UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Escuela Técnica Superior de Ingeniería Informática

Grado en Ingeniería Informática

Trabajo de Fin de Grado

Diseño de un protocolo de comunicaciones sin pérdidas sobre Ethernet para clústers de computadores

---

Autor: Ignacio Ballester Tester

Tutor: Federico Silla Jiménez

Julio de 2015

Resumen

La pila de protocolos TCP/IP y en particular el protocolo TCP ha sido diseñado para permitir comunicaciones fiables en entornos donde se pueden producir pérdidas de paquetes. En este sentido, el protocolo TCP ha demostrado su gran utilidad en la actual Internet, cuya complejidad ha excedido toda previsión inicial.

No obstante, la pila de protocolos TCP/IP presenta una sobrecarga tanto en tiempo de procesamiento como en espacio de cabeceras que, aunque necesarias en Internet, podría ser redundante en otros entornos más sencillos como el interior de un cluster de computadores. Nótese que en este entorno ni siquiera se contempla el uso de routers.

Este trabajo de fin de grado ha consistido en la implementación de un nuevo protocolo de red para paso de mensajes apoyado en Ethernet, sin más protocolos intermedios, para clusters de computadores y su posterior estudio, comparando el ancho de banda resultante con el de TCP. Una vez estudiados los casos en los que el protocolo implementado podía resultar poco eficiente, se ha trabajado en asegurar que sea totalmente fiable y se ha implementado también, de forma muy básica control de congestión. Al final, se ha conseguido desarrollar un protocolo fiable y que obtiene mayores prestaciones que TCP aunque aún existen lineas de investigación abiertas.

Palabras clave Ethernet, pila de protocolos TCP/IP, redes de área local, ancho de banda

Abstract

The TCP/IP protocol stack and particularly the TCP protocol has been designed to allow reliable communications on environments where packet lost could happen. In this sense the TCP protocol has proved its great usefulness on the Internet, which complexity has exceeded any initial prediction.

However the TCP/IP protocol stack turns up to have an overloading in both, process time and headers size, which although needed on Internet, could be redundant in other simpler environments like a computer cluster. Note that en this environment it's not even consider the use of routers.

This end of degree project involved the implementation of a new network protocol for message passing, supported by Ethernet, without any other intermediate protocol, for computer's clusters and its further study comparing the bandwidth results with TCP. Once studied the cases where the implemented protocol could be inefficient, work has been made to ensure that is completely reliable and has also implemented very basic form of congestion control. In the end, a reliable and with better performance than TCP protocol has successfully been developed, even though there are still open lines of research.

Keywords Ethernet, TCP/IP protocol stack, local area network, bandwidth

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Visión general

Dentro de un *cluster* de computadores es muy común que se deba transferir información entre varios nodos. Esta transferencia se realiza habitualmente utilizando la red Ethernet y la pila de protocolos TCP/IP, ya que es el estándar de comunicaciones más extendido dentro de las redes informáticas. TCP se creó a principios de los años 70 y desde entonces no se ha dejado de investigar para que se pudiera adaptar a las nuevas tecnologías, así como para mejorar sus prestaciones y ampliar sus capacidades para que pudiera funcionar en los nuevos escenarios que se iban planteando, como comunicaciones desde un extremo a otro del planeta o comunicación inalámbrica, entre otros.

En el contexto del interior de un *cluster* de computadores, que en esencia es una red de área local con algunas peculiaridades, no se necesita soporte para adaptar las tramas a conexión inalámbrica, ni que IP introduzca un tiempo de vida, que se decrementa cada vez que la trama pasa por un router, dado que está no va a pasar nunca por el router hacia el exterior con lo que este campo nunca se va a utilizar.

Tampoco se necesitan todos los mecanismos de control de congestión implementados en TCP. Como se puede observar, las comunicaciones dentro del cluster no presentan la complejidad de las comunicaciones en Internet y, por tanto, tampoco es necesaria la complejidad de TCP. De hecho, se podría incluso prescindir también del protocolo IP, dado que los paquetes que se generan no van a atravesar diferentes redes.

Sin embargo, hay que ser realistas y reconocer que una comunicación directamente sobre Ethernet sería contraproducente a menos que se establecieran unos mecanismos de control de flujo básicos. Precisamente, la necesidad de estos mecanismos es una de las principales razones por las que se utiliza la pila de protocolos TCP/IP dentro de un cluster, dado que de esta forma se evita tener que implementarlos.

No obstante, el uso de TCP/IP dentro de un cluster no es gratis, dada la importante sobrecarga que presentan. Por una parte, por la red Ethernet solo pueden transitar tramas con una longitud máxima de 1500 bytes. Cuando se utiliza la pila de protocolos TCP/IP, ambos protocolos ocupan mínimo 20 bytes hasta un máximo de 60 bytes cada uno con sus cabeceras. Con lo que en el mejor de los casos estamos hablando de 40 bytes perdidos en información de control en su mayoría innecesaria para la comunicación que vamos a realizar dentro del cluster.

Por otra parte, además de la sobrecarga en cuanto a espacio dentro de la trama también hay que tener en cuenta la sobrecarga en cuanto a tiempo de procesamiento de los paquetes. Ambas sobrecargas pueden ocasionar una pérdida de prestaciones en la transmisión de volúmenes de datos de tamaño medio o grande.

Por supuesto, hay información en las cabeceras de TCP/IP necesaria para que toda la información llegue a su destino, así como que se pueda reconstruir después de haber sido segmentada, con todo esto, ¿es necesaria toda la potencia de TCP/IP dentro de un *cluster* de computadores?

1.2. Objetivos del trabajo

El objetivo de este trabajo es diseñar e implementar en lenguaje C, un nuevo protocolo de transporte, que garantice la entrega de toda la información que se le suministre así como su reconstrucción en el destino y, que no necesite apoyarse en más protocolos que aquel o aquellos que necesite para la transmisión y recepción de la información. Si conseguimos no utilizar la pila de protocolos TCP/IP, el protocolo va a disponer de la totalidad de la trama Ethernet, en la cual colocará una pequeña cabecera con información de control.

Al no utilizar estos protocolos, también nos libramos del coste que tiene establecer los parámetros de sus cabeceras y el coste intrínseco de TCP, ya que este resulta muy pesado al incluir mucha sobrecarga para conseguir todo lo que ofrece: fiabilidad, entrega en orden, establecimiento de la conexión, etc.

