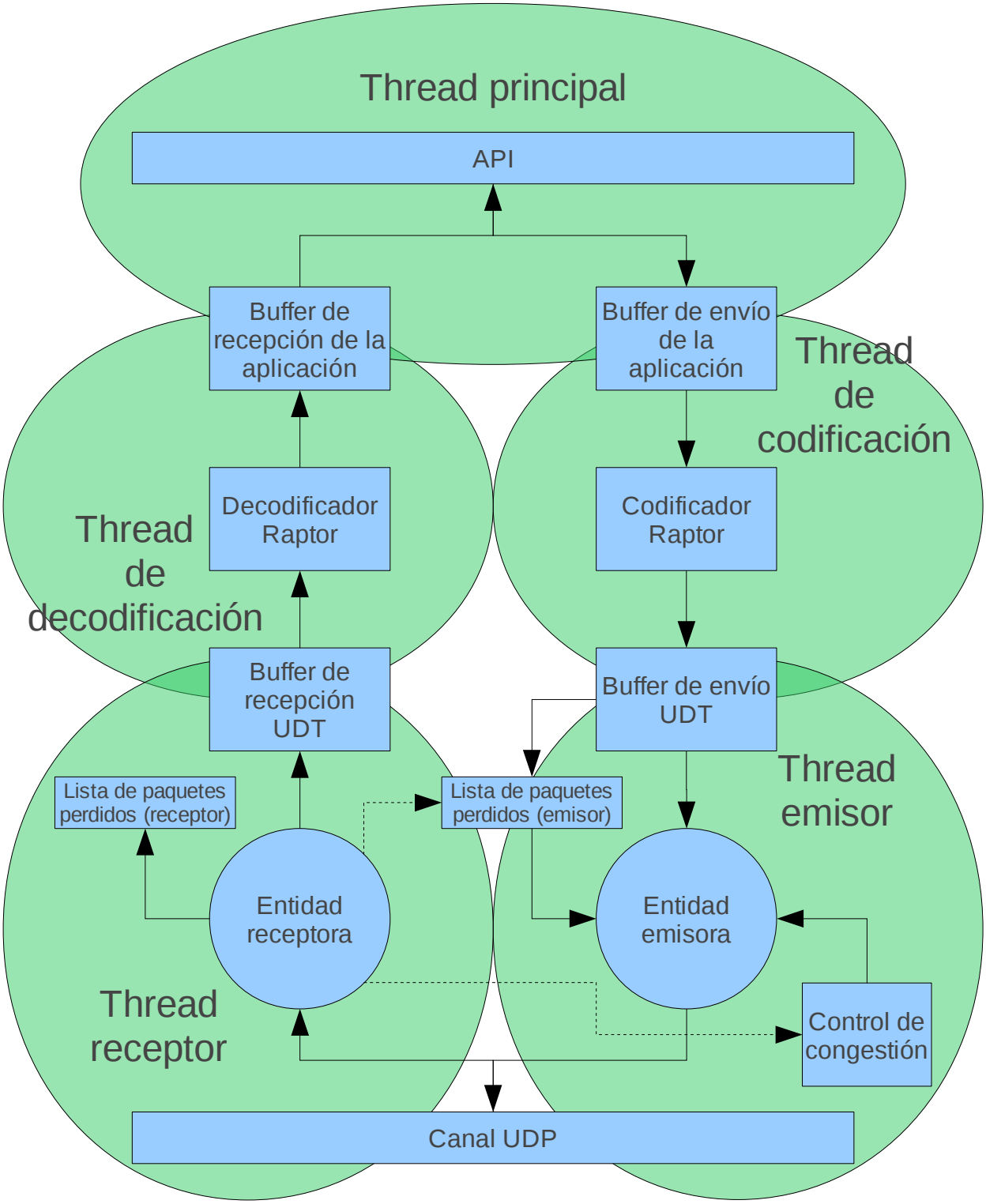

Figura 23: Diagrama de bloques de la librería RDT

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

La implementación de este thread se ha realizado a imagen y semejanza de los threads de Java, creándose una clase que también es un thread y que cuenta con un método run y otro método start: el primero contiene el código que ejecutará el thread independiente, y el segundo es el método encargado de lanzar el thread 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 diseño de threads de java lográndose un gran nivel de encapsulación, lo que permite una depuración mucho más exhaustiva de las funciones de codificación y decodificación Raptor, obteniendo así un proceso de codificación y decodificación mucho más robusto.

