UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

Escuela de Ingeniería de Telecomunicación y Electrónica

PROYECTO FIN DE CARRERA

IMPLEMENTACIÓN DE UN PROTOCOLO ESTÁNDAR CON ARQ PARA COMUNICACIONES HF

Titulación: Ingeniero en Electrónica
Autor: D. Víctor Alonso Eugenio
Tutores: Dr. D. Iván A. Pérez Álvarez
Dr. D. Roberto Esper-Chaín Falcón
D. Himar Alonso Díaz
Fecha: Julio de 2017

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

Escuela de Ingeniería de Telecomunicación y Electrónica

PROYECTO FIN DE CARRERA

IMPLEMENTACIÓN DE UN PROTOCOLO ESTÁNDAR CON ARQ PARA COMUNICACIONES HF

HOJA DE FIRMAS

Firma de los tutores

Fdo. Dr. D. Iván A. Pérez Álvarez	Fdo. Dr. D. Roberto Esper-Chaín Falcón	Fdo. D. Himar Alonso Díaz

Firma del alumno

Fdo. D. Víctor Alonso Eugenio

Fecha: Julio de 2017

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

Escuela de Ingeniería de Telecomunicación y Electrónica

PROYECTO FIN DE CARRERA

IMPLEMENTACIÓN DE UN PROTOCOLO ESTÁNDAR CON ARQ PARA COMUNICACIONES HF

HOJA DE EVALUACIÓN

Calificación: _____

Presidente	**Secretario**	**Vocal**
Fdo.: _____	Fdo.: _____	Fdo.: _____

Fecha: Julio de 2017

```
victoralonso@pfc ~/PFC/ $ diff agradecimientos_2011.txt agradecimientos_2017.
    txt

10c10
< A los personajes del McMorrigans, a las amistades verdaderas, a las
    cervezas de barril y a los packs de cocacola. Fer, sé que dentro de 20
    años quedaremos a tomar una cerveza (y tú un refresco) y podremos brindar
    por 35 años de amistad.
---
> A los personajes del McMorrigans, a las amistades verdaderas, a las
    cervezas de barril y a los packs de cocacola.
30a31,40
>
> Ha pasado mucho tiempo desde el último (y primer) documento en el que tuve
    un oportunidad de escribir unos agradecimientos. Unas personas se han ido,
     otras han llegado, y casi todas las nombradas anteriormente se mantienen
    ahí. Vaya por delante que no pretendo que estos agradecimientos sean como
    aquellos, pues las personas otrora agradecidas, permanencen agradecidas en
     este documento, tal y como muestra el digno formato aquí presentado.
    Incluso, intentaré presentarlos sin nombrar a nadie de manera concreta, lo
     que los hace más ambiguos y divertidos, y por su puesto, menos
    comprometedores para mi.
>
> Desde entonces, han pasado muchas personas nuevas por mi vida que me han
    ayudado a crecer en diferentes ámbitos. Muchas de las personas con las que
     comparto espacio laboral cada día son geniales y me aportan mucho como
    persona y como profesional. Son la ostia. De verdad que pocos sitios habrá
     para trabajar con la calidad humana que se respira ahí, y más de uno es
    un jodido crack, algunos de ellos con los que he compartido viajes y con
    otros, cafés y de diversos temas. Y de vez en cuando se hacía pan ;-).
    Unos vienen y otros van, y van a sitios tan diversos como Madrid, Reino
    Unido o Australia. Es una utopía que algún día nos podamos reunir todos a
    trabajar en algún proyecto que nos implicase a todos, pero molaría. Les
    ofrezco un agradecimiento de corazón a todos y cada uno de ellos.
>
> También como tutores figuran personas de las que he aprendido muchísimo, y
    es al parecer una cadena en la cual he enganchado simultáneamente a dos
    eslabones precedentes a mi. Son geniales, unos cracks absolutos, y han
    contribuido de manera decisiva en mi corta carrera profesional; se puede
    incluir a este par de cracks, además, otro compañero de trabajojo (no es
    una errata).
>
> Me Incomodaría Acabar Urgidamente estos agradecimientos sin citar
    enérgicamente (1GJ) a mi familia. Ellos han estado ahí siempre, y estoy
    seguro que lo seguirán estando; es por eso que no puedo hacer más que
    citar a mis palabras escritas en el 2011, y decir que no ha cambiado ni un
     ápice lo que pienso.
>
```

Índice general

Índice de figuras

Parte I

Memoria

Capítulo 1

Introducción y antecedentes

1.1. Antecedentes

Las comunicaciones en la banda HF, que abarca el rango comprendido entre 3 a 30MHz, tienen especial interés tanto en las aplicaciones civiles como militares debido a la posibilidad de establecer enlaces *transhorizonte*. Un ejemplo de ello es el uso de esta banda en la aviación civil para el establecimiento de comunicaciones de emergencia y seguridad entre aeronaves y estaciones terrenas muy alejadas entre sí. Todas las aeronaves que realizan trayectos transoceánicos o polares están obligadas a llevar a bordo equipos de comunicaciones de este tipo.

El sistema de comunicaciones que está implementado actualmente a nivel mundial es el propuesto por el Grupo de Expertos sobre Comunicaciones Móviles Aeronáuticas (AMCP) de la Organización de Aviación Civil Internacional (OACI) y que está basado en técnicas monoportadora siguiendo un esquema muy similar a la norma militar MIL-STD-188-110A [1].

La *División de Ingeniería de Comunicaciones* del *Instituto para el Desarrollo Tecnológico y la Innovación en Comunicaciones* (IDeTIC) de la *Universidad de Las Palmas de Gran Canaria* (ULPGC) ha establecido una línea de trabajo conjunta con el *Grupo de Aplicaciones del Procesado de la Señal* (GAPS) de la *Universidad Politécnica de Madrid* (UPM), bautizada con el nombre *HFDVL (HF Data + Voice Link)* con el objetivo de mejorar las prestaciones del sistema, respetando los requisitos impuestos por el estándar, utilizando técnicas de modulación de tipo multiportadora.

Esta línea de trabajo comenzó en 1997 y tiene el soporte tanto de proyectos privados con una entidad pública como AENA (Aeropuertos Españoles y Navegación Aérea)[2][3], como proyectos financiados por el MICINN (Ministerio de Ciencia e Innovación)[4][5]. En ellos se ha desarollado la capacidad de diseñar y construir módems propios OFDM (Orthogonal Frequency Division Multiplex) además de demostrar su viabilidad para transmitir voz interactiva digital y datos.

El establecimiento de enlaces *transhorizonte* se logra con la transmisión ionosférica mediante reflexión en las capas de la atmósfera situadas entre 50 y 500 km de altura. Este tipo de transmisión puede sustituir o complementar la necesidad de enlace vía

satélite, que exige un alto coste del sistema, dependencia de un dispositivo intermedio y que no tiene cobertura en zonas polares. Sin embargo, al establecer el enlace ionosférico a través de un medio natural que depende de las condiciones atmosféricas, fenómenos meteorológicos, radiación solar o situación geográfica, se dispone de un canal variante, con un importante nivel de ruido y con desvanecimientos planos y selectivos [6].

La posibilidad de establecer diferentes caminos para establecer la comunicación, denominado como *efecto multitrayecto*, es la principal causa de que el canal presente un desvanecimiento selectivo, lo que conlleva la modificación de amplitud y fase de una parte de la banda de forma diferente a otras partes [7]. Por otro lado, debido a la naturaleza cambiante del canal en el tiempo, aparecen nulos profundos que pueden ocasionar errores a ráfagas.

Para paliar los efectos del desvanecimiento selectivo se utilizan las modulaciones multiportadora, de forma que los datos a transmitir se reparten entre las subportadoras de datos. De todas las modulaciones multiportadora, la más empleada es OFDM.

Por otro lado, la presencia de nulos profundos hace que algunas de las subportadoras OFDM resulten demasiado atenuadas y se pierda la información que transportaban. Existen diferentes técnicas que permiten solventar los efectos destructivos de los nulos profundos dependiendo de las características impuestas por el tipo de sistema de comunicación a utilizar, entre ellas, la latencia o retardo en la comunicación y la disponibilidad o no de información sobre el estado del canal.

Las técnicas tradicionales que permiten evitar la pérdida de información son la codificación en tiempo del canal y el entrelazado, sistema conocido como COFDM (*Coded OFDM*). Para minimizar la tasa de error de bit (*BER*) se debe aumentar la longitud del entrelazado con lo que aumenta el retardo en la comunicación. Otras técnicas han sido investigadas e implementadas en simulaciones o entornos reales, dentro del contexto de la línea de trabajo HFDVL, todas ellas enfocadas en el tratamiento y la mejora en la señal; esto deja un ámbito totalmente inexplorado y es en el que se plantean los objetivos de éste proyecto: **establecer un protocolo de comunicación**.

1.2. El Sistema HFDVL

El desarrollo de este proyecto se encuentra dentro del Sistema HFDVL. Este sistema, que se enmarca en la línea de investigación ya citada, pretende ser un producto final, proveyendo al operador de unas capacidades de comunicación propias de HF, y con unas características de fiabilidad y throughput mejoradas.

El Sistema HFDVL al completo consta de una plataforma hardware diseñada *ad-hoc*, sobre la que se ejecuta el sistema operativo GNU/Linux. Además, varios programas se ejecutan automáticamente al arrancar el sistema, y son éstos los que proveen los servicios de comunicación. Desde el punto de vista del operador, es la interfaz gráfica o *GUI* la que le permite comandar las comunicaciones HF.

Para completar la cadena de comunicaciones del Sistema HFDVL, es necesario

Figura 1.1: Plataforma hardware del Sistema HFDVL

disponer de una radio cuyo ancho de banda sea al menos 2,8kHz[1]. El intercambio de señal con el Sistema HFDVL se hace en banda base, y de ahí la necesidad de la radio.

1.2.1. Servicios de comunicaciones ofrecidos por el Sistema HFDLV

Las interfaces de entrada/salida dispuestas por el hardware para la interacción con los usuarios son:

- Pantalla táctil en el frontal del equipo hardware.
- Dos puertos USB, tanto para ratón/teclado como para dispositivos de almacenamiento masivo.
- Micrófono y altavoz de operador, para realizar las comunicaciones vocales.

Apoyándose en las interfaces de entrada salida, los servicios de comunicación que el modem da al operador, son:

Voz analógica

Este modo de comunicación, transfiere la señal de la radio al altavoz de operador (estando en recepción) de manera directa, y el audio del micro de operador a la entrada

[1] Típico en los transceptores comerciales de HF

de la radio (estando en transmisión).

Voz digital

La señal de voz presente en el micrófono de operador, y que se desea transmitir, será codificada a travéz de un VOCODER, y posteriormente modulada por el módem HFDVL. En recepción se realiza el proceso inverso, reproduciendo la señal resultante por el auricular de operador.

Ficheros

Al configurar este modo, es posible transmitir ficheros entre dos módems. La fuente de origen de los ficheros puede ser el USB, o también una carpeta compartida a través del protocolo de red SAMBA. Los ficheros de recepción se almacenan en otra carpeta compartida, o bien pueden ser extraidos a un dispositivo USB a través de la interfaz gráfica.

SMS

La interfaz gráfica permite ser configurada para mostrar un chat de mensajes, de manera que un operador pueda trabajar directamente sobre la pantalla del móvil conectando simplemente un teclado USB. Este modo de trabajo codifica los mensajes (SMS) de manera redundante en un único entrelazado de transmisión, de manera que se convierte en el modo de trabajo más robusto del Sistema HFDVL. Sin embargo, no provee de corrección de errores ni notificación de entrega, ya que simplemente transmite el mensaje desado, sin protocolo de corrección de errores.

STANAG 5066

El trabajo implementado en este PFC se enmarca en este servicio de comunicación. Debido a la estandarización de STANAG 5066, de cara a los diferentes programas que quieran usar el servicio, se ha podido implementar *a lo largo del desarrollo del proyecto de investigación*, diferentes aplicaciones que quedan fuera del ámbito de este PFC. Esas aplicaciones, cuyo desarrollo se justifica en el proyecto *Coincidente*[15] son, entre otras:

HFMail.

Cliente ethernet.

HFChat.

1.3. Objetivos

En el presente proyecto se desea conocer, evaluar e implementar el protocolo de comunicaciones STANAG 5066. Este estándar es ampliamente conocido en el ámbito militar, ya que añade un gran grado de confiabilidad a las transmisiones de datos que se realizan.

El desarrollo propuesto en este proyecto queda justificado por la cada vez mayor dificultad de disminuir la *BER* mediante técnicas de tratamiento de señal, por lo tanto, la solución propuesta es la de añadir un protocolo de comunicaciones. La línea de trabajo HFDVL ha estado asociada a proyectos desarrollados para el Ministerio de Defensa de España[8], lo cual ha permitido conocer las necesidades y los requisitos en las comunicaciones HF con fines militares, y permitido descrubrir la utilidad del éstandar STANAG 5066.

Además, el estándar citado define un protocolo de comunicación de cara a las aplicaciones que quieran hacer uso de él, y esto permite una gran flexibilidad a la hora de reutilizar aplicaciones ya existentes y que usen el estándar STANAG 5066; lo cual es un importante punto a favor para fomentar la penetración (en un mercado tan complicado como las comunicaciones militares), debido a que los operadores podrán seguir usando las aplicaciones a las que están acostumbrados.

El estándar STANAG 5066 define tres capas, de manera similar al modelo OSI, aunque no es posible establecer una extrapolación evidente: La capa SIS se encarga de solicitar conexiones con nodos remotos, en función de las necesidades de las aplicaciones, como funcionalidad más importante; la capa CAS gestiona las conexiones con los nodos remotos; la capa DTS realiza gran parte del trabajo de la STANAG 5066, como por ejemplo, el segmentado y reconstrucción de los paquetes en fragmentos más pequeños; su direccionamiento, cálculo del CRC, etc... La propuesta del proyecto mantendrá esta estructura de tres capas, comunicándose cada capa exclusivamente con su inmediata superior o inferior, siendo la capa absolutamente inferior la *capa módem*, y la capa absolutamente superior la *capa aplicación*, quedando:

$$\text{Aplicación} \Leftrightarrow \text{SIS} \Leftrightarrow \text{CAS} \Leftrightarrow \text{DTS} \Leftrightarrow \textit{Módem}$$

La implementación se hará en el lenguaje de programación C intentando hacer uso del estándar POSIX. El Sistema Operativo para el que se realizará dicha implementación será la distribución de GNU/Linux, Open SuSE 11.4, aunque gracias al uso de POSIX podría ser compilado para todas las distribuciones modernas de GNU/Linux. La elección del lenguaje C es debida a la versatilidad de lenguaje, la cantidad de recursos disponibles a nivel de sistema, y la eficiencia de la ejecución de el código compilado. Se descartaron lenguajes bastante populares actualmente como Python o Java por ser lenguajes interpretados y su baja eficiencia en el caso de Python, y la necesidad de tener instalada una máquina virtual Java (JVM) y el entorno de ejecución (JRE), en el caso de Java.

Cabe decir que el desarrollo del servidor STANAG 5066 y el Sistema HFDVL al completo, está enmarcado en un proyecto de investigación, en el que participan varios desarrolladores.

1.4. Contenido de la memoria

El presente documento está organizado como se indica a continuación:

Capítulo 2: Descripción de la arquitectura software implementada en el Sistema HFDVL, y cómo se integrará el software que se va a desarrollar.

Capítulo 3: Explicación del funcionamiento básico del estándar STANAG 5066, su división entre capas, y detalles del documento.

Capítulo 4: Detalle de la implementación del servidor STANAG 5066, haciendo referencias a las partes clave del mismo. Se describe cómo se lleva a cabo la implementación siguiendo las pautas definidas por el estándar.

Capítulo 5: Verificaciones realizadas al sistema, además de diferentes entornos reales de operación en los que ha sido probado.

Capítulo 6: Conclusiones del proyecto.

1.4.1. Notas al lector

A lo largo del presente documento es probable que se omitan ciertos prefijos, sobre todo al hacer referencia a funciones del código, variables, etc. Por ejemplo, las funciones de la capa DTS normalmente vienen precedidas del prefijo `S5066_DTS_`.

También hace uso en este documento del término *librerías* de código, para referirnos a *bibliotecas* que implementan una funcionalidad. Si bien es más correcto el segundo término, el primero es el más utilizado en la jerga informática.

Capítulo 2

El Sistema HFDVL

El protocolo propuesto para ser implementado en este PFC es, en sí mismo, un protocolo de comunicaciones pensado para disponer de un módem de comunicaciones con el que comunicar diferentes nodos del que hacer uso. Haciendo un símil con la pila OSI [11], este protocolo implementaría la capa "enlace de datos" [10, p.4], mientras que el módem HFDVL implementaría el nivel físico.

Dentro de la línea de trabajo HFDVL, ya ha sido desarrollado el módem SDR que usará en este proyecto, y cuenta con su propia línea de investigación, con el propósito de dar la máxima fiabilidad a las comunicaciones en la banda de HF. Este módem trata de mejorar las prestaciones y la BER respecto a módems clásicos, y sobre todo, ser más robusto respecto a otros módems y modulaciones, a través de modulaciones multiportadora y otras técnicas, como el uso de entrelazados de los símbolos tras pasarlos a través de un codificador LDPC. Estos detalles técnicos se encuentran fuera del ámbito de este proyecto.

Este capítulo describirá el funcionamiento del software que implementa el Sistema HFDVL, al menos desde el punto de vista de las capas superiores que configuran y comandan el módem, y cómo el software desarrollado durante este proyecto podrá usar los recursos porporcionados por el sistema descrito.

2.1. El Sistema HFDVL como sistema de transmisión cliente/servidor

El Sistema HFDVL, como pieza software completa, cuenta con múltiples elementos divisibles en partes más elementales, conformando diferentes componentes software que tienen significado en sí mismo, como podría ser el propio modulador/demodulador SDR (módem HFDVL), librerías de acceso al audio del sistema, librería de control de PTT, librería para la coordinación de multihilo, además de otra multitud de librerías usadas internamente. Estas piezas se usan conjuntamente creando los diferentes archivos ejecutables.

La arquitectura software elemental implementada por el Sistema HFDVL para ha-

cer útil la entrada/salida de datos del sistema, es una estructura cliente/servidor. La Figura 2.1 describe cómo está concebido el Sistema HFDVL, de manera que:

- El **módem HFDVL** es la parte software que se encarga de hacer la modulación y demodulación, esto es, traducir la señal HF, la cuál le llega como muestras de audio, a un "stream" de datos y viceversa. Este proceso es dependiente de la configuración que tenga: densidad de la constelación, LDPC, entrelazado, etc. Se implementa como una librería dentro del *servidor*.
- El **servidor** es el software que instancia y controla la librería módem, y se encarga de gestionar la entrada y salida de los datos y de la señal relativos al módem, además de instanciar tantos módems como les haya sido pedido, y asociarlos con cada dispositivo de audio. También configura sus características de modulación nombradas anteriormente (constelación, LDPC...). Además se encarga de establecer comunicación con los dispositivos de audio del que cada módem toma las muestras en recepción, o escribe sus muestras en transmisión. Las decisiones del *servidor* relativas a la configuración del módem, y a cuándo y qué *transmitir*, son tomadas desde el Cliente, el cual se comunica con el *servidor* a través de una conexión TCP/IP. El **servidor** se concibe como ejecutable.
- El **cliente** es el encargado final de proveer los datos a transmitir por el módem HFDVL, y también es el responsable último de la gestión los datos recibidos. Además, tal y como se mencionó, el cliente se encarga de gestionar las configuraciones del módem HFDVL. La parte Cliente se concibe como un ejecutable propio e independiente.

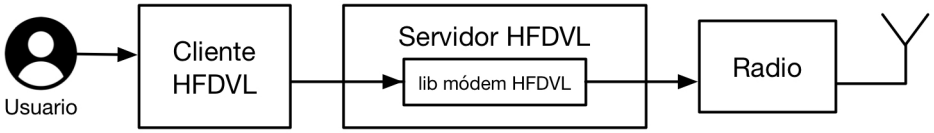


Figura 2.1: Descripción simple de la cadena completa de comunicación en un nodo

2.2. Librería módem HFDVL

Esta parte del software es la encargada de traducir las muestras digitalizadas de la señal, en un *stream* de bits, y viceversa. Para hacer esa traducción, se ha de configurar de la misma manera el transmisor y el receptor. La Figura 2.2 describe visualmente qué queremos decir por traducción de *stream* de bits a muestras digitalizadas.

La forma de onda que usa este módem es una modulación OFDM con 73 portadoras, de las cuales 60 son de datos, y 13 son pilotos para la estima del canal. Por el momento, la cantidad de portadoras es invariable y no se puede configurar, y está diseñado para que encaje en sistemas de transmisión de hasta 2,8kHz de ancho de banda, típico de los transceptores HF comerciales.

Los diferentes parámetros de configuración que sí se pueden configurar son los relativos a las 60 portadoras de datos, y éstos son:

Constelación. Cada una de las portadoras de la OFDM utiliza una modulación QAM. La densidad hacer referencia a cuántos símbolos diferentes admite esta modulación.

Codificación LDPC. Técnica de transmisión de información, por la que se puede corregir errores en un mensaje transmitido a través de un canal ruidoso.

Entrelazado. Distribuye los símbolos salientes del codificador LDPC a través de todo el bloque de entrelazado. Esto permite que si se produce una ráfaga de errores concentrada en un punto de los datos, estos datos presentes en el canal de comunicaciones están entrelazados, por lo que los errores serán distribuidos por todo el bloque de entrelazado a la hora de realizar el desentrelazado. De esta manera, el decodificador LDPC mejora la capacidad de recuperación de datos.

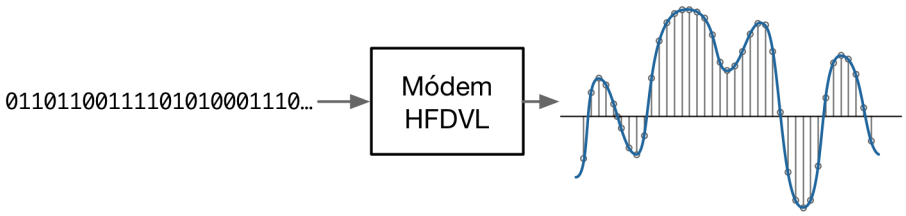


Figura 2.2: Conversión de bits a muestras de audio

El módem HFDVL cuenta con un estimador de la calidad del canal, calculando el parámetro SNR. En la Figura 2.3 se ejemplifica el cálculo de la SNR a partir de la modulación QAM, donde la potencia de la constelación está normalizada a 1.

Se pueden diferenciar tres estados diferentes en el proceso de recepción, que permiten identificar y procesar de manera adecuada las diferentes partes de una transmisión. Algunos cambios de estado producen eventos de interés para el protocolo de comunicaciones propuesto, y que ha de diferenciar entre los enventos en tramisión y en recepción. Podemos observar los estados, los eventos que producen las transiciones y los eventos generados en la Figura 2.4

IDLE. Estado de *reposo*, en el que simplemente se encuentra a la espera de detectar el sincronismo.

DATA. El módem se encuentra activamente demodulando símbolos OFDM y entregando bits demodulados al *servidor*.

EOT. Se ha detectado una secuencia de fin de transmisión, que indica al módem que ha de terminar de demodular.

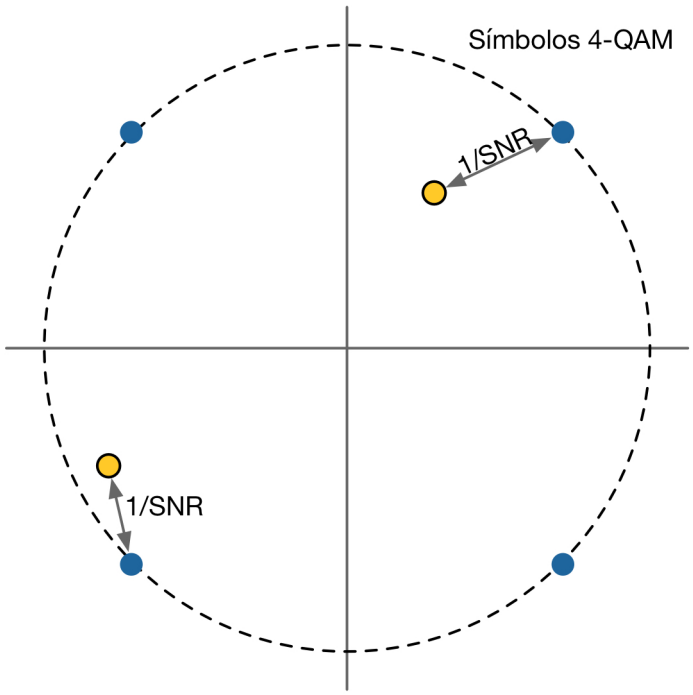


Figura 2.3: Estima de la SNR a partir de la modulación QAM

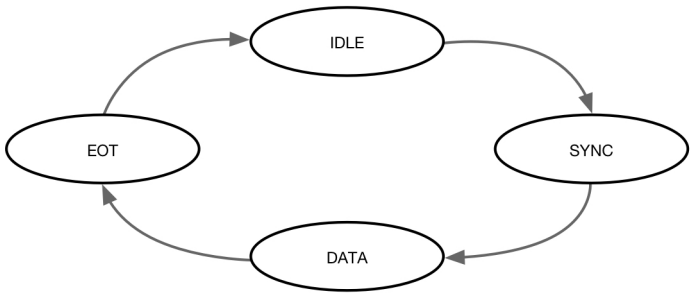


Figura 2.4: Simplificación de la máquina de estados implementada por la librería del módem

2.2.1. Secuencia de una comunicación HFDVL

En la transmisión de datos por parte del modulador HFDVL (que se implementa en la *librería módem*) se puede diferenciar tres partes fundamentales de la señal en la secuencia de transmisión. Esto se explica mejor en la Figura 2.5, en la que se puede observar cada parte tanto en el dominio temporal como de frecuencia.

Sincronía. La modulación OFDM que implementa el módem necesita que el receptor esté sincronizado con el transmisor a la hora de poder extraer la información de los diferentes símbolos OFDM. Esto quiere decir, que si el receptor intenta extraer información de un símbolo del cual no ha identificado bien el momento de inicio del mismo, estará demodulando ruido, por tanto, la parte de *sincronía* es una de las partes fundamentales de la comunicación y es la que se transmite

al inicio.

Esta parte de sincronía está compuesta por un tono de 1,5kHz, para detección de señal y sincronismo grueso en frecuencia. Le sigue una secuencia PSK pseudo aleatoria para un sincronismo temporal fino, y seguido de un símbolo OFDM de COX [12], que se encarga del sincronismo fino en frecuencia, obteniendo un error inferior a 3Hz.

Datos. Posteriormente a la sincronía tiene lugar la transmisión de datos. Como hemos mencionado anteriormente, esta parte usará los 2,8kHz disponibles de ancho de banda, colocando 73 portadoras, 60 de ellas portadoras de datos, y 13 de ellas pilotos para la estima del canal.

Esta transmisión tendrá una duración proporcional a la duración del entrelazado configurado, ya que sólo se puede transmitir un número entero de bloques de entrelazado.

EOT. Una vez finalizada la modulación de los datos, se incorporará a la transmisión la parte final, en la que se indica el fin de transmisión o EOT (*End of Transmission*). Este EOT cuenta con dos partes, el FEOT y el SEOT. El FEOT indica el fin de los datos modulados, siendo de menor duración que el SEOT. Por su parte el SEOT tiene con el propósito de dar robustez a la señalización de fin de transmisión, ya que un receptor podría quedar demodulando una transmisión inexistente (ya finalizada) por tiempo indeterminado. La librería módem también incorpora una protección con la que daría por terminada una recepción automáticamente en caso de demodular una señal con una calidad inferior a un umbral durante un tiempo prolongado. La duración del SEOT es de 6 segundos.

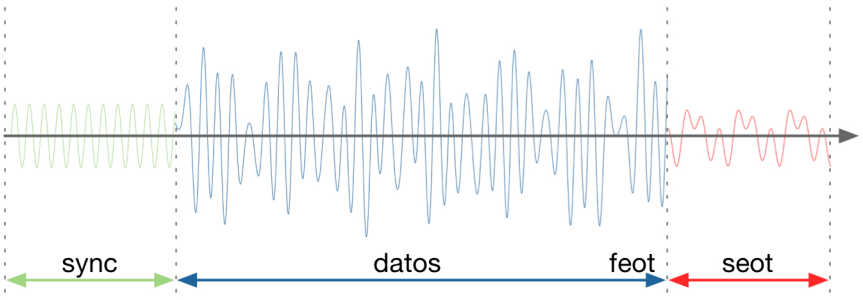


Figura 2.5: Señal generada en una secuencia completa de transmisión

2.3. Servidor

Uno de los dos archivos ejecutables de la arquitectura software es el *servidor*. Podemos describir la función del servidor como la de gestionar la librería módem y otras librerías, para traducir los bits de información a muestras de audio y que éstas sean escritas en los correspondientes dispositivos, y viceversa, recuperar las muestras de audio de los correspondientes dispositivos para obtener los bits de datos. La Figura 2.6 describe simplificadamente cómo se comunica la librería módem dentro del servidor.

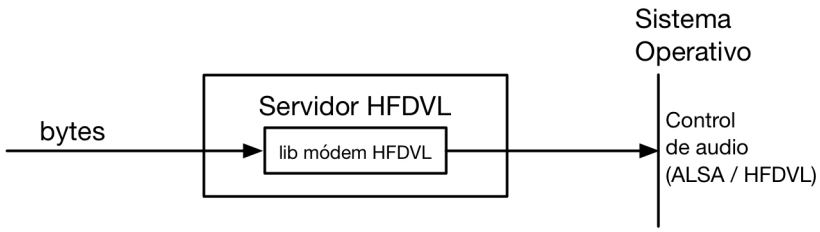


Figura 2.6: Diagrama básico de cómo el servidor comunica a la librería módem

El servidor se encarga de coordinar diferentes librerías, como son librería módem o la librería de audio entre otras, y configurar todo el sistema a petición del cliente, de manera que el servidor actúa bajo órdenes de éste. Una de las tareas imprescindibles del servidor es la de intercambiar permanentemente muestras de audio entre el dispositivo de audio y el módem. En este intercambio de muestras *half-duplex*, encontramos los siguientes tres casos:

IDLE. (audio ⇒ módem). El servidor recoge muestras de audio que le entrega a la librería módem, pero ésta no genera ningún tipo de datos demodulados, ya que aún no se ha detectado la sincronía y por tanto no se ha comenzado a demodular.

RX. (audio ⇒ módem). El servidor recoge muestras de audio que le entrega a la librería módem, y esta responde con bloques de datos del tamaño del entrelazado, demodulados y disponibles para ser enviados al cliente. La Figura 2.7 describe temporalmente los eventos de entrega de muestras y de disponibilidad de datos.

TX. (módem ⇒ audio). El servidor recibe datos a transmitir provenientes del cliente. Estos datos se entregan al módem HFDVL, y ésta a su vez devuelve muestras para ser escritas en el dispositivo de audio.

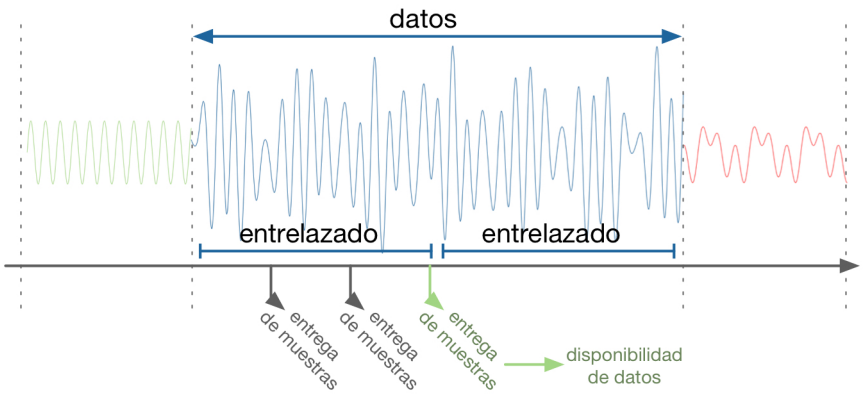


Figura 2.7: Eventos temporales de entrega de muestras y de datos demodulados en recepción

El servidor, bajo órdenes del cliente, instancia diferentes módems a través de la librería módem (normalmente hay tantos módems software como canales de recepción,

para hacer lo que se denomina, combinación de señales en recepción), configurándolos en el mismo momento de instanciación con los parámetros relativos a la modulación/demodulación. De esta manera, si quisiéramos cambiar los parámetros de trabajo del módem, el cliente debería comandar la destrucción de los módems existentes y posteriormente la creación de unos nuevos con los parámetros deseados.

Los dispositivos de audio también se abren o cierran a petición del cliente, proporcionando además el controlador hardware y ruta de los dispositivos a utilizar. Este aspecto es de interés a título informativo, para poder completar el esquema mostrado en la Figura 2.8, y comprender la flexibilidad del sistema.

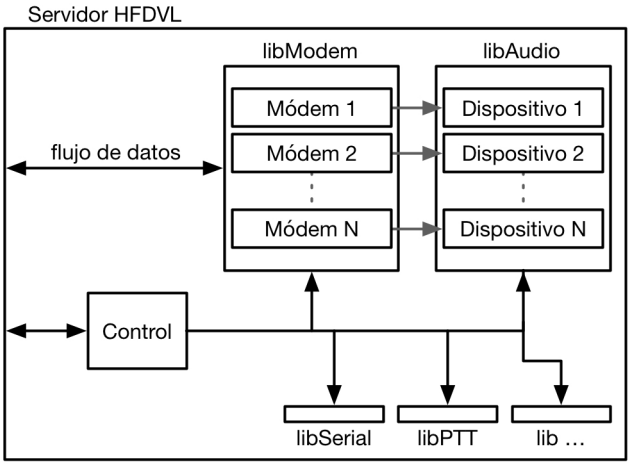


Figura 2.8: Arquitectura simplificada del servidor

Además de incluir los datos recibidos/a transmitir, la comunicación cliente/servidor también incluye comandos para la configuración del módem como ya vimos, y cierta información del estado del módem, como puede ser el estado en el que se encuentra el módem: IDLE/RX/TX. Durante el desarrollo del protocolo, esto será fundamental para implemetar un control de acceso al medio muy simple con el que evitar colisiones entre diferentes nodos transmisores.