Una vez implementado el protocolo comprobaremos sus prestaciones creando una aplicación muy sencilla que pondremos a prueba utilizando, para el envío de los datos, TCP o el protocolo implementado. En base al valor que consigamos de tiempo de ejecución calcularemos el ancho de banda efectivo para el tamaño del mensaje del cliente sin tener en cuenta el tamaño de la información de control utilizada. Al fin y al cabo, en el caso de TCP tampoco se tiene en cuenta, a la hora de calcular el ancho de banda efectivo la información de control ni los segmentos retransmitidos.

1.3. Estructura del documento

La estructura de este trabajo final de grado es la siguiente:

En el segundo capítulo, ***Internet: arquitectura y prestaciones***, se introduce la arquitectura de protocolos que se utiliza al mandar mensajes a través de Internet y se cuantifican las prestaciones, en función de ancho de banda, que estos alcanzan en los casos de estudio que se detallan en dicho capítulo.

En el tercer capítulo, ***Programa cliente-servidor en TCP***, se detalla la creación de una aplicación, que utilizando TCP, envía información entre el *host* cliente y servidor, y se prueba realizando varias versiones cada una obviando parte de la sobrecarga de TCP, con el fin de posteriormente comparar el protocolo implementado con estas pruebas.

En el cuarto capítulo, ***El protocolo implementado***, se expondrá el diseño teórico del protocolo, así como su implementación primero sobre UDP y una vez visto que resulta rentable,

directamente sobre Ethernet. Seguidamente de cada implementación se muestran las pruebas realizadas y se comparan con las obtenidas anteriormente con TCP.

En el quinto capítulo, ***Conseguir robustez en la fiabilidad***, se plantean mejoras a la hora de conseguir que el protocolo sea fiable en todos los escenarios, se implementan y prueban.

En el sexto capítulo, ***Implementando control de congestión***, se describirá la forma de introducir vagamente control de congestión en el protocolo implementado y se demostrará su efectividad.

En el último capítulo, ***Conclusiones***, se valorarán los datos obtenidos en los capítulos anteriores y se propondrán posibles líneas futuras de investigación.

Capítulo 2

Internet: arquitectura y prestaciones

2.1. Arquitectura en capas de protocolos

Generalmente para proporcionar estructura al diseño de protocolos de red, se organizan los protocolos (y el hardware y el software de red que los implementan) en capas. Cada protocolo pertenece a una capa. Una capa ofrece una serie de servicios a la capa que se encuentra inmediatamente por encima de ésta, para ello realizan ciertas acciones en dicha capa al tiempo que se utilizan los servicios que ofrece la capa que tiene directamente debajo de ella. Las capas de protocolos presentan ventajas conceptuales y estructurales. La estructuración de los protocolos de red en capas presenta una forma estructurada de estudiar y organizar los componentes del sistema y, además, la modularidad facilita la actualización de los componentes del sistema sin que se tengan que modificar el resto de capas.

Cuando los protocolos de las distintas capas se toman en conjunto se habla de pila de protocolos. La pila de protocolos de Internet consta de cinco capas: capa de aplicación, capa de transporte, capa de red, capa de enlace y capa física; como se muestra en la Figura 2.1. A continuación, se va a explicar cada capa siguiendo un enfoque ascendente.

Figura 2.1: Pila de protocolos de Internet

2.1.1. Capa física

La capa física se encarga de transportar los bits individuales de un nodo de la red al siguiente. Mediante la variación de propiedades físicas, como son el voltaje o la corriente, se consigue

enviar datos mediante un cable. El comportamiento de la señal se puede representar matemáticamente, mediante series de Fourier. La intensidad de las variaciones en el voltaje así como el lapso entre cada una de ellas depende del cable por el que se transmite la señal.

Es por esto que, los protocolos que operan en esta capa dependen del enlace y, por tanto, dependen del medio de transmisión del enlace. Por ejemplo, es el protocolo de la capa de enlace, en este caso Ethernet, el que selecciona el protocolo de la capa física. Ethernet dispone de muchos protocolos de la capa física: uno para cable de cobre de par trenzado, otro para cable coaxial, otro para fibra óptica, etc. En cada caso, los bits se desplazan a través del enlace de forma distinta.

2.1.2. Capa de enlace de datos

Los paquetes que transitan por la red de Internet pasan a través de una serie de routers desde que salen de su origen hasta que llegan a su destino. Para trasladar un paquete de un nodo al siguiente, la capa de red confía en los servicios de la capa de enlace. Los servicios proporcionados por esta capa dependen del protocolo que se emplee en el enlace: hay protocolos que proporcionan entrega fiable a nivel de enlace entre el nodo transmisor y el nodo receptor, otros están diseñados para detectar colisiones de paquetes, etc. Entre los protocolos de la capa de enlace se incluyen IEEE 802.3 (Ethernet) e IEEE 802.11 (WiFi).

Ethernet

Apareció a mediados de la década de 1970 y ha ido evolucionando y creciendo, manteniéndose dominante sobre otras tecnologías LAN (Token Ring, FDDI, ATM). Puede decirse que Ethernet ha sido a las redes de área local lo que Internet a las redes globales. Ethernet fue la primera LAN de alta velocidad ampliamente implantada. Más barata y simple que el resto de tecnologías, y, además, se defendía produciendo versiones que operaban a velocidades iguales o incluso mayores.

Examinemos la trama que utiliza Ethernet:

Figura 2.2: Formato de una trama Ethernet

- *Preámbulo*. (8 bytes). La trama Ethernet comienza con este campo de 8 bytes. Cada uno de los siete primeros bytes tiene el valor 10101010 y el último byte tiene el valor 10101011. Los siete primeros bytes sirven para "despertar" a los adaptadores de recepción y sincronizar sus relojes con el reloj del emisor. Los últimos 2 bits del octavo byte del preámbulo alertan al adaptador receptor de que va a llegar "información importante".
- *Dirección de destino*. (6 bytes). Contiene la dirección IEEE conocida común mente como MAC, del adaptador de destino. En estás direcciones los primeros tres bytes identifican a

la organización fabricante de la tarjeta y los tres últimos identifican a la tarjeta de red.