Esta clase cuenta también con un método, el método nil, que simplemente copia 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 thread se lleva a cabo con el método start, que invoca a la librería pthread para crear el nuevo thread.

```
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 thread difiere del utilizado por la librería UDT. En la librería UDT se usa un método estático con el código a ejecutar por el thread en lugar de usar un método del objeto que posee el nuevo thread. 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 thread esperar a que el buffer se rellene 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 thread, y la longitud del envío, que acabamos de obtener al leer la información, 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 bufferes 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 bufferes 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();
```

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 thread 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 thread 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. Permite detener de manera segura la ejecución de los threads 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 threads es crucial, ya que al cerrar la librería se destruyen estructuras de datos como los buffers que son utilizadas por el thread. Si no se estableciera este método de finalización, al cerrar la librería el thread 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 thread 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 thread codificador, el método donde se implementa todo el proceso de decodificación es el método start y 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 thread 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 guarda del bucle comprueba dos cosas. La primera es que verdaderamente se haya conseguido leer algo del buffer y 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 lo mismo. 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 resetea.

```
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 la orden al núcleo de la librería 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 necesaria reservarla y liberarla en cada iteración del thread. 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 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.

6.2.- 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.

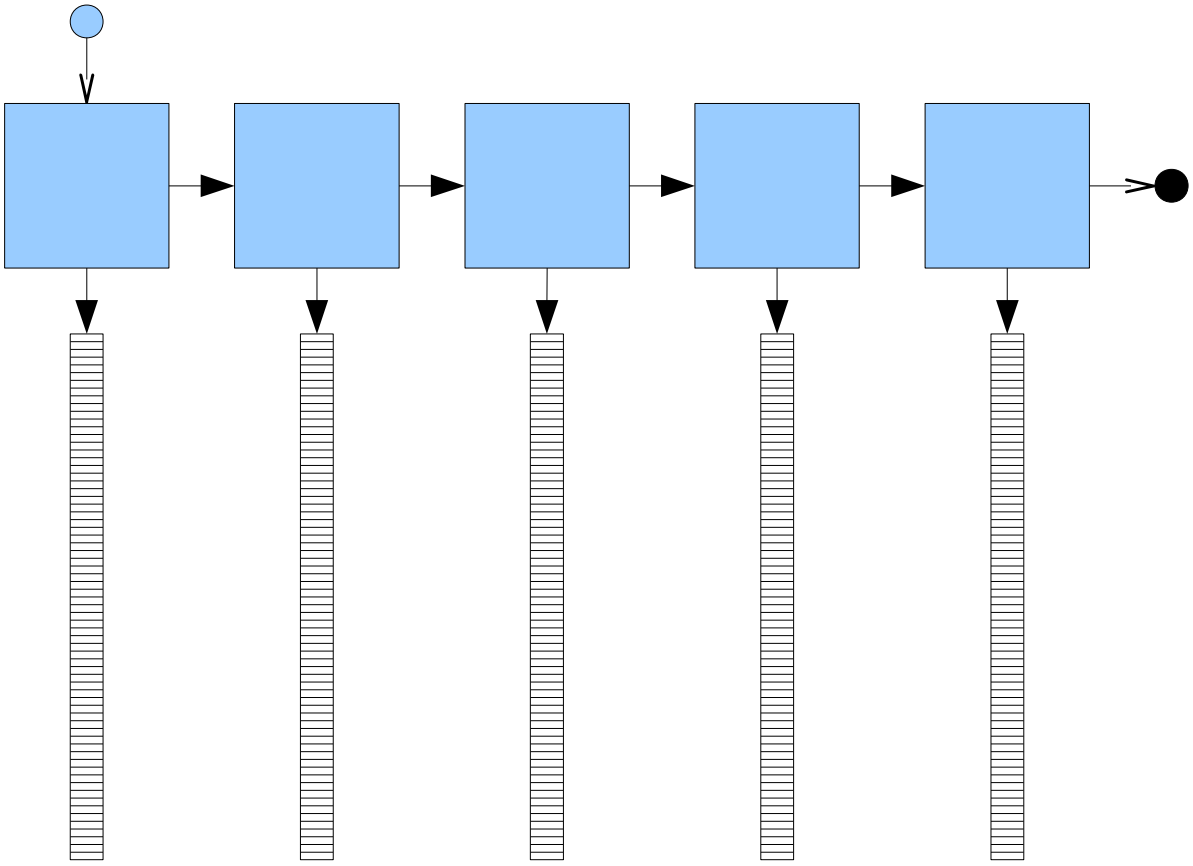

Figura 24: Diagrama de la estructura de SimplerBuffer

La declaración tanto de los bufferes como de los threads se realiza en el nucleo 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 de los buffers, es necesario duplicar también el sistema de semáforos de la librería. Esta duplicación se hace siguiendo las pautas en la implementación previa en la librería UDT. Así pues, se definen los nuevos semáforos y variables de condición, se incluyen en 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 thread se suspenda indefinidamente, queda seleccionar el momento en el que dicho thread se despertará y continuará con su cometido. Así pues, cada vez que se envía un paquete nuevo, el thread de escritura se despertará; de igual manera, cada vez que se recibe un paquete se despertará el thread 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 thread 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.

6.3.- 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 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 en dos puntos distintos similares a la siguiente.