2.4. Cliente / GUI

El *cliente*, también un archivo binario ejecutable, es el responsable de seleccionar y enviar al servidor los datos a ser modulados, y de gestionar los datos provenientes del servidor cuando se encuentra en recepción, por ejemplo, guardándolos en un fichero. Al haber implementado una arquitectura cliente/servidor, el cliente puede estar en una máquina diferente al servidor, siempre que exista una comunicación TCP/IP entre ellos. Para el desarrollo de este proyecto se ha obviado los modos de trabajo AUDIO y SMS, al no ser de interés para esta aplicación.

El uso común y evidente del cliente es ser la interfaz de comunicación entre un usuario y el servidor, aunque el cliente podría ser que no requiriera de la interacción de este usuario; podríamos contemplar una implementación del cliente específicamente

diseñada para estaciones autónomas, como podría ser una estación meteorológica en la que el cliente recogería automáticamente los datos de los sensores de la estación y se los entregara al servidor para ser transmitidos.

El cliente, al ser el encargado de ordenar la configuración de módem, audio y otros parámetros al servidor, podría permitir al usuario actuar sobre la configuración deseada de la librería módem (constelación, etc).

La Figura 2.9 ilustra los modos de trabajo del cliente: **continuo** y **STANAG 5066**, y cómo cambia el flujo de los datos en función de la selección del modo de trabajo.

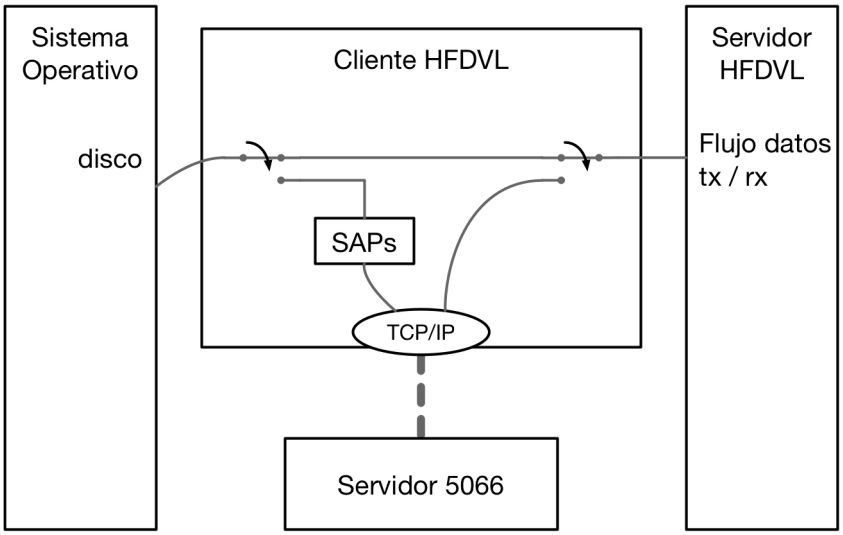


Figura 2.9: Ruta de los datos tanto transmisión como recepción, para modo continuo y modo STANAG 5066

Modo continuo. Este modo de trabajo es simple y se explica en este documento para conocer la base de comunicación cliente→servidor→librería módem. En él, los datos a transmitir, con origen **siempre** de ficheros, se entregan al servidor tal cual se leen del fichero seleccionado, y completado con zero-padding en caso de no llenar un número entero de entrelazados. De la misma manera, los datos recibidos desde el servidor se escriben en un fichero (un fichero nuevo es creado cada vez que se sincroniza una nueva recepción), incluyendo los ceros de *padding* que se hayan podido añadir en el nodo transmisor y arrastrando los errores que se hayan podido generar en la demodulación. Hay que destacar que estas transmisiones son demoduladas simultáneamente por tantos receptores como hayan sido capaces de sincronizar, de manera que se asemeja a una *transmisión broadcast* a todos los receptores que estén a la escucha en el canal de transmisión.

Modo STANAG 5066. En este modo de trabajo se intercala el servidor STANAG 5066 (que desarrollaremos a lo largo del proyecto) en la ruta de los datos, además de incluir clientes STANAG 5066 para hacer de interfaz con este servidor. De manera más sencilla podríamos asemejar esta estructura a la que vemos en la Figura 2.10, en la que además podemos apreciar las diferentes capas que se definen en el STANAG 5066, cómo se intercala dentro del flujo de datos cliente↔servidor.

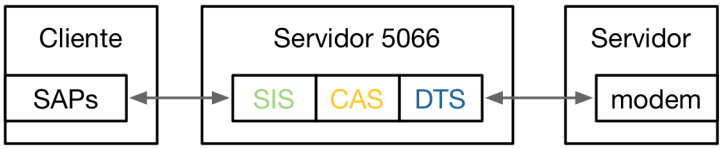


Figura 2.10: Representación de las diferentes capas lógicas en el flujo de los datos

La comunicación entre el cliente y el servidor STANAG 5066 se hace a través de una conexión TCP/IP. Típicamente el servidor STANAG 5066 se ejecutará dentro de la misma máquina que ejecute el servidor.

La conmutación entre el modo continuo y el modo STANAG 5066 se hace de manera interna en el cliente.

2.5. Rol del servidor STANAG 5066

El papel del servidor STANAG 5066 es múltiple. Téngase en cuenta que el módem actúa como sistema de transmisión sin protocolo; el módem simplemente se centra en minimizar la BER (*Bit Error Rate*) con técnicas en la forma de onda. La implementación del STANAG 5066 dota al sistema de:

Funciones de comunicación en red. Cada nodo tiene asignada una dirección propia (y que además habrá de ser única), y se incorpora en cada paquete un campo de dirección de destino, lo que posibilita la direccionabilidad de la información transmitida. Cada nodo es responsable de descartar la información que no va destinada a su dirección.

Certeza de que los datos recibidos son correctos. Cada paquete incorpora un CRC (*Cyclic Redundancy Check*) por lo que se tiene certeza en el receptor de si los datos han llegado correctamente o no. En el modo de transmisión NO-ARQ es posible recibir datos erróneos.

Modo de transmisión sin errores. El modo de transmisión ARQ hace que el receptor vaya confirmando los fragmentos de los mensajes que van llegando correctamente, por lo que el transmisor retransmite los que han llegado incorrectamente y continúa a medida que se van recibiendo completos de manera correcta; esto hace que exista una garantía en el nodo transmisor de que la información va a llegar correctamente, y si se ha producido un fallo puntual, se retransmitirán los segmentos erróneos.

Confirmación de entrega. El nodo transmisor, al enviar información usando el modo ARQ puede solicitar confirmación de entrega, ya que de manera implícita el receptor ha ido confirmando la recepción correcta de los mensajes para poder continuar con las transmisiones.

Estas características tienen un coste en rendimiento, por el hecho de incluir un protocolo de comunicaciones, y otras limitaciones que impone el STANAG 5066, como puede ser que las transmisiones no duren más de dos minutos, debiendo parar la transmisión para permitir a otros nodos acceder al medio. Aunque no exista ningún nodo dispuesto a transmitir.

Capítulo 3

El estándar STANAG 5066

El estándar STANAG 5066 [?] define un protocolo de comunicaciones de datos, específico para comunicaciones HF y diseñado para dar abstracción de la electrónica subyacente en cada nodo de comunicaciones.

Este protocolo provee interoperabilidad en dos interfaces principales: primera, que el estándar describe como *Common Air Interface*, y que corresponde a la interfaz que utilizan los nodos para intercambiar información entre ellos, normalmente a través de una radio; la segunda es la que describe cómo los clientes o usuarios han de interactuar con la"subred", es decir, cómo intercambiar datos con el servidor STANAG 5066 para acceder a esta subred de comunicaciones (la subred existente a través de HF).

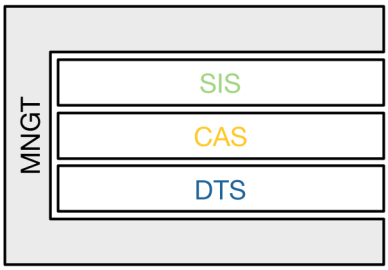


Figura 3.1: Estructura de capas definidas por el estándar STANAG 5066

Para facilitar la descripción e implementación del estándar, éste divide su definición en las capas que se muestran en la Figura 3.1. Cada capa se comunica con su homóloga remota a través de unidades de información llamadas PDUs (*Protocol Data Units*), y lo hace usando los servicios de comunicaciones que le provee la capa inmediatamente inferior. Esto queda más claro si observamos la Figura 3.2, en la que puede verse como la capa SIS, por ejemplo, genera un S_PDU que cede a la capa CAS para que ésta de manera transparente, haga entrega *tal cual* a la capa SIS remota.

Las PDUs son elementos de información que queremos hacer llegar a la capa remota del mismo nivel, y por convención, están precedida de la letra U, S, C o D, dependiendo de la capa de origen, Usuario (Aplicación), SIS, CAS o DTS respectivamente.

Cada capa que hace uso de los servicios de transporte de capas inferiores a ella, lo

hace a través de primitivas. Ésto se explica en la Sección 3.2.

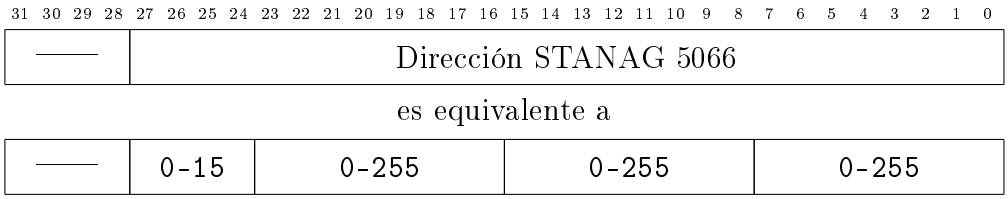
Sólo las comunicaciones Aplicación ↔ SIS y DTS ↔ DTS usan primitivas que están completamente definidas por el estándar para garantizar interoperabilidad, y las comunicaciones entre capas SIS ↔ CAS y CAS ↔ DTS quedan a disposición de cada implementación, lo cual nos proporciona libertad y facilidad para tratar las primitivas sin necesidad de codificarlas según ninguna imposición.

Para la transmisión de información, que es al fin y al cabo la labor principal del estándar, las aplicaciones envian U_PDUs a través de las primitivas *_UNIDATA_*; al *payload* enviado por una aplicación se le puede llamar independientemente U_PDU o UNIDATA.

Existen dos tipos básicos de servicio, el ARQ y el NO-ARQ:

- **ARQ**. Su uso es para transmisión de datos que requieran fiabilidad de que van a ser entregados de manera correcta. Requiere confirmación explicita por parte del nodo receptor de las secciones de U_PDUs que se han ido recibiendo correctamente, para poder retransmitir las incorrectas. Esto hace que no siempre sea posible usar este modo con nodos que, por algún motivo, no puedan usar la transmisión para hacer las confirmaciones.
- **NO-ARQ**. Está pensado para complementar al modo ARQ y poder comunicarse con un nodo receptor que no pueda confirmar las recepciones, o bien, transmisiones simples que no requieran confiabilidad, por ejemplo, en determinadas situaciones, un METAR[1]. A pesar de no garantizar la integridad de los datos recibidos, el nodo receptor sabe en todo momento si la información ha sido correctamente recibida. También permite la transmisión punto → multipunto.

Para identificar el destinatario al que se desea enviar información, cada nodo tiene una **dirección de subred de 28 bits**, en el rango $0 - 268{,}435{,}456$ ó $0 - 2^{28}$. Si agrupáramos la dirección en 4 bytes, en el cual el más significativo sólo serán válidos los 4 bits inferiores, se puede observar una equivalencia con direcciones IPv4, lo que facilita su representación en formatos ya conocidos, y que en nuestro caso dejaría un rango de direcciones permitidas desde `0.0.0.0` hasta `15.255.255.255`.



3.1. SAPs

Para poder comunicarse a través del STANAG 5066, lo primero que ha de hacer un cliente es notificar al servidor que va a empezar a usar sus servicios de comunicación,

[1] Informe meteorológico utilizado en aeronáutica.

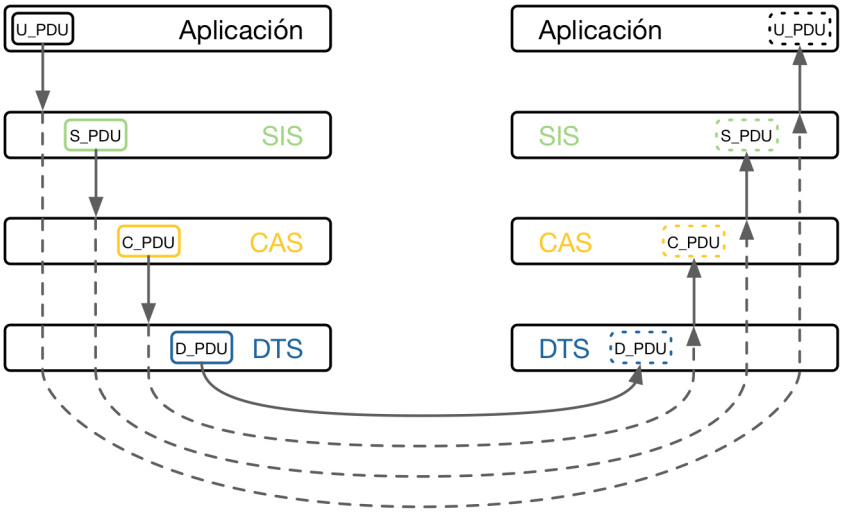


Figura 3.2: Esquema de cómo cada capa encapsula las PDUs de la capa superior

y para ello ha de asociarse a alguno de los SAP (*Service Access Point*) si estuviera disponible. Los SAPs son identificados a través del identificador de SAP (*SAP ID*), que es un número entre 0 y 15; por tanto puede haber hasta 16 clientes conectados de manera simultánea.

La solicitud de vinculación con un SAP se hace a través de la S_Primitiva `S_BIND_REQUEST`, y el cliente recibirá la aceptación (`S_BIND_ACCEPTED`) o rechazo (`S_BIND_REJECTED`) proveniente del servidor STANAG 5066. El uso de las primitivas se explica en la Sección 3.2.

Cuando un cliente hace la solicitud de vinculación con un SAP, indica el campo `rank`, que también es un valor entre 0 y 15, y con el que se indica la importancia del cliente asociado. Esto indica al servidor STANAG 5066 cuánta prioridad ha de dar a este cliente a la hora de darle recursos; si el servidor STANAG 5066 determina que no tiene recursos suficientes podría optar por desconectar a un cliente con `rank` bajo, para centrar los recursos en un cliente con un `rank` superior.

Es destacable, que a la hora de enviar información a un nodo remoto, se ha de especificar, además de la dirección del nodo, el SAP al que va dirigido.

3.2. Primitivas

Las primitivas son sentencias estructuradas que permiten a una capa intercambiar órdenes, instrucciones, notificaciones o datos con la capa adyacente, es decir, las PDUs nombradas anteriormente, y que contienen los datos a hacer llegar al nodo remoto, han de ser codificadas en una primitiva, que indica a la capa receptora de la PDU la orden a realizar. De acuerdo con el estándar, las primitivas se precederán de la letra S, C o D en función de la capa inferior que intervenga en la comunicación; podemos verlo en la Figura 3.3.

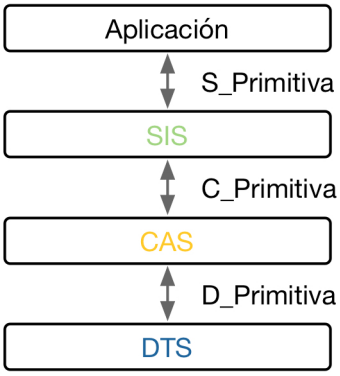


Figura 3.3: Primitivas para el intercambio de información entre capas

Pongamos un ejemplo de comunicación:

1. Inicialmente el cliente solicita vincularse
   Cliente → *SIS* : `S_BIND_REQUEST`.
2. El servidor STANAG 5066 acepta la petición
   SIS → *Cliente* : `S_BIND_ACCEPT`.
3. El cliente quiere transmitir una PDU a otro nodo (por ejemplo, en modo ARQ con confirmación de entrega)
   Cliente → *SIS* : `S_UNIDATA_REQUEST`.
4. El servidor STANAG 5066 confirma que la PDU ha sido entregada al nodo remoto
   SIS → *Cliente* : `S_UNIDATA_REQUEST_CONFIRM`.

3.3. UNIDATAS (*U_PDUs*)

Como hemos visto, las UNIDATAS o U_PDUs son las unidades de información que un cliente envía para que sea entregado a otro cliente en un nodo remoto.

Las U_PDUs tienen varios parámetros además del *payload* a transmitir. Estos parámetros son:

- **Prioridad**. Número entero en el rango entre 0 y 15, que indica al servidor STANAG 5066 qué prioridad ha de dar a dicho paquete en la transmisión
- **SAP de destino**. SAP al que se entregará el paquete en el nodo destino.
- **Dirección de destino**. Dirección de la subred STANAG 5066 del nodo de destino.
- **Modo de entrega**. Aquí se indica si es tipo ARQ o NO-ARQ, además de especificar parámetros específicos de cada una de los modos. **ARQ**: tipo de confirmación

y orden de entrega (forzosamente en orden o ir liberando los paquetes completamente recibidos en recepción). **NO-ARQ**: orden de entrega (igual que modo ARQ) y la cantidad mínima de retransmisiones.

- **TTL o tiempo de vida del paquete**. Tiempo de validez del paquete desde el momento en que lo entrega al servidor STANAG 5066.
- ***Payload***. Información útil a transmitir al nodo remoto.

El estándar contempla además, dos tipos de UNIDATAS, las normales, y las EXPEDITED, las cuales cuentan con una prioridad especial, y con un modo diferente de gestión a la hora de hacer el intercambio de información con los nodos remotos.

3.3.1. La Fragmentación de los paquetes D_PDU en la DTS

Cuando las U_PDUs son recibidos por la capa SIS, ésta las procesa generando una S_PDU (con la que se comunicará con la SIS remota), y solicita a la capa CAS (a través de una C_Primitiva) que los transmita. La capa CAS genera una C_PDU y solicita a la DTS que los transmita, y la DTS, al contrario que las otras capas no genera una única D_PDU, sino que fragmenta la C_PDU en segmentos de X bytes[2], generando una cantidad N de D_PDUs. Esto se describe mejor en la Figura 3.4

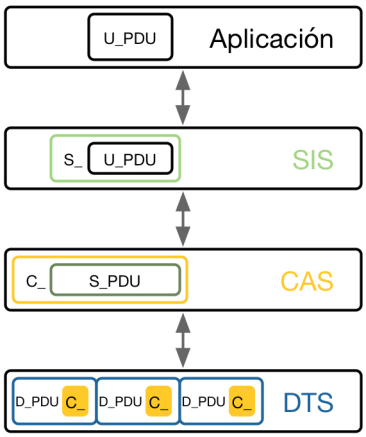


Figura 3.4: Segmentación de las C_PDUs en la DTS en múltiples D_PDUs

La fragmentación se emplea tanto en modo ARQ como en NO-ARQ, y su principal función es la de evitar la corrupción de paquetes muy grandes, limitando el tamaño de éstos a la hora de ser transmitidos por HF. El tamaño de la fragmentación es un compromiso dependiente de la calidad del canal HF, en el que si hacemos las divisiones muy pequeñas, el *throughput* se verá afectado por el *overhead*, y si hacemos divisiones muy grandes, aumentamos la probabilidad de error en el paquete, creando la necesidad de retransmitir/descartar el paquete. La evolución de *throughput vs tamaño de D_PDU*, teniendo en cuenta estos dos factores, se puede apreciar en la Figura 3.5, que está calculada para una BER de 10^{-4}. Se puede considerar un valor de BER bastante común según la experiencia del equipo diseñador de la forma de onda HFDVL.

[2]80 bytes para nuestra plataforma

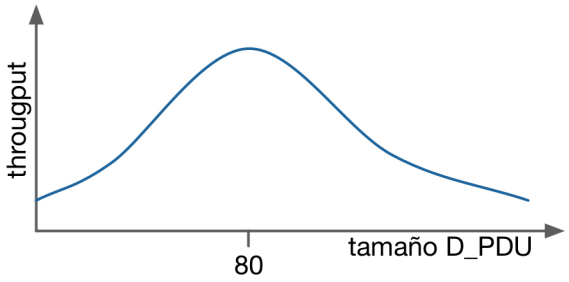


Figura 3.5: Throughput vs tamaño de D_PDU para una BER de 10^{-4}

La evolución de esta curva viene descrita por las siguientes ecuaciones:

$$\text{throughput\_por\_overhead} = \frac{\text{payload}}{\text{overhead} + \text{payload}} \tag{3.1}$$

$$\text{PER} = 1 - (1 - \text{BER})^{\text{tam\_d\_pdu}} \tag{3.2}$$

$$\text{rtx\_paquete} = \frac{1}{1 - \text{PER}} \tag{3.3}$$

$$\text{payload\_por\_rtx} = \text{rtx\_paquete} \times \text{tam\_d\_pdu} \tag{3.4}$$

3.3.2. ARQ y NO-ARQ

Para poder enviar U_PDUs a nodos remotos, se ha de usar los servicios ARQ o NO-ARQ. Las características de cada uno de estos servicios son diferentes, y los clientes habrán de seleccionar el servicio a utilizar en función de las necesidades de la aplicación que implementen.

- El modo **ARQ** es un modo de transmisión con **solicitud de retransmisión** de las D_PDUs que hayan llegado de manera incorrecta. Por tanto, es imprescindible que el nodo receptor sea capaz de transmitir los denominados **ACK**s, que son los paquetes mediante los que se confirman las D_PDUs recibidas correctamente e incorrectamente. El mecanismo con el que funciona el modo ARQ es el de **ventana deslizante**, que es ampliamente utilizado en otros protocolos con ARQ. La Figura 3.6 muestra cómo se rellenaría la apertura de la ventana deslizante con varias C_PDUs.

  Además de este mecanismo, el STANAG 5066 utiliza la **repetición selectiva** de las D_PDUs recibidas incorrectamente; esto quiere decir que los ACK transportan información, no sólo del último segmento consecutivo recibido correctamente, sino también de los segmentos *sueltos* que se hayan recibido correctamente más allá del primer elemento erróneo. A pesar de no permitir mover la ventana más allá del último elemento consecutivo correcto, hace que las retransmisiones sean

mucho más cortas que si no contara con este mecanismo selectivo, ya que solo se incluirán aquellos segmentos erróneos.

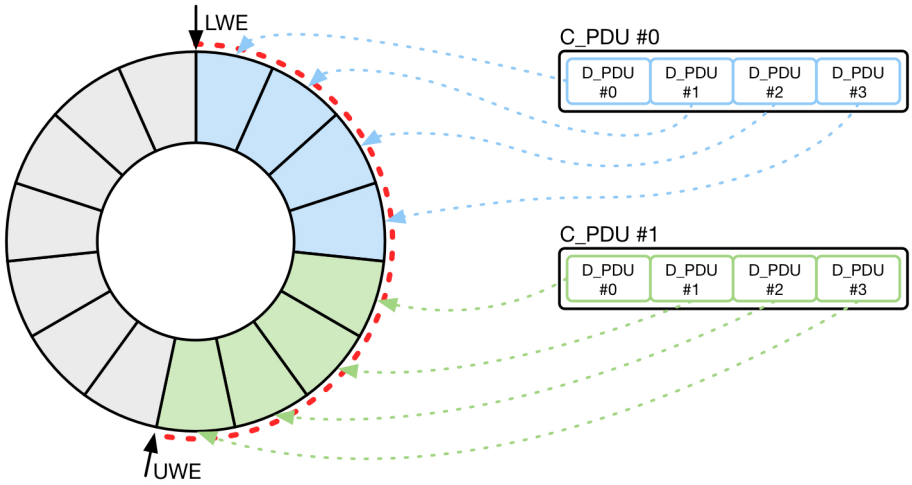


Figura 3.6: Ventana deslizante de las transmisiones ARQ

El modo ARQ también provee de la capacidad de confirmar al cliente transmisor de la entrega de los datos, y cuenta con dos tipos de confirmaciones: **confirmación de nodo** y **confirmación de cliente**.

- La **confirmación de nodo** es la que se genera de manera natural, al conocer el propio servidor STANAG 5066 de qué C_PDUs se han confirmado todas las D_PDUs en las que se subdivide, se pueden generar las confirmaciones de entrega de los UNIDATA. Estas confirmaciones **se generan en el nodo local**, es decir, no se transmite por HF ninguna información extra para poder disponer de ella.
- La **confirmación de cliente** es una confirmación diferente y su selección es mutuamente excluyente de la de nodo. La confirmación de cliente se genera en la capa SIS remota: Una vez se haya hecho el proceso de recomposición de las D_PDUs, y se tenga la certeza de que se puede entregar la U_PDU resultante a un cliente en el SAP correspondiente, es la capa SIS remota la que genera una S_PDU con el propósito de confirmar la entrega de la U_PDU inicial. Esta S_PDUs de confirmación es transmitida por HF, lo que hace que esta confirmación sea **más fiable**, pero **más costosa** en términos de utilización del canal. En la Figura 3.7 se puede observar el diagrama de secuencia, en el que se genera la confirmación de cliente.

- El modo **NO-ARQ** podría considerarse un modo *pasivo* en cuanto a la integridad de datos, porque el receptor no interviene de manera activa, sino que se limita a recibir paquetes y recomponer las U_PDUs. Que se pueda recomponer correctamente o no una U_PDU, dependerá de los errores en la demodulación de los bits recibidos, por ejemplo, debido a una interferencia de gran potencia.

  Para compensar esta debilidad, el modo NO-ARQ cuenta con un sistema de retransmisión de las D_PDUs, y de reconstrucción inteligente en recepción, que se describe en la Figura 3.8.

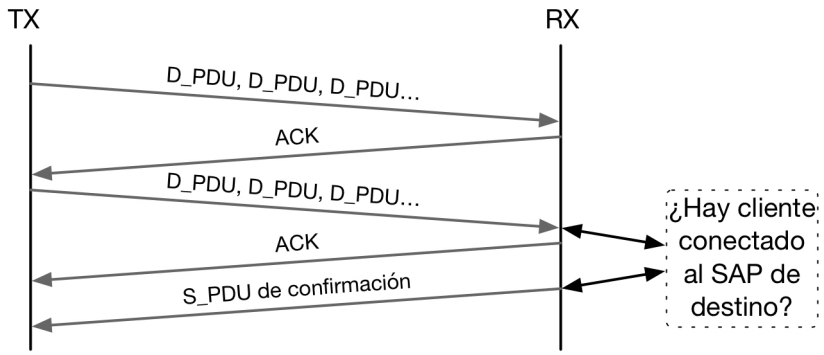


Figura 3.7: Diagrama de secuencia en la confirmación de cliente de una U_PDU

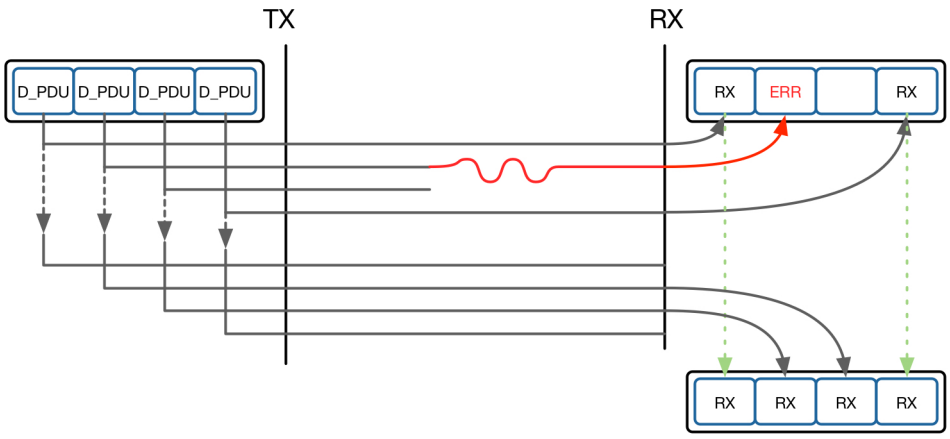


Figura 3.8: Diagrama de secuencia del mecanismo de reconstrucción del modo NO-ARQ

El sistema receptor irá almacenando los segmentos que vaya recibiendo, estén correctos o no, y mantendrá un control de los mismos, por si en futuras recepciones llegan correctamente segmentos que estaban erróneos o sin recibir. Las C_PDUs reconstruidas se liberarán hacia la capa CAS local en el momento de tener todos los segmentos correctos, o en el momento que expire su validez temporal de recepción. Esto es, un campo de control mediante el cual el transmisor indica al cliente durante cuánto tiempo es posible que vuelva a hacer retransmisisones de cada C_PDU, y que permite al receptor gestionar la *liberación* hacia la capa CAS de C_PDUs que estén parcialmente recibidas, evitando así bloqueos de espera indefinida de retransmisiones.

A la hora de enviar una U_PDU, el cliente indica también el número mínimo de retransmisiones que ha de hacer el sistema, en un valor en el rango $0 - 15$.

3.3.3. TTL, prioridad

Los campos **TTL** (*Time To Live*) y **prioridad** se especifican a la hora de pedir la transmisión de un UNIDATA. Son dos de los campos de las primitivas `S_UNIDATA_REQUEST` y `S_EXPEDITED_UNIDATA_REQUEST`.

El **TTL** indica la cantidad de tiempo que un UNIDATA es válido para ser enviado a través de la subred STANAG 5066. Una vez expirado su TTL, el paquete debe ser descartado. Si se especifica un TTL de 0, el paquete será válido durante la cantidad máxima de tiempo admisible en el sistema[3].

El campo **prioridad** representa la prioridad que ha de dar el sistema a ese paquete, siendo 0 la menor, y siendo 15 la mayor prioridad posible. El servidor STANAG 5066 debe *intentar* respetar las prioridades, ya que *forzar* a respetar las prioridades podría interferir con otros criterios de optimización, dependiendo de la implementación que se haga.

3.4. División entre capas SIS, CAS y DTS

El estándar STANAG 5066 define tres capas diferentes en las que se ha dividir su implementación. Esta división se hace para facilitar tanto la compresión de estándar, como la implementación resultante.

Para garantizar la interoperabilidad, el estándar define completamente cómo ha de ser la comunicación Aplicación $\Leftrightarrow$ SIS y también $\text{DTS}_{local} \Leftrightarrow \text{DTS}_{remota}$. Garantizar una comunicación estándar Aplicación $\Leftrightarrow$ SIS, permite la implementación del servidor STANAG 5066 pueda utilizarse con aplicaciones de otros desarrolladores diferentes, y no sólo del desarrollador del servidor STANAG 5066 en cuestión. A la vez, la estandarización DTS $\leftrightarrow$ DTS permite que se intercomuniquen dos nodos implementados por desarrolladores diferentes se comuniquen normalmente. La Figura 3.9 ilustra esta idea.

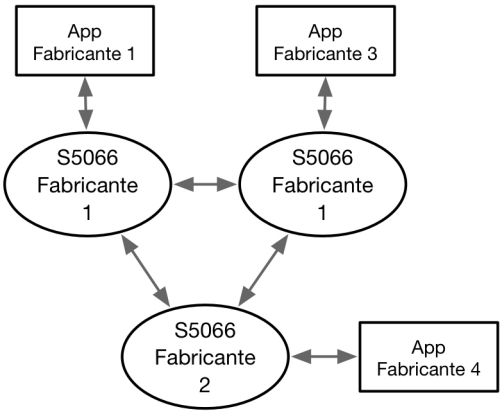


Figura 3.9: Ejemplo de uso de aplicaciones y servidores desarrollados por diferentes fabricantes

El STANAG 5066 no define un protocolo de comunicaciones entre capas SIS $\rightarrow$ CAS y CAS $\rightarrow$ DTS; aunque sí define primitivas C_Primitivas y D_Primitivas, no define exactamente cómo se codifican, etc.

[3]El STANAG 5066 define este tiempo como de libre selección por el implementador del protocolo

3.4.1. SIS

La capa SIS es la interfaz de comunicación entre las aplicaciones (clientes) y la subred STANAG 5066. El estándar define estrictamente la codificación de las comunicaciones entre Aplicación y SIS, a través de S_Primitivas [10, Anexo A.2].

Entre las tareas definidas para la capa SIS se encuentra:

- *Gestionar las conexiones de los clientes a los SAP.* Ha de llevar el control de la lista de SAP disponibles, además de los recursos a disposición, para poder gestionar los `S_BIND_REQUEST`.
- *Ser la interfaz de comunicación con los clientes.* Es el punto de entrada de los datos a transmitir, punto de salida de los datos recibidos a través de la subred HF y ha de encargarse de comunicarse con los clientes.
- *Iniciar y terminar los physical links.* Los datos ARQ requieren una notificación explícita de que el nodo de destino está disponible, y eso se hace a través de los enlaces físicos (*physical link*) que se explican en la Sección 3.5.
- *Gestionar las confirmaciones de cliente.* De los dos tipos de notificación, la *de cliente* necesita una gestión por parte de la capa SIS del nodo receptor, ya que es necesario comprobar si existe un cliente disponible en el SAP de destino al que entregar los datos, y generará una S_PDU de confirmación o denegación segú corresponda.
- *Notificar a los clientes de diferentes eventos internos al STANAG 5066.* Diferentes eventos pueden ocurrir durante la ejecución del servidor STANAG 5066 , como por ejemplo la creación de un *soft link* por iniciativa de un nodo remoto, o que se han llenado los *buffers* para almacenar datos a transmitir (y los clientes han de parar la transmisión de datos). Es la capa SIS la encargada de notificar a los clientes de estos eventos.