- *Dirección de origen*. (6 bytes). Contiene la dirección IEEE del adaptador que transmite la trama hacia la LAN.
- *Campo de tipo*. (2 bytes). Permite a Ethernet multiplexar los protocolos de la capa de red. Este campo indica que protocolo de la capa superior se ha encapsulado en la trama Ethernet.
- *Campo de datos*. (46 a 1500 bytes). Este campo transporta el datagrama IP. La unidad máxima de transmisión de Ethernet es de 1500 bytes y el tamaño mínimo es de 46 bytes. En caso de intentar enviarse una trama de longitud menor a 46 bytes, esta se rellena hasta tener el tamaño mínimo.
- *Comprobación de redundancia cíclica*. (4 bytes). El propósito del campo CRC es permitir que el adaptador del receptor detecte los errores de bit que se hayan podido producir en la trama, en su recorrido entre transmisor y receptor. Notese que, no se puede verificar plenamente que los datos no han sido manipulados, porque CRC es incapaz de detectar si se han producido cambios deliberados y no aleatorios. Así, CRC resulta útil para verificar la integridad del mensaje pero no para saber si este es correcto, no obstante este no es el objetivo de este campo de comprobación. Para este fin existen protocolos en las capas altas de la pila de protocolos.

2.1.3. Capa de red

La capa de red recibe de la capa de transporte un paquete y una dirección destino, y proporciona el servicio de suministrar ese paquete a la capa de transporte del host destino. Los protocolos de esta capa permiten el direccionamiento lógico de los paquetes, es decir, se encargan de que los paquetes viajen a través de las distintas subredes hasta llegar a su destino. Incluso se encarga de descartar los paquetes que se han perdido, que llevan demasiado tiempo en la red, etc.

Los paquetes de esta capa reciben a menudo el nombre de datagramas e incluye, entre sus protocolos, el protocolo IP y suele hacerse referencia a ella simplemente como capa IP. Pero contiene multitud de protocolos de enrutamiento que determinan las rutas que los datagramas siguen entre los orígenes y los destinos.

IP

IP son las siglas para *Internet Protocol*. El formato de los datagramas de IPv4, el estándar de este protocolo que se utiliza actualmente aunque se está migrando al estandar IPv6, se muestra en la Figura 2.3. Los campos clave de la cabecera de los datagramas de IP son los siguientes:

- *Número de versión*. 4 bits que especifican la versión del protocolo IP del datagrama. A partir de aquí, el router y los computadores que reciben el datagrama puede determinar cómo interpretar el resto del datagrama IP.

Figura 2.3: Cabecera del protocolo IP

- *Longitud de cabecera*. 4 bits para determinar dónde comienzan realmente los datos del datagrama IP. Normalmente el tamaño de una cabecera IP es de 20 bytes. Este campo se expresa en palabras de 32 bits. Es decir, para un tama no de cabecera de 20 bytes, este campo contiene un valor igual a 5.

- *Tipo de servicio*. Para poder diferenciar entre distintos tipos de datagramas IP.

- *Longitud del datagrama*. Longitud total del datagrama IP. Este campo tiene una longitud de 16 bits, por tanto, el tamaño máximo teórico es de 65.535 bytes, aunque rara vez sobrepasa los 1.500 bytes dada la limitación que impone Ethernet.

- *Identificador, flags, desplazamientos de fragmentación*. Son parámetros relativos a la fragmentación de datagramas, cosa que ha sido eliminada en la versión 6 de IP, dada la sobrecarga que genera en los routers.

- *Tiempo de vida*. Para garantizar que los datagramas no circulan eternamente por la red si se pierden, este campo se decrementa cada vez que el paquete pasa por un router. Si el nuevo valor es igual a cero, entonces el datagrama se descarta..

- *Protocolo*. El valor de este campo indica el protocolo específico de la capa de transporte al que se pasarán los datos contenidos en el datagrama IP.

- *Suma de comprobación de cabecera*. Ayuda a los elementos de la red a detectar errores de bit en la cabecera de un datagrama IP, pero no sirve para comprobar la integridad de los datos.

- *Direcciones IP de origen y destino*. Identifican al nodo emisor del datagrama y al receptor del mismo respectivamente.

- *Opciones*. Permite ampliar la cabecera IP, introduciendo información de control que no tiene cabida en la cabecera fija. Su longitud puede ser de cero bits.
- *Relleno*. En caso de que el tamaño del campo opciones sea mayor que cero, este campo se encarga de completarlo para que su longitud sea múltiplo de 32 bits.
- *Datos*. Contiene el segmento de la capa de transporte que va a entregarse al destino. Puede transportar datagramas UDP o TCP así como otro tipo de protocolos, como mensajes ICMP, que es un subprotocolo de IP para el control y la notificación de errores.

2.1.4. Capa de transporte

La capa de transporte de Internet transporta los mensajes de la capa de aplicación de un nodo terminal de la aplicación a otro. Los protocolos de esta capa se comunican directamente con el protocolo IP, el cual, como ya se ha mencionado no garantiza la entrega de los paquetes, ni la entrega en orden de los mismos y tampoco garantiza la integridad de los datos contenidos en ellos.

En Internet existen dos protocolos de transporte, TCP y UDP. Los paquetes, intercambiados mediante TCP se denominan segmentos, mientras que en el caso de UDP, la únidad de transferencia se denomina datagrama. Ambos protocolos, extienden la entrega host a host, que debe proporcionar un protocolo de esta capa, a una entrega proceso a proceso mediante lo que se denomina multiplexación y demultiplexación de la capa de transporte, además de proporcionar un servicio de comprobación de la integridad de los datos del mensaje contenido en el datagrama IP, al incluir campos de detección de errores en sus respectivas cabeceras, que al contrario que IP verifican tanto la integridad de la cabecera como de los datos.

El concepto de entrega proceso a proceso implica que un host puede tener varios *sockets* abiertos simultáneamente escuchando a distintas comunicaciones o enviando información a diversos destinos. Esta funcionalidad se implementa mediante lo que se denominan puertos. Un puerto, en general un número entre 0 y 65535 ya que los protocolos utilizan 16 bytes sin signo para representar el puerto, permite otorgar un identificador a cada comunicación. Permitiendo así que varías comunicaciones se sucedan simultáneamente sin que una interfiera con la otra. Los números de puerto son conocidos y están restringidos para ser utilizados por el protocolo de aplicación correspondiente.