```
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 pondrá el nuevo sistema de control de flujo en base a ráfagas que hemos introducido en los apartados anteriores.

6.4.- 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 en el 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 momento 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 destroza todo el sistema de control y estimación de flujo.

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.

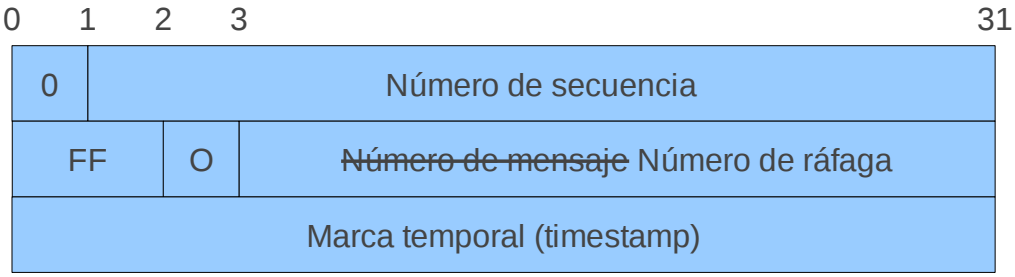

Figura 25: Modificación en la cabecera de los mensajes de datos

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 si comprueba se 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 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.

7.- Validación y pruebas

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

En la primera fase, se realizaron un conjunto de 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 este 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 uncliente 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 correctamento 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