La SIS se comunica con otras capas SIS remotas a través de unidades de datos S_PDUs. Los ocho tipos de S_PDU existentes son:

0. `DATA`. El *payload* ha de ser entregado como U_PDU al cliente local.
1. `DATA_DELIVERY_CONFIRM`. Es la S_PDU de confirmación que se genera cuando a un nodo receptor llega una S_PDU tipo 0 (DATA) con requisito de confirmación de cliente.
2. `DATA_DELIVERY_FAIL`. Se genera cuando llega una S_PDU con requisito de confirmación de cliente, pero no hay un cliente conectado al SAP de destino.
3. `HARD_LINK_ESTABLISHMENT_REQUEST`. Solicita a un nodo remoto la creación de un *Hard Link*.
4. `HARD_LINK_ESTABLISHMENT_CONFIRM`. Acepta la solicitud de creación de un *Hard Link*.

5. `HARD_LINK_ESTABLISHMENT_REJECTED`. Rechaza la solicitud de creación de un *Hard Link*.

6. `HARD_LINK_TERMINATE`. Notifica a un nodo remoto la terminación de un *Hard Link* que haya sido creado.

7. `HARD_LINK_TERMINATE_CONFIRM`. Confirma la recepción de la notificación de finalización de *Hard Link* existente.

Una de las tareas principales de la capa SIS, es determinar el inicio/finalización de los *physical links*. Para la transmisión de datos ARQ es imprescindible que exista una sesión *soft link* o *hard link* vigente. Si un cliente enviase datos a un nodo remoto con el que no existe enlace, la SIS ha de solicitar a la CAS la creación de uno nuevo; mientras tanto, los datos a transmitir han de ser almacenados en la SIS hasta que se resuelva la creación del enlace.

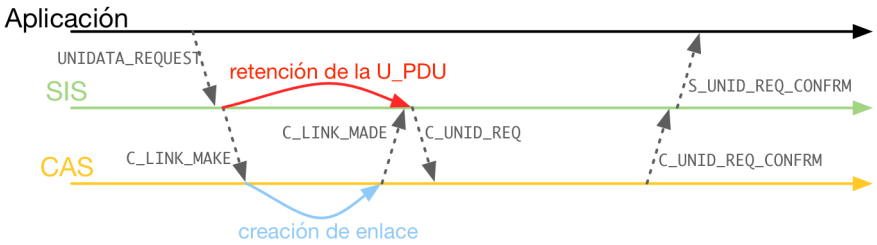


Figura 3.10: Diagrama secuencial del proceso de creación de enlace y envio de datos ARQ

3.4.2. CAS

La capa CAS es la más simple del protocolo ya que su tarea se limita a la gestión de la creación y la ruptura de *physical links*, solicitados por parte de la capa SIS, a demás de ser un *bypass* para el flujo de datos.

Si la capa SIS indentifica la necesidad de crear un *physical link* con un nodo remoto, hará una solicitd a la capa CAS. Ésta se encargará de comunicarse con la capa CAS remota para gestionar el enlace (A través de una C_PDU), o en caso de que no haya comunicación, notificar a la SIS local de este inconveniente. Esto lo hace a través de las C_Primitivas relacionadas con los *physical link*, `C_PHYSICAL_LINK_*`. Las C_PDUs relacionadas con los *physical link* se transmiten haciendo uso del servicio *EXPEDITED NO-ARQ*.

El intento de enlace será enviado un número N de veces, por si el destinatario no ha podido recibirlo a la primera. La cantidad de intentos antes de dar el enlace por fallido, y el tiempo de espera entre intentos, es de elección libre para el implementador.

Respecto a la transferencia de datos, la CAS hace de *bypass* entre SIS y DTS, convirtiendo las primitivas `C_UNIDATA_*` en `D_UNIDATA_*` y viceversa.

3.4.3. DTS

La capa DTS es la responsable de la transmisión eficiente de los datos a través del enlace HF de las unidades de datos proveniente de la capa CAS (C_PDUs). Esta eficiencia se logra principalmente con tres mecanismos: la fragmentación de las C_PDUs en elementos más pequeños (D_PDUs), añadiendo CRC a cada segmento D_PDU, e implementando mecanismos de repetición, ARQ y NO-ARQ.

Cada una de las D_PDUs transmitidas a través de la interfaz HF se divide en tres partes: sincronismo, cabecera y una parte opcional de dato. Las dos últimas incorporan un código CRC de comprobación. Los tipos de D_PDU son:

0. `DATA-ONLY`. PDU para enviar datos tipo ARQ . Entre los campos que contiene se encuentran los relativos a la sincronización de la ventana deslizante.
1. `ACK-ONLY`. Aquí se encuentran los ACKs relativos a las transmisiones ARQ . Se especifican tanto el útlimo segmento que llegó correctamente de manera consecutiva, como todos los segmentos posteriores no consecutivos que hayan llegado bien.
2. `DATA-ACK`. Es una combinación entre las dos PDUs anteriores. Se envian ACKs de recepciones ARQ y se envían paquetes ARQ a ese mismo nodo.
3. `RESET/WIN-RESYNC`. Resetea los estados relacionados con la ventana deslizante.
4. `EXP-DATA-ONLY`. Igual que la PDU tipo 0 `DATA-ONLY`, pero siendo EXPEDITED. Se mantiene como una D_PDU separada para diferenciar ventanas y poder dar una prioridad inmediata sin necesidad de vaciar ventanas ya existentes.
5. `EXP-ACK-ONLY`. Contiene los ACKs referentes a la D_PDU `EXP-DATA-ONLY`.
6. `MANAGEMENT`. Para transferir paquetes de gestión.
7. `NON-ARQ-DATA`. Usada para transmitir paquetes NO-ARQ.
8. `EXP-NON-ARQ-DATA`. Igual que la anterior, pero tipo EXPEDITED, con más prioridad.

9–14 -. Sin implementar. Reservado para futuras implementaciones.

15 `WARNING`. Utilizada para notificar errores a un nodo remoto, por ejemplo, si un nodo nos envía un paquete ARQ sin haber hecho un *physical link* previamente, hemos de enviar un paquete de este tipo para notificarlo de este hecho.

La DTS ha de llevar el control de las ventanas deslizantes, tanto para hacer las transmisiones, como para las recepciones, por cada nodo con el que queramos mantener comunicaciones ARQ. Esto tiene un coste en memoria relativamente elevado, ya que mientras sea más elevado el número de nodos simultáneos con los que querríamos mantener comunicaciones ARQ, más *estados de transmisión* hay que mantener en memoria.

Igualmente para las transmisiones NO-ARQ, sobre todo en la parte de recepción, ya que la DTS deberá mantener en memoria todos los segmentos de C_PDUs que enventualmente pueda recibir de manera incorrecta, para posteriormente poder generar una C_PDU completa.

En caso de que la DTS no pueda regenerar una C_PDU completamente en modo NO-ARQ, deberá notificar mediante marcadores (punteros) los inicios y tamaños de las secciones que han llegado incorrectamente, o que no han llegado, como se explica en la Figura 3.11.

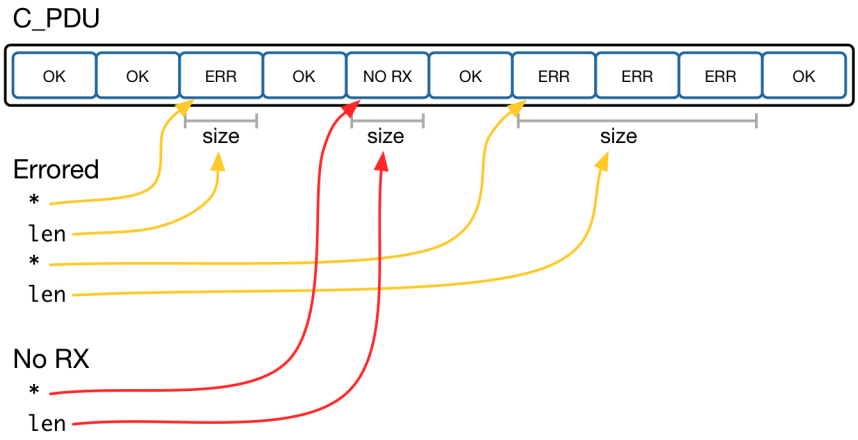


Figura 3.11: Ejemplo de cómo la DTS ha de marcar los segmentos erróneos y los no recibidos

3.5. Los enlaces (physical links y soft links)

Uno de los mecanismos internos que define el STANAG 5066 es el enlace, y define tres tipos, *physical link*, *soft link* y *hard link*.

- **Physical link**. Es el **enlace básico** sobre el que se basan los *soft* y *hard links*. Se establece mediante la capa CAS, haciendo envío de las correspondientes C_PDUs de solicitud, y de aceptación. Es solicitado por la capa SIS con propósito de dar soporte a un *soft* o a un *hard link*.

- **Soft link**. Es un enlace iniciado por requerimiento de la capa SIS, y requiere un *pysical link* activo. Este tipo de enlace es el enlace básico necesario para la transferencia de paquetes ARQ; cuando un cliente quiere enviar datos ARQ a un nodo remoto y no hay enlace activo con ese nodo, la SIS se ha de encargar de establecer un *soft link*, para eso hace falta pedir a la CAS el establecimiento de un *physical link*

- **Hard link**. Son enlaces iniciados por los clientes, no por la capa SIS. Su propósito es que los clientes sean conocedores del estado del enlace con el nodo al que solicitan el *hard link*. Esta característica se ha implementado en este proyecto a

través de primitivas S_MANGEMENT_MSG_INDICATION, mediante la que se indica la creación o ruptura de los *soft links* creados por iniciativa de la SIS.

3.6. Flow on/off

Estas S_Primitivas son órenes SIS ⇒ Cliente para que los clientes paren cualquier tipo de transmisión de datos al servidor STANAG 5066, o para volver a reestablecer la posibilidad de envío de información.

Es importante por parte de los clientes implementar esta primitiva, ya que si el servidor STANAG 5066 se está quedando sin recursos, enviará S_DATA_FLOW_OFF a todos los clientes conectados. Si alguno de los clientes continúa enviando datos, podría ser desconectado unliateralmente del servidor STANAG 5066.

3.7. Paquetes MANAGEMENT_MSG

El estándar STANAG 5066 define la primitiva S_MANAGEMENT_MSG_REQUEST y S_MANAGEMENT_MSG_INDICATION como primitivas de intercambio de mensajes de gestión del servidor STANAG 5066 responsable del nodo local, y la única imposición respecto a estas primitivas, es que el servidor sólo ha de aceptar primitivas provenientes de clientes que hayan conectado con rank igual a 15.

Además, el estándar deja a libre disposición del implementador qué tipo de mensajes de gestion SIS ⇒ Cliente o Cliente ⇒ SIS pueden ser intercambiados, tanto para el control del servidor STANAG 5066 como para la notificación de eventos al cliente.

Esta primitiva es usada por la implementación realizada en este Proyecto, y hemos definido varios tipos de mensaje propios en ambos sentidos. Ya que estas definiciones no se recogen en el estándar, se describen con detalle:

- **Cliente ⇒ SIS**:
  - ASK FOR LOCAL ADDRESS (0x01).
    Usada para que los clientes puedan saber la dirección del nodo local. Cuando el cliente se conecta al servidor STANAG 5066 no sabe de antemano cuál es la dirección HF del nodo local. Esta primitiva permite al cliente preguntar al servidor por su dirección.

	7 6 5 4 3 2 1 0
0x00 ⋮ 0x04	Cabecera STANAG 5066
0x05	S_Prim type = 0x12
0x06	MSG TYPE = 0x01

- `ASK FOR MODEM CONFIG (0x02).`

  Reservado para poder preguntar por la configuración de módem. No ha sido implementado.

- `JOIN A NEW GROUP (0x03).`

  Pide al servidor STANAG 5066 que añada un nuevo grupo a la lista de grupos a los que pertenece. Es posible que el servidor STANAG 5066 responda con la nueva lista de grupos a través de un mensaje `S_MANAGEMENT_MSG_INDICATION` de tipo `joined groups`.

	7 6 5 4 3 2 1 0
0x00 … 0x04	Cabecera STANAG 5066 ⋮
0x05	S_Prim type = 0x12
0x06	MSG TYPE = 0x03
0x07 … 0x0A	Dirección del grupo

- `QUIT A GROUP (0x04).`

  Pide al servidor STANAG 5066 que salga de uno de los grupos a los que pertenece.

	7 6 5 4 3 2 1 0
0x00 … 0x04	Cabecera STANAG 5066 ⋮
0x05	S_Prim type = 0x12
0x06	MSG TYPE = 0x04
0x07 … 0x0A	Dirección del grupo

- `ASK FOR JOINED GROUPS (0x05).`

  Pide al servidor STANAG 5066 que responda al cliente con la lista de grupos a los que pertenece.

	7 6 5 4 3 2 1 0
0x00 0x04	Cabecera STANAG 5066 ⋮
0x05	S_Prim type = 0x12
0x06	MSG TYPE = 0x05

- **SIS ⇒ Cliente**:

  - `LINK UP (0x04)`. Enviado de manera asíncrona por la capa SIS en cuanto se considera un enlace abierto. Existen situaciones ambiguas en las que se considera enlace abierto, por ejemplo, estando en un nodo A, recibimos solicitudes de enlace del nodo B, pero nuestras aceptaciones de enlace no llegan al nodo B, por tanto, nosotros (nodo A) consideramos enlace abierto, pero realmente no hay enlace bidireccional.

	7 6 5 4 3 2 1 0
0x00 0x04	Cabecera STANAG 5066 ⋮
0x05	S_Prim type = 0x13
0x06	MSG TYPE = 0x04
0x07 0x08 0x09 0x0A	Dirección del nodo

  - `LINK DOWN (0x01)`. Este mensaje es enviado de manera asíncrona por la capa SIS en cuanto el enlace no cumple los requisitos de actividad.

	7 6 5 4 3 2 1 0
0x00 0x04	Cabecera STANAG 5066 ⋮
0x05	S_Prim type = 0x13
0x06	MSG TYPE = 0x01
0x07 0x08 0x09 0x0A	Dirección del nodo

- `LOCAL ADDRESS (0x02)`. El servidor informa de su dirección de nodo local HF.

	7 6 5 4 3 2 1 0
0x00 … 0x04	Cabecera STANAG 5066 ⋮
0x05	S_Prim type = 0x13
0x06	MSG TYPE = 0x02
0x07 … 0x0A	Dirección local del nodo

- `JOINED GROUPS (0x05)`. Esta información se envía por el servidor STANAG 5066 cada vez que se modifica la lista de grupos, o cuando se pide explícitamente la lista de grupos a los que pertenece el servidor.

	7 6 5 4 3 2 1 0
0x00 … 0x04	Cabecera STANAG 5066 ⋮
0x05	S_Prim type = 0x13
0x06	MSG TYPE = 0x05
0x06	Nº de grupos (N)
0x07 … 0x0A	Dirección de grupo #1
⋮	
0x07+4(N-1) … 0x0A+4(N-1)	Dirección de grupo #N

- `MODEM CONFIG (0x03)`.

  Reservado para informar con la configuración de módem. No ha sido implementado.

3.8. Mensaje a grupos

El STANAG 5066 no define cómo se ha de implementar los mensajes a grupos, sino simplemente proporciona un campo *booleano* (un único bit) con el que indicar si la dirección de destino especificada es una dirección de nodo local, o una dirección de grupo. Se hace evidente que el filtrado en recepción por parte de la DTS de paquetes *nodo a nodo* dirigidos al nodo local, se ha de hacer comparando con la dirección de nodo local conocida, sólo si el paquete *no tiene* el bit de grupo inactivo (puesto a 0). El estándar no define cuál ha de ser el mecanismo a seguir en caso de que el bit de grupo estuviera activo.

En este apartado se describen las decisiones de diseño tomadas respecto al tratamiento de paquetes con el bit de grupo.

1. Cada nodo ha de guardar una lista de los grupos a los que pertenece. Un nodo puede pertenecer a múltiples grupos.
2. El filtrado de los paquetes dirigidos a los grupos será responsabilidad del propio nodo, que habrá de ignorar los paquetes dirigidos a grupos a los que no pertenece.
3. La decisión de entrar o salir de los grupos no es responsabilidad del servidor STANAG 5066. Por tanto, ninguna de las capas podrá tomar la decisión autónoma de entrar a un grupo.
4. El mecanismo de entrada y salida de los grupos es a través de primitivas `S_MANAGEMENT_MSG_INDICATION` y un parámetros de arranque específico del servidor STANAG 5066. Por tanto, la gestión de grupos es responsabilidad del gestor de red, a través de los parámetros de arranque del servidor, o a través de un cliente STANAG 5066 conectado con `rank` 15, con capacidad de enviar las primitivas `S_MANAGEMENT_MSG_INDICATION`.

Estas definiciones no cambian nada respecto a las implementaciones en clientes STANAG 5066, porque, aunque no se define cómo se ha de implementar a nivel de protocolo, el uso consiste simplemente en activar o desactivar el bit de grupo.

Capítulo 4

Arquitectura del servidor STANAG 5066

La definición inicial de la arquitectura del servidor STANAG 5066 ha sido una parte clave del desarrollo. El estándar no da pautas hacia una implementación; y hacer un buen diseño de la arquitectura es clave para poder realizar un desarrollo de manera continuada, sin tener que replantear aspectos fundamentales del diseño a lo largo de la misma, lo que obliga a hacer un estudio pormenorizado a nivel de diseño para plantear la arquitectura. La aplicación desarrollada se ejecutará bajo GNU/Linux y ha de ser programada en C, debido a la imposición de la plataforma y a la experiencia del equipo de trabajo.

Como reglas de codificación del proyecto, se decidió que el lenguaje de programación será C, en su versión C99, así como la obligación de usar diferentes *flags* de compilación. Esto se verá en la Sección 4.4.

Una de las incógnitas iniciales es cómo se comunican cliente y servidor. En el estándar sólo se define la codificación de los datos en las comunicaciones Cliente ⇒ SIS, no cuál ha de ser el medio de comunicación. Además, no todas las S_Primitivas SIS ⇒ Cliente contienen un campo de SAP de destino, con lo que se deduce que la comunicación con cada cliente ha de realizarse a través de un canal *punto a punto*, y esto significa que ha de haber una interfaz de comunicación para cada cliente conectado.

La solución implementada se basa en el protocolo TCP/IP, por ser éste un mecanismo sencillo y extendido para otros protocolos de comunicaciones. Para usar este mecanismo, el servidor STANAG 5066 exportará un puerto TCP/IP a través de un socket, al que se conectarán los clientes. Es necesario que el servidor mantenga una tabla de asociación SAP ⇔ Cliente TCP para indentificar a qué cliente TCP (socket) ha de enviar qué S_Primitiva dependiendo del SAP de destino.

Las capas SIS, CAS y DTS han sido implementadas por separado, componiendo cada una, una librería que ha de ser utilizada por el servidor STANAG 5066. Para la comunicación entre capas se ha definido una estructura de datos común, así como una API que será descrita más adelante.

Ya que las capas SIS, CAS y DTS se centran en el procesado de las S_Primitivas y

PDUs, se definió una capa transversal de gestión, la capa MANAGEMENT o MNGT, que se encarga de el intercambio de datos entre las capas, de la gestión de los temporizadores exportados por éstas y también de las comunicaciones externas, como el control de la comunicación con el cliente STANAG 5066 para entregar los datos a los clientes y al módem HFDVL.

Como se describe de manera más ampliada en la Figura 4.1, el servidor STANAG 5066 no se comunica directamente con los clientes STANAG 5066, sino que lo hace a través del cliente HFDVL, así que la tabla de asociación SAP ⇔ Cliente TCP ha de estar alojada en este último, sin obviar que ha de existir un protocolo de comunicaciones entre el cliente HFDVL y el servidor STANAG 5066 para coordinar esto.

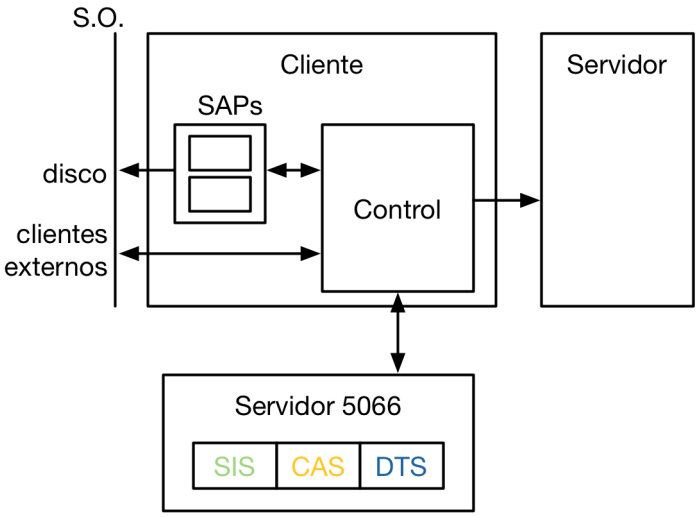


Figura 4.1: Bloques lógicos que intervienen en el control de las comunicaciones cliente HFDVL y servidor STANAG 5066

Tal y como muestra la Figura 4.2, la comunicación entre capas se hace a través de la capa MNGT, que es la que realiza las llamadas a las diferentes funciones de la API de cada una de las capas, y recoge los datos resultantes para seguir con el procesado entregándolos a la capa siguiente.

4.1. Arquitectura en librerías y capa MNGT

La identificación de las funciones pertenecientes a cada capa sirve para poder trazar una división clara de tareas. Tras un análisis del estándar hemos decidido definir una capa añadida a SIS, CAS y DTS, y cuya inclusión permitirá que estas tres capas centren sus funciones en implementar el estándar. Esta capa realiza tareas de gestión y se le ha denominado MANAGEMENT, o *MNGT* de manera abreviada.

Sus tareas son las de realizar todas las funciones relacionadas con el bucle de eventos de la aplicación servidor, tanto de las conexiones TCP/IP, como de los temporizadores exportados por las capas; además se encarga de desempaquetar los datos provenientes del cliente HFDVL y de encapsularlos en estructuras estándar para el procesado de la

SIS y DTS. Es posible que el procesado de estos datos genere más datos de respuesta destinados a otras capas o incluso hacia fuera del servidor (clientes o módem). La capa MNGT se encargará de propagar esos datos a la capa correspondiente.

La implementación en librerías de las diferentes capas tiene varias ventajas en un proyecto como éste:

- Establecer separación de las tareas en el equipo de trabajo, y permitir la coordinación el bloques más pequeños. Cada capa es un paquete software individual.
- Compartimentar el desarrollo del código. Cada capa se gestiona de manera individual, y libera versiones conforme a la evolución del código. A la hora de mantener una configuración sencilla y operativa, es conveniente tener cada paquete por separado.
- Se puede abstraer a cada capa del procesado inherente a hacer una aplicación (excepto a la capa MNGT, cuya labor es precisamente esa), como son los bucles de eventos, comunicaciones por sockets, etc...

La interfaz para hacer uso de las tres capas SIS, CAS y DTS fue definida de manera muy similar para todas ellas, de hecho, sólo la DTS cuenta con una excepción, `from_modem`. Todas las capas cuentan con las siguientes funciones de entrada, haciendo referencia al origen de los datos (relativos a dicha capa). Se aprecia en la Figura 4.2 y se describe a continuación:

- `from_upper(void)`. Este punto de entrada entrega datos provenientes de la capa inmediatamente superior a la capa tratada. Es posible que la capa genere datos de salida, tanto a la capa superior como a la capa inferior.
- `from_lower(void)`. Se entregarán datos provenientes de la capa inferior. Existe la posibilidad de la capa llamada genere datos para las capas superior e inferior.
- `from_timer(void)`. Esta llamada es provocada por los timers que la propia capa haya configurado previamente. No tiene como propósito proveer datos de entrada, sino generar cambios de estado o datos de salida de manera asíncrona, sin depender de un estímulo proveniente de otra capa.
- `from_back_for_more(void)`. Las estructuras de intercambio de datos entre capas están pensadas para contener una única PDU o Primitiva cada vez, y es posible que las capas requieran generar múltiples PDUs cuando se les haga una llamada `from_upper`, `from_lower` o `from_timer`. Para ello, las capas indicarán que requieren ser llamadas de nuevo para seguir entregando datos (habrán de mantener el estado interno). Esta función recoge la necesidad de volver a ser llamada para seguir entregando datos de salida.
- **(DTS)** `from_modem`. Utilizada para notificar la DTS los cambios de estados de la librería módem: TX, RX o IDLE. Al ser la capa encargada de la gestión de las transmisiones y recepciones, ha de tener conocimiento de el estado del módem, sobre todo para implementar un protocolo de acceso al canal.

- **(DTS)** `change_modem_bps(void)`. El estándar STANAG 5066 tiene una imposición de 2 minutos de uso máximo consecutivo del canal. La DTS ha de tener conocimiento del throughput del módem, para transmitir $120 \times bps$ bits en cada bloque de transmisión.

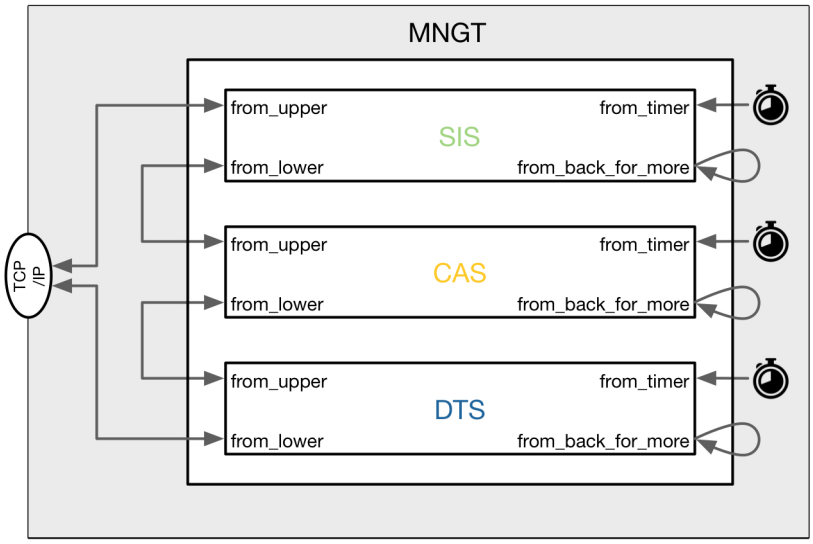


Figura 4.2: Comunicación entre capas y gestión de la MNGT

Para mejorar la trazabilidad y evitar *leaks* de memoria, hemos establecido como una regla de codificación de las libererías, que está **prohibido reservar memoria durante la ejecucición** de la misma; únicamente se puede reservar memoria en la función de inicialización.

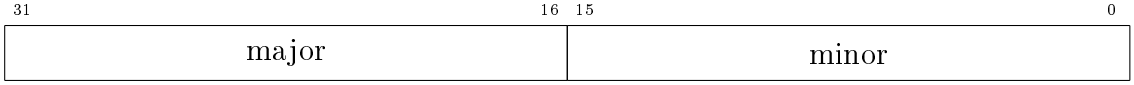
4.2. API de las capas y estructura de datos entre capas

La API en uso para las tres capas se ha definido de manera común a ellas, excepto las dos funciones específicas de la DTS, como ya hemos visto. Cada capa tiene un conjunto de funciones comunes, que son: `version(void)`, `init(...)`, `destroy(...)`, `process_from_upper(...)`, `process_from_lower(...)`, `process_from_timer(...)`, `process_from_back_for_more(...)`, precedido del prefijo "`S5066_{nombre_de_la_capa}_(...)`", de manera, que para la DTS, quedaría `S5066_DTS_init(...)(...)`, por ejemplo. Durante esta sección obviaremos el nombre de la capa para las funciones comunes a las tres, no así para las específicas de la DTS. La definición de la API se presenta a continuación:

```
uint32_t S5066_version(void);

void* S5066_init(int *timer_fds,
  int *ntimers,
  uint32_t node_address,
```

```
  /* ONLY DTS */ const uint32_t *group_address_list,
  /* ONLY DTS */ int group_list_sz);

int S5066_destroy(void *handler);

int S5066_process_from_upper(void *handler,
  /* ONLY SIS */ struct S5066_SIS_clnt_info *clientinfo,
  struct S5066_data *data,
  struct S5066_data *ret);

int S5066_process_from_lower(void *handler,
  /* ONLY SIS */ struct S5066_SIS_clnt_info *clientinfo,
  struct S5066_data *data,
  struct S5066_data *ret);

int S5066_process_from_timer(void *handler,
  /* ONLY SIS */ struct S5066_SIS_clnt_info *clientinfo,
  struct S5066_data *data,
  struct S5066_data *ret,
  int timer_index);

int S5066_process_from_back_for_more(void *handler,
  /* ONLY SIS */ struct S5066_SIS_clnt_info *clientinfo,
  struct S5066_data *data,
  struct S5066_data *ret );

/* ONLY DTS */
int S5066_DTS_process_from_modem(void *handler,
  struct S5066_data *data,
  struct S5066_data *ret,
  int modem_status);

/* ONLY DTS */
void S5066_DTS_change_modem_bps(void *handler,
  uint32_t new_bps);
```

La función `version(void)` devuelve el número de versión en un **uint32**, que se codifica como se muestra a continuación:



4.2.1. Inicialización

Cada librería ha de ser inicializada a través de la función `init(...)`. La capa MNGT entrega a esta función un puntero **int** `*timer_fds` que apunta a un array de

int, de tamaño especificado en **int** `*ntimers`, y notifica la dirección del nodo local en **uint32_t** `node_address`. La capa correspondiente rellenará el array **int** `*timer_fds` con los descriptores de ficheros que haya instanciado, y devolvolviendo la cantidad de elementos usados en el array en `ntimers`.

Al hacer la inicialización, la capa devuelve un puntero **void***, que apunta al *handler* de la capa. Internamente, el *handler* apunta a la zona de memoria utilizada por la librería de la capa para guardar el estado y los datos no entregados. La reserva de memoria tiene lugar en la llamada `init(...)`. Además, este puntero ha de ser entregado como el primer argumento en cada una de las demás funciones de API. Esto es para que la función tenga acceso a la memoria en la que se alojan los datos relativos al estado de la capa.

Todas las reservas de memoria que puedan ser utilizadas durante las diferentes llamadas al API, han de ser hechas en la llamada a esta función.

4.2.2. Funciones para procesar datos, `process_from_...`

De manera general puede considerarse que en la API existen dos funciones para entrada de datos (`from_upper(...)` y `from_lower(...)`), una para eventos asíncronos (`from_timer(...)`), y una más auxiliar para extraer datos pendientes (`from_back_for_more(...)`). Además del *handler*, estas funciones tienen como argumentos los punteros **struct** `S5066_data`*data y **struct** `S5066_data`*ret, que hacen la labor de entrada y salida de datos de las capas.

Las funciones para entrada de datos `from_upper(...)` y `from_lower(...)` proporcionan una interfaz para la entrada de los datos provenientes de las capas superior e inferior respectivamente; por ejemplo en la CAS, `from_upper(...)` haría referencia a datos provenientes de la capa SIS. Los datos provenientes de la respectiva capa estarán disponibles **siempre** en la variable `*data`, siendo `*ret` una variable auxiliar para salida de datos, y cuyo contenido será ignorado.

Las variables `*data` y `*ret` hacen referencia al intercambio de datos en la trayectoria directa al que hace referencia la llamada, como podemos observar en la Figura 4.3. Tanto `*data` como `*ret` son usadas para la salida de datos, de manera que `*data` es reescrita. La variable `*data` hace referencia a la salida de datos en el mismo sentido de la entrada, y `*ret` en el sentido opuesto, es decir, para `from_upper(...)`, `*data` contendrá la salida destinada a la capa inferior, y `*ret` los datos devueltos a la capa superior.

Por convenio, para las funciones `from_timer(...)`, `from_back_for_more(...)` y `from_modem(...)` se tomará el sentido como si fuera llamado desde la capa superior, es decir, la salida generada con destino a la capa superior deberá estar escrito en `*ret` y los datos con sentido la capa inferior han de guardarse en `*data`.

Las particularidades para cada capa, como es la variable **struct** `S5066_SIS_clnt_info *clientinfo` en la SIS, o las funciones `from_modem` y `change_modem_bps` serán tratados en la sección correspondiente a cada capa.

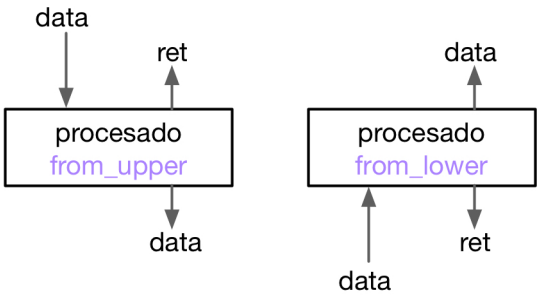


Figura 4.3: Dirección de la comunicación que representan las variables `*data` y `*ret`

Las llamadas a las funciones `process_from_...` tienen como valor de retorno un **int**, que puede tomar los siguientes valores definidos como constantes en los archivos de la API:

```
#define S5066_OUTPUT_NONE              0x00
#define S5066_OUTPUT_ERROR             0x01
#define S5066_OUTPUT_LOWER             0x02
#define S5066_OUTPUT_UPPER             0x04
#define S5066_OUTPUT_BACK_FOR_MORE     0x08
```

Podemos observar que los valores son desplazamientos de un bit, de manera que la capa pueda indicar varios valores de salida haciendo una operación `OR lógica` entre ellos. La descripción de los posibles valores es:

- `OUTPUT_NONE`. Una vez procesados los datos de entrada, y hacer los cambios de estados pertinentes, la capa informa de que no tiene ningún dato de salida. Un ejemplo puede ser en el que ha saltado un temporizador de caducidad de una U_PDU (vencimiento de su TTD), y tras la llamada `from_timer`, se ha desechado la U_PDU y no se generan datos para ninguna de las capas adyacentes.