TCP

TCP es un protocolo fiable, que garantiza la entrega y la entrega en orden; y orientado a la conexión, esto quiere decir que antes de que un proceso de la capa de aplicación pueda enviar datos a otro, los dos *hosts* deben primero “establecer una comunicación” entre ellos. Esto se realiza enviando varios segmentos TCP preliminares para definir los parámetros de la transferencia de datos que se va a realizar, así como iniciar las variables de TCP asociadas con la conexión.

Dado que el protocolo TCP se ejecuta solo en los sistemas terminales, que son los únicos que acceden a los datos de la capa de transporte contenida en la trama, los elementos intermedios

de la red (routers y switches) no mantienen el estado de la conexión TCP. Ambos elementos de la red son completamente inconscientes de la existencia de una conexión TCP. Por su lado un switch solo analiza la capa de enlace y un router analiza además la capa de red.

TCP sigue un modelo de protocolo en tres fases para iniciar una conexión. Según este modelo el emisor envía al receptor un segmento especial sin datos de aplicación, el receptor cuando recibe este segmento, asigna los *buffers* y variables TCP a la conexión y envía al emisor un nuevo segmento especial indicando que se ha establecido la conexión. Al recibir este segmento el cliente asigna sus propios *buffers* y variables y envía un nuevo segmento indicando que confirma el establecimiento de la conexión, este segmento puede contener datos de aplicación.

Una vez completados estos tres pasos, cliente y servidor pueden enviarse segmentos con datos de la capa de aplicación el uno al otro, cuando el receptor ha recibido un número determinado de segmentos correctamente, notifica al cliente envíandole un paquete de reconocimiento. Hasta que uno de los dos desee terminar la comunicación. El protocolo seguido para cerrar una conexión TCP es el siguiente: el *host* que desea terminar la conexión envía al otro host un segmento especial indicando el fin de la conexión. El *host* que recibe este segmento envía al *host* que lo ha emitido un segmento de reconocimiento del fin de la conexión y a continuación envía un segmento de fin de conexión al otro equipo.

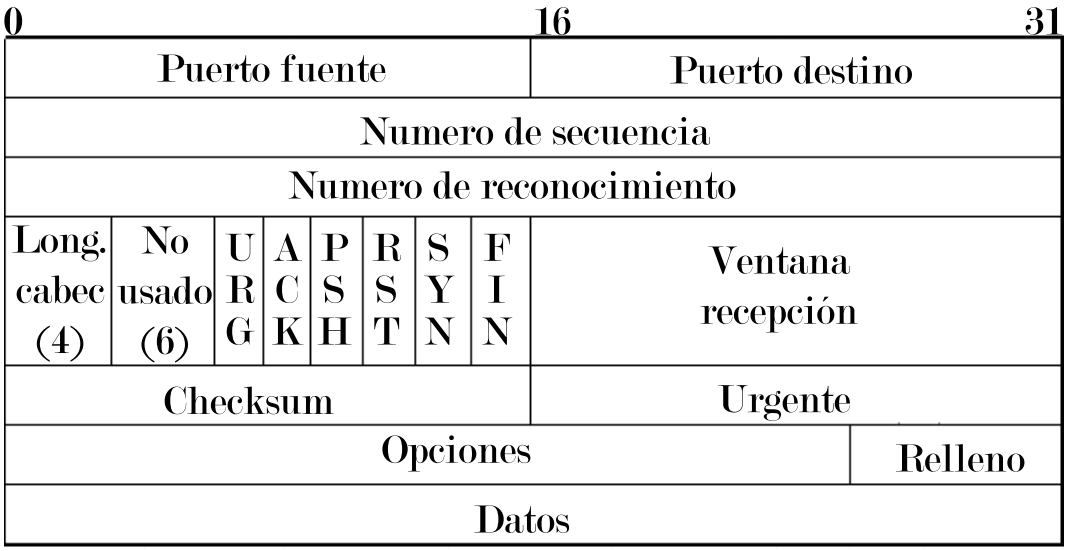

Figura 2.4: Cabecera del protocolo TCP

Acontinuación analizaremos los campos de la cabecera de TCP que se puede observar en la Figura 2.4:

- *Puerto fuente*. (16 bits). Identifica el puerto emisor.
- *Puerto destino*. (16 bits). Identifica el puerto receptor.
- *Número de secuencia*. (32 bits). Identifica con que byte del mensaje que se está enviando, se corresponde el primer byte de datos del segumento.

- *Número de reconocimiento*. (32 bits). Contiene el valor del siguiente número de secuencia que el servidor espera recibir.
- *Longitud de cabecera*. (4 bits). Especifica el tamaño de la cabecera en palabras de 32 bits.
- *No usado*. (3 bits). Para uso futuro. Debe estar a 0.
- *URG*. (1 bit). Indica que el campo del puntero a datos urgente es válido.
- *ACK*. (1 bit). Indica que el campo de reconocimiento es válido.
- *PSH*. (1 bit). Push. El receptor debe pasar los datos a la aplicación tan pronto como sea posible.
- *RST*. (1 bit). Reset. Reinicia la conexión cuando falla un intento de conexión, o al rechazar paquetes no válidos.
- *SYN*. (1 bit). Synchronice. Sincroniza los números de secuencia para iniciar la conexión.
- *FIN*. (1 bit). Para que el emisor solicite la liberación de la conexión.
- *Ventana recepción*. (16 bits). Indicar el número de bytes que un receptor está dispuesto a aceptar.
- *Checksum*. (16 bits). Utilizado para la comprobación de errores tanto en la cabecera como en los datos.
- *Urgente*. (16 bits). Indica la posición del último byte de datos marcado como urgente.
- *Opciones*. Para poder añadir características no cubiertas por la cabecera fija.
- *Relleno*. Para asegurarse que la cabecera acaba con un tamaño múltiplo de 32 bits.
- *Datos*.