```

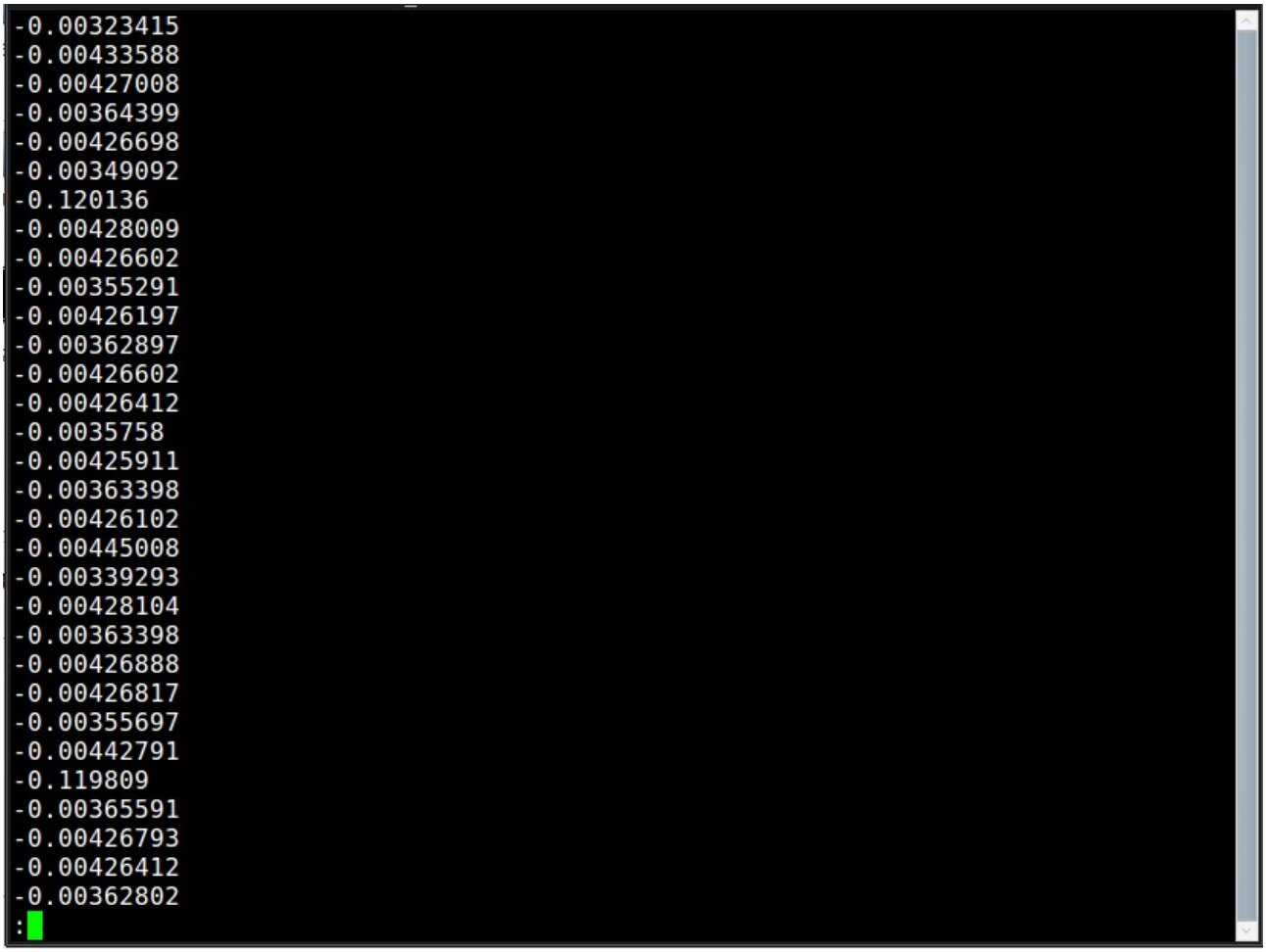

Figura 26: Diferencia temporal de emisión entre paquetes

En la figura aparece la diferencia entre los tiempos de emisión de los paquetes de datos. Se puede ver que entre todos los paquetes 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. Se ha fijado la tasa objetivo a 100 paquetes por segundo, por tanto, 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 puede comprobar que se cumplen todas esas restricciones en la figura.

Así pues, comprobado 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 realizó una pareja de programas de transmisión de ficheros idéntica a la anterior salvo por el hecho de que trabajaba con TCP.

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. Se usó un fichero de un tamaño de 5MB aproximadamente. Para la realización de todas esas pruebas se escribió un script en shell 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 siguientes, que han sido representados en 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.

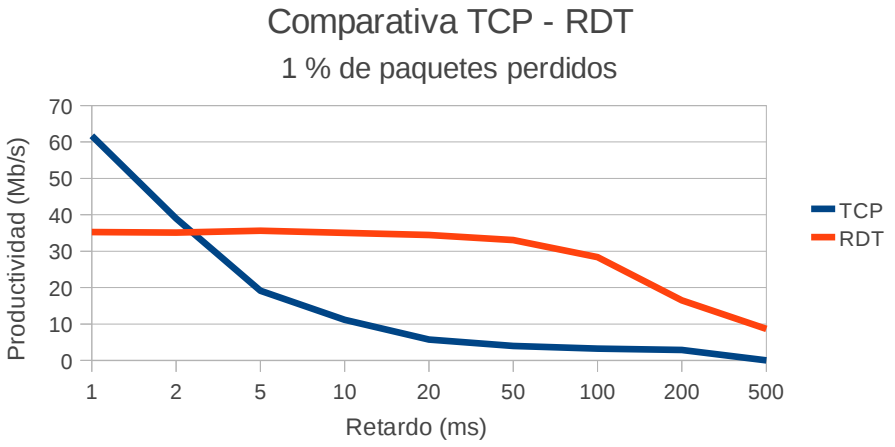

Figura 27: Comparativa TCP - RDT