- `OUTPUT_ERROR`. Durante el procesado de los datos o los cambios de estado internos, se ha producido un error, y no hay garantía de que los datos hayan podido ser correctamente procesados.

- `OUTPUT_LOWER`. La capa ha generado datos de salida con destino a la capa inferior. Si la función que se llamó ha sido `from_upper`, `from_timer` o `from_back_for_more`, se encontrarán en `*data`, si la llamada fue desde `from_lower`, los datos se encontrarán en `*ret`.

- `OUTPUT_UPPER`. Durante el procesado, la capa generó datos hacia la capa superior, y se encontrarán de manera inversa a `OUTPUT_LOWER`, dependiendo de la función que fue llamada.

- `OUTPUT_BACK_FOR_MORE`. Además de poder haber generado datos `LOWER` o `UPPER`, se puede notificar que es necesario que la MNGT tenga que recoger más datos, y eso se notifica mediante este valor de retorno.

La figura Figura 4.4 ejemplifica cuál sería el proceso de llamadas dependiendo de los valores de retorno sucesivos, tras una llamada inicial a `from_upper`.

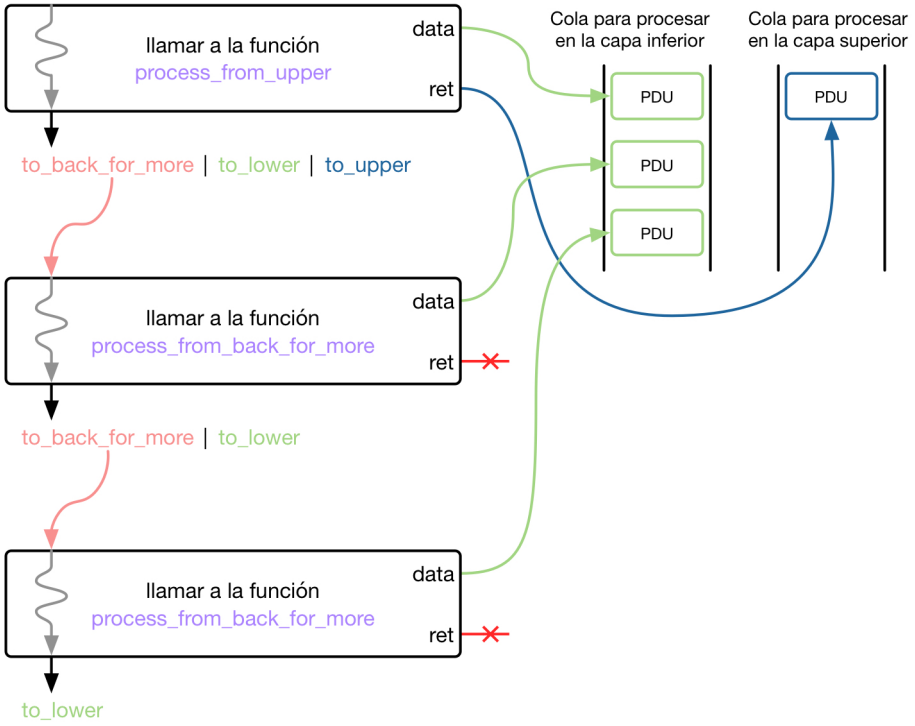


Figura 4.4: Ejemplo de procesado de múltiples salidas, tras la llamada a la función `process_from_upper`

4.2.3. Estructura de datos para intercambiar información entre capas

El intercambio de información entre capas se hace a través de las variables `*data` y `*ret`, siendo ambas del mismo tipo **struct** `S5066_data`. La estructura de datos no cambia en ninguna de las capas, haciendo más fácil la copia de los datos. Al ser relativamente pocos campos y el tipo más frecuente **uint8_t** no va a suponer en un gran uso de memoria (relativo a la plataforma en la que se ejecutará), y por simplicidad hemos decidido mantener los campos incluso en situaciones que sabemos que no van a hacer falta, como por ejemplo, en la comunicación Cliente ⇔ SIS.

La estructura está pensada para contener todos los datos relativos a una primitiva de cualquier capa. Los campos relativos a cada primitiva se encontrarán en diferentes partes de la estructura dependiendo de qué comunicacion entre capas estemos estudiando.

La definición del tipo **struct** `S5066_data` la presentamos a continuación:

```
struct S5066_data {
  /* cross-layer information */
  int16_t  s_pdu_id;  /*!< ID used by SIS for PDU confirmation */
```

```
  uint8_t  prim_type;
  uint8_t  transmission_mode;
  uint8_t  delivery_confirm;
  uint8_t  delivery_order;
  uint8_t  min_rtx;
  uint8_t  priority;
  uint8_t  dst_sap_id;
  uint8_t  src_sap_id;
  uint32_t time_to_die;
  uint32_t src_address;
  uint32_t dst_address;
  uint8_t  link_type;
  uint8_t  warning_type;
  uint8_t  warning_reason;
  uint8_t  reject_reason;
  uint8_t  is_group_address;
  /*  data buffer information */
  uint8_t  *buf;
  uint32_t buflen;
  uint32_t bufdataini;
  uint32_t bufdatalen;
  /* errored blocks array */
  uint16_t *error_blocks;
  uint16_t error_blocks_count;
  uint16_t *non_rcvd_blocks;
  uint16_t non_rcvd_blocks_count;
};
```

Podríamos hacer una división de la estructura en dos grandes bloques: El **bloque de parámetros**, que en el código se indica como `cross-layer information` y **bloque de datos**, indentificado como `data buffer information` junto con `errored blocks array`. El bloque de datos corresponde con un array de **uint8_t**, que sirve tanto para codificar Primitivas completas de manera *binaria*, como para albergar el *payload* en comunicaciones internas entre capas. El bloque de parámetros albergará los diferentes parámetros de la comunicación en comunicaciones internas entre capas.

Todos los datos relativos a una primitiva estarán contenidos en estos dos bloques, aunque no necesariamente usando los dos de manera simultánea. A lo largo de la cadena Cliente ⇒ Módem HFDVL podemos identificar los siguientes escenarios de uso de esta estructura de datos:

- Cliente ⇔ SIS. En este caso, la S_Primitiva se encuentra completamente codificada en `*buf`. En el estándar [10, Anexo A.2.2], se describe cómo ha de estructurarse la codificación para cada S_Primitiva.
- SIS ⇔ CAS. Los datos presentes en `*buf` corresponden a la S_PDU (si hubiese). Ya que la S_PDU no tiene los datos relativos a la transmisión en sí, como por ejemplo (para transmisión) dirección de destino, modo de transmisión, SAP de

destino, (y para recepción), dirección de origen, SAP de origen, etc, estos datos se han de codificar en el bloque de parámetros. Existen C_Primitivas de notificación de enlace, que no tienen S_PDU asociada, por lo tanto se ignorará el campo `*buf`.

- CAS ⇔ DTS. Caso muy similiar al anterior, en que las C_PDU no contienen datos relativos a la transmisión y éstos han de ser codificados en el bloque de parámetros. En `*buf` se encontrará la C_PDU, si corresponde.

- DTS ⇔ Módem HFDVL. En esta comunicación, todos los datos estarán en `*buf`, ya que éste se interpreta como el bloque de bytes a transmitir por el módem. De la misma manera que en las comunicaciones Cliente ⇔ SIS, las D_PDUs están completamente definidas y todos los parámetros necesarios para la comunicación están presentes en la codificación *binaria*.

La Figura 4.5 describe cómo evolucionan los datos de las Primitivas dentro de la estructura **struct** `S5066_data` a medida que van pasando por las diferentes capas, siendo este proceso interpretable de manera bidireccional. Se aprecia que desde la S_Primitiva hasta la D_Primitiva, la U_PDU se mantiene sin cambios, sólo cambiando las cabeceras, y siendo las D_DPU las que trocean las C_PDUs.

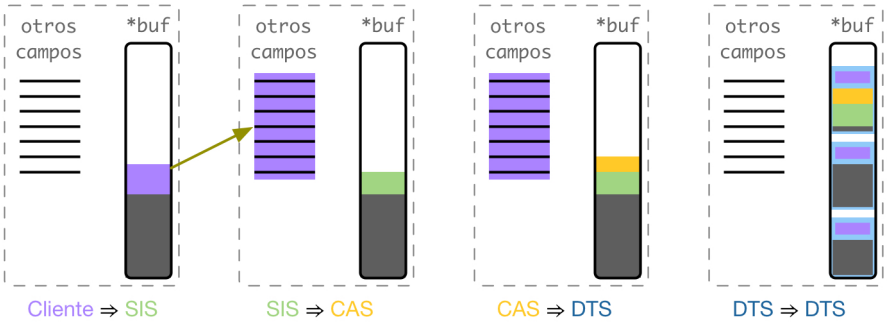


Figura 4.5: Uso de los diferentes campos de la estructura **struct** `S5066_data` en función de la primitiva que se codifique en ella

El hecho que la U_PDU no se modifique durante el procesado de las diferentes capas, nos llevó a tomar una decisión de diseño para un manejo de memoria óptimo: los datos en `*buf` han de estar alineados con el final del buffer, como se observa en la Figura 4.6, de manera que se utiliza este buffer de manera óptima. Los únicos datos que cambian en `*buf` a lo largo del procesado son las cabeceras.

Además, de esta manera se puede calcular el tamaño de `*buf` como el tamaño máximo de U_PDU más el tamaño máximo de las cabeceras añadidas.

En el caso de la comunicación DTS ⇔ Módem HFDVL, el buffer tiene un tamaño diferente, ya que no se intercambian Primitivas, sino los datos que han de ser transmitidos y recibidos por el módem. El tamaño de este buffer viene descrito por el tiempo máximo de transmisión multiplicado por los *bps* máximos del módem.

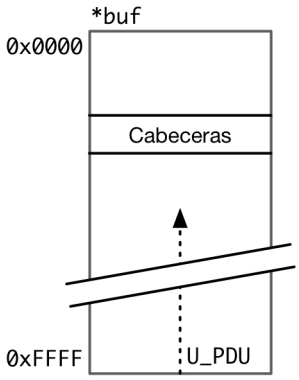


Figura 4.6: La alineación de los datos en `*buf` se hace respecto a las últimas posiciones de memoria del buffer

4.3. La pseudo-capa MNGT

Como ya hemos mencionado, la capa MNGT es la que dota del mecanismo de intercomunicación entre las diferentes capas, y que éstas, estando desarrolladas como librerías, necesitan de una aplicación que implemente el bucle de eventos, el cual también se implementa dentro. De manera simple, la MNGT se puede resumir como la aplicación que instancia las capas y las intercomunica. Por este motivo, aunque durante todo el documento le llamamos "capa" MNGT por simplicidad, más bien habría que considerarla una pseudo-capa, porque toma muchas más funciones que las capas responsables del protocolo propiamente dicho.

Anteriormente se ha ilustrado en la Figura 4.1 que la MNGT se comunica con el cliente HFDVL, y que es el cliente HFDVL el que dispondrá de las comunicaciones tanto con los clientes STANAG 5066, como con el módem.

Para este propósito, la capa MNGT implementa un protocolo de comunicación particular a través de TCP/IP, para encapsular los datos provenientes o con destino tanto al módem como a los clientes STANAG 5066 y esto permite a la capa SIS (interfaz superior del STANAG 5066) y a los clientes STANAG 5066, implementar el protocolo y las comunicaciones a través de S_Primitivas tal cual se describen en el estándar [10, Anexo A.2], puesto que a que el protocolo Servidor STANAG 5066 ⇔ Cliente HFDVL hace transparente la comunicación Cliente ⇔ SIS.

4.3.1. Gestión de las comunicaciones entre las capas

Respecto a la comunicación entre capas, la capa MNGT implementa una cola de elementos **struct** `S5066_data` para cada uno de los sentidos de entrada de cada capa, es decir, la capa SIS tiene dos colas de entrada, `from_lower` y `from_upper`, igual que CAS y DTS. Esto es así para poder manejar más fácilmente toda la gestión de información. Cada **struct** `S5066_data` de salida, genera inmediatamente un elemento más en la cola de entrada correspondiente.

Para facilitar la implementación, y también para minimizar la cantidad de bytes co-

piados, cada cola tiene un número de elementos fijos, los cuales se marcan como usados o como libres; sin embargo, estos elementos no son transpasados a otras colas, sino que se intercambia el puntero `*buf` en caso de ser necesario, y se copia el bloque de parámetros. Recordemos que a la hora de realizar una llamada `from_lower`, `from_upper`, ..., se han de pasar como parámetros dos punteros a **`struct`** `S5066_data`, `*data` y `*ret`. La estructura `*data` es la inmediatamente siguiente en la cola de entrada, y la variable `*ret`, es una estructura auxiliar *ad hoc* para poder recoger los datos con la dirección opuesta a la dirección de la función llamada.

Si la llamada a la función de procesado `from_...` tiene alguna salida `OUTPUT_...` disponible, se obtendrá un elemento libre de la cola de entrada de la capa correspondiente, y se copiará a este los elementos del bloque de parámetros de la variable de salida correspondiente, además, se intercambiarán los punteros al array `*buf`, ahorrando la copia de este buffer.

Existe una excepción respecto al intercambio de punteros de `*buf`, y es que el buffer `*buf` de los elementos de la cola de entrada `from_upper` destinada al módem, y la cola de entrada `from_lower` destinada a la DTS, tienen un tamaño mucho mayor, ya que el tamaño de este buffer está ajustado para el intercambio de información con el módem, tal y como se comentó al final del punto anterior.

4.3.2. Bucle de eventos

La capa MNGT es la responsable de implementar el bucle de eventos asociado a la aplicación servidora. En este bucle de eventos ha de incluir la comprobación de todos los descriptores de ficheros asociados al funcionamiento normal de la aplicación, como pueden ser sockets, o temporizadores internos.

Además de éstos, MNGT ha de incorporar todos los descriptores de ficheros que las capas hayan inicializado como identificadores de temporizadores. Para ello, ha de mantener una tabla `indice_fd` ⇔ `{timer_fd, capa}`, en la que se asocie cada temporizador, `indice_fd`, con la capa que lo instanció y con un identificador único dentro de la capa `{timer_fd, capa}`. De esta manera las capas pueden crear eventos en temporizadores, asociados a diferentes eventos dentro del protocolo STANAG 5066, como por ejemplo, la caducidad de un *physical link*.

4.4. Makefile y compilación

En un proyecto software, una parte fundamental del desarrollo es la configuración del entorno de edición y compilación. Debido a la experiencia del equipo de desarrollo en otros proyectos, se decidió no usar ningún tipo de IDE para el mantenimiento y compilación del proyecto. Esta tarea se puede hacer de manera *sencilla* a través de un interprete de comandos y un *makefile*, que al fin y al cabo es lo que hacen (aunque de manera automatizada) los entornos como Qt: mantener un control de los archivos y generar un makefile.

El editor de texto para programar el código es de libre elección, y no es algo transcendental; aunque durante este proyecto se han usado editores como Brackets, Atom, Qt Editor, Kate, Geany y Sublime Text.

La compilación se lleva a cabo a través de un *interprete de comandos* usando la aplicación `make`, que requiere de la existencia de un makefile correctamente configurado. Un makefile configurado adecuadamente minimiza las ediciones que necesitará a medida que el proyecto crezca incluyendo más archivos de código. De hecho, el makefile utilizado para este proyecto en cada una de las librerías, en su objetivo más completo, genera tanto la librería para enlazado estático, para enlazado dinámico, y una versión de documentación automatizada con `doxygen`.

El makefile contiene, además, los *flags* de compilación, así como parámetros relativos al compilador como puede ser el estandar de C a usar (en nuestro caso, C99). Algunos de los flags usados son: `-Wall`, `-pedantic` y `-fsigned-char`.

También es necesario disponier de la herramienta `gcc`, para poder compilar las liberías, y compilar el server, así como enlazarlo con las librerías de las capas. El makefile para la capa MNGT está configurado para hacer el *enlazado* con las tres librerías de las capas SIS, CAS y DTS, bien de manera estática o bien de manera dinámica. Durante este proyecto se ha tenido la preferencia de generar los ejecutables *enlazados de manera estática* por el hecho de estar autocontenidos, y no hacer necesaria la instalación de unas librerías en el sistema que ninguna otra aplicación va a usar.

El makefile para cada librería se puede encontrar en el Apéndice A.

Para poder coordinar el desarrollo con otros miembros del equipo de desarrollo, trabajamos haciendo uso de un repositorio SVN. Esto ha permitido ir desarrollando cada una de las capas en local, e ir haciendo subiendo los cambios a medida que se iban implementando funcionalidades, de manera que la parte del equipo dedicada a desarrollar la capa MNGT podía actualizar cada capa, y generar una nueva compilación con las partes recién desarrolladas.

4.5. Implementación de la capa SIS

La principal tarea de la capa SIS es la de actuar de interfaz entre los clientes y el *stack* STANAG 5066, por tanto es la responsable de codificar y decodificar las primitivas tal como se definen en el estándar [10, Anexo A.2.2]. También es la que se encarga de hacer las peticiones de *pysical link* a la CAS y de declarar esos *links* como expirados.

Para la comunicación ARQ es imprescindible que exista un *physical link* con el nodo de destino. Si un cliente STANAG 5066 envía una U_PDU con modo de transmisión ARQ y no existe *physical link* creado con el destino, es necesario crearlo; mientras dicho *physical link* se está creando, es necesario mantener las U_PDUs retenidas, debido a esto, la capa SIS también ha de implementar una cola de U_PDUs.

4.5.1. Estructura de ficheros

Los ficheros de esta librería están dispuestos como se describe en la Figura 4.7. En ella podemos apreciar tres directorios, tags, branches y working; esta es la estructura habitual de un repositorio SVN (cambiando working por trunk), en la cual para este PFC no se usaron los dos primeros. En working se encuentran todos los archivos de código de este proyecto.

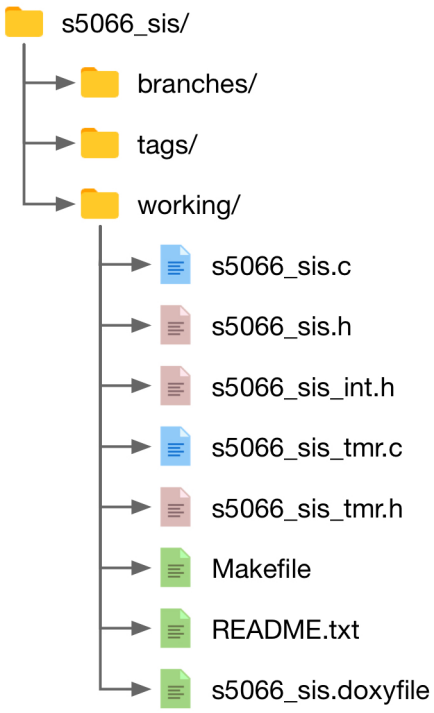


Figura 4.7: Estructura de ficheros en el directorio de trabajo de la capa SIS

Se destacan como iconos en color azul los ficheros .c, habiendo en esta capa SIS únicamente dos, sis.c y timer.c, y en color rosa los ficheros de cabeceras .h, existiendo tres, sis.h, sis_int.h y timer.h. En verde encontramos otros ficheros como Makefile, o los relativos al doxygen, README.txt y sis.doxyfile.

- **sis.c**, alberga la implementación de las funciones públicas de la librería, `from_*`, las de inicialización y la de petición de versión.
- **sis_int.h**, declara todas las estructuras de datos y funciones internas que serán usadas por la librería.
- **sis.h**, contiene la declaración de las funciones ya nombradas de la API, `from_*`, además de las de inicialización y consulta de versión. También declara las estructuras de datos a usar `struct S5066_SIS_data`.
- **Makefile**, contiene la configuración de compilación. La versión de la capa está definida aquí, y se introduce como macro de compilación.
- **sis.doxyfile**, contiene la configuración para generar el documento doxygen referente a este código.

4.5.2. Handler

El *handler* de la capa SIS, que recordemos, es una estructura que se crea en la función `init(...)`, está definido en el fichero sis_int.h, porque es una variable modificable únicamente por la capa SIS. Su existencia es necesaria para poder mantener el estado entre llamadas de las funciones ya descritas de la API.

En el siguiente fragmento de código podemos ver su definición:

```
struct S5066_SIS_handler {
    struct S5066_sap_info_t     *sap_usage[S5066_NSAPS];
    struct S5066_node_info_t    *node[S5066_NNODES];
    struct S5066_U_PDU_info_t   *u_pdu_queue[S5066_U_PDU_QUEUE_SIZE];
    struct S5066_new_timer_info_t *new_timer[S5066_NTIMERS];
    struct S5066_SIS_back_for_more_t back_for_more;
    uint32_t node_addr; /* local node address */
    uint32_t default_ttl;
    uint16_t ntimers;
    uint16_t dlvry_conf_max_rtrn_bytes;
    uint8_t max_invalid_prim;
    uint8_t invalid_prims;
    uint8_t free_u_pdus;
    uint8_t sis_flow_status;
    uint8_t dts_flow_status;
    int timerfd;
    int next_timer;
};
```

La memoria para alojar el *handler* se reserva en la llamada `init(...)`, donde además se inicializan todos los campos. Podemos observar como existe un campo dedicado a la gestión de los SAPs, que es `struct S5066_sap_info_t *sap_usage[S5066_NSAPS]`, un campo para la gestión de los estados de enlace de nodos remotos, `struct S5066_node_info_t *node[S5066_NNODES]`, otro para la gestión de los diferentes timers `struct S5066_new_timer_info_t *new_timer[S5066_NTIMERS]`, y por último, destacable también `struct S5066_U_PDU_info_t *u_pdu_queue[S5066_U_PDU_QUEUE_SIZE]`, que almacena los U_PDU que lo requieran.

4.5.3. Codificación/decodificación de las S_Primitivas

Las S_Primitivas se codifican tal y como las define el estándar en [10, Anexo A.2.2], siendo escritas en la variable `*buf`. Esta variable contiene el *stream* de bytes provenientes del (o con destino al) socket, con destino al cliente STANAG 5066 conectado al SAP correspondiente.

Recordemos que la conexión Cliente ⇔ SIS se hace a través de un socket TCP/IP, y para el servidor es necesario mantener una tabla de correspondencia SAP ⇔

Cliente TCP . En relación a esta correspondiencia se ha diseñado la estructura **struct** `S5066_SIS_clnt_info *clientinfo`, que está presente en todas las funciones de la API de procesado de datos, y tiene la siguiente estructura:

```
struct S5066_SIS_clnt_info {
    uint8_t command;
    uint8_t msap_id;
};
```

Esta estructura se tendrá en cuenta en caso de que alguna llamada a las funciones de procesado de datos devuelva un valor `OUTPUT_UPPER`, y su propósito es indicar en la varible `msap_id` al cliente a quién ha de entregar la S_Primitiva, y un comando asociado, presente en `command`. Los comandos tienen relación con dos propósitos:

- **Indicar estados de asociación a los SAP al servidor**. Ya que el estado de asociación del SAP se transmite al cliente a través de la S_Primitiva correspondiente (`S_BIND_ACCEPTED`/`S_BIND_REJECTED`/`S_UNBIND_INDICATION`)
- **Pedir al servidor confirmación de entrega al SAP concreto**. Para minimizar condiciones de carrera, en la que el cliente haya podido desconectarse sin notificar a la capa SIS (cuando el equipo donde se ejecuta el cliente se apaga repentinamente, por ejemplo)

Respecto a la implementación del código de codificación y decodificación de las primitivas, es imprescindible la existencia del puntero **uint8_t** `*pini`.

Para hacer la decodificación, este puntero se sitúa apuntando al inicio de la S_Primitiva con la siguiente instrucción:

```
uint8_t *pini = data->buf + data->bufdataini;
```

Con esto, ya se puede ir accediendo a los diferentes campos de la codificación de la primitiva, simplemente accediendo añadiendo un offset a la posición de memoria apuntada por este puntero, por ejemplo, para obtener el tipo de primitiva usaremos esta instrucción:

```
const uint8_t s_prim_type = pini[0];
```

Como aclaración, el campo `type` se encuentra en la posición `0` porque la capa MNGT elimina la cabecera de la comunicación Cliente ⇔ SIS, que se describe en el estándar [10, Anexo A.2.2.2].

En la parte de codificación, se utiliza la misma técnica del puntero **uint8_t** `*pini`, colocándolo inicialmente en la parte posterior del buffer, escribiendo primero la U_PDU y actualizando el valor de `*pini` para apuntar a la posición más alta de los datos escritos en el buffer. Por último, se escribe la cabecera correspondiente. Este proceso se describe gráficamente en la Figura 4.8.

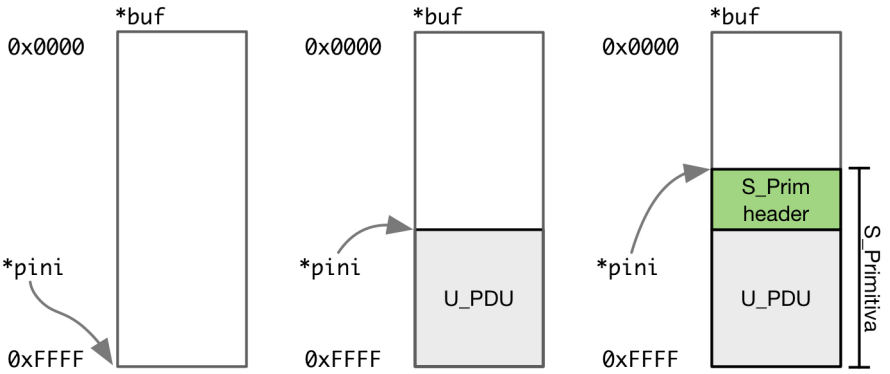


Figura 4.8: Utilización del puntero `*pini` por la capa SIS

4.5.4. Gestión de timers

Los timers son una parte fundamental de la librería, y se utilizan para dos propósitos: el primero es controlar el *timeout* asociado al envío SIS ⇒ Cliente de la primitiva `KEEP_ALIVE`; el segundo propósito, y más importante, es controlar el tiempo de inactividad de los *soft links* con cada nodo remoto, de manera que tras un tiempo concreto de inactividad, el *soft link* se dará por roto debido a la falta de actividad.

El control de timers se hace a través de diversas funciones definidas en el archivo sis_tmr.h e implementadas en sis_tmr.c. Estas funciones se utilizan para modificar de manera centralizada y ordenada la estructura **struct** `S5066_new_timer_info_t *new_timer`.

Cuando uno de los timer expira, la capa MNGT recibirá el señal del Sistema Operativo, y hará la llamada a la función de la API `process_from_timer(...)`, en la cual pasará como argumento el índice del timer, y de esta manera el proceso puede identificar a qué evento corresponde.

La implementación usada en esta capa, es una versión mejorada de la primera aproximaxión hecha durante el desarrollo de esta capa. En la primera versión de gestión de timers, a cada evento programado se le asocia un timer, de manera que la cantidad de eventos simultáneos que la SIS puede controlar está limitada por la cantidad de timers que la MNGT permita a la capa crear en su función `init`.

Para evitar esto, y además, optimizar la gestión de recursos del Sistema Operativo, la SIS sólo usará un timer del sistema, guardando una lista de eventos pendientes en **struct** `S5066_new_timer_info_t *new_timer`. Cuando expira el timer (y se llama a `process_from_timer(...)`), se buscará el evento más reciente, ya que es este evento el que corresponde gestionar, y una vez gestionado y eliminado de la lista, se buscará el siguiente; por último, se ajusta el tiempo del timer al tiempo correspondiente de expiración de este evento.

Esta gestión optimizada es posible porque la precisión requerida en el timer es del orden de segundos, incluso podría hacerse la aproximación gruesa a varios segundos. La expiración de los enlaces no requiere una coordinación crítica entre los nodos, lo

cual no hace importante la exactitud, y sumando la alta latencia de las comunicaciones por HF, (es posible que cuando expire un evento, se esté recibiendo una transmisión de hasta 2 minutos, imposibilitando el uso del canal) permite esta aproximación gruesa.

4.5.5. Gestión de los enlaces

Una de las tareas fundamentales de la capa SIS, es la gestión de la creación y ruptura de enlaces. Es la capa SIS la que conoce la existencia o no de enlace con un nodo remoto, y en función de los modos de transmisión con los que se solicitan transmitir las U_PDUs, decide si es necesario crear un enlace con el nodo de destino.

Los enlaces son imprescindibles para una comunicación ARQ. El hecho de requerir enlace antes de "aceptar" la transmisión de cualquier paquete ARQ, hace que se ahorre mucho tráfico ARQ que tiene el requisito de respuesta del nodo remoto. Por tanto, tiene sentido que exista un proceso previo (creación de enlace) que sea responsable de comprobar si el nodo remoto está disponible.

El proceso completo de existencia de los enlaces se describe en la Figura 4.9. Existen cinco estados posibles de un *soft link* que ayudan a mantener un control de proceso de creación y ruptura de los mismos.

Los eventos que generan transiciones entre los diferentes estados provienen de las funciones `process_from_upper(...)`, `process_from_lower(...)`, y `process_from_timer(...)`, que son tanto la llegada de nuevas U_PDUs para ser transmitidas, la actualización del estado del *physical link* por parte de la capa CAS, o el temporizador que informa de la expiración de un enlace por inactividad.

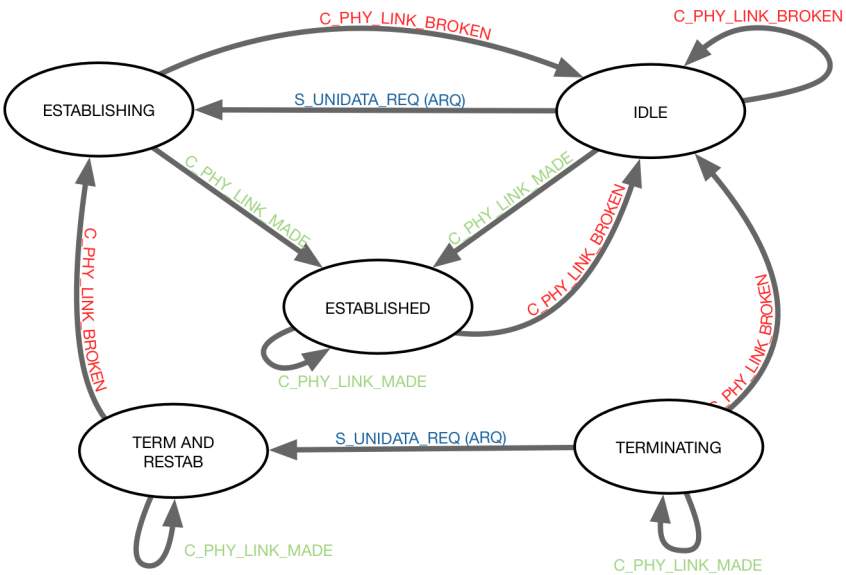


Figura 4.9: Estados de los *soft links* implementados por la SIS, y sus transiciones

4.5.6. Buffers de U_PDUs (mientras se hace el enlace)

En el caso de que un cliente STANAG 5066 quiera transmitir U_PDUs con el modo ARQ a un nodo con el que no exista enlace vigente, se ha de realizar el proceso de enlazado. Mientras se está gestionando la creación del enlace, todas las U_PDUs con modo ARQ han de ser "retenidas" antes de transmitirse.

Cabe decir que si el enlace con el nodo remoto ya existe, la U_PDU será directamente encapsulada en una S_PDU y transmitida a la capa CAS, igual que las U_PDUs que tienen modo NO-ARQ.

Para el almacenamiento temporal de las U_PDUs destinadas a estos nodos con los que se está creando el enlace, se ha definido el buffer **struct** `S5066_U_PDU_info_t *u_pdu_queue` en el *handler*. Esta variable es un *array* de tamaño `S5066_U_PDU_QUEUE_SIZE`, lo que provoca que sea posible que este buffer se llene en caso que un cliente transmita muchas U_PDUs en un corto espacio de tiempo. El mecanismo para informar a los clientes que es necesario que paren la transmisión de U_PDUs se define en el Apartado 4.5.7, o la capa SIS podría desconectarles unilateralmente.

4.5.7. Gestión de flow on/off

Las primitivas `FLOW_ON` y `FLOW_OFF` se usan para forzar a los clientes que paren de enviar datos al servidor STANAG 5066, normalmente usadas porque el servidor se está quedando sin recursos.

La capa SIS es la responsable de enviar esta primitiva a los clientes, aunque no siempre porque sean sus buffers los que ese han llenado. Recordemos que hay un caso en el que la SIS acumulará las U_PDUs a transmitir (U_PDUs con modo de transmisión ARQ y destinado a nodos sin enlace creado). Sin embargo, hay otros múltiples casos en el que se limita a actuar como *bypass* de la información. En ese caso, es la capa DTS la que se responsabiliza de almacenar los datos a transmitir.

Por tanto, existen dos posibles orígenes por el que se puede producir un *flow off*: SIS o DTS.

Es necesario mantener el estado del *flow* de la capa DTS para poder limitar los eventos del *flow* de la capa SIS. Esto se puede apreciar en la Figura 4.10, en la que se ven los diferentes estados de *flow* por pares SIS-DTS, y tanto los eventos que provoca el cambios de estado, como los eventos generados por los cambios de estado.