UDP

UDP es un protocolo, de la capa de transporte, simple que realiza las tareas mínimas que debe de hacer un protocolo de esta capa. A parte de la función de multiplexación/demultiplexación y de un mecanismo de comprobación de errores, no añade nada a IP. Básicamente, UDP proporciona un servicio sin conexión no fiable a la aplicación que lo utiliza. DNS es un ejemplo de protocolo de la capa de aplicación que utiliza UDP. ¿Pero por qué utilizar UDP en vez de TCP, si de esta forma obtendríamos un servicio fiable? Existen diversas razones que aportan sentido al uso de UDP frente a TCP, simplemente porque UDP se adapta mejor a sus necesidades, no porque uno sea mejor que el otro:

- Mejor control en el nivel de aplicación de cuándo se envían los datos. UDP empaqueta los datos de la capa de aplicación en el momento en que los recibe e inmediatamente entrega el datagrama a la capa de red. Esta característica es no solo deseable sino necesaria en

aplicaciones en tiempo real, y resulta preferible implementar un sistema de tolerancia ante la pérdida de tramas que sufrir el retardo que puede producirse al enviar segmentos TCP.

- Sin establecimiento de la conexión. UDP inicia la transmisión mediante origen y destino sin preliminares y por tanto no sufre ningún retardo a causa del establecimiento de la conexión.
- Sin información del estado de la transmisión. TCP mantiene información del estado de la conexión, de los *buffers* de envío y recepción, de los parámetros de control de la congestión y de secuencia, etc, para poder ofrecer un servicio fiable. Por el contrario, UDP no mantiene información sobre el estado de la transmisión, ni ninguno de estos parámetros y por tanto no sufre el sobre coste que si padece TCP.
- Poca sobrecarga a causa de la cabecera de los paquetes. Los segmentos UDP solo requieren de una cabecera de 8 bytes, mientras que los de TCP contienen una cabecera de 20 bytes.
- Poca sobrecarga en lo referente al procesamiento de los datagramas UDP. Dado que UDP es tan sencillo, la recepción de uno de estos paquetes lleva asociadas muy pocas acciones, lo que conlleva una sobrecarga temporal baja.

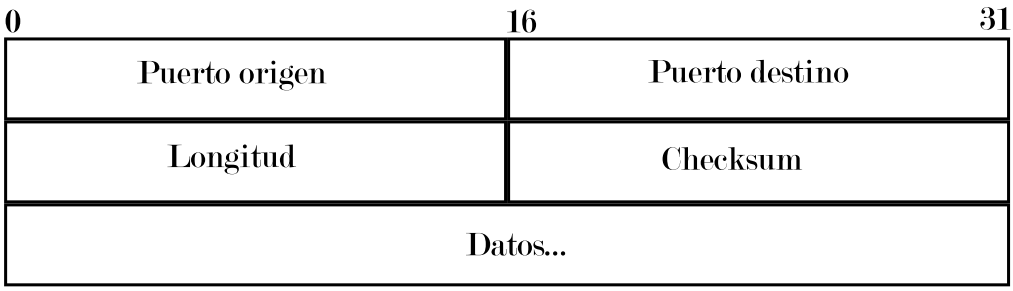

Figura 2.5: Cabecera del protocolo UDP

A continuación se explican los campos de la cabecera de UDP que se puede observar en la Figura 2.5:

- *Puerto origen*. (16 bits). Identifica el proceso de origen.
- *Puerto destino*. (16 bits). Identifica el proceso de recepción. Este campo junto al anterior se utilizan para ofrecer comunicación proceso a proceso.
- *Longitud*. (16 bits). Especifica la longitud del segmento UDP en bytes, incluyendo la cabecera.
- *Checksum*. (16 bits). Para detectar si se han introducido errores en el segmento. El emisor aplica un algoritmo de suma sobre los datos y escribe el valor obtenido en este campo. Posteriormente el receptor aplica el mismo algoritmo sobre los datos que ha recibido y

compara el resultado con el valor contenido en el campo de la cabecera; si ambos valores son iguales se puede suponer que los datos no se han visto modificados, pero si por el contrario los valores no son idénticos, los datos se han modificado.

- *Datos*. Datos de la aplicación.

Aunque UDP ofrece un mecanismo de detección de errores, mediante el campo *Checksum*, no ofrece ningún tipo de algoritmo para recuperarse de esto, por lo que la mayoría de las implementaciones de UDP directamente descartan las tramas que no pasan el test. Precisamente, esta falta de recuperación automática frente a errores en la transmisión hace que muchas aplicaciones utilizan TCP.

2.1.5. Capa de aplicación

La capa de aplicación incluye multitud de protocolos, tales como el protocolo HTTP (que permite la solicitud y transferencia de documentos web), SMTP (que permite el envío de mensajes de correo electrónico), FTP (que permite la transferencia de archivos entre dos nodos terminales) y DNS (que permite la traducción de los nombres de dominio "legibles" de por ejemplo, las páginas web, a direcciones de red manejables por IP). A los paquetes de información de la capa de aplicación se les denomina mensajes.

2.2. Prestaciones de TCP y UDP

A día de hoy existen multitud de estándares de Ethernet cada uno de ellos ofrece una velocidad de transferencia máxima, desde las primeras versiones de 10 Mbits por segundo, versiones para cable coaxial hasta llegar a las versiones desarrolladas para fibra óptica. Entre las implementaciones más potentes se encuentra el actual Gigabit Ethernet que ofrece una tasa de datos en bruto de 1.000 Mbps y cuenta con compatibilidad para las versiones anteriores. Esta versión de 1 Gbps, aunque quizá sea la más extendida hoy en día, no es en cambio la de más prestaciones, dado que existen otras versiones que funcionan a 40 Gbps.

Las pruebas que vamos a realizar van a ser con ordenadores que cuentan con tarjetas de red preparadas para funcionar a 1 Gbit Ethernet en bruto, pero sería de mucha utilidad saber cual es el ancho de banda neto de la red, tanto si utilizamos TCP como si utilizamos UDP, para cuando comparemos las prestaciones de lo desarrollado con las tecnologías que ya existen. Para esto vamos a utilizar un programa llamado "ipref" [10], concretamente en su tercera versión estable [9], que es una herramienta que permite medir el rendimiento de la red con ambos protocolos.