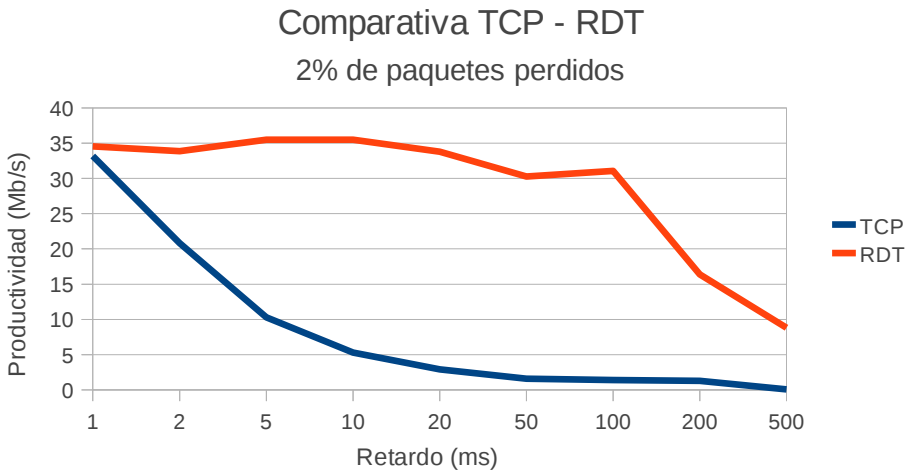

Figura 28: Comparativa TCP - RDT

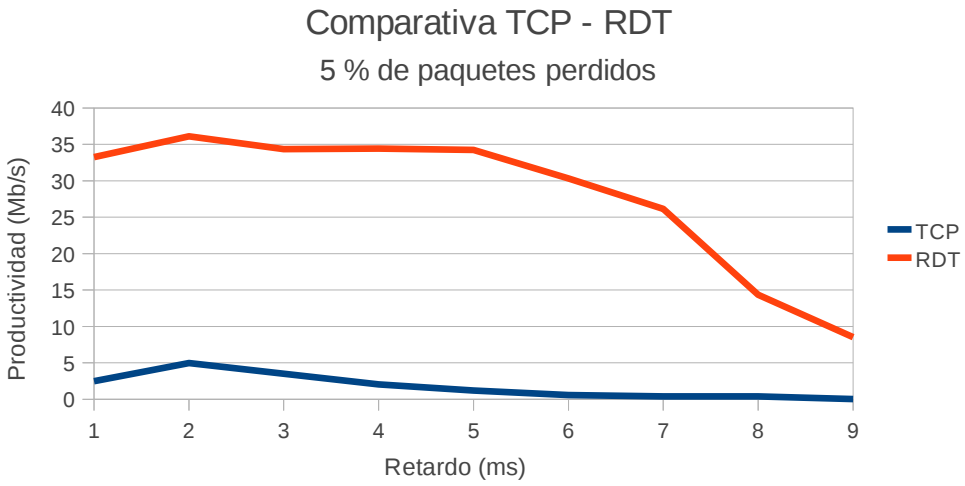

Figura 29: Comparativa TCP - RDT

En este primer trío de figuras, se ve como evoluciona la productividad de TCP y RDT conforme se varía el retardo. 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.

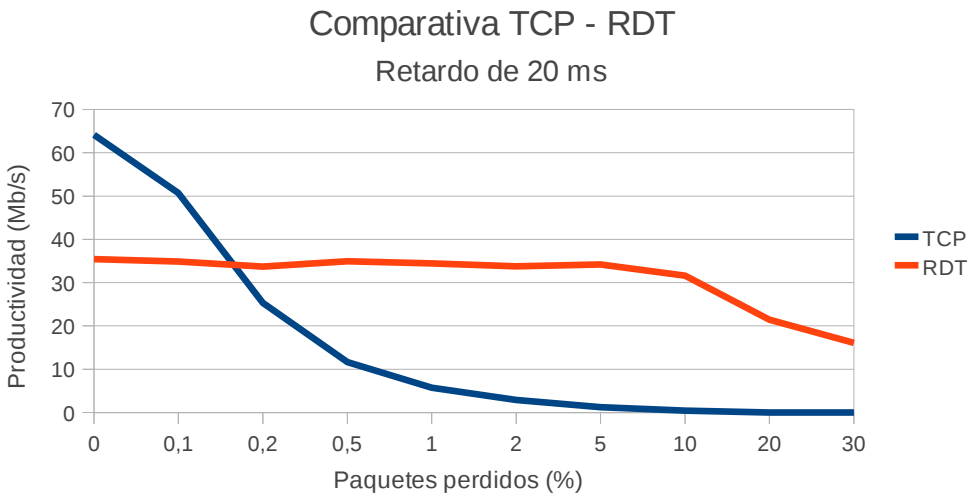

Figura 30: Comparativa TCP - RDT

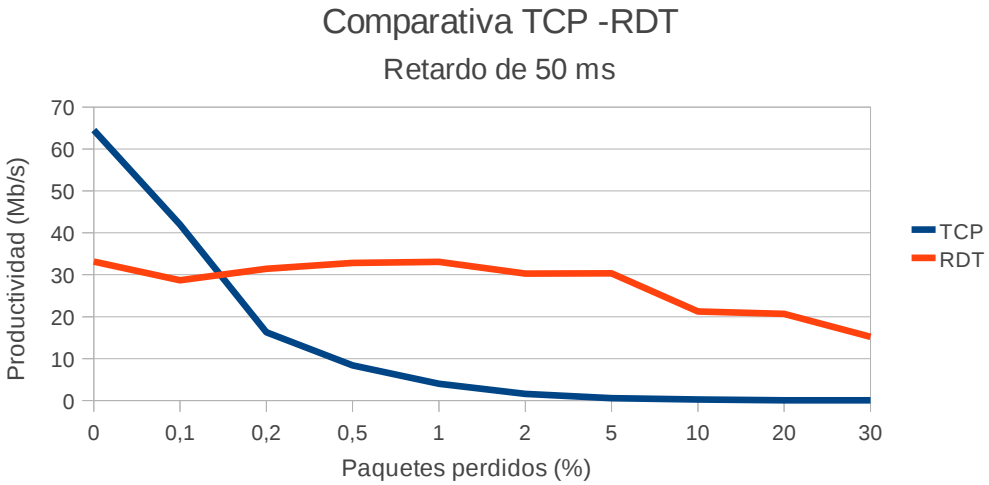

Figura 31: Comparativa TCP - RDT

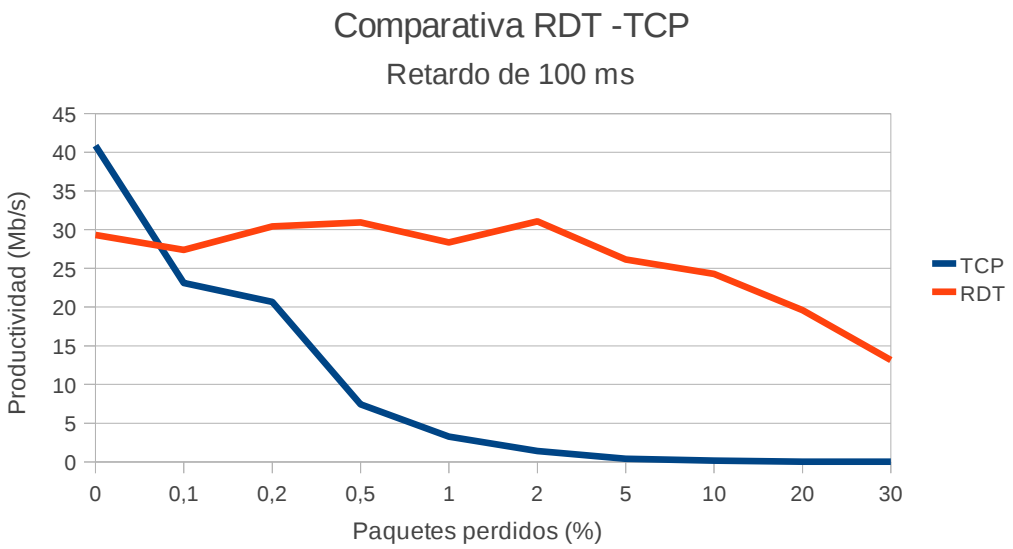

Figura 32: Comparativa TCP - RDT

En este segundo trío de imágenes se ve como evoluciona la productividad de TCP y RDT conforme se varía el porcentaje de paquetes perdidos. 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.

8.- Conclusiones y trabajo futuro

La eficiencia del protocolo TCP sobre las redes inalámbricas se ha visto ampliamente cuestionada en todos los ámbitos posibles. Muchas son las soluciones que se han propuesto a este problema, de muy distintos ámbitos y con muy distintos enfoques. En el presente documento se a descrito una nueva propuesta 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.