En la capa SIS, el evento que provoca un cambio en su estado del *flow* es que el número de buffers libres del array **struct** `S5066_U_PDU_info_t *u_pdu_queue` esté por debajo de `S5066_SIS_FLOW_OFF_THRESHOLD`, por lo que se generará un *flow off* de la SIS, o que vuelva a estar por encima del valor `S5066_SIS_FLOW_ON_THRESHOLD`, en cuyo caso se generará un *flow on* de la SIS.

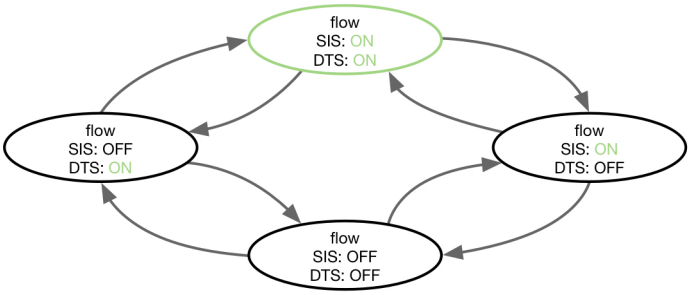


Figura 4.10: Estados del flujo de cliente, en función de las capas SIS y DTS

4.6. Implementación de la capa CAS

La responsabilidad de la capa CAS se centra en ejecutar la creación y ruptura de los *physical links* bajo petición de la capa SIS. Además, ha de actuar como *bypass* de las S_PDUs, tanto de transmisión como de recepción.

Existen dos tipos de C_Primitivas, las que están relacionadas con los links (tarea principal de la CAS), y las que están relacionadas con los "UNIDATA". En el primer caso se encuentran las primitivas `C_PHYSICAL_LINK_...`; en el segundo caso están las `C_UNIDATA_...` y `C_EXPEDITED_UNIDATA_...`.

En el caso de las C_Primitivas relacionadas con los "UNIDATA", la capa CAS únicamente encapsulará los S_PDUs asociadas dentro de un C_PDU de tipo 0. Este proceso se explica en el Apartado 4.6.3.

4.6.1. Estructura de ficheros

De manera similar a la capa SIS, los ficheros de esta librería se organizan como se describe en la Figura 4.11. Vemos los tres directorios de trabajo del svn, tags, branches y working; en working se encuentran todos los archivos de código y otros de este proyecto.

- **cas.c**, alberga la implementación de las funciones públicas de la librería, `from_*`, las de inicialización y la de petición de versión.
- **cas_int.h**, declara todas las estructuras de datos y funciones internas que serán usadas por la librería.
- **cas.h**, contiene la declaración de las funciones ya nombradas de la API, `from_*`, además de las de inicialización y consulta de versión. También declara las estructuras de datos a usar `struct S5066_CAS_data`.
- **Makefile**, contiene la configuración de compilación. La versión de la capa está definida aquí, y se introduce como macro de compilación.
- **cas.doxyfile**, contiene la configuración para generar el documento doxygen referente a este código.

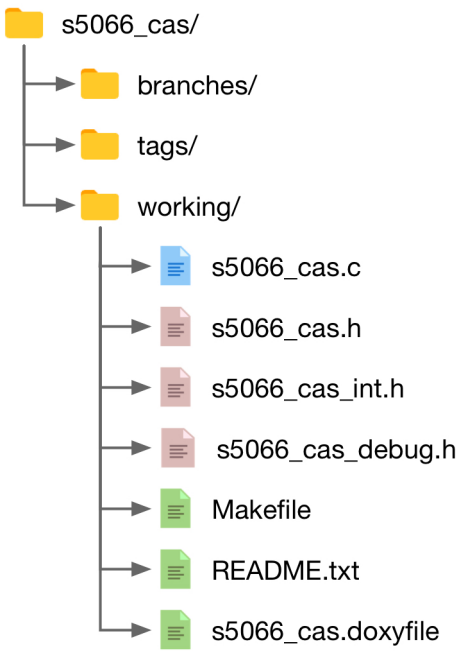


Figura 4.11: Estructura de ficheros en el directorio de trabajo de la capa CAS

4.6.2. Handler

El handler de la capa CAS mantiene una simplicidad acorde con las tareas que tiene que realizar. Podemos destacar la variable `struct S5066_CAS_links_t *link`, que es un array de tamaño `S5066_CAS_MAX_PHY_LINKS`. La definición del handler la podemos ver a continuación:

```
struct S5066_CAS_handler {
    uint32_t node_addr;
    uint8_t phlink_req_max_repeats;
    uint8_t phlink_priority;
    struct S5066_CAS_back_for_more_t back_for_more;
    struct S5066_CAS_links_t *link[S5066_CAS_MAX_PHY_LINKS];
};
```

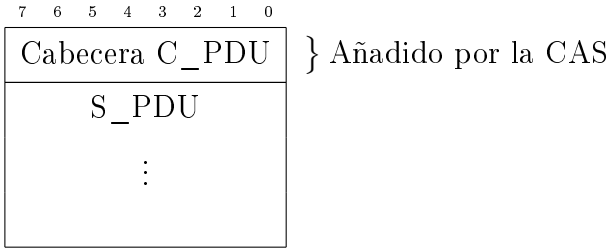
Este handler no tiene que almacenar ninguna PDU, y la función fundamental es mantener el estado de los diferentes *physical links* que se hayan creado con los diferentes nodos remotos. Para ello, existe la variable `struct S5066_CAS_links_t *link` que hemos destacado anteriormente. La gestión de los links se detalla en el Apartado 4.6.4.

4.6.3. Bypassing de primitivas

Recordemos que las estructuras `struct S5066_data` son iguales para las tres capas y esto facilita el proceso de *bypassing* de los datos desde la SIS hasta la DTS. Cuando la capa CAS recibe una llamada a la función `process_from_upper(,)`

y el campo `*data->prim_type` tiene el valor S5066_C_UNIDATA_REQUEST o S5066_C_EXP_UNIDATA_REQUEST.

Para ello, actualiza el valor de `*data->prim_type` a la correspondiente D_Primitiva, dejando los demás campos sin modificar, y añadiendo la cabecera de la C_PDU al buffer `*buf`, como se muestra en el siguiente diagrama:



De la misma manera, hay otras primitivas (tanto el sentido SIS ⇒ CAS como DTS ⇒ CAS) de las que no se toca el buffer `*buf`; sólo se cambia el tipo de primitiva, y se genera la salida para el siguiente paso. Estas primitivas están destinadas a hacer una comunicación interna SIS ⇔ DTS, por tanto, la CAS ha de ser completamente transparete. Un ejemplo de estas primitivas, es la de solicitud (por parte de la SIS) a la DTS que añada un nuevo grupo a su lista de grupos a los que pertenece.

4.6.4. Gestión de los enlaces

La gestión de los enlaces por parte de la capa CAS se hace manteniendo un estado para cada enlace, de manera muy similar a la capa SIS. La Figura 4.12 muestra los estados definidos dentro de la capa CAS para los *soft links*, además de los eventos que generan una transición, y de qué eventos de salida tras una transición.

Recordemos que la capa CAS, inicia un proceso de enlazado tras recibir la solicitud de la capa SIS local. También puede declarar un enlace como *existente* tras recibir una solicitud con origen remoto.

4.6.5. Gestión de timers

Para la capa CAS, la gestión de timers es sencilla, y está únicamente asociada a los eventos de creación y ruptura de enlace, siempre que sea una solicitud iniciada por el nodo local.

Cuando la SIS solicita a la capa CAS la creación o ruptura del enlace, la capa CAS transmite un paquete NO-ARQ al nodo remoto, en concreto a la CAS remota. En el momento de entregar el paquete NO-ARQ a la DTS, la CAS local arranca un temporizador que en el momento de su expiración hará que la creación/ruptura de enlace se considere como no respondida, y ejecute un nuevo intento de comunicación. Esto se repetirá un número de veces definido como S5066_CAS_DFLT_MAX_REPEATS, antes de dar la creación/ruptura como fallida.

En caso de que la ruptura sea fallida, el nodo local considerará unilateralmente el

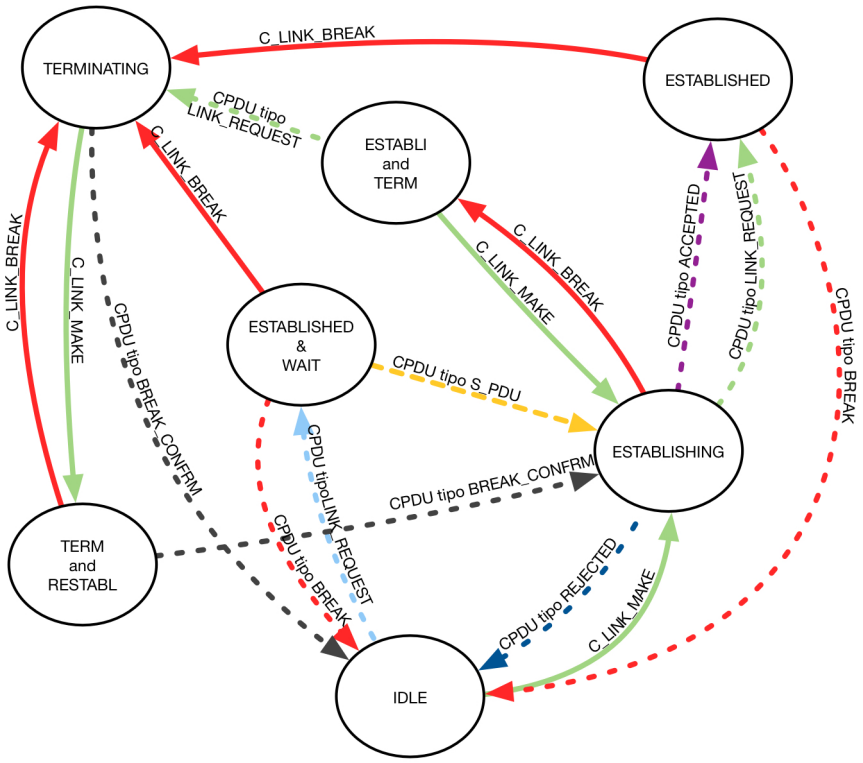


Figura 4.12: Estados de los *physical links* implementados por la CAS, y sus transiciones

enlace como cerrado; la situación de desincronización de los estados no es crítica, pero puede dar lugar a que un nodo que aún considera enlace abierto, transmita una gran cantidad de datos antes de ser informado que el nodo de destino no tiene conocimiento de ese enlace.

4.7. Implementación de la capa DTS

Esta capa es la interfaz con otros nodos a través de HF. La DTS, se encarga de la comunicación con los demás nodos de la subred STANAG 5066 a través de las comunicaciones HF, y es la que está en contacto "directo" con el módem.

Una de las tareas de la DTS es la de fragmentar las C_PDUs en D_PDUs de un tamaño máximo notablemente menor al tamaño máximo admitido para una C_PDU, para la cadena de transmisión, y además ha de reconstruir esta fragmentación en la cadena de recepción.

La cadena de recepción comienza con una de las piezas fundamentales y delicadas de esta capa: la máquina de estados de decodificación en recepción. Esta máquina de estados recorre el array de bytes recibidos desde el módem, buscando la secuencia de sincronismo, y una vez detectada, va recorriendo e interpretando la cabecera, posteriormente calculando el CRC.

Es fundamental una implementación correcta y minuciosa de esta máquina de es-

tados, ya que los datos provenientes del módem pueden contener bytes erróneos, por tanto, mantener un control del inicio de las secuencias detectadas, permite volver por el byte inmediatamente siguiente al de la secuencia en el caso de que no se decodifique una cabecera válida.

La DTS es la responsable de la implementación del protocolo ARQ, manteniendo los estados de transmisión y/o recepción, en función del papel que desempeñe el nodo en concreto. Cada nodo con el que se quiera intercambiar información ha de mantener su propia información de estado ARQ, además de almacenar todos las D_PDUs temporalmente hasta poder componer el C_PDU completo y poder pasarlo a la capa CAS.

Además, ha de tener conocimiento del estado de los *physical links*. Esto contradice el principio de separación entre capas propuesto en la arquitectura OSI [11], pero en caso de que la DTS esté recibiendo mensajes ARQ de un nodo sin conexión, permite ignorar estos paquetes sin modificar su estado de recepción ARQ, facilitando una futura sincronización con el nodo remoto. Además, realiza los descartes de los datos en la propia capa DTS, minimizando los requisitos de computación local.

En el caso de las transmisiones NO-ARQ, la parte de transmisión es responsable de las retransmisiones correspondientes de cada C_PDU. En recepción se ha de almacenar cada D_PDU hasta recibir correctamente todas las correspondientes a una C_PDU, durante un tiempo finito especificado en cada segmento transmitido.

Para el control de acceso al canal, la DTS recibe el estado (TX/RX/IDLE) del módem a través de la función `process_from_modem(...)`. Esto permite implementar mecanismos de contención, en el que, por ejemplo, se inhiba la generación de salida en caso de estar recibiendo datos.

La implementación de la DTS, a la hora de procesar datos de entrada y generar datos de salida, se ha hecho manteniendo una división clara de estos dos procesos. En cualquier llamada `process_from_lower`, `process_from_upper`, `process_from_timer`, `process_from_modem` o `process_from_back_for_more` lo primero que se hace es modificar los datos del handler según corresponda, en función de los datos que se entreguen o de la función que se haya llamado (por ejemplo, `from_timer`), y posteriormente se analiza el handler para determinar si, tras la pertinente modificación, hay datos de salida con destino a la capa CAS o al módem.

A pesar de que el estándar STANAG 5066 limita los modos simultáneos en los que se puede estar trabajando ARQ y NO-ARQ, de manera que hay que iniciar 'sesiones'para cada uno de ellos, la implementación desarrollada elimina dichas limitaciones.

4.7.1. Estructura de ficheros

El directorio de trabajo está compuesto por las tres carpetas ya mencionadas que corresponden al uso del repositorio SVN. Dentro de la carpeta working es donde se encuentran los archivos de código fuente que corresponden a la capa DTS.

La Figura 4.13 muestra los archivos existentes en el directorio working. En la si-

guiente lista describimos un resumen de las funcionalidades implementadas en cada fichero:

- **dts.c**, alberga la implementación de las funciones públicas de la librería, `from_*`, las de inicialización y la de petición de versión. Además, implementa las funciones `send_to_lower` y `send_to_upper`, que son las encargadas de generar los datos de salida hacia la CAS y el módem.
- **dts_int.h**, declara todas las estructuras de datos (como el handler) y funciones internas que serán usadas por la librería.
- **dts.h**, contiene la declaración de las funciones ya nombradas de la API, `from_*`, además de las de inicialización y consulta de versión. También declara las estructuras de datos a usar `struct S5066_DTS_data`.
- **Makefile**, contiene la configuración los procedimientos de compilación. La versión de la capa está definida aquí, y se introduce como macro de compilación.
- **dts.doxyfile**, contiene la configuración para generar el documento doxygen referente a este código.
- **dts_arq.c y .h**, se encuentran las funciones relativas al protocolo ARQ, búsqueda y marcación de segmentos D_PDU, cálculo de la ventana de transmisión, etc...
- **dts_buf.c y .h**, contiene las funciones de control de los buffers para almacenar las C_PDUs. Extrae y devuelve buffers del *pool* de buffers que tiene la DTS, entre otras funcionalidades.
- **dts_crc.c y .h**, organiza las operaciones de CRC para transmisión y recepción.
- **dts_dbg.c y .h**, las funciones de depurado (generación de logs), así como diferentes macros están en este fichero.
- **dts_dec.c y .h**, aloja las funciones de procesado de D_PDU para recepción. Estas funciones son llamadas por la máquina de estado de decodificación en recepción (cuyo código está en dts_rxm.c.)
- **dts_edb.c y .h**, unas funciones avanzadas de generación logs se encuentran en este fichero.
- **dts_enc.c y .h**, las funciones de codificación de D_PDU a partir de parámetros se agruapan aquí.
- **dts_grp.c y .h**, funcionalidades de grupo, que se encargan de la modificación de la estructura correspondiente en el handler, para, por ejemplo, entrar o salir de grupos.
- **dts_rxm.c y .h**, aloja la máquina de estado de decodificación en recepción.
- **dts_stm.c y .h**, define las estructuras `st_mach`, que son las encargadas de guardar el estado ARQ y NO-ARQ con los nodos remotos. Las funciones que modifican estas estructuras también se encuentran en este archivo.

- **dts_tmr.c y .h**, contiene las funciones para la gestión de los timers.
- **dts_txm.c y .h**, diversas funciones de generación de datos de salida de HF se encuentran aquí. Estas funciones son apoyo fundamental de la función `send_to_lower`.

Figura 4.13: Estructura de ficheros en el directorio de trabajo de la capa DTS

4.7.2. Gestión de timers

De manera similar a la capa CAS, la gestión de los timers se hace con un descriptor de fichero (un temporizador en el Sistema Operativo) por cada evento que se quiera controlar.

Se han definido siete tipos de timers, que se detallan en la siguiente lista

- S5066_DTS_TIMER_NON_ARQ_WE y S5066_DTS_TIMER_ENON_ARQ_WE. Son usados para "despachar" C_PDU tipo NO-ARQ que hayan llegado incompletas. Pongamos el caso de una C_PDU que consta de 10 fragmentos y solo se han recibido 8, y además el nodo transmisor ya ha hecho todas las retransmisiones pertinentes, por lo tanto, no se va a recibir más datos de esta C_PDU. Este temporizador previene que estos datos queden ocupando memoria indefinidamente.

  De manera genérica, el nodo receptor arranca (o actualiza) un temporizador asociado a cada C_PDU de la que recibe información válida, y cuando éste expira, entrega la C_PDU a la capa CAS.

- S5066_DTS_TIMER_NON_ARQ_WE_PREVENT y S5066_DTS_TIMER_NON_ARQ_WE_PREVENT. Cuando una C_PDU de tipo NO-ARQ se ha recibido completamente bien, se entrega inmediatamente a la capa CAS, y es posible que el nodo transmisor siga generando repeticiones de esta misma C_PDU. Por tanto, si recibimos información asociada a la C_PDU ya recibida hemos de ignorarla. Se arranca un temporizador de este tipo asociado al campo `c_pdu_id`, ya que el valor del `c_pdu_id` se recicla siguiendo una cuenta con módulo 4096.

- S5066_DTS_TIMER_ARQ_ACK. Este temporizador se arranca en el momento de escribir datos correspondientes a una C_PDU tipo ARQ, y está asociado a cada C_PDU. El propósito del temporizador es controlar un tiempo prudencial en el que se debería haber recibido los ACKs de las D_PDUs transmitidos, y en el momento de expiración se marcará la C_PDU asociada como que necesita ser retransmitida. El ajuste del tiempo de este temporizador se basa en los tiempos estimados una transmisión de ACK.

- S5066_DTS_TIMER_OUTPUT_LOWER. El acceso y la ocupación del canal demostró ser un punto conflictivo desde el momento en el que tres nodos participan de manera activa en las comunicaciones. Este temporizador controla un tiempo 'pseudo-aleatorio'durante el cual el nodo local no puede realizar ninguna transmisión a partir de la finalización de la última recepción (evento RX $\Rightarrow$ IDLE), esto evita que varios nodos colisionen permanentemente a la hora de transmitir datos. Cabe recordar que la DTS recibe el estado del módem, lo que permite implementar este mecanismo.

- S5066_DTS_TIMER_WATCHDOG_BLOCK_OUTPUT. Para prevenir algún error en el que la DTS se quede con la transmisión inhibida permanentemente, se implementa este timer que hace la función de watchdog. Recordemos que el tiempo máximo de transmisión de un nodo según la norma STANAG 5066 es de dos minutos, así que se programa a un valor suficientemente alto, diez minutos, como para poder considerar un error si la DTS permanece bloqueada durante todo ese tiempo.

La recepción de los eventos de temporizador, se realiza en la funcion `from_timer(...)`, como en las otras capas.

4.7.3. Máquina de estado de decodificación en recepción

Desde el punto de vista de la capa MNGT, el punto de entrada a la cadena de recepción es la función `process_from_lower(...)`. Para la DTS, el primer paso en la cadena de recepción es hacer la decodificación de las D_PDUs que se encuentren a lo largo del buffer de recepción. La Figura 4.14 nos da una idea de cómo se ven afectados los datos desde el punto de vista de la DTS tras una transmisión por HF.

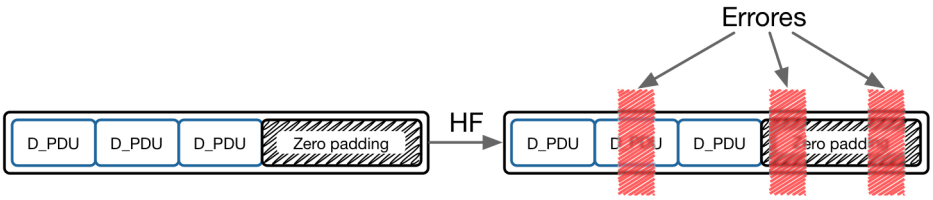


Figura 4.14: Introducción de errores en la información (D_PDUs) tras el paso por el canal de HF

La función encargada de hacer la decodificación de los datos recibidos es `process_rx_buffer(...)`. Esta función implementa una máquina de siete estados, todos relacionados con los diferentes *pasos* a dar durante una recepción para decodificar las D_PDUs. El primer paso es detectar la secuencia de sincronismo descrita en [10, Anexo C.3.2]. Posteriormente decodificar la cabecera y comprobar el CRC, y en caso de que la cabecera esté correcta y que exista *payload* (hay cabeceras que no tienen *payload*) proceder a copiar el contenido del *payload* y verificar su CRC. Estos pasos se muestran en la Figura 4.15

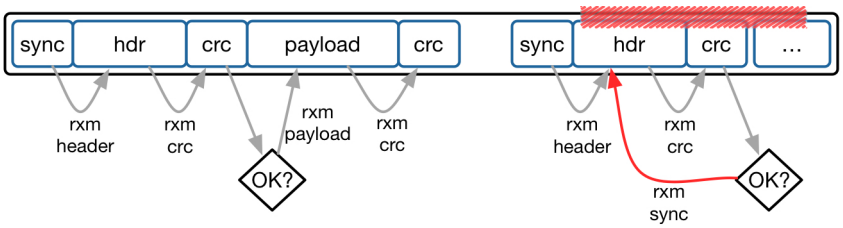


Figura 4.15: Proceso del buffer de recepción por la máquina de estado de decodificación

Uno de los motivos de hacer la decodificación de esta manera, es poder mantener el estado y los datos decodificados en caso que una D_PDU pueda estar dividida en dos buffers de recepción por parte del módem. Este caso se ejemplifica en la Figura 4.16.

Los estados que implementa esta máquina se codifican la siguiente enumeración

```
/*!
States for the D_PDU RX state machine
```

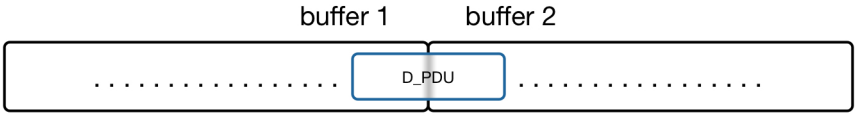


Figura 4.16: Una D_PDU puede estar repartida en dos buffers de recepción consecutivos

```
*/
enum {
    S5066_DTS_RX_IDLE = 1, /*!< Waiting for SYNC_BYTES to be received */
    S5066_DTS_RX_SYNC,     /*!< First SYNC_BYTE received */
    S5066_DTS_RX_HEADER,   /*!< SYNC sequence received */
    S5066_DTS_RX_HDRCRC,   /*!< Rcvng and processing CRC on header */
    S5066_DTS_RX_SKIPDATA, /*!< Skip data field */
    S5066_DTS_RX_DATA,     /*!< Processing data */
    S5066_DTS_RX_DATACRC   /*!< Data received (and saved) */
};
```

- RX_IDLE. La maquina de estado recorrerá la lista de bytes buscando el primero de los dos bytes de sincronismo. En el momento que lo encuentre, pasará al estado RX_SYNC.
- RX_SYNC: En este estado comprueba que el byte leído es el segundo byte de sincronismo. En caso de que sea correcto, pasará a RX_HEADER, en caso contrario, volverá a RX_IDLE.
- RX_HEADER: Este es el estado de decodificación de la cabecera. A medida que recorre los bytes, irá escribiendo los diferentes campos leídos en la estructura `handler->d_pdu`. Además, ha de tener en cuenta los bytes que ha leído previamente, porque los campos presentes y el tamaño de la cabecera depende del tipo de D_DPU enviada, y esto se codifica en el *nibble* más significativo del primer byte tras la secuencia de sincronismo.
- RX_HDRCRC: Una vez procesada la cabecera, se pasa a este estado, en el que se recorren y guardan los dos bytes correspondientes al CRC de cabecera. Cuando se ha guardado el segundo byte se comprueba el CRC. En caso de que el CRC sea correcto, se dará por procesada la cabecera y se procesará el "*payload*" si procediera. Si el CRC fuera incorrecto, se volvería a analizar el primer byte que fue procesado como cabecera.
- RX_SKIPDATA: Si el *payload* proveniente en la D_PDU ya ha sido recibido con anterioridad (una repetición NO-ARQ o una retransmisión ARQ por no haber recibido los ACKs, por ejemplo), se entrará en este estado, que saltará el índice de byte analizado en el buffer de recepción, con lo que se optimiza el análisis.

  La existencia de este estado puede parecer innecesaria, ya que lo más evidente es que tras el estado RX_HDRCRC se salte directamente a RX_IDLE incrementando

el índice de análisis, pero esto provocaría comportamientos no deseados en caso que este "*payload*" esté dividido en dos buffers diferentes de recepción.

- RX_DATA: En este estado se copian los bytes correspondientes del *payload* de la D_PDU en el buffer C_PDU que haya sido determinado en el procesado de la cabecera. Si todos los datos correspondientes a una D_PDU no están presentes en lo que resta de buffer de recepción, se mantendrá el estado hasta la siguiente llamada de la función `process_rx_buffer`, habiendo decrementado los bytes que sí pudieron ser copiados.

- RX_DATACRC: Una vez copiado el *payload*, se procede a calcular el CRC de la misma manera que con la cabecera. En el caso que los datos sean incorrectos y la transmisión sea NO-ARQ, se marcará el segmento como recibido incorrectamente, y si la transmisión es ARQ será descartado.

4.7.4. Array de C_PDUs

Para almacenar tanto las C_PDUs "a transmitir" como los que están en proceso de recepción, se usa una array de C_PDUs que está declarado como un campo de datos del handler. Estos elementos son de tipo **struct** `S5066_DTS_c_pdu`, y el array se define como **struct** `S5066_DTS_c_pdu *c_pdu_buf[S5066_DTS_C_PDU_BUFFERS]`. La declaración de esta estructura se muestra a continuación:

```
/*!
C_PDU structure. Used for both RX and TX
*/
struct S5066_DTS_c_pdu {
    enum C_PDU_BUF_STATUS status;
    int nmach; /*!< associated state machine */
    /* necessary information from/for upper layers */
    uint16_t s_pdu_id;
    enum service_type_t service_type;
    uint8_t transmission_mode;
    uint8_t delivery_confirm;
    uint8_t delivery_order;
    uint32_t time_to_die;
    uint32_t src_address;
    uint32_t dst_address;
    uint8_t priority;   /*!< 4 bits */
    uint8_t group_address;
    /* C_PDU variables for DTS  */
    uint8_t max_rtx;
    uint8_t rtx_left;
    uint16_t c_pdu_rx_win;
    uint16_t seg_size;
    uint16_t last_seg_size;
    uint16_t nsegments;
    uint16_t current_seg;
```

```
    uint16_t bytes_left;
    uint64_t c_pdu_id;
    enum seg_info_t seg_info[S5066_DTS_MAX_NSEGMENTS];
    unsigned int total_d_pdus;  /*!< # of rcvd D_PDUs for curr C_PDU */
    /* buffer variables */
    uint8_t buf[S5066_DTS_C_PDU_BUFFER_SIZE];
    uint32_t buflen;
    uint32_t bufdataini;
    uint32_t bufdatalen;
};
```

Esta estructura tiene un propósito ambivalente, sirviendo tanto para transmisión como para recepción. Se puede destacar el array `uint8_t buf[S5066_DTS_C_PDU_BUFFER_SIZE]`, cuyo propósito es almacenar el *payload* asociado al C_PDU.

Entre los campos destinados a la transmisión se encuentran, por ejemplo, `priority`, o `rtx_left`, que se usa únicamente para llevar el control de las retransmisiones pendientes si se ha de transmitir con el modo NO-ARQ.

Una limitación inherente a este array es su número de elementos, `S5066_DTS_C_PDU_BUFFERS`. Cuando el número de elementos libres en el array cae por debajo de `S5066_DTS_C_PDU_MIN_SAFE_BUFFERS`, la DTS lanza una C_Primitiva con intención de informar a la SIS que ha de que parar el flujo por parte de los clientes.

Cuando los elementos libres vuelven a estar por encima de `S5066_DTS_C_PDU_MIN_RECOVERY_BUFFERS`, se lanza una C_Primitiva liberando el flujo por parte de la DTS.

La existencia de dos valores diferentes para parar y reactivar el flujo, permite la configuración con un ciclo de histéresis en el proceso. Este proceso se muestra en la Figura 4.17.

Array de segmentos

Otro campo destacable es el array `enum seg_info_t seg_info[S5066_DTS_MAX_NSEGMENTS]`, que almacena el estado de cada uno de los segmentos, tanto para el caso de transmisión como de recepción.

Los posibles valores asociados a cada uno de los elementos del array son del tipo `enum seg_info_t`, que se define a continuación.

```
enum seg_info_t {
    S5066_DTS_NONRCVD = 0,
    S5066_DTS_RCVDERR,
    S5066_DTS_RCVDOK,
    S5066_DTS_NO_TX,
    S5066_DTS_TX_NOT_ACK,
```

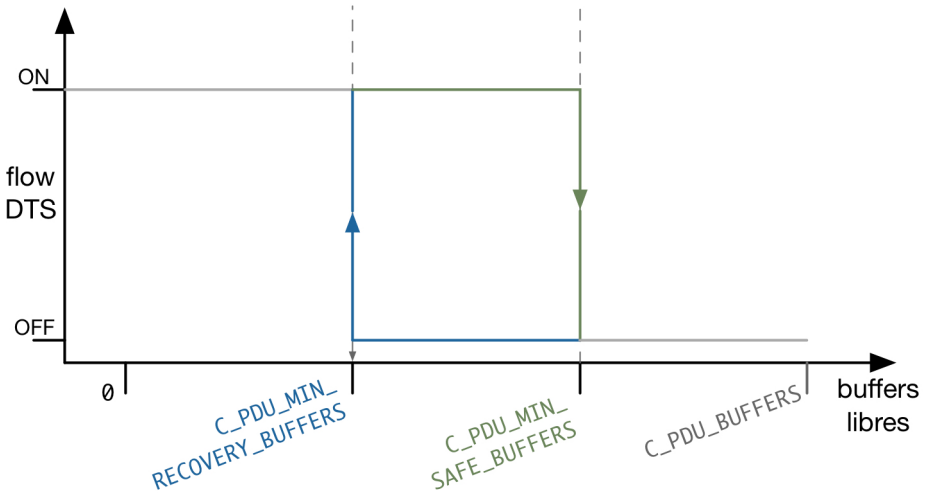


Figura 4.17: Condiciones de activación/desactivación por parte de la DTS del flujo de datos de los clientes

```
    S5066_DTS_TX_ACK
};
```

- **Transmisión**. Sólo se usará para el modo de transmisión ARQ. El estado de cada uno de los segmentos se usará para marcar los segmentos que hayan sido confirmados y que tengan FSN (*Frame Sequence Number*) superior al LWE (*Lower Window Edge*), esto es: los que hayan sido confirmados mediante ACKs selectivos. Los segmentos que estén marcados con el estado `S5066_DTS_TX_ACK` no serán retransmitidos.

  Cuando **todos** los segmentos de una C_PDU están marcados como `S5066_DTS_TX_ACK`, y también se cumpla la condición que tenga **confirmación de nodo**, se procederá a generar la confirmación. Este proceso se realiza en la función `send_to_upper(...)`.

- **Recepción**. El uso de este campo en recepción se hace sólo en el modo de transmisión NO-ARQ. Este array sirve para marcar la "calidad" de recepción del segmento: no recibido, recibido con errores, o recibido correctamente. Estos tres valores se codifican como `S5066_DTS_NONRCVD`, `S5066_DTS_RCVDERR` y `S5066_DTS_RCVDOK`, respectivamente.

4.7.5. Array de nodos remotos, o *st_mach*

Para llevar el control de las comunicaciones con cada uno de los nodos remotos, se diseñó la estructura `struct stmach_t`, cuyo propósito es albergar todas las variables de estado asociadas tanto a las transmisiones ARQ como a las NO-ARQ.

Cada *st_mach* en uso tiene asociada una dirección única de HF correspondiente a un nodo remoto. Además, en su estructura existen múltiples variables con el fin de

coordinar las transmisiones y recepciones con dicho nodo.