Para utilizar iperf se requiere de dos ordenadores conectados a través de cable de red a la misma red de área local. Un nodo se establece como el servidor ejecutando en la terminal el comando "iperf3 -s", y el otro como cliente ejecutando el comando "iperf3 -c [IP]", si quiere una conexión TCP, o el comando "iperf3 -c [IP] -u -b 10G" si quiere hacer la comunicación mediante UDP. IP es la dirección IP del servidor de la forma "192.168.0.1". El parámetro "-u" indica a iperf que la comunicación se realiza mediante UDP la cual, por defecto no utiliza todo el ancho de banda de la red, es por eso que mediante el parámetro "-b 10G" le indicamos a iperf

que envíe las tramas utilizando un ancho de banda máximo de 10Gbps, dado que el ancho de banda máximo por defecto en UDP es de 1 Mbit por segundo. Imponiendo este ancho de banda máximo conseguimos que se envíen los paquetes empleando la potencia máxima de la red.

```
[igbaltes@jp67 ~]$ iperf3 -s
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from 192.168.0.2, port 33681
[  5] local 192.168.0.1 port 5201 connected to 192.168.0.2 port 33682
[ ID] Interval           Transfer     Bandwidth
[  5]   0.00-1.00   sec   108 MBytes   906 Mbits/sec
[  5]   1.00-2.00   sec   112 MBytes   940 Mbits/sec
[  5]   2.00-3.00   sec   112 MBytes   943 Mbits/sec
[  5]   3.00-4.00   sec   112 MBytes   941 Mbits/sec
[  5]   4.00-5.00   sec   112 MBytes   941 Mbits/sec
[  5]   5.00-6.00   sec   112 MBytes   941 Mbits/sec
[  5]   6.00-7.00   sec   112 MBytes   941 Mbits/sec
[  5]   7.00-8.00   sec   112 MBytes   941 Mbits/sec
[  5]   8.00-9.00   sec   112 MBytes   941 Mbits/sec
[  5]   9.00-10.00  sec   112 MBytes   941 Mbits/sec
[  5]  10.00-10.04  sec  4.12 MBytes   940 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  5]   0.00-10.04  sec  1.10 GBytes   939 Mbits/sec   41             sender
[  5]   0.00-10.04  sec  1.10 GBytes   938 Mbits/sec                  receiver
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figura 2.6: Servidor iperf TCP entre los AMD Opteron

En la Figura 2.6 y la Figura 2.7 podemos ver una ejecución de iperf utilizando TCP entre equipos con procesador AMD. Estos equipos cuentan con cuatro procesadores AMD Barcelona a 2.2GHz y 16 GBytes de memoria RAM. La primera figura muestra la salida de este programa en el servidor, mientras que la segunda muestra la salida en el cliente. El receptor recibe cada segundo 112 MBytes de datos enviados por el cliente, al añadir a este valor el tamaño de las cabeceras llegamos a un ancho de banda aproximado de 940 Mbits/s. La columna "retr" indica el número de retransmisiones que TCP ha realizado y la de "cwnd" indica el tamaño de la ventana de congestión de TCP. El servidor da menos información que el cliente, dado que solo aporta información de los bytes recibidos y del ancho de banda. Las dos últimas filas de ambos procesos muestran un resumen de la comunicación tanto desde el punto de vista del emisor como del receptor, ofreciendo el total de bytes transferidos y el ancho de banda medio.

Por otra parte cuando se utiliza UDP en lugar de TCP, en la Figura 2.8 y la Figura 2.9 se puede ver la salida del servidor y el cliente respectivamente, se observa una diferencia en los datos que se muestran frente al otro protocolo. El cliente en esta ocasión informa del número total de datagramas UDP enviados cada segundo. Por su parte, el servidor informa del *jitter*, la variación de la latencia y de los datagramas perdidos y los totales que se debían recibir.

- A modo de resumen, la comunicación entre los dos ordenadores AMDs consigue:
  - Utilizando TCP un ancho de banda máximo de aproximadamente 940 Mbps. De un máximo ofrecido por la red de 1 Gbit/s, TCP alcanza casi el total, perdiendo solo 60 Mbps.

```
[igbaltes@jp66 ~]$ iperf3 -c 192.168.0.1
Connecting to host 192.168.0.1, port 5201
[  4] local 192.168.0.2 port 33682 connected to 192.168.0.1 port 5201
[ ID] Interval           Transfer     Bandwidth       Retr  Cwnd
[  4]   0.00-1.00   sec   113 MBytes   949 Mbits/sec    0    437 KBytes
[  4]   1.00-2.00   sec   112 MBytes   942 Mbits/sec    1    245 KBytes
[  4]   2.00-3.00   sec   112 MBytes   943 Mbits/sec    5    117 KBytes
[  4]   3.00-4.00   sec   112 MBytes   940 Mbits/sec    5    132 KBytes
[  4]   4.00-5.00   sec   112 MBytes   943 Mbits/sec    5    127 KBytes
[  4]   5.00-6.00   sec   112 MBytes   943 Mbits/sec    5    143 KBytes
[  4]   6.00-7.00   sec   112 MBytes   938 Mbits/sec    5    150 KBytes
[  4]   7.00-8.00   sec   113 MBytes   941 Mbits/sec    5    116 KBytes
[  4]   8.00-9.00   sec   112 MBytes   942 Mbits/sec    5    127 KBytes
[  4]   9.00-10.00  sec   112 MBytes   944 Mbits/sec    5    137 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-10.00  sec  1.10 GBytes   942 Mbits/sec   41             sender
[  4]   0.00-10.00  sec  1.10 GBytes   942 Mbits/sec                  receiver

iperf Done.
[igbaltes@jp66 ~]$
```

Figura 2.7: Cliente iperf TCP entre los AMD Opteron

- Mientras tanto, utilizando UDP conseguimos un ancho de banda de casi 960 Mbps, perdiendo solo 40 Mbps con respecto al máximo de la red y perdiéndose 45 de los 144775 datagramas que se han enviado en total.