Además, el presente proyecto es solo una primera aproximación al problema por esta vía, y se espera que afinando más la solución implementada se pueda mejorar bastante el rendimiento y aumentar la distancia entre TCP y RDT.

Este ha sido un trabajo largo y duro y, aún así, tras este proyecto queda bastante trabajo por hacer. Una parte de él ya se ha introducido antes, y otra parte son mejoras de detalles concretos y otras requieren un trabajo sobre la librería en más profundidad.

Una gran parte del trabajo es la estimación de los parámetros óptimos. Hay parámetros como son el alfa y el beta del sistema de control de flujo que se han dejado en un valor fijo, sin comprobar si un valor distinto puede mejorar el comportamiento del protocolo. En el futuro habrá que realizar un estudio del rendimiento en más profundidad para determinar el valor óptimo de los parámetros.

El booleano useRaptor del que tanto se ha hablado adquiere un valor fijo en el momento de compilar la librería. Una posible modificación sería el incluir la posibilidad de que el valor de esta variable se pudiera establecer usando el sistema de opciones de sockets que ofrecía la librería UDT.

La librería UDT ofrecía capacidad para ser compilada para Windows o Linux, en función de las variables de compilación definidas. Esta capacidad se ha perdido en la nueva librería RDT dado que las librerías Raptor solo están implementadas para entornos Linux/Unix. Una posible ampliación de la actual implementación sería volver a recuperar esta compatibilidad.

También se debería estudiar mejorar el sistema de pruebas. Si bien todos los resultados que se han obtenido han sido precisos, el tiempo empleado para obtenerlos ha sido demasiado grande. Además, cuando se daban condiciones extremas de retardo (> 500 ms.) y pérdida de paquetes (> 30 %) no ha podido ser posible obtener datos.

Una importante ampliación que mejoraría el rendimiento de la librería notablemente sería el proporcionar soporte a multithreading. Esto permitiría una codificación de los bloques en paralelo, permitiendo una mejora en el rendimiento en fases claves como el cambio de bloque de codificación.

9.- 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