La estructura declarada para las st_mach se muestra a continuación:

```
/*
DTS connection state machine
*/
struct S5066_DTS_stmach_t {
   uint32_t remote_addr;
   enum S5066_DTS_stmach_connection_t conn;
   /* NON-ARQ variables */
   uint16_t c_pdu_id_non_arq_tx;
   uint16_t c_pdu_id_non_arq_rx;
   uint16_t c_pdu_id_exp_non_arq_tx;
   uint16_t c_pdu_id_exp_non_arq_rx;
   int c_pdu_nonarq_curr_nbuffer;
   /* ARQ variables */
   uint16_t c_pdu_id_exp_arq_tx;
   uint16_t c_pdu_id_exp_arq_rx;
   uint16_t c_pdu_id_arq_rx;
   uint64_t c_pdu_id_arq_tx;
   int c_pdu_arq_curr_nbuffer;
   int keep_alive_send;
   uint8_t rx_lwe, rx_uwe, tx_lwe, tx_uwe;
   uint8_t exp_rx_lwe, exp_rx_uwe, exp_tx_lwe, exp_tx_uwe;
   uint8_t acks_rcvd;
   uint8_t rtx_needed;
   uint8_t arq_tx_times_wo_acks;
   uint8_t pending_tx_ack;
   uint8_t selected_to_tx_arq;

   dllist rx_sequential_lst;
   dllist rx_seg_lst;
   dllist node_pool_lst;
   dllist exp_rx_sequential_lst;
   dllist exp_rx_seg_lst;
   dllist exp_node_pool_lst;
   struct S5066_DTS_tx_seq_arq_t tx_seq[S5066_DTS_SEQ_NUM_SIZE];
   struct S5066_DTS_rx_buf_arq_t rx_buf[S5066_DTS_SEQ_NUM_SIZE];
   struct S5066_DTS_tx_seq_arq_t tx_seq_exp[S5066_DTS_SEQ_NUM_SIZE];
   struct S5066_DTS_rx_buf_arq_t rx_buf_exp[S5066_DTS_SEQ_NUM_SIZE];

   /* Management status */
   uint16_t mngt_tx_frm_id;
   uint16_t mngt_rx_frm_id;
   uint8_t mngt_status;

   /* "Warning D-PDU" management */
   uint8_t tx_warning_reason;
```

```
    uint8_t rx_warning_reason;
    uint8_t tx_warning_type;
    uint8_t rx_warning_type;
};
```

Como se puede ver, en estas estructuras se alojan las variables de estado de las ventanas de transmisión, tanto en transmisión (`tx_lwe` y `tx_uwe`), como en recepción (`rx_lwe`, `rx_uwe`). También se encuentran las listas doblemente enlazadas que tienen función de dar soporte al ordenado de las D_PDUs tipo ARQ recibidas de manera desordenada, en `rx_buf`.

Otra variable, como `mngt_status`, se usa para marcar el *modo management* en un nodo. Esto se describe en el Apartado 4.7.8.

También se mantiene el control de las C_PDUs que se han transmitido y del valor de `c_pdu_id` que ha ido asignando. Igualmente se realiza el proceso equivalente en recepción, almacenando los diferentes `c_pdu_id` que han sido recibidos, para evitar enviar a la CAS varias veces una misma C_PDU en sus sucesivas retransmisiones.

4.7.6. Función de generación de datos `send_to_upper`

Para generar los datos destinados a la capa CAS, todas las funciones `process_from_...` hacen en algún punto del código una llamada a la función `send_to_upper`, de manera que se mantiene centralizado el análisis del handler independientemente de la función desde que se llame.

Esta función busca cinco tipos de eventos que ha de transmitir hacia la capa CAS, aunque no necesariamente el destino es para CAS, sino para SIS o incluso el Cliente. Los eventos que busca, por orden de prioridad son:

1. **C_PDU** completamente recibida. Si una C_PDU ha sido completamente recibida, sea ARQ o NO-ARQ, se entregará a la CAS y liberará el buffer correspondiente a dichos datos, devolviéndolo al *pool* del handler.

2. **Keep Alive de nodo**. Este dato se transmite con el fin de informar a la SIS, que a pesar que no se hayan podido recibir C_PDUs completas, hay actividad con un nodo en concreto. El propósito de esta información es evitar largos tiempos de inactivdad en la SIS (que es la responsable de terminar los enlaces por inactividad) por el hecho de recibir una C_PDU muy fragmentada. En la Figura 4.18 apreciamos la generación de estos eventos.

3. **Warnings RX**. Si la DTS ha recibido un *warning* de un nodo remoto, se notificará a la CAS para que tome las acciones pertinentes. Los *warning* normalmente están asociados a la recepción de datos ARQ sin tener conocimiento de enlace creado previamente.

4. **Warnings TX**. Si la DTS ha transmitido un *warning*, igualmente se notificará a la CAS para que tome las acciones pertinentes.

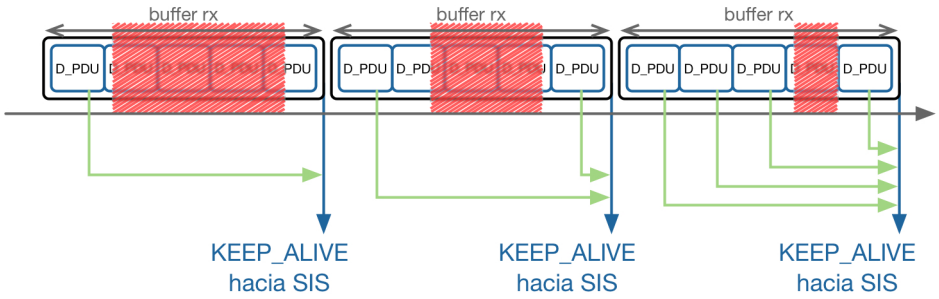


Figura 4.18: Generación de primitivas "keep alive" hacia la capa SIS, informando de actividad con un nodo remoto

5. **Flow on/off**. Si los buffers libres para C_PDUs se están agotando, ha de informar a la SIS para que cese el flujo, al menos, por parte de los clientes.

Esta función, además de generar la primitiva asociada a un evento para ser entregada a la CAS, comprueba si existe algún otro evento que no haya podido ser codificado por la limitación de "una primitiva por estructura **struct** `S5066_DTS_data`". En ese caso, además de generar la salida `OUTPUT_UPPER`, generará la salida `OUTPUT_BACK_FOR_MORE`, para que la capa MNGT llame a la función `back_for_more`, donde se generarán las salidas pendientes.

4.7.7. Función de generación de datos `send_to_lower`

La generación de datos con dirección hacia el módem se centralizan en esta función. Tiene la característica que sólo generará salida `OUTPUT_LOWER` y nunca `OUTPUT_BACK_FOR_MORE`. Esto se debe a que, como excepción, mútliples primitivas D_PDUs pueden ser codificadas en una estructura **struct** `S5066_DTS_data`. Una vez esta función genera los datos a ser transmitidos en el momento de ser llamada, y no tiene sentido volver a generar datos de salida hacia el módem habiendo tenido la oportunidad de generarlos previamente.

Puede darse el caso en el que se limite la generación de D_PDUs, pero siempre impuestos por el estándar. Esto hace que se considere el buffer de transmisión como lleno, hasta que la DTS reciba un evento de fin de transmisión de los datos entregados al módem.

La secuencia de búsqueda de datos a transmitir se hace priorizando unos servicios frente a otros, y dentro de cada servicio, aplicando un criterio de priorización particular. El orden de búsqueda según los servicios es:

1. **RESET/RESYNC** (D_PDU tipo 3). Utilizados para resincronizar estados ARQ. Es la más prioritaria de todas, y en caso de escribirse algún D_PDU de este tipo se terminará la búsqueda, comenzando la transmisión.
2. **MANAGEMENT** (D_PDU tipo 6). Este tipo de D_PDU se usan para transmitir comandos EOW, que es un campo con el que transmitir comandos de con-

figuración entre nodos. Además, también se puede añadir un campo adicional de *extended message*.

3. **EXPEDITED NO-ARQ** (D_PDU tipo 8). Son los fragmentos de las C_PDUs con modo de transmisión NO-ARQ y tipo *expedited*. Las comunicaciones CAS ⇔ CAS para la gestión de *soft links* usan este servicio.
4. **EXPEDITED ACK-ONLY** (D_PDU tipo 5).
5. **ACK-ONLY** (D_PDU tipo 1).
6. **EXPEDITED ARQ** (D_PDU tipo 4).
7. **ARQ** (D_PDU tipo 0).
8. **NO-ARQ** (D_PDU tipo 7).
9. **WARNING** (D_PDU tipo 15).

Como podemos observar, no hemos implementado la primitiva **DATA-ACK**, ya que es equivalente a una primitiva **ACK** seguida de una primitiva **DATA**, e implementarla complica innecesariamente la lógica de transmisión.

Adicionalmente, a la hora de codificar la D_PDU se ha de tener en cuenta tamaño total del buffer a transmitir, para poder escribir el campo EOT (*End Of Transmission*) en las diferentes D_PDU, decrementando correctamente el EOT de manera sucesiva hasta llegar a 0,5 segundos, en el último segmento.

4.7.8. Modo management

Cuando algún nodo remoto al que se está intentando transmitir datos ARQ no está respondiendo, tras un número determinado de transmisiones sin respuesta, definido en la macro `S5066_DTS_MNGT_RETRIES_TO_NOT_RESPNDG`, se marca dicho nodo con el flag `S5066_DTS_MNGT_NOT_RSPDNG`.

Lo que provoca este flag, es que se dejen de generar datos ARQ destinados a ese nodo, permitiendo usar los dos minutos de transmisión para otros nodos que sí puedan estar activos.

Como mecanismo para recuperar las comunicaciones con dicho nodo, se generará una secuencia de D_PDUs de MANAGEMENT (tipo 6). Si el nodo remoto recibe esta D_PDU, responderá con un ACK del último paquete ARQ recibido. Si el nodo local recibe dicho ACK, reestablecerá el envío de los datos ARQ.

Uno de los beneficios de esta técnica, además de permitir priorizar la transferencia de información con nodos con los que sí hay actividad, es la de prevenir descoordinaciones de transmisiones en escenarios con dos únicos nodos, en el que ambos intentan transmitir una gran cantidad de información (ocupando los dos minutos que permite el protocolo). Esta situación se ve en la Figura 4.19.

La menor proporción de tiempo dedicado a transmisión respecto al tiempo de espera por los ACKs, añadiendo el tiempo aleatorio de acceso al canal, hace que sea más sencillo que ambos nodos se vuelvan a sincronizar en cuanto a las transmisiones, y actúen sin colisiones.

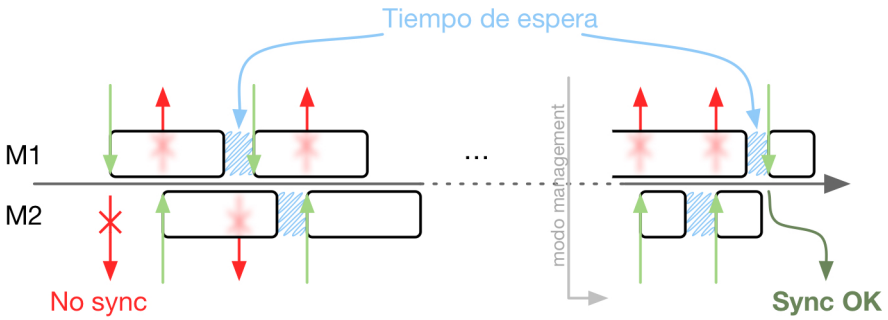


Figura 4.19: Evolución temporal de dos sistemas transmitiendo de manera simultánea, y cómo el *modo management* ayuda a re-sincronisar los nodos

4.7.9. Segmentación en transmisión

Tanto los datos ARQ como los NO-ARQ son encapsulados en D_PDUs. Si la C_PDU tiene un tamaño mayor que el *payload* permitido por cada D_PDU, las C_PDUs son segmentadas, y cada uno de estos segmentos, transmitidos en una D_PDU.

El tamaño de segmentación está definido en `handler->seg_size`, lo que significa que puede cambiar en tiempo de ejecución. Sin embargo, una C_PDU no puede tener segmentos de diferentes tamaños, a excepción del último.

Cuando la DTS recibe una nueva C_PDU proveniente de la capa CAS, la guarda en uno de los buffers de `handler->c_pdu_buf`, y asigna el `seg_size` vigente en el handler. A partir de los datos guardados en esta estructura, se irán escribiendo, cuando corresponda, los diferentes segmentos en el buffer de transmisión.

Existen dos funciones principales para la codificación y escritura de los segmentos en el buffer de transmisión: para ARQ está `d_pdu_build_arq`, y para el modo NO-ARQ existe `d_pdu_build_nonarq`.

4.7.10. La ventana circular

La técnica de ventana circular, o ventana deslizante, es ampliamente utilizada [14] en protocolos ARQ con petición de repeticiones selectivas.

En el caso del STANAG 5066, se define la ventana circular de transmisión como un array circular de 256 elementos, en la cual sólo puede haber 128 elementos consecutivos a partir de el primero sin confirmar recepción. A medida que el nodo receptor va

confirmando los elementos iniciales, se añaden más elementos a la ventana existente de transmisión.

El comportamiento de la ventana está definido por el estándar el el punto [10, Anexo C.6.2]. En él se define que el FSN es un número que se incrementa con "módulo 256". Además introduce los elementos de control LWE y UWE (*Upper Window Edge*)).

En el punto [10, Anexo C.6.2] se define además una serie de propiedades que ha de cumplir la ventana de transmisión, entre las cuales destacamos las siguientes:

1. El LWE indica la D_PDU transmitida, con el menor FSN y que no haya sido confirmada aún, dentro de la ventana de transmisión vigente.
2. El UWE indica la D_PDU transmitida, con el mayor FSN dentro de la ventana de transmisión vigente (haya sido confirmada o no).
3. El tamaño de la ventana de transmisión viene definido por la ecuación $UWE - (LWE - 1)$, tomando la operación "$-$" como operador aritmético modular, con módulo 256.
4. Una ventana de transmsión vacía se representa con:
   - $LWE = 0$
   - $UWE = 255$

   Por lo tanto, usando la ecuación anterior en una aritmética "modulo 256", se obtiene:

   $\text{Win\_sz} = UWE - (LWE - 1)$

   $\text{Win\_sz} = 255 - (0 - 1)$

   $\text{Win\_sz} = 255 - (255)$

   $\text{Win\_sz} = 0$
5. El tamaño máximo de la ventana de transmisión es 128.

Se puede destacar que el límite de 128 elementos en la ventana de transmisión es fundamental para poder operar con aritmética "módulo 256". El algoritmo desarrollado utiliza esta limitación para poder ordenar los segmentos según su FSN cuando la secuencia de FSN traspasa la frontera $255 \rightarrow 0$, en la que el segmento con $FSN = 0$ es posterior al segmento con $FSN = 255$. En este caso, la ecuación $FSN(255) < FSN(0)$.

El siguiente extracto de código muestra cómo se implementa la inserción ordenada de los segmentos recibidos, siendo `node_fsn` el FSN del nodo recibido, y `comp_fsn` el FSN del elemento con el que se está comparando para ser insertado anterior o posteriormente a él. Esto se amplía en el Apartado 4.7.11

```
if (node_fsn > comp_fsn) {
   if ( (node_fsn - comp_fsn) <= S5066_DTS_MAX_ALLOWABLE_WIN_SZ ) {
      dl_insert_after(node, comp, list);
      return NULL; /* success */
```

```
    } else if (comp->prev == NULL) {
        dl_prepend(node, list);
        return NULL; /* success */
    } else {
        comp = comp->prev; /* move to previous node */
    }
} else if (comp_fsn > node_fsn) {
    if ( (comp_fsn - node_fsn) > S5066_DTS_MAX_ALLOWABLE_WIN_SZ ) {
        dl_insert_after(node, comp, list);
        return NULL; /* success */
    } else if (comp->prev == NULL) {
        dl_prepend(node, list);
        return NULL; /* success */
    } else {
        comp = comp->prev; /* move to previous node */
    }
}
```

Además de la ventana deslizante se implementa un mecanismo de ACKs selectivos, mediante el cual el nodo receptor no sólo informa de cúal ha sido el FSN del último segmento correcto de manera consecutiva (para poder avanzar la ventana), sino que informa de los segmentos intercalados que se han recibido correctamente.

Esta información no permite avanzar el LWE y por tanto no inserta elementos nuevos, pero en caso que muchos elementos intermedios hayan sido confirmados correctamente no se incluirán en la nueva transmisión, generando un ahorro de tráfico y acelerando la retransmisión de los segmentos que no han sido confirmados.

La Figura 4.20 describe cómo evoluciona la ventana de transmisión en función de la actualización del FSN o de los ACKs selectivos. Se observa como en la *iteración 1* no se desplaza el LWE, sin embargo se transmiten menos segmentos debido a los ACKs selectivos. En la *iteración 2* se avanza el LWE cinco posiciones, permitiendo que se inserten los segmentos $143 \rightarrow 147$ a la ventana de transmisión.

4.7.11. Recomposición en recepción

Para la recomposición de los datos recibidos, las técnicas usadas para ARQ y para NO-ARQ son diferentes.

NO-ARQ

El primer paso en la recepción de segmentos NO-ARQ, es buscar el buffer de C_PDU con el mismo `c_pdu_id`, procedente del mismo nodo remoto, y si no se ha podido encontrar, se asigna uno nuevo, ya que esto supone que es el primer segmento recibido (no necesariamente el segmento inicial) correspondiente al `c_pdu_id` actual.

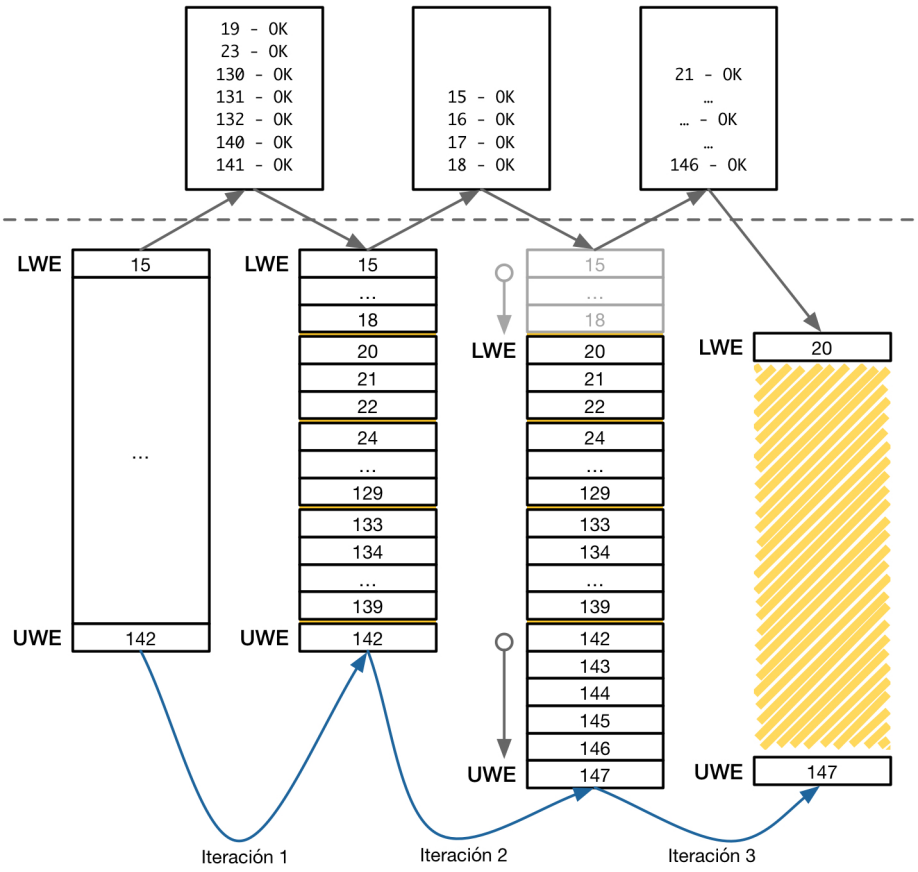


Figura 4.20: Ventana de transmisión, a medida que se van confirmando segmentos con ACKs selectivos

Posteriormente, en caso de que exista una recepción previa del segmento analizado, se ha de comprobar el estado de la recepción. Hay tres estados posibles: no recibido (`S5066_DTS_NONRCVD`), recibido erróneamente (CRC incorrecto - `S5066_DTS_RCVDERR`) y recibido correctamente (CRC correcto - `S5066_DTS_RCVDOK`) . En caso que el estado del segmento actual sea igual que el anterior, no se actualizará, y en caso que sea mejor (`S5066_DTS_RCVDOK` mejor que `S5066_DTS_RCVDERR` mejor que `S5066_DTS_NONRCVD`) se reemplazará el contenido previo.

Si todos los segmentos de la C_PDU se han marcados como `S5066_DTS_RCVDOK`, se considerará como completamente recibido. Esta labor es realizada dentro de la función `send_to_upper`, descrita en el Apartado 4.7.6.

ARQ

En la recepción de segmentos ARQ no se escriben los datos directamente en el buffer de datos del C_PUD correspondiente, sino que en cada segmento recibido (incluida la información extra de la D_PDU) se escribe en un elemento **struct** `S5066_DTS_rx_buf_arq_t`, que será insertado por orden en una lista doblemente enlazada de elementos **struct** `S5066_DTS_rx_buf_arq_t`. El orden vendrá determinado por el campo `Transmission Frame Sequence Number` (FSN) de cada D_PDU.

La inserción ordenada en función del FSN se hace teniendo en cuenta que la ventana de transmisión es de 256 elementos, y nunca se podrá haber transmitido más de 128 elementos simultáneos. Estas limitaciones permite operar con aritmética "módulo 256", en la cual se cumple que 0 es posterior a 255.

El algoritmo de inserción tiene en cuenta si la diferencia entre los FSN comparados es mayor a 128. En caso de que así sea, se asume que estamos analizando FSN que están cerca de la frontera $255 \rightarrow 0$, y corregirá el FSN menor, abandonando momentáneamente la aritmética "módulo 256" para incrementar precisamente en 256 su valor, colocándolo necesariamente por encima del valor más alto posible de FSN.

Para la generación de C_PDUs completas a partir de los fragmentos ordenados, se han de tener en cuenta los bits de `c_pdu_start` y `c_pdu_end`, existentes en los D_PDUs.

Capítulo 5

Pruebas y resultados

Para la verificación del sistema hemos realizado una serie de pruebas controladas, en las que se han utilizado diferentes configuraciones de comunicaciones, tanto con comunicación directa entre módems, como a través de un canal HF.

Además, el Sistema HFDVL ha sido utilizado para diferentes pruebas en entornos reales en colaboración con diferentes cuerpos de las Fuerzas Armadas.

Las pruebas realizadas se enfocan en comprobar los siguientes puntos:

1. Verificar la creación y ruptura de enlaces ARQ.
2. Comprobar la cadena de transmisión y recepción de paquetes ARQ.
3. Comprobar la cadena de transmisión y recepción de paquetes NO-ARQ punto a punto.
4. Verificación del funcionamiento de los mensajes a grupos.
5. Confirmar la generación de primitivas `FLOW_ON` y `FLOW_OFF`.
6. Medidas de latencia.
7. Medidas de rendimiento.

Para todas las pruebas, excepto las pruebas de larga duración en laboratorio, se usó la configuración del módem más robusta (y más lenta), que es 4-QAM, con LDPC 4/5, y entrelazado largo (Ver Sección 2.2).

5.1. Pruebas de laboratorio

A la hora de iniciar las verificaciones del servidor STANAG 5066, se realizaron pruebas con tres Sistemas HFDVL conectados de manera directa a través del audio, que equivaldría a una comunicación radio *perfecta*, sin pérdidas, distorsiones, ni efectos derivados de una transmisión HF.

Esto nos permitió realizar todas las pruebas programadas en un entorno ideal, probando el funcionamiento correcto de las diferentes lógicas implementadas: creaciones de enlaces, mensajes ARQ y NO-ARQ, confirmación de mensajes entregados, mensajes a grupos, etc...

Para realizar estas pruebas, usamos una herramienta desarrollada con tal propósito, llamada *Subnet Management Client*, que permite usar los servicios básicos provistos por el servidor STANAG 5066.

La topología de pruebas usada se describe en la Figura 5.1.

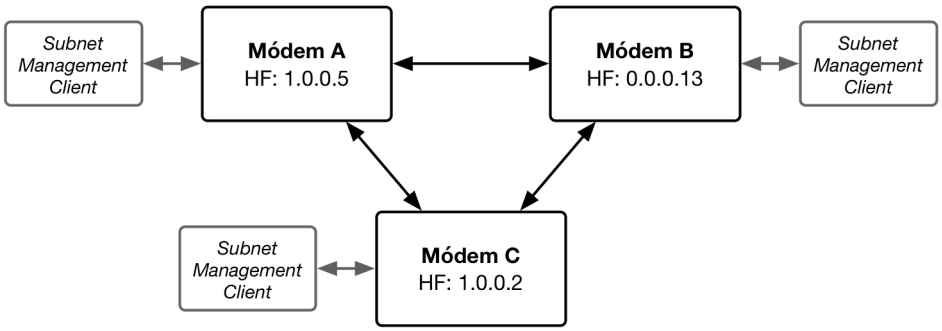


Figura 5.1: Topología usada para hacer las pruebas de laboratorio

5.1.1. La aplicación Subnet Management Client

La principal herramienta de pruebas usada ha sido una aplicación desarrollada en el contexto del Proyecto Coincidente [15]. Esta herramienta se llama *Subnet Management Client*, y se ciñe a la definición del cliente con el mismo nombre, descrito en el Anexo F del estándar STANAG 5066 [10, Anexo F.1].

Con esta herramienta se puede hacer usos de los servicios básicos de transmisión provistos por el servidor STANAG 5066. Estos servicios incluyen transmisiones ARQ y sus confirmaciones (tanto a nivel de cliente, como a nivel de nodo), NO-ARQ, así como las notificaciones de `FLOW ON` y `FLOW OFF`.

Su interfaz es en línea de comandos, por ser un entorno habitual para desarrolladores, permitiendo además concentrar el esfuerzo en la implementación funcional. Además de poder ejecutarla en equipos Linux en los que nos encontremos trabajando, la interfaz en línea de comandos nos permite ejecutar la aplicación a través de una sesión SSH, sin necesidad de habilitar la conexión del servidor gráfico.

La interfaz, que se muestra en la Figura 5.2, se divide en tres bloques: el de mensajes, situado en el área 1, el de información que se encuentra en el área 2, y el de acciones alojado en el área 3.

Área 1 Aquí se muestra un histórico de los mensajes intercambiados con nodos remotos, además de otros eventos por parte del servidor, como enlaces creados/rotos o activación y desactivación de flujo de datos.

Área 2 En esta área se muestra información relativa al estado de la conexión con el servidor STANAG 5066, así como el estado del mismo, como puede ser el estado del flujo de datos, nodos con los que existen enlaces creados, o los grupos NO-ARQ a los que pertenece el nodo local.

Área 3 Las posibles acciones que podemos realizar se muestran aquí, como enviar mensajes, o entrar y salir de grupos NO-ARQ.

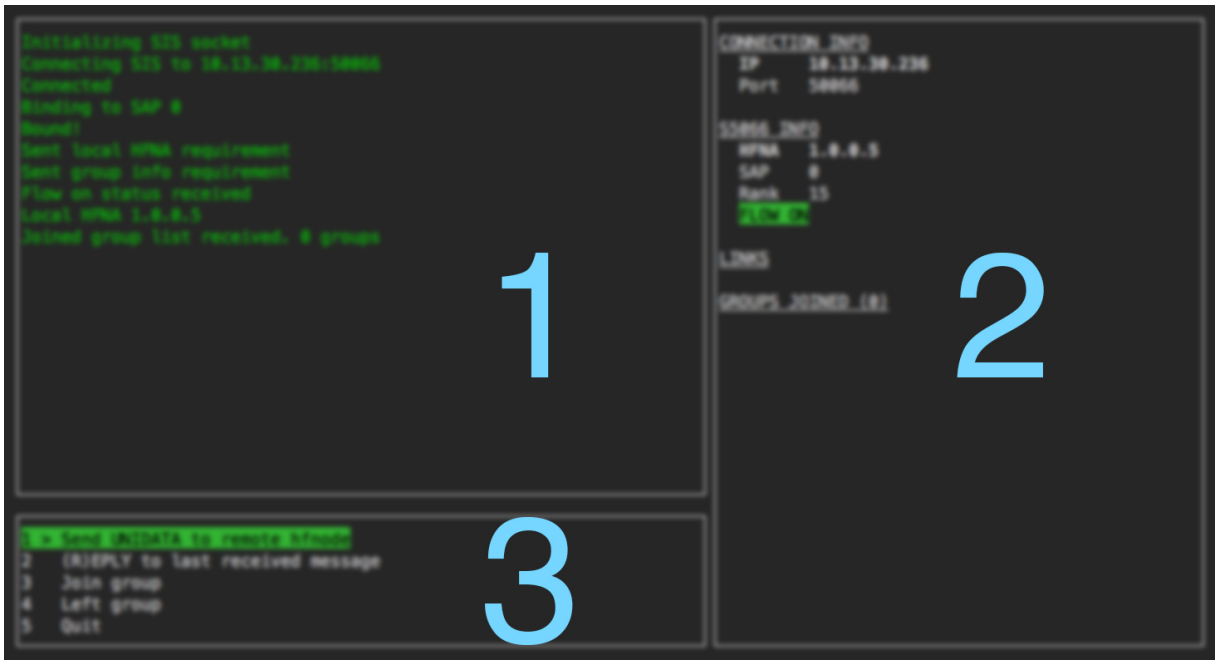


Figura 5.2: Interfaz en línea de comandos del Subnet Management Client

5.1.2. Verificar la creación y ruptura de enlaces ARQ

La herramienta *Subnet Management Client* nos indica los enlaces vigentes en el servidor STANAG 5066 en el área de información, como se ilustra en la Figura 5.3.

Para la verificación de la creación de los enlaces, se comenzó con un servidor STANAG 5066 en el que no se hubieran creado enlaces durante su ejecución y se envió un mensaje ARQ a un nodo remoto. Recordemos que el modo de transmisión ARQ necesita de la existencia de un enlace, así que antes de la transmisión del mensaje, el servidor tratará de crear el enlace necesario.

Una vez creado el enlace, mantuvimos un tiempo de inactividad suficiente para que la capa SIS comenzara el protocolo de ruptura de enlace.

Además, se realizaron pruebas de enlaces simultáneos con dos nodos, como se ilustra en la Figura 5.4.

Se comprobó que la creación y ruptura de enlaces, tras 500 segundos de inactividad, se realizaba de manera correcta.

Figura 5.3: Información relativa a los enlaces vigentes en el servidor STANAG 5066

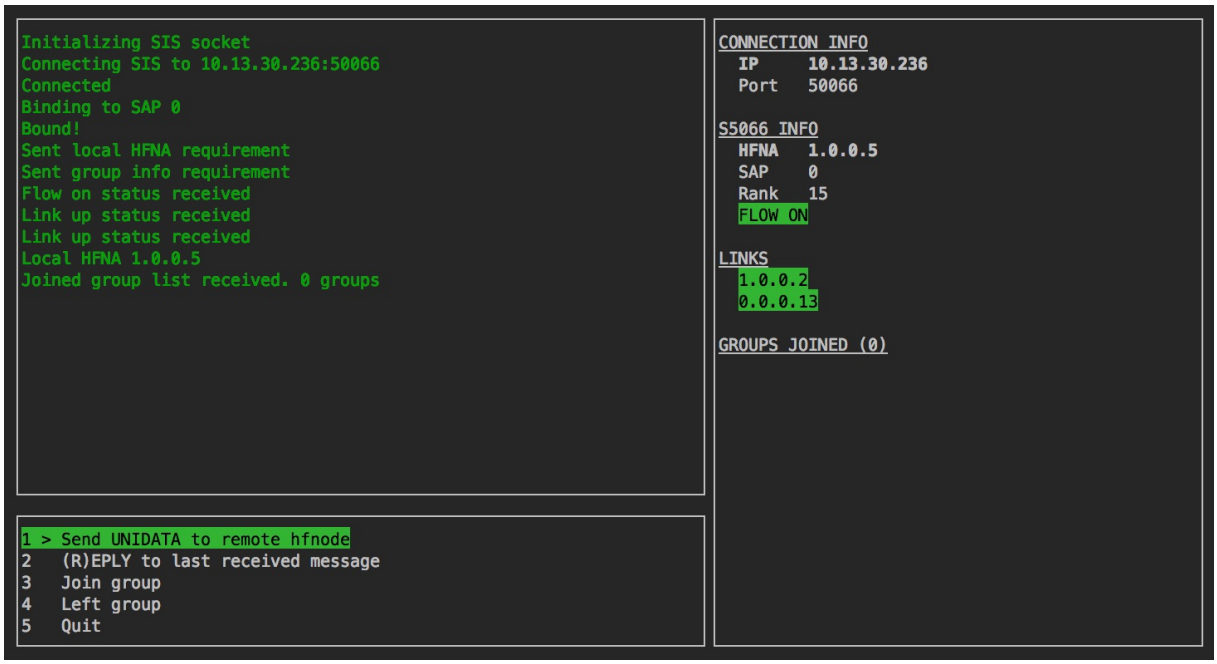


Figura 5.4: Creación de dos enlaces simultáneos, con los nodos `0.0.0.13` y `1.0.0.2`

5.1.3. Comprobar la cadena de transmisión y recepción de paquetes ARQ

Para la verificación de la cadena de transmisión y recepción de paquetes ARQ, probamos enviar mensajes de manera alterna dos nodos diferentes, el `0.0.0.13` y el `1.0.0.2`.

Además, de manera aleatoria, durante la transmisión de algunos mensajes desconectamos los cables entre los módems, de manera que no se pudiera realizar la comunicación. Después de un tiempo de espera que no fuera suficiente para romper el enlace, se reconectaron los cables, reestableciéndose la comunicación, y permitiendo que llegaran los mensajes al destino.

Se puede comprobar la transmisión y recepción correcta de los mensajes en la Figura 5.5, en la que se muestra la recepción de los mensajes desde cada nodo al que iba dirigido, además de las confirmaciones en el nodo receptor.