```
igbaltes@jp74:~> iperf3 -s
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
Accepted connection from 192.168.0.1, port 49166
[  5] local 192.168.0.3 port 5201 connected to 192.168.0.1 port 33287
[ ID] Interval           Transfer     Bandwidth       Jitter    Lost/Total Datagrams
[  5]   0.00-1.00   sec  24.5 MBytes   206 Mbits/sec  0.091 ms  9435/12576 (75%)
[  5]   1.00-2.00   sec  28.4 MBytes   239 Mbits/sec  0.100 ms  10976/14617 (75%)
[  5]   2.00-3.00   sec  28.0 MBytes   235 Mbits/sec  0.095 ms  11044/14633 (75%)
[  5]   3.00-4.00   sec  28.2 MBytes   237 Mbits/sec  0.087 ms  11016/14625 (75%)
[  5]   4.00-5.00   sec  27.9 MBytes   234 Mbits/sec  0.094 ms  11047/14619 (76%)
[  5]   5.00-6.00   sec  27.9 MBytes   234 Mbits/sec  0.130 ms  11051/14622 (76%)
[  5]   6.00-7.00   sec  27.9 MBytes   234 Mbits/sec  0.141 ms  11065/14631 (76%)
[  5]   7.00-8.00   sec  28.1 MBytes   235 Mbits/sec  0.068 ms  11031/14624 (75%)
[  5]   8.00-9.00   sec  27.7 MBytes   233 Mbits/sec  0.095 ms  11070/14619 (76%)
[  5]   9.00-10.00  sec  27.9 MBytes   234 Mbits/sec  0.078 ms  11054/14631 (76%)
[  5]  10.00-10.04  sec  1.00 MBytes   226 Mbits/sec  0.104 ms  414/542 (76%)
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Jitter    Lost/Total Datagrams
[  5]   0.00-10.04  sec  1.10 GBytes   945 Mbits/sec  0.104 ms  109203/144739 (75%)
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figura 2.8: Servidor iperf UDP entre el AMD (cliente) y el Pentium III (servidor)

Con el fin de analizar de forma más completa el coste computacional de los protocolos TCP y UDP, vamos a realizar una prueba similar en la que uno de los nodos, va a ser reemplazado por un equipo más viejo (y por tanto menos potente), exactamente un ordenador con procesador Pentium III a 1GHz y con 1GB de memoria RAM. En este escenario, el equipo con procesador AMD será primero el cliente mientras que el nodo con procesador Pentium hará el papel de servidor. Posteriormente, intercambiaremos los papeles de los nodos.

```
[igbaltes@jp67 ~]$ iperf3 -c 192.168.0.3 -u -b 1G
Connecting to host 192.168.0.3, port 5201
[  4] local 192.168.0.1 port 33287 connected to 192.168.0.3 port 5201
[ ID] Interval           Transfer     Bandwidth       Total Datagrams
[  4]   0.00-1.00   sec   103 MBytes   863 Mbits/sec  13166
[  4]   1.00-2.00   sec   114 MBytes   958 Mbits/sec  14623
[  4]   2.00-3.00   sec   114 MBytes   958 Mbits/sec  14625
[  4]   3.00-4.00   sec   114 MBytes   958 Mbits/sec  14623
[  4]   4.00-5.00   sec   114 MBytes   958 Mbits/sec  14622
[  4]   5.00-6.00   sec   114 MBytes   959 Mbits/sec  14625
[  4]   6.00-7.00   sec   114 MBytes   958 Mbits/sec  14623
[  4]   7.00-8.00   sec   114 MBytes   958 Mbits/sec  14622
[  4]   8.00-9.00   sec   114 MBytes   958 Mbits/sec  14623
[  4]   9.00-10.00  sec   114 MBytes   958 Mbits/sec  14625
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Jitter    Lost/Total Datagrams
[  4]   0.00-10.00  sec  1.10 GBytes   949 Mbits/sec  0.104 ms  109203/144739 (75%)
[  4] Sent 144739 datagrams

iperf Done.
[igbaltes@jp67 ~]$
```

Figura 2.9: Cliente iperf UDP entre el AMD (cliente) y el Pentium III (servidor)

La comunicación entre un AMD y el Pentium III siendo el Pentium el servidor consigue:

- 570 Mbits por segundo utilizando TCP, mostrando de nuevo que utilizar TCP resulta en una sobrecarga excesiva para el Pentium.
- Utilizando UDP, el caso que muestran la Figura 2.8 y la Figura 2.9, nos damos cuenta de los problemas que puede suponer tener un equipo lento como receptor. La perdida de 109203 datagramas de los 144739 totales que se han enviado hacen que el ancho de banda real en el servidor sea de tan solo 230 Mbps a pesar del hecho de que ambas ejecuciones muestran como ancho de banda el valor percibido por el cliente en su envío.

La comunicación entre un AMD y el Pentium III siendo el AMD el servidor alcanza:

- Los 770 Mbps utilizando TCP, dejando claro que el Pentium III es mucho menos potente, por lo que le cuesta más tiempo construir las cabeceras así como mantener la conexión TCP.
- Mientras tanto utilizando UDP conseguimos un ancho de banda de 950 Mbits/s. Vemos que el Pentium III sí que es capaz de transmitir tramas casi igual de rápido que uno de los AMD, cuando el protocolo que se utiliza no tiene un sobrecoste elevado.

Finalmente hemos completado este estudio utilizando ordenadores modernos basados en procesadores Xeon de segunda generación (ES-2620v2 a 2,2 GHz y 32 GB de memoria RAM) y también procesadores ATOM de última generación (C2750 a 2,4 GHz y 32 GB de memoria RAM). En este caso, las comunicaciones entre los Atoms y los Xeon son muy parecidas entre sí, siendo ambos equipos de características similares:

- Utilizando TCP como protocolo para la comunicación se consigue un ancho de banda de 940 Mbits/s, casi idéntico al valor máximo que obteníamos con los AMD.

- Mientras, con UDP se consigue un ancho de banda de 955 Mbits/s, un poco menos que en los AMDs, con la salvedad de que estos equipos no sufren ninguna perdida de tramas.

2.3. Conclusiones

En este capítulo se ha hecho un resumen de la pila de protocolos TCP/IP y se han analizado las prestaciones de TCP y UDP en diversos entornos. Las pruebas realizadas indican de forma clara que el protocolo TCP presenta una sobrecarga de procesamiento importante, que puede llegar incluso a reducir de forma notable las prestaciones de las comunicaciones para aquellos procesadores menos potentes. En el caso de UDP, se ha comprobado que las prestaciones se pueden llegar a mantener bajo determinadas circunstancias, aunque la perdida de paquetes hace que las prestaciones también se reduzcan.