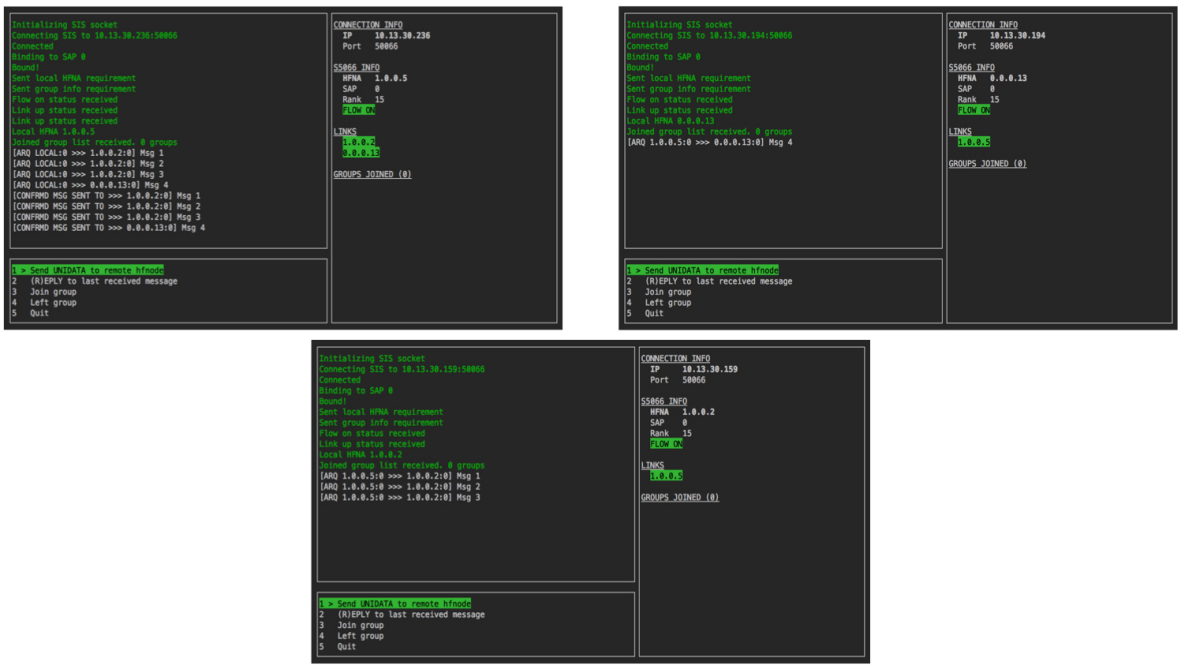


Figura 5.5: Historial de un intercambio de mensajes ARQ y sus confirmaciones de recepción

Para concluir esta prueba, se realizó la transmisión de mensajes destinados a un SAP al cual no hubiera ningún cliente conectado, y configurando dicho mensaje con el parámetro de confirmación de cliente. Esto debe generar un rechazo de los datos transmitidos, y pudimos comprobar que se hace de manera correcta, tal y como se muestra en la Figura 5.6.

5.1.4. Comprobar la cadena de transmisión y recepción de paquetes NO-ARQ punto a punto

De manera similar a la verificación de la cadena ARQ, en este caso realizamos una serie de transmisiones usando el modo NO-ARQ.

Podemos verificar, como se ilustra en la Figura 5.7, que la recepción de los mensajes transmitidos se llevó a cabo sin ningún problema, siendo recibidos únicamente por el nodo destinatario.

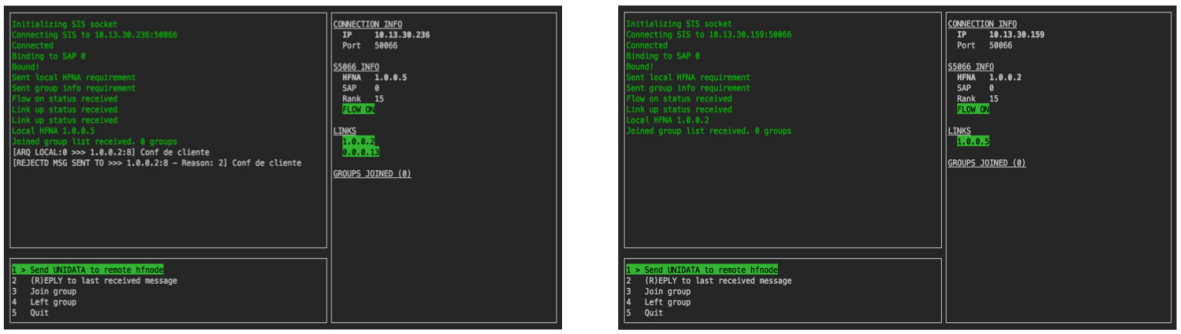


Figura 5.6: Transmisión de un mensaje ARQ con confirmación de nodo a un SAP sin cliente conectado

Adicionalmente, leyendo las trazas del servidor STANAG 5066 comprobamos que se hace correctamente la retransmisión de las D_PDU NO-ARQ, así como la recomposición en recepción.

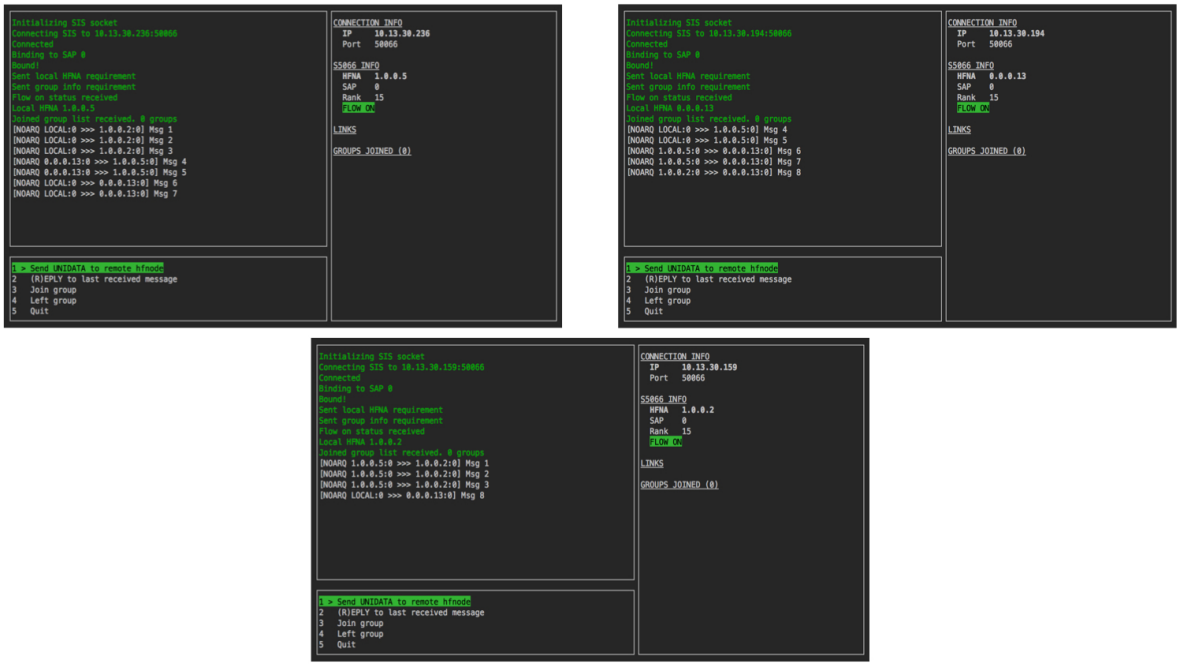


Figura 5.7: Historial de un intercambio de mensajes NO-ARQ

5.1.5. Verificación del funcionamiento de los mensajes a grupos

En el caso de los mensajes a grupos, se ha de comprobar dos lógicas que se han implmenentado: por un lado la parte de unirse y salir de grupos, que se realiza localmente bajo petición del Subnet Management Client (o cualquier otro cliente que implemente las primitivas descritas en la Sección 3.7); por otro lado, comprobar que el filtrado de los destinos NO-ARQ sólo acepte los paquetes destinados a los grupos a los que pertenecemos.

Para ejecutar esta prueba se configuraron los tres módem de la manera que se muestra en la Tabla 5.1:

Grupos Modem A	Grupos Modem B	Grupos Modem C
4.0.0.0	4.0.0.0	4.0.0.0
5.0.0.0	5.0.0.0	
6.0.0.0		

Tabla 5.1: Grupos en los que se incluyó cada módem

Una vez se han incluido cada módem en los grupos correspondientes, se enviaron varios mensajes desde cada módem, destinado a los grupos a los que pertenecían los módems remotos. También se realizaron transmisiones destinadas a grupos a los que no pertenecía ningún modem para comprobar que se realizaba correctamente el filtrado.

Para finalizar, cada módem se dejó sin ningún grupo, y se volvió a mandar varios mensajes de prueba destinado a los grupos a los que anteriormente sí pertenecían.

Un ejemplo de las pruebas realizadas se ilustra en la Figura 5.8.

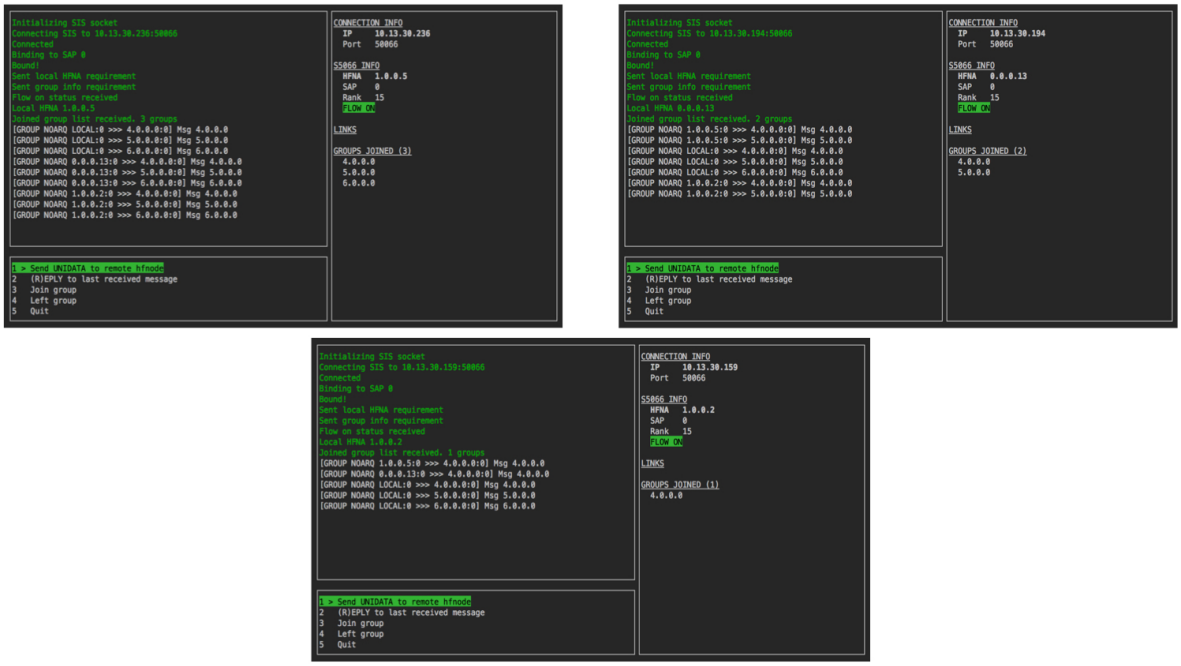


Figura 5.8: Mensajes intercambiados en la prueba de mensajes a grupos

5.1.6. Verificación de primitivas `FLOW_ON` y `FLOW_OFF`

Durante todas las pruebas anteriores, se pudo comprobar que el servidor STANAG 5066 generaba las primitiva `FLOW_ON` y `FLOW_OFF`. Su comportamiento es correcto ya que tras el envío de varias `UNIDATA_REQUEST` se enviaba el `FLOW_OFF`, y posteriormente tras la transmisión de éstas, se enviaba el pertinente `FLOW_ON`.

5.1.7. Medidas de latencia

Para las medidas de latencia se contemplan tres casos:

- Mensajes NO-ARQ
- Mensajes ARQ
- Creación de enlaces

Estas pruebas se realizaron **siempre** con los servidores STANAG 5066 en estado de reposo, y sin información pendiente de transmitir en los buffers internos. El motivo de esta decisión es que la latencia podría variar mucho dependiendo del uso del canal por parte de otros módems, o incluso el uso del servidor STANAG 5066 local por parte de otros clientes SAP.

Como se ha citado, el estándar STANAG 5066 impone un límite máximo de dos minutos durante el que cada nodo podría transmitir, con lo que podría darse el caso en el que realizemos la medición de un paquete que tardaría dos minutos en comenzar a ser transmitido si un nodo remoto considera hacer uso del canal durante este tiempo.

En la ejecución de estas pruebas transmitimos varios mensajes NO-ARQ o ARQ, midiendo el tiempo desde el momento del envío en el Subnet Management Client local, hasta la recepción en el Subnet Management Client remoto. Se realizaron 30 envíos de mensajes para cada servicio, y 10 creaciones de enlace.

Los resultados se pueden observar en la Tabla 5.2

Servicio	Tiempo medio	Desviación estándar
Mensajes NO-ARQ	6.13 s	0.15 s
Mensajes ARQ	6.20 s	0.09 s
Confirmación ARQ	18.37 s	0.14 s
Notificación de enlace en nodo remoto	6.34 s	0.21 s
Notificación de enlace en nodo local	18.40 s	0.18 s

Tabla 5.2: Medidas de latencia

Podemos apreciar que las comunicaciones que se realizan en una primera transmisión, como son los mensajes ARQ y NO-ARQ, así como la notificación de la creación de enlace en nodo remoto, tarda algo más de 6 segundos en producirse; esto es aproximadamente el tiempo de sincronismo más un entrelazado.

Las comunicaciones que requieren "ida y vuelta", como son la confirmación de los mensajes ARQ y la notificación de enlace en el nodo local, se demoran algo más de 18 segundos. Corresponde al tiempo de dos transmisiones completas, más el tiempo de SEOT de la primera transmisión y un tiempo de espera antes de la segunda transmisión de un segundo.

5.1.8. Pruebas de larga duración

Uso del GUI del Sistema HFDVL

Las pruebas de larga duración fueron realizadas usando el GUI del Sistema HFDVL. El GUI cuenta con un cliente STANAG 5066 que está preparado para transmitir ficheros. Este cliente, troceará el fichero seleccionado en segmentos del tamaño de una `S_PDU`, alrededor de 2kbytes. La recomposición de esas S_PDUs en el fichero de recepción queda en manos del cliente del GUI que hace el papel de receptor.

Estas pruebas servirán para verificar que tras el envío repetido de `S_PDU`, el servidor STANAG 5066 no sufre ninguna corrupción de los datos, y sus estructuras internas siguen funcionando correctamente.

También, sirven para realizar una medida del *throughput* neto del Sistema HFDVL junto con el servidor STANAG 5066.

Durante estas pruebas se transmitió un archivo de 10Mbytes con la configuración más lenta y con la más rápida del módem HFDVL. Esto corresponde a velocidades de 1.800 bps y 8.640 bps respectivamente.

Al hacer el md5sum[1] de los ficheros transmitidos en origen y en destino, concidieron:

```
hfdvl@modem1:~/sent_files> md5sum file10mb-1.file
c104043ae74d85b7fcaea50c2d2ffa55  file10mb-1.file

hfdvl@modem2:~/received_files> md5sum file10mb-1.file
c104043ae74d85b7fcaea50c2d2ffa55  file10mb-1.file


hfdvl@modem1:~/sent_files> md5sum file10mb-2.file
5b4e802eaa5acd50faa0f1f5bdeb56dd  file10mb-2.file

hfdvl@modem2:~/received_files> md5sum file10mb-2.file
5b4e802eaa5acd50faa0f1f5bdeb56dd  file10mb-2.file
```

Posteriormente a las pruebas se pudo seguir usando el Subnet Management Client de manera normal para transmitir mensajes en cualquier servicio correctamente, con lo que podemos considerar que el servidor STANAG 5066 ha mantenido las estructuras internas sin corromper.

5.1.9. Medidas de rendimiento

Con los resultados de las pruebas de larga duración se puede obtener una medida del rendimiento del *throughput* neto del sistema, usando el servidor STANAG 5066.

[1]Realizar el Hash MD5 de un fichero nosdevuelve una cadena de 128 bits calculada a partir del contenido del mismo. Utilizando esta herramienta podemos comprobar si un fichero ha sido modificado

En la Tabla 5.3, se presentan los tiempos requeridos para la transmisión del fichero de 10Mbytes en cada configuración.

Velocidad	Tamaño archivo	Tiempo de transmisión	Throughput bruto	Throughput neto (eficiencia)
1.800 bps	10.485.760 bytes	102.265 s (28h 24m 35s)	1.800 bps	820 bps (45 %)
8.640 bps	10.485.760 bytes	54.694 s (15h 11m 34s)	8.640 bps	1533 bps (17 %)

Tabla 5.3: Tiempos de transmisión y rendimiento neto del Sistema HFDVL con el servidor STANAG 5066

Se puede observar en la Tabla 5.3 que el rendimiento decrece a medida que la velocidad bruta aumenta, a pesar que el *throughput* neto también aumenta. Esto es debido a que los tiempos utilizados para transmitir el *payload* decrecen (aumentando el *throughput*), pero los tiempos necesarios para realizar los ACKs, esperas, etc, quedan constantes (disminuyendo la eficiencia). Esto lo vemos en la Figura 5.9.

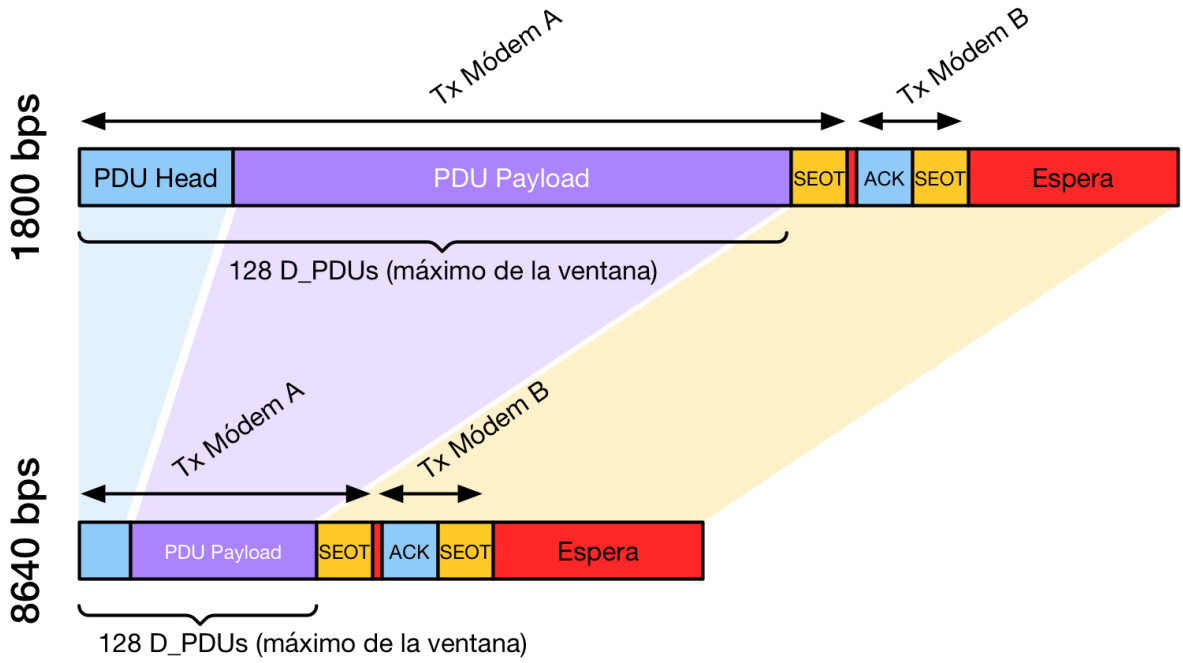


Figura 5.9: Uso del tiempo de transmisión para diferentes velocidades brutas del módem HFDVL

5.2. Pruebas a través de enlace HF real

Tras la realización satisfactoria de las pruebas en un entorno controlado, comenzamos a probar el Sistema HFDVL y el servidor STANAG 5066 en un enlace de HF real ULPGC ⇔ UPM (Madrid).

Las pruebas a través de un enlace real de HF introducirán errores en las comunicaciones, poniendo a prueba la integridad de los mensajes NO-ARQ tras las repeticiones, así como la solicitud de repeticiones selectivas del modo ARQ.

Las pruebas se realizaron utilizando el cliente HFMail, desarrollado dentro del contexto del Proyecto Coincidente [15], y descrito en el estándar STANAG 5066 [10, Anexo F.5]. Para ello, se automatizó el envío de correos electrónicos entre servidores conectados a los extremos de la comunicación HF, tal y como se muestra en la Figura 5.10. Desde el punto de vista del servidor STANAG 5066, se trata de S_PDUs de la misma manera que sucede con el Subnet Management Client.

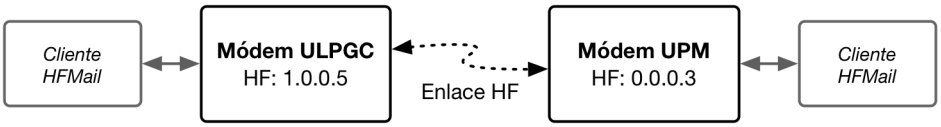


Figura 5.10: Topología de pruebas a través de enlace HF real

El propósito del cliente HFMail es proveer de un cliente que se encargue de automatizar el uso del servidor STANAG 5066, ya que se programó el envío de correos de manera automática desde el momento en que no haya correos pendientes de enviar. El cliente HFMail tratará de entregar los correos de manera constante, incluso cuando no se pueda generar el enlace HF entre dos nodos (en horas nocturnas[2] no hubo comunicación).

La constancia en el envío de datos por parte del cliente HFMail permitió estudiar el comportamiento del servidor STANAG 5066 con una alta tasa de errores en las comunicaciones, ya que a medida que se producía la transición entre las horas óptimas y las menos óptimas, se iban produciendo paulatinamente más errores en el canal, generando una exigencia mayor a la hora de repetir cada vez más segmentos ARQ.

Lo que pudimos comprobar es que el funcionamiento del servidor STANAG 5066 es correcto y robusto, solicitando las repeticiones de los segmentos que no han sido recibidos correctamente en modo ARQ, y generando la mejor combinación posible en los mensajes NO-ARQ.

[2]El sistema se programó con una frecuencia constante óptima para las horas diurnas por lo que durante la noche eran raras ocasiones en las que podía haber comunicación

Capítulo 6

Conclusiones y líneas futuras

6.1. Conclusiones

A la vista de los resultados podemos concluir, de forma global, que se ha podido implementar de forma exitosa el protocolo STANAG 5066 en combinación con el Sistema HFDVL.

Las pruebas realizadas en transmisiones por radio han servido para detectar y corregir errores en la implementación, así como para optimizar diferentes secciones del código.

Cabe destacar varias decisiones de diseño que se han demostrado acertadas, ya que han permitido flexibilidad en el desarrollo en equipo, así como mantener unas buenas prácticas de programación:

- Implementación de las capas como librerías. Las capas mantienen una independencia entre sí, manteniendo las funcionalidades perfectamente acotadas en la definición del estándar. Queda acotado también el ámbito de procesado de cada capa, siendo accesibles únicamente a través de llamadas a las funciones de API que definen.

  Haber definido una API común a las tres capas hace que sea muy sencillo alternar la programación entre las capas.

- La configuración del proceso de compilación a través de un Makefile bien estructurado ha permitido que tanto las librerías que implementaran las distintas capas como el servidor STANAG 5066 se hayan podido compilar en las diferentes máquinas del equipo de trabajo.

  Como ejemplo de utilidad del Makefile, a la hora de realizar pruebas de portabilidad para compilar la aplicación en diferentes arquitecturas, tan solo fue necesario añadir un flag de compilación en cada uno de los Makefiles.

- Además del aislamiento de las capas SIS, CAS y DTS, la existencia de la capa MNGT para coordinar los mensajes entre ellas, ha sido fundamental. Esta capa provee funciones de soporte (temporización de eventos y traspaso de mensajes),

permitiendo abordar en las otras tres una implementación del estándar más clara y ordenada.

La versatilidad y sencillez de esta arquitectura permite corregir errores o implementar características con una gran flexibilidad.

Se añade a estas conclusiones directamente relacionadas con el desarrollo del software, algunas otras relativas a las comunicaciones HF.

- El protocolo STANAG 5066 está diseñado para comunicaciones HF de bajo *throughput* y alta latencia. Esto quiere decir que una simple comunicación de unos pocos kilobytes puede demorarse minutos, y en ocasiones, un usuario que no esté habituado a enfrentarse a estas limitaciones puede cometer excesos en los tamaños de transmisiones.
- En ocasiones, el usuario puede encontrarse con falta de información a la hora de operar un enlace de comunicaciones. El protocolo STANAG 5066 no define una manera estándar de proporcionar información de su estado interno (por ejemplo, *soft links* activos, o cuántas C_PDUs hay encoladas), por tanto, en base a la experiencia se ha ido ampliando la información que el servidor STANAG 5066 transmite al usuario.

El desarrollo de este proyecto presenta una oportunidad para líneas futuras de trabajo e investigación, en relación a la eficiencia del protocolo y a técnicas de acceso al canal para evitar colisiones. Además, hay multitud de puntos en el que el estándar STANAG 5066 permite al implementador de protocolo hacer ampliaciones y mejorar el comportamiento de los nodos.

6.2. Líneas futuras de trabajo y puntos abiertos en el STANAG 5066

A lo largo del desarrollo de este proyecto se han detectado muchas posibles optimizaciones. Éstas no se han podido llevar a cabo por la limitación temporal y la concentración de esfuerzos en realizar una implementación funcional y libre de errores.

Una compilación de las diferentes optimizaciones que se han identificado se muestra a continuación:

- Implementar una limitación de flujo (FLOW OFF) en función de la actividad de cada cliente, y no a través de los elementos libres del array de C_PDUs. La implementación actual no penaliza individualmente a los clientes más activos. También podría tenerse en cuenta el *rank*, o el nodo de destino, de manera que se podría parar el flujo destinado a un nodo de destino en concreto.
- Optimizar la cantidad de datos transmitidos en función del tamaño del entrelazado del módem. Por ahora, la DTS sólo recibe la información de la velocidad del

módem, para poder cumplir con la limitación de los dos minutos consecutivos en tramsisión.

Optimizar en función del entrelazado podría significar que se transmitirían 110 elementos, ya que los 18 restantes hasta completar 128 infrautilizarían un bloque de entrelazado.

- Eliminar el array de C_PDUs existente en la DTS, y convertirlo en una lista doblemente enlazada de elementos, al igual que se hace con los segmentos en recepción ARQ.

  Además, esto corregiría un potencial fallo existente, en el cual podría desordenarse la entrega de las C_PDUs a los nodos remotos. Este fallo sólo se daría una vez cada 2^{64} C_PDUs transmitidas, ya que es dependiente de la variable **uint64_t** `c_pdu_id_arq_tx` presente en las *st-mach*.

- Implementación de más S_Primitivas de información de tipo `S_MANAGEMENT_...`. Algunos parámetros de los que podría informar son: Porcentaje de utilización de los recursos (C_PDUs, st_mach, ...); throughput medio de TX/RX en las últimas 1 hora, 2 horas, 6 horas, 24 horas...

- Desarrollo de un modo equivalente a *IP forward*, de manera que se puedan configurar nodos HF como enrutadores de D_PDUs. Este punto es delicado y muy ambicioso, ya que entran en juego multitud de factores, como puede ser que la red de nodos pueda ser visible o no en función de la propagación de la ionosfera, con lo que el modo *forward* sería perjudicial en un momento dado.

- Reserva de memoria dinámica para C_PDUs y para st_mach. Actualmente la inicialización de la DTS reserva toda la memoria que cree que pueda ser necesaria para su ejecución. Eso hace que normalmente tenga muchos recursos inutilizados.

  La reserva de recursos es un punto crítico en sistemas embebidos, como puede ser el Sistema HFDVL. Realizar una optimización y enfocar la gestión de memoria hacia una reserva/liberación dinámica de los recursos podría hacer el sistema más eficiente.

  Por su puesto, esta implementación conlleva el peligro de *leaks* de memoria, o de la reserva incontrolada de recursos, por lo que no debe ser un desarrollo menor.

- Experimentar diferentes modos de optimizar las comunicaciones, como puede ser:

  - Informar a cada nodo de la topología de red para que haga las optimizaciones oportunas.
  - Conocer si la red está limitada a un enlace punto-a-punto reduce los tiempos de espera.
  - Tiempos de vida de enlace variables en función de un histórico de de actividad.
  - Investigar una nueva estrategia para el *modo management*, que pueda ser dependiente del número de nodos existentes en lar ed.

También cabría explorar la configuración de ventanas deslizantes más grandes, y mayores tamaños de D_PDU; soluciones que ya se prueban en implementaciones de otras compañías, como ISODE [13].

Parte II

Bibliografía

Bibliografía

[1] Department of the Army, Information Systems Engineering Command, MIL-STD-188-110A: *Interoperability and Performance Standards for Data Modems*. Philadelphia, PA. Naval Publications and Forms Center, Attn. NPODS, Sep 1991.

[2] *"Desarrollo y Evaluación de un Módem Avanzado Multiportadora en HF para Comunicaciones Aéreas"*. Proyecto-referencia: CN-36/00-33017. Financiado por AENA (Aeropuertos Españoles y Navegación Aérea).

[3] *"Generación de prototipos operativos de un módem HF multiportadora y evaluación de prestaciones en entornos reales"*. Proyecto-referencia: 240-033-0051. Financiado por AENA (Aeropuertos Españoles y Navegación Aérea).

[4] *"Módem para Voz Digital Interactiva en Comunicaciones Ionosféricas"*. Proyecto-referencia: TEC2004-06915-C03-01/TCM.

[5] *"M3 en HF: Sistemas Multiportadora, Multibanda, Multiantena en Comunicaciones HF"*. Proyecto-referencia: TEC2007-67520-C02-02/02/TCM

[6] Sklar, B. "Rayleigh fading channels in mobile digital communications systems Parts 1 and 2: Characterization and Mitigation". *IEEE Communications Magazine*, pp.90-101 y pp.102-109, Julio 1997.

[7] Proakis, J.G. *Digital communications*, 3d ed. New York, McGraw-Hill, 1995.

[8] España. Anuncios de licitaciones públicas y adjudicaciones, número 824. *Anuncio de formalización de contratos de: Área Económica de la Dirección General de Armamento y Material DGAM. Objeto: Programa: Dn8644 Coincidente Ëdesarrollo, Implementación y Prototipado de una Forma de Onda Española en Hf para la Transmisión de Voz Digital y Datos en el Ámbito de Defensa (Mdef-Hfdvl)¨ Expediente: 1003211004400.*
BOE, 10 de enero de 2010, núm. 8, p. 1049

[9] ITU-R F.1487. "Testing of HF modems with bandwidths of up to about 12 kHz using ionospheric channel simulators" . *ITU-R Technical Report*, 2000.

[10] NATO Standarization Agreement: *"Profile for High Frequency (HF) Radio Data Communications"*. STANAG 5066. Version 2.0.

[11] ITU-T X-224. "Open Systems Interconnection - Protocol for providing the connection-mode transport se". *ITU-T*, 1995.

[12] Timothy M. Schmidl and Donald C. Cox. "Robust Frequency and Timing Synchronization for OFDM". *IEEE Transactions On Communications*, pp.1613-1621, Diciembre 1997.

[13] ISODE. "Extending STANAG 5066 to improve ARQ Performance over Wideband HF Radio". URL https://www.isode.com/whitepapers/extending-stanag-5066.html

[14] V. Jacobson, R. Braden, D. Borman. "TCP Extensions for High Performance". RFC 1323. Mayo 1992.

[15] España. Contratación del Sector Público. *Anuncio de formalización de contratos de: Subdirección General de Adquisiciones de Armamento y Material DGAM. Objeto: "Conectividad e interoperabilidad de sistemas IP en redes HF mediante el sistema HFDVL NB/WB (IP-HFDVL)". Programa Coincidente DN8644. Expediente: 1003215004100.*
BOE, 30 de noviembre de 2015, núm. 286, p. 49959

Parte III

Pliego de condiciones

Pliego de condiciones

Los recursos hardware empleados para el desarrollo del presente Proyecto Fin de Carrera son:

3 x Módem HFDVL Plataforma necesaria para hacer las pruebas con los servidores HFDVL. Al menos tres, para poder probar los mensajes a grupos.

Ordenador Personal En él hemos llevado a cabo el diseño y la implementación del servidor STANAG 5066, así como la redacción y maquetación de este documento. Las características principales de este PC son:

- Procesador Intel Core 2 Duo a 2,8GHz.
- 4Gbytes de memoria RAM.
- 500Gbytes de disco duro.

Switch Ethernet 3com 3CFSU05 Para interconectar por Ethernet el PC y los módems, además de otros dispositivos de la red.

Cables de red Los cables empleados para conectar el PC y los módems al switch son de categoría 5, y de 2m de longitud cada uno.

Impresora Develop ineo+ 220 Utilizada para la impresión de parte de la bibliografía y manuales necesarios para el desarrollo del proyecto, así como del presente documento.

Los recursos software utilizados para desarrollar los objetivos del presente proyecto son los siguientes:

Sistema Operativo openSUSE 11.4 Sistema operativo instalado en el PC.

Software HFDVL Versión validada y ejecutándose en mayo de 2017.

Compilador GCC para Linux v4.3.1 Se ha empleado para compilar el servidor STANAG 5066.

Atom y TeXLive v2015.20160222 Editor y procesador de de textos LaTeX para la redacción de la memoria del PFC.

Highlight v3.18 Aplicación para dar formato LaTeX a los fragmentos de código insertados en este documento.

LibreOffice Draw v5.1.4 Aplicación de la suite ofimática *LibreOffice* que ha sido empleada para diseñar algunas de las las gráficas incluidas en esta memoria.

Inkscape v0.91 Aplicación de diseño gráfico vectorial utilizada para la creación de otras de las gráficas de esta memoria.

Apple iWork Keynote v7.1.1 Aplicación con la que se ha realizado la presentación de este Proyecto Fin de Carrera.

Parte IV

Presupuesto

Presupuesto

El presupuesto del presente Proyecto Fin de Carrera se ha realizado según los criterios que establece el Colegio Oficial de Ingenieros de Telecomunicación (COIT), tal y como se indica a continuación:

- Trabajo tarifado por tiempo empleado.
- Amortización del inmovilizado material.
  - Amortización del material hardware.
  - Amortización del material software.
- Redacción del proyecto.
- Derechos de visado del COIT.
- Gastos de tramitación y envío.

En este presupuesto se han aplicado los baremos publicados por el Colegio Oficial de Ingenieros de Telecomunicación para el año 2008[1], pues los correspondientes a los años posteriores no han sido publicados por indicación de las autoridades de la competencia del Ministerio de Economía, Industra y Competitividad.

Una vez analizados cada uno de los criterios establecidos se aplicarán los impuestos vigentes para obtener el coste total del presente Proyecto Fin de Carrera.

P.1. Trabajo tarifado por tiempo empleado

Se contabilizan los gastos que corresponden a la mano de obra, según el salario correspondiente a la hora de trabajo de un ingeniero. El COIT propone utilizar la siguiente fórmula:

$$H = (C \times 75 \times H_n) + (C \times 95 \times H_e) \quad € \tag{P.1}$$

donde:

[1] `http://www.coit.es`

- H son los honorarios totales por el tiempo dedicado.
- C es el coeficiente de corrección en función del número de horas trabajadas definido por el COIT por tramos.
- H_n son las horas normales trabajadas dentro de la jornada laboral.
- H_e son las horas especiales trabajadas.

Se estima que para la realización del presente Proyecto Fin de Carrera se ha trabajado durante 9 meses, desde agosto de 2016 hasta abril de 2017. Considerando 20 días laborables al mes, con una dedicación a tiempo completo de 8 horas por jornada laboral, el número de horas normales asciende a:

$$H_n = 9 \times 20 \times 8 = 1.440 \text{ horas} \tag{P.2}$$

Dado que no se ha trabajado fuera del horario laboral, el número de horas especiales es cero. Por otro lado, el coeficiente corrector a aplicar es el que está asignado por el COIT para un exceso de 1.080 horas y cuyo valor es de 0,4. Finalmente, obtenemos el coste total de los honorarios, que asciende a:

$$H = (0,4 \times 75 \times 1440) + (0,4 \times 95 \times 0) = 43.200{,}00 \text{ €} \tag{P.3}$$

El trabajo tarifado por tiempo empleado asciende a la cantidad de *cuarenta y tres mil doscientos euros.*

P.2. Amortización del inmovilizado material

En el inmovilizado material se consideran tanto los recursos hardware como software empleados para la realización de este Proyecto Fin de Carrera.

Se estipula el coste de amortización para un período medio de 5 años utilizando un sistema de amortización lineal en el que se supone que el inmovilizado material se deprecia de forma constante a lo largo de su vida útil. La cuota de amortización anual se calcula haciendo uso de (P.4).

$$\text{Cuota anual} = \frac{\text{Valor de adquisición} - \text{Valor residual}}{\text{Número de años de vida útil}} \tag{P.4}$$

donde el *valor residual* es el valor teórico que se supone que tendrá el elemento en cuestión después de su vida útil.

P.2.1. Amortización del material hardware

Dado que la duración de este Proyecto Fin de Carrera es de 9 meses y es inferior al periodo de 5 años estipulado para el coste de amortización, los costes serán los derivados del tiempo de utilización de cada recurso. Todos los recursos hardware se han utilizado durante 9 meses.

En la tabla P.1 se relaciona el hardware necesario para la realización del proyecto, indicando para cada recurso su valor de adquisición, valor residual y coste de amortización.

Recurso	**Valor de adquisición**	**Valor residual**	**Coste de la amortización**
Ordenador personal Intel Core 2 Duo 2,8GHz	1.020,00 €	150,00 €	130,50 €
Total	**1.020,00 €**	**150,00 €**	**130,50 €**

Tabla P.1: Precios y costes de amortización del hardware.

El coste total del material hardware asciende a *ciento trenita euros con cincuenta céntimos*.

P.2.2. Amortización del material software

Para el cálculo de los costes de amortización del material software se considerarán, al igual que con el material hardware, que los costes serán los derivados del tiempo de utilización de cada recurso.

La tabla P.2 muestra los elementos software necesarios para la realización del proyecto, así como su valor de adquisición, valor residual y coste de amortización.

Recurso	**Valor de adquisición**	**Valor residual**	**Coste de la amortización**
Sistema operativo openSUSE 11.4	0,00 €	0,00 €	0,00 €
Compilador GCC para Linux	0,00 €	0,00 €	0,00 €
Atom, TeXLive, Highlight (LaTeX)	0,00 €	0,00 €	0,00 €
LibreOffice Draw	0,00 €	0,00 €	0,00 €
Inkscape	0,00 €	0,00 €	0,00 €
Apple iWork Keynote	16,34 €	5,00 €	0,05 €
Total	**16,34 €**	**5,00 €**	**0,05 €**

Tabla P.2: Precios y costes de amortización del software.

Todos los recursos software se han utilizado durante 9 meses, a excepción de las aplicaciones Atom, TeXLive, Highlight, LibreOffice Draw e Inkscape, que se han utilizado durante 1 mes, y la aplicación Keynote, durante 1 semana.

Por tanto, el coste total del material software asciende a la cantidad de *cinco céntimos de euro*.

P.3. Redacción del proyecto

El COIT recomienda utilizar (P.5) para determinar el coste asociado a la redacción de la memoria del proyecto.

$$R = 0,07 \times P \times C \tag{P.5}$$

donde:

- R son los honorarios por la redacción del proyecto.
- P es el presupuesto.
- C es el coeficiente reductor de honorarios en función del presupuesto.

El valor del presupuesto P se calcula sumando los costes de las secciones anteriores correspondientes al trabajo tarifado por tiempo empleado y a la amortización del inmovilizado material, tanto hardware como software. Esta suma de costes se muestra en la tabla P.3.

Concepto	**Coste**
Trabajo tarifado por tiempo empleado	43.200,00 €
Amortización del material hardware	130,50 €
Amortización del material software	0,05 €
Total (P)	**43.330,55 €**

Tabla P.3: Presupuesto incluyendo trabajo y amortización

El coeficiente corrector a aplicar, según el establecido por el COIT para el tramo comprendido entre 30.050 € y 60.101 €, es de 0,9. Por tanto, se obtiene:

$$R = 0,07 \times 43.330,55 \times 0,9 = 2.729,82 \text{ €} \tag{P.6}$$

El coste de la redacción del proyecto asciende a *dos mil setecientos veintinueve euros con ochenta y dos céntimos*.

P.4. Derechos de visado del COIT

El COIT establece que para la redacción de proyectos y trabajos en general, los derechos de visado para 2008 se calculan en base a (P.7).

$$V = 0,006 \times P \times C \tag{P.7}$$

donde:

- V es el coste de visado del proyecto.
- P es el presupuesto.
- C es el coeficiente reductor en función del presupuesto.

El valor del presupuesto P se halla sumando los costes de las secciones anteriores correspondientes al trabajo tarifado por tiempo empleado, a la amortización del inmovilizado material, tanto hardware como software, y a la redacción del proyecto. Esta suma se muestra en la Tabla P.4.

Concepto	**Coste**
Trabajo tarifado por tiempo empleado	43.200,00 €
Amortización del material hardware	130,50 €
Amortización del material software	0,05 €
Redacción del proyecto	2.729,82 €
Total (P)	**46.060,37 €**

Tabla P.4: Presupuesto, añadiendo la redacción del proyecto.

En este caso, el coeficiente reductor C es el asignado para el tramo entre 42.070,70 € y 63.106,05 €, cuyo valor es de 0,45. Así, aplicando (P.7) con los datos de la tabla P.4 y el coeficiente especificado se obtiene:

$$V = 0,006 \times 46.060,37 \times 0,45 = 124,36 \text{ €} \tag{P.8}$$

Los costes por derechos de visado del presupuesto ascienden a *ciento veinticuatro euros con treinta y seis céntimos.*

P.5. Gastos de tramitación y envío

Los gastos de tramitación y envío están estipulados en *doce euros* (12,00 €)

P.6. Aplicación de impuestos y coste total

La realización del presente Proyecto Fin de Carrera está gravada por el Impuesto General Indirecto Canario, I.G.I.C., en un siete por ciento (7 %). En la tabla P.5 se muestra el presupuesto final con los impuestos aplicados.

El presupuesto total del proyecto *"Implementación de un protocolo estándar con ARQ para comunicaciones HF"* asciende a *cuarenta y nueve mil cuatrocientos treinta euros con cincuenta céntimos.*

Concepto	**Coste**
Trabajo tarifado por tiempo empleado	43.200,00 €
Amortización del material hardware	130,50 €
Amortización del material software	0,05 €
Redacción del proyecto	2.729,82 €
Derechos de visado del COIT	124,36 €
Gastos de tramitación y envío	12,00 €
Total (Sin IGIC)	**46.196,73 €**
IGIC (7 %)	3.233,77 €
Total	**49.430,50 €**

Tabla P.5: Presupuesto total del Proyecto Fin de Carrera.

El ingeniero proyectista

Fdo: D. Víctor Alonso Eugenio
En Las Palmas de Gran Canaria, a 10 de junio de 2017

Parte V

Apéndices

Apéndice A

Makefiles

La aplicación `make` es una herramienta de generación o automatización de código, para compilar bibliotecas o aplicaciones. Para ello se sirve de los llamados *makefiles*, en los que se definen los objetivos de compilación, qué parametros o qué bibliotecas son necesarios, etc. Para hacer uso de esta utilidad solo tenemos que incluir en el directorio raíz de compilación un fichero con el nombre `Makefile` y ejecutar en la consola:

```
$ make [objetivo(s)]
```

El código de los Makefiles de las tres librerías de las capas se presentan a continuación:

SIS

```
#################################################################
# The architecture to compile against is given by BITS=32 or BITS=64
during
# make invocation (by default 32 bits).

## User defined compile-time options

## Debug and runtime flags
FLAGS_RUN = -O2
#FLAGS_DEBUG = -O0 -g
FLAGS_DEBUG = -O2

MACROS = -D_GNU_SOURCE -D_POSIX_C_SOURCE=199309L
ifndef DEBUGCODE
MACROS += $(FLAGS_RUN) -DNDEBUG
else
ifeq ($(strip $(DEBUGCODE)),0)
MACROS += $(FLAGS_RUN) -DNDEBUG
else
```

```
MACROS += $(FLAGS_DEBUG)
endif
endif


DEFAULTBITS = 64
## Architecture to compile for. If you not define BITS outside
Makefile,
## compilation will be for the default value inside the conditional.
## If you want to compile for a given architecture, let's say 64 bits,
without
## modifying this makefile, just do:
## $ BITS = 64 make
ifneq (,$(findstring arm,$(shell arch)))
   #arm architecture
   # check whether to generate 32 or 64 bits binaries
   ifeq ($(strip $(BITS)),)
      BITS = $(DEFAULTBITS)
   endif
   MBITS =
else
   #intel architecture
   # check whether to generate 32 or 64 bits binaries
   ifeq ($(strip $(BITS)),)
      BITS = $(DEFAULTBITS)
   endif
   MBITS = -m$(BITS)
endif

ifeq ($(BITS),32)
else
ifeq ($(BITS),64)
else
$(error Wrong architecture. Should be 32 or 64)
endif
endif


# Set OPT_BASE to default path, if it has been not set in exec
parameters
ifeq ($(strip $(OPT_BASE)),)
   OPT_BASE = $(HOME)/opt
endif

OPTCOMMONINSTALLPATH =  $(OPT_BASE)/s5066/sis
OPTLIBINSTALLPATH =  $(OPTCOMMONINSTALLPATH)/lib
OPTINCLUDEINSTALLPATH = $(OPTCOMMONINSTALLPATH)/include
```

```
## Library version
MAJOR = 0
MINOR = 1
LIB_MAJOR_VERSION = $(MAJOR)
LIB_MINOR_VERSION = $(MINOR)

############################################################
## Definitions

MKDIR_P = mkdir -p
DOXYGEN = doxygen
CC = gcc
LDD = gcc
AR = ar
CFLAGS = $(MBITS) -std=c99 $(MACROS) -Wall -pedantic -fsigned-char -I.
LFLAGS = $(MBITS) -L. -L/usr/lib/$(BITS) -L/usr/lib

############################################################
## File names and Makefile internal definitions
DOXYFILE = s5066_sis.doxyfile

LIB_NAME = s5066sis$(BITS)
GENERIC_NAME = lib$(LIB_NAME)
SONAME = $(GENERIC_NAME).so.$(LIB_MAJOR_VERSION)
DYNAMIC_NAME = $(GENERIC_NAME).so.$(LIB_MAJOR_VERSION).$(LIB_MINOR_VERSION)
STATIC_NAME = $(GENERIC_NAME).$(LIB_MAJOR_VERSION).$(LIB_MINOR_VERSION).a
SIS = s5066_sis
TMR = s5066_sis_tmr
SISINTERNAL = s5066_sis_int
############################################################
## Compile rules
# verbatim rule's names
.phony: all docs libs dynamic static clean clean-docs clean-dist
optinstall
# clear all built-in suffix rules
.SUFFIXES:

libs: dynamic static

all: libs docs

docs:
   $(DOXYGEN) $(DOXYFILE)
dynamic: $(DYNAMIC_NAME)

static: $(STATIC_NAME)
```

```
clean-dist: clean clean-docs


## Object file
$(SIS)_$(BITS).o: $(SIS).c $(SIS).h $(SISINTERNAL).h
   $(CC) $(CFLAGS) $(INCPATH) -DSIS_VERSION_MAJOR=$(MAJOR) \
   -DSIS_VERSION_MINOR=$(MINOR) -c $< -o $@ -fPIC

$(TMR)_$(BITS).o: $(TMR).c $(SIS).h $(SISINTERNAL).h
   $(CC) $(CFLAGS) $(INCPATH) -DSIS_VERSION_MAJOR=$(MAJOR) \
   -DSIS_VERSION_MINOR=$(MINOR) -c $< -o $@ -fPIC

## Dynamic library
$(DYNAMIC_NAME): $(SIS)_$(BITS).o $(TMR)_$(BITS).o
   $(LDD) $(CFLAGS) $(LFLAGS) -shared -Wl,-soname,$(SONAME) -o $@ $^
   - ln -sf $(DYNAMIC_NAME) $(SONAME)
   - ln -sf $(SONAME) $(GENERIC_NAME).so

## Static library
$(STATIC_NAME): $(SIS)_$(BITS).o $(TMR)_$(BITS).o
   - $(AR) -rcs $@ $^
   - ln -sf $@ $(GENERIC_NAME).a

optinstall: libs dirs
   cp $(GENERIC_NAME).a ${OPTLIBINSTALLPATH}/
   cp $(STATIC_NAME) ${OPTLIBINSTALLPATH}/
   cp $(GENERIC_NAME).so ${OPTLIBINSTALLPATH}/
   cp $(SONAME) ${OPTLIBINSTALLPATH}/
   cp $(DYNAMIC_NAME) ${OPTLIBINSTALLPATH}/
   cp s5066_sis.h ${OPTINCLUDEINSTALLPATH}

dirs: ${OPTLIBINSTALLPATH} ${OPTINCLUDEINSTALLPATH}

${OPTCOMMONINSTALLPATH}:
   ${MKDIR_P} ${OPTCOMMONINSTALLPATH}
${OPTLIBINSTALLPATH}: ${OPTCOMMONINSTALLPATH}
   ${MKDIR_P} ${OPTLIBINSTALLPATH}
${OPTINCLUDEINSTALLPATH}: ${OPTCOMMONINSTALLPATH}
   ${MKDIR_P} ${OPTINCLUDEINSTALLPATH}


## Cleaning away
clean:
   rm -f $(STATIC_NAME) $(DYNAMIC_NAME) $(SONAME) $(GENERIC_NAME).a \
   $(GENERIC_NAME).so $(SIS)_$(BITS).o $(TMR)_$(BITS).o
clean-docs:
   rm -rf html latex
```

CAS

```
#################################################################
# The architecture to compile against is given by BITS=32 or BITS=64
during
# make invocation (by default 32 bits).

## User defined compile-time options

## Debug and runtime flags
FLAGS_RUN = -O2
#FLAGS_DEBUG = -O0 -g
FLAGS_DEBUG = -O2

MACROS = -D_GNU_SOURCE -D_POSIX_C_SOURCE=199309L
ifndef DEBUGCODE
MACROS += $(FLAGS_RUN) -DNDEBUG
else
ifeq ($(strip $(DEBUGCODE)),0)
MACROS += $(FLAGS_RUN) -DNDEBUG
else
MACROS += $(FLAGS_DEBUG)
endif
endif


DEFAULTBITS = 64
## Architecture to compile for. If you not define BITS outside
Makefile,
## compilation will be for the default value inside the conditional.
## If you want to compile for a given architecture, let's say 64 bits,
without
## modifying this makefile, just do:
## $ BITS = 64 make
ifneq (,$(findstring arm,$(shell arch)))
   #arm architecture
   # check whether to generate 32 or 64 bits binaries
   ifeq ($(strip $(BITS)),)
      BITS = $(DEFAULTBITS)
   endif
   MBITS =
else
   #intel architecture
   # check whether to generate 32 or 64 bits binaries
   ifeq ($(strip $(BITS)),)
      BITS = $(DEFAULTBITS)
   endif
   MBITS = -m$(BITS)
```

```
endif

ifeq ($(BITS),32)
else
ifeq ($(BITS),64)
else
$(error Wrong architecture. Should be 32 or 64)
endif
endif


# Set OPT_BASE to default path, if it has been not set in exec
parameters
ifeq ($(strip $(OPT_BASE)),)
   OPT_BASE = $(HOME)/opt
endif

OPTCOMMONINSTALLPATH =  $(OPT_BASE)/s5066/cas
OPTLIBINSTALLPATH =  $(OPTCOMMONINSTALLPATH)/lib
OPTINCLUDEINSTALLPATH = $(OPTCOMMONINSTALLPATH)/include


BACKUPDATE = $(shell date +%a-%d-%b-%H-%M)

## Library version
MAJOR = 0
MINOR = 1
LIB_MAJOR_VERSION = $(MAJOR)
LIB_MINOR_VERSION = $(MINOR)

################################################################
## Definitions

MKDIR_P = mkdir -p
DOXYGEN = doxygen
CC = gcc
LDD = gcc
AR = ar
CFLAGS = $(MBITS) -std=c99 $(MACROS) -Wall -pedantic -fsigned-char -I.
LFLAGS = $(MBITS) -L. -L/usr/lib/$(BITS) -L/usr/lib

################################################################
## File names and Makefile internal definitions
DOXYFILE = s5066_cas.doxyfile

LIB_NAME = s5066cas$(BITS)
GENERIC_NAME = lib$(LIB_NAME)
SONAME = $(GENERIC_NAME).so.$(LIB_MAJOR_VERSION)
```

```
DYNAMIC_NAME = $(GENERIC_NAME).so.$(LIB_MAJOR_VERSION).$(LIB_MINOR_VERSION)
STATIC_NAME = $(GENERIC_NAME).$(LIB_MAJOR_VERSION).$(LIB_MINOR_VERSION).a
STABLE_FOLDER = ../../stable
BACKUP_FOLDER = backups$(BITS)

CAS = s5066_cas
CASINTERNAL = s5066_cas_int
################################################################################
## Compile rules
# verbatim rule's names
.phony: all docs libs dynamic static clean clean-docs clean-dist
optinstall
# clear all built-in suffix rules
.SUFFIXES:

all: libs docs

docs:
   $(DOXYGEN) $(DOXYFILE)
dynamic: $(DYNAMIC_NAME)

static: $(STATIC_NAME)

libs: dynamic static

clean-dist: clean clean-docs


## Object file
$(CAS)_$(BITS).o: $(CAS).c $(CAS).h $(CASINTERNAL).h
   $(CC) $(CFLAGS) $(INCPATH) -DCAS_VERSION_MAJOR=$(MAJOR) \
   -DCAS_VERSION_MINOR=$(MINOR) -c $< -o $@ -fPIC

## Dynamic library
$(DYNAMIC_NAME): $(CAS)_$(BITS).o
   $(LDD) $(CFLAGS) $(LFLAGS) -shared -Wl,-soname,$(SONAME) -o $@ $^
   - ln -sf $(DYNAMIC_NAME) $(SONAME)
   - ln -sf $(SONAME) $(GENERIC_NAME).so

## Static library
$(STATIC_NAME): $(CAS)_$(BITS).o
   - $(AR) -rcs $@ $^
   - ln -sf $@ $(GENERIC_NAME).a

optinstall: libs dirs
   cp $(GENERIC_NAME).a ${OPTLIBINSTALLPATH}/
   cp $(STATIC_NAME) ${OPTLIBINSTALLPATH}/
   cp $(GENERIC_NAME).so ${OPTLIBINSTALLPATH}/
```

```
    cp $(SONAME) ${OPTLIBINSTALLPATH}/
    cp $(DYNAMIC_NAME) ${OPTLIBINSTALLPATH}/
    cp s5066_cas.h ${OPTINCLUDEINSTALLPATH}

dirs: ${OPTLIBINSTALLPATH} ${OPTINCLUDEINSTALLPATH}

${OPTCOMMONINSTALLPATH}:
    ${MKDIR_P} ${OPTCOMMONINSTALLPATH}
${OPTLIBINSTALLPATH}: ${OPTCOMMONINSTALLPATH}
    ${MKDIR_P} ${OPTLIBINSTALLPATH}
${OPTINCLUDEINSTALLPATH}: ${OPTCOMMONINSTALLPATH}
    ${MKDIR_P} ${OPTINCLUDEINSTALLPATH}

## Cleaning away
clean:
    rm -f $(STATIC_NAME) $(DYNAMIC_NAME) $(SONAME) $(GENERIC_NAME).a \
    $(GENERIC_NAME).so $(CAS)_$(BITS).o
clean-docs:
    rm -rf html latex
```

DTS

```
#################################################################
# The architecture to compile against is given by BITS=32 or BITS=64
during
# make invocation (by default 32 bits).

## User defined compile-time options

## Debug and runtime flags
FLAGS_RUN = -O2
#FLAGS_DEBUG = -O0 -g
FLAGS_DEBUG = -O2

MACROS = -D_GNU_SOURCE -D_POSIX_C_SOURCE=199309L
ifndef DEBUGCODE
MACROS += $(FLAGS_RUN) -DNDEBUG
else
ifeq ($(strip $(DEBUGCODE)),0)
MACROS += $(FLAGS_RUN) -DNDEBUG
else
MACROS += $(FLAGS_DEBUG)
endif
endif
```

```
DEFAULTBITS = 64
## Architecture to compile for. If you not define BITS outside
Makefile,
## compilation will be for the default value inside the conditional.
## If you want to compile for a given architecture, let's say 64 bits,
without
## modifying this makefile, just do:
## $ BITS = 64 make
ifneq (,$(findstring arm,$(shell arch)))
   #arm architecture
   # check whether to generate 32 or 64 bits binaries
   ifeq ($(strip $(BITS)),)
      BITS = $(DEFAULTBITS)
   endif
   MBITS =
else
   #intel architecture
   # check whether to generate 32 or 64 bits binaries
   ifeq ($(strip $(BITS)),)
      BITS = $(DEFAULTBITS)
   endif
   MBITS = -m$(BITS)
endif

ifeq ($(BITS),32)
else
ifeq ($(BITS),64)
else
$(error Wrong architecture. Should be 32 or 64)
endif
endif


# Set OPT_BASE to default path, if it has been not set in exec
parameters
ifeq ($(strip $(OPT_BASE)),)
   OPT_BASE = $(HOME)/opt
endif

OPTCOMMONINSTALLPATH =  $(OPT_BASE)/s5066/dts
OPTLIBINSTALLPATH =  $(OPTCOMMONINSTALLPATH)/lib
OPTINCLUDEINSTALLPATH = $(OPTCOMMONINSTALLPATH)/include


## Library version
MAJOR = 1
MINOR = 0
```

```
LIB_MAJOR_VERSION = $(MAJOR)
LIB_MINOR_VERSION = $(MINOR)

#############################################################
## Definitions

MKDIR_P = mkdir -p
DOXYGEN = doxygen
CC = gcc
LDD = gcc
AR = ar
CFLAGS = $(MBITS) -std=c99 $(MACROS) -Wall -pedantic -fsigned-char -I.
LFLAGS = $(MBITS) -L. -L/usr/lib/$(BITS) -L/usr/lib

#############################################################
## File names and Makefile internal definitions
DOXYFILE = s5066_dts.doxyfile

LIB_NAME = s5066dts$(BITS)
GENERIC_NAME = lib$(LIB_NAME)
SONAME = $(GENERIC_NAME).so.$(LIB_MAJOR_VERSION)
DYNAMIC_NAME = $(GENERIC_NAME).so.$(LIB_MAJOR_VERSION).$(LIB_MINOR_VERSION)
STATIC_NAME = $(GENERIC_NAME).$(LIB_MAJOR_VERSION).$(LIB_MINOR_VERSION).a

ARQ = s5066_dts_arq
BUF = s5066_dts_buf
CRC = s5066_dts_crc
ENC = s5066_dts_enc
DBG = s5066_dts_dbg
EDB = s5066_dts_edb
DEC = s5066_dts_dec
DTS = s5066_dts
DTSINTERNAL = s5066_dts_int
LST = s5066_dts_lst
RXM = s5066_dts_rxm
STM = s5066_dts_stm
TMR = s5066_dts_tmr
TXM = s5066_dts_txm
GRP = s5066_dts_grp
#############################################################
## Compile rules
# verbatim rule's names
.phony: all docs libs dynamic static clean clean-docs clean-dist
directories optinstall
# clear all built-in suffix rules
.SUFFIXES:

libs: dynamic static
```

```
all: libs docs

docs:
	$(DOXYGEN) $(DOXYFILE)

dynamic: $(DYNAMIC_NAME)

static: $(STATIC_NAME)


clean-dist: clean clean-docs

## Object files
$(DTS)_$(BITS).o: $(DTS).c $(DTS).h $(DTSINTERNAL).h
	$(CC) $(CFLAGS) $(INCPATH) -DDTS_VERSION_MAJOR=$(MAJOR) \
	-DDTS_VERSION_MINOR=$(MINOR) -c $< -o $@ -fPIC

$(CRC)_$(BITS).o: $(CRC).c $(CRC).h
	$(CC) $(CFLAGS) $(INCPATH) -DDTS_VERSION_MAJOR=$(MAJOR) \
	-DDTS_VERSION_MINOR=$(MINOR) -c $< -o $@ -fPIC

$(DBG)_$(BITS).o: $(DBG).c $(DBG).h
	$(CC) $(CFLAGS) $(INCPATH) -DDTS_VERSION_MAJOR=$(MAJOR) \
	-DDTS_VERSION_MINOR=$(MINOR) -c $< -o $@ -fPIC

$(EDB)_$(BITS).o: $(EDB).c $(EDB).h
	$(CC) $(CFLAGS) $(INCPATH) -DDTS_VERSION_MAJOR=$(MAJOR) \
	-DDTS_VERSION_MINOR=$(MINOR) -c $< -o $@ -fPIC

$(RXM)_$(BITS).o: $(RXM).c $(RXM).h
	$(CC) $(CFLAGS) $(INCPATH) -DDTS_VERSION_MAJOR=$(MAJOR) \
	-DDTS_VERSION_MINOR=$(MINOR) -c $< -o $@ -fPIC

$(BUF)_$(BITS).o: $(BUF).c $(BUF).h
	$(CC) $(CFLAGS) $(INCPATH) -DDTS_VERSION_MAJOR=$(MAJOR) \
	-DDTS_VERSION_MINOR=$(MINOR) -c $< -o $@ -fPIC

$(TMR)_$(BITS).o: $(TMR).c $(TMR).h
	$(CC) $(CFLAGS) $(INCPATH) -DDTS_VERSION_MAJOR=$(MAJOR) \
	-DDTS_VERSION_MINOR=$(MINOR) -c $< -o $@ -fPIC

$(STM)_$(BITS).o: $(STM).c $(STM).h
	$(CC) $(CFLAGS) $(INCPATH) -DDTS_VERSION_MAJOR=$(MAJOR) \
	-DDTS_VERSION_MINOR=$(MINOR) -c $< -o $@ -fPIC

$(LST)_$(BITS).o: $(LST).c $(LST).h
	$(CC) $(CFLAGS) $(INCPATH) -DDTS_VERSION_MAJOR=$(MAJOR) \
```

```
   -DDTS_VERSION_MINOR=$(MINOR) -c $< -o $@ -fPIC

$(ENC)_$(BITS).o: $(ENC).c $(ENC).h
   $(CC) $(CFLAGS) $(INCPATH) -DDTS_VERSION_MAJOR=$(MAJOR) \
   -DDTS_VERSION_MINOR=$(MINOR) -c $< -o $@ -fPIC

$(DEC)_$(BITS).o: $(DEC).c $(DEC).h
   $(CC) $(CFLAGS) $(INCPATH) -DDTS_VERSION_MAJOR=$(MAJOR) \
   -DDTS_VERSION_MINOR=$(MINOR) -c $< -o $@ -fPIC

$(ARQ)_$(BITS).o: $(ARQ).c $(ARQ).h
   $(CC) $(CFLAGS) $(INCPATH) -DDTS_VERSION_MAJOR=$(MAJOR) \
   -DDTS_VERSION_MINOR=$(MINOR) -c $< -o $@ -fPIC

$(TXM)_$(BITS).o: $(TXM).c $(TXM).h
   $(CC) $(CFLAGS) $(INCPATH) -DDTS_VERSION_MAJOR=$(MAJOR) \
   -DDTS_VERSION_MINOR=$(MINOR) -c $< -o $@ -fPIC

$(GRP)_$(BITS).o: $(GRP).c $(GRP).h
   $(CC) $(CFLAGS) $(INCPATH) -DDTS_VERSION_MAJOR=$(MAJOR) \
   -DDTS_VERSION_MINOR=$(MINOR) -c $< -o $@ -fPIC

## Dynamic library
$(DYNAMIC_NAME):       $(DTS)_$(BITS).o $(CRC)_$(BITS).o $(DBG)_$(BITS).o
$(EDB)_$(BITS).o $(RXM)_$(BITS).o $(BUF)_$(BITS).o $(TMR)_$(BITS).o
$(STM)_$(BITS).o $(LST)_$(BITS).o $(ENC)_$(BITS).o $(DEC)_$(BITS).o
$(ARQ)_$(BITS).o $(TXM)_$(BITS).o $(GRP)_$(BITS).o
   $(LDD) $(CFLAGS) $(LFLAGS) -shared -Wl,-soname,$(SONAME) -o $@ $^
   - ln -sf $(DYNAMIC_NAME) $(SONAME)
   - ln -sf $(SONAME) $(GENERIC_NAME).so

## Static library
$(STATIC_NAME):        $(DTS)_$(BITS).o $(CRC)_$(BITS).o $(DBG)_$(BITS).o
$(EDB)_$(BITS).o $(RXM)_$(BITS).o $(BUF)_$(BITS).o $(TMR)_$(BITS).o
$(STM)_$(BITS).o $(LST)_$(BITS).o $(ENC)_$(BITS).o $(DEC)_$(BITS).o
$(ARQ)_$(BITS).o $(TXM)_$(BITS).o $(GRP)_$(BITS).o
   - $(AR) -rcs $@ $^
   - ln -sf $@ $(GENERIC_NAME).a

optinstall: libs dirs
   cp $(GENERIC_NAME).a ${OPTLIBINSTALLPATH}/
   cp $(STATIC_NAME) ${OPTLIBINSTALLPATH}/
   cp $(GENERIC_NAME).so ${OPTLIBINSTALLPATH}/
   cp $(SONAME) ${OPTLIBINSTALLPATH}/
   cp $(DYNAMIC_NAME) ${OPTLIBINSTALLPATH}/
   cp s5066_dts.h ${OPTINCLUDEINSTALLPATH}

dirs: ${OPTLIBINSTALLPATH} ${OPTINCLUDEINSTALLPATH}
```

```
${OPTCOMMONINSTALLPATH}:
    ${MKDIR_P} ${OPTCOMMONINSTALLPATH}
${OPTLIBINSTALLPATH}: ${OPTCOMMONINSTALLPATH}
    ${MKDIR_P} ${OPTLIBINSTALLPATH}
${OPTINCLUDEINSTALLPATH}: ${OPTCOMMONINSTALLPATH}
    ${MKDIR_P} ${OPTINCLUDEINSTALLPATH}

## Cleaning away
clean:
    rm -f $(STATIC_NAME) $(DYNAMIC_NAME) $(SONAME) $(GENERIC_NAME).a \
    $(GENERIC_NAME).so $(DTS)_$(BITS).o $(CRC)_$(BITS).o $(DBG)_$(BITS).o
$(EDB)_$(BITS).o $(RXM)_$(BITS).o $(BUF)_$(BITS).o $(TMR)_$(BITS).o
$(STM)_$(BITS).o $(LST)_$(BITS).o $(ENC)_$(BITS).o $(DEC)_$(BITS).o
$(ARQ)_$(BITS).o $(TXM)_$(BITS).o $(GRP)_$(BITS).o
clean-docs:
    rm -rf html latex
```