Nótese que en estas pruebas se ha usado un viejo procesador (Pentium III) para modelar el caso de procesador poco potente. No obstante, hay numerosos ejemplos de procesadores actuales con potencia reducida, como pueden ser todos aquellos procesadores para entornos embebidos. Por otra parte, nótese también que otra posibilidad en lugar de reducir la potencia de los procesadores es utilizar una red más potente junto a procesadores modernos. Por ejemplo, podría plantearse cuál sería la pérdida de prestaciones cuando un procesador actual utiliza el protocolo TCP en redes de altas prestaciones que proporcionan ancho de banda de 40, 56 o 100 Gbps.

Capítulo 3

Programa cliente-servidor en TCP

El objetivo del estudio realizado en este trabajo final de grado, es diseñar e implementar un nuevo protocolo de transporte, para utilizarse en redes de área local. Pero además de esto se han de medir sus prestaciones y comparar con el estándar (IP y TCP). Para esto se ha implementado un sencillo programa en lenguaje C, una aplicación de emisor-receptor simple que utiliza el protocolo TCP. Las prestaciones que obtengamos con este programa se compararán posteriormente con las de los desarrollos realizados.

En el lado del cliente se han de suministrar cuatro parámetros: la IP del receptor, el puerto en el que se ha de realizar la comunicación, un puntero a la zona de memoria donde se encuentra la información que se debe enviar y un número entero que indica el tamaño en bytes de la información a enviar. Por su lado, al receptor es necesario especificarle el puerto en el que debe esperar la conexión, a su vez recibe un segundo parámetro, un puntero a entero en el que escribirá el tamaño del mensaje. El valor de retorno del receptor es un putero a la zona de memoria donde ha alojado el mensaje. El funcionamiento del programa es el siguiente:

- El receptor crea un *socket* escuchando en todas sus direcciones IP y espera a que le llegue una petición de conexión.
- El emisor crea un *socket* dirigido a la IP que se le pasa por parámetro, y envía una petición de conexión.
- Cuando el emisor recibe la confirmación de que la conexión se ha establecido envía un mensaje con el tamaño en bytes de la información que quiere transmitir y, a continuación, envía la información, al acabar termina la ejecución del método.
- El receptor recibe primero, garantizado por TCP, el tamaño del mensaje y realiza un *malloc*, una petición de memoria, para reservar una zona de memoria para el mensaje y, a continuación. A continuación, recibe el resto del mensaje, lo almacena en la zona de memoria que ha reservado y la función retorna un puntero al comienzo de esta zona.

Una vez implementado y visto que el programa funciona se requiere introducir en el código mediciones temporales para saber cual es el tiempo de ejecución del programa para distintos tamaños de mensaje. En lenguaje C lo que se hace es invocar a una función que devuelva el valor como un número entero de la fecha actual. Con dos mediciones como esta y una diferencia se obtiene el tiempo transcurrido entre inicio y fin.

Tal como se ha construido el programa, no se puede iniciar la medición del tiempo en el receptor antes de esperar la conexión puesto que el retraso en el arranque del emisor condicionaría el tiempo obtenido. Por lo tanto, se requiere medir el tiempo de ejecución en el cliente.

Para esto, se obtiene el valor del reloj justo antes de realizar, en el cliente, la petición de conexión al servidor, pero necesitamos modificar el programa para poder medir el fin de la comunicación. Ahora, el receptor una vez finalizada la recepción de todos los bytes de la información enviaba una trama que hace la función de ACK (*acknowledgement*), como se denomina en TCP, un reconocimiento pero a nivel de aplicación. Con esta modificación se puede correctamente medir el tiempo de ejecución del programa.

Para medir el paso del tiempo en C podemos utilizar la función *gettimeofday()* [4] que permite obtener los segundos y milisegundos transcurridos desde el uno de enero de 1970 en formato UTC, o *clock_gettime(CLOCK_REALTIME, [...])*, [3], que funciona igual que la anterior pero devuelve los segundos y nanosegundos transcurridos. Una vez realizadas una serie de pruebas, para comparar las prestaciones de ambas funciones, se decidió emplear la segunda función dado que padece un menor retardo de ejecución y en caso de ser necesario obtenemos mayor precisión.

De esta forma obtener el tiempo de ejecución es tan simple como: realizar una llamada a *clock_gettime()* justo antes de efectuar la conexión en el emisor y volver a llamarla cuando se recibe el ACK por parte del receptor. Con estas dos mediciones temporales una simple diferencia nos muestra el tiempo que ha llevado enviar la información.

TCP es un protocolo complejo en el que intervienen muchos parámetros. Por tanto, se hace necesario valorar cómo influye cada uno de estos parámetros en las prestaciones de TCP, dado que el objetivo de este sencillo programa es servir de referencia a nuestro protocolo, que presentaremos más adelante. Así pues, vamos a elaborar unas cuantas versiones más del programa en TCP. A la versión que obtiene el tiempo de envío de la forma definida en el párrafo anterior la denominaremos “TCP N” (N de versión normal).

En la segunda versión vamos a obviar el tiempo de conexión de TCP, con lo que vamos a mover la medición temporal de antes de ejecutar la llamada a *accept()* a antes de llamar por primera vez a una instrucción *sendto()* y vamos a denominar a esta version “TCP C” (C de *connect*). En la tercera y última versión, vamos a abrir la ventana de transmisión de TCP antes de realizar el envío, por lo que vamos no vamos a modificar el momento en el cual iniciamos la medición temporal, pero se va a introducir antes de ésta el envío de un megabyte de datos sin información útil desde el cliente al servidor, con la finalidad de establecer cuanto tarda en enviarse el mensaje si la conexión TCP ya ha sido capaz de llegar a un valor óptimo de configuracion. Antes de obtener el valor del reloj el cliente esperará un reconocimiento por parte del servidor indicando que ha finalizado la recepción de la primera tanda de información. Esta versión se denomina “TCP W” (W de *window*, de ventana de congestión).

Todos los resultados de ancho de banda que se muestran en el presente trabajo, proceden de la medía aritmética del tiempo necesario para el envío de los datos de la aplicación entre cliente y servidor utilizando el protocolo de transporte correspondiente, de diez ejecuciones previa eliminación de los valores extremos extraídos de un total de doce ejecuciones.

En las siguientes gráficas podemos observar los resultados obtenidos con esta aplicación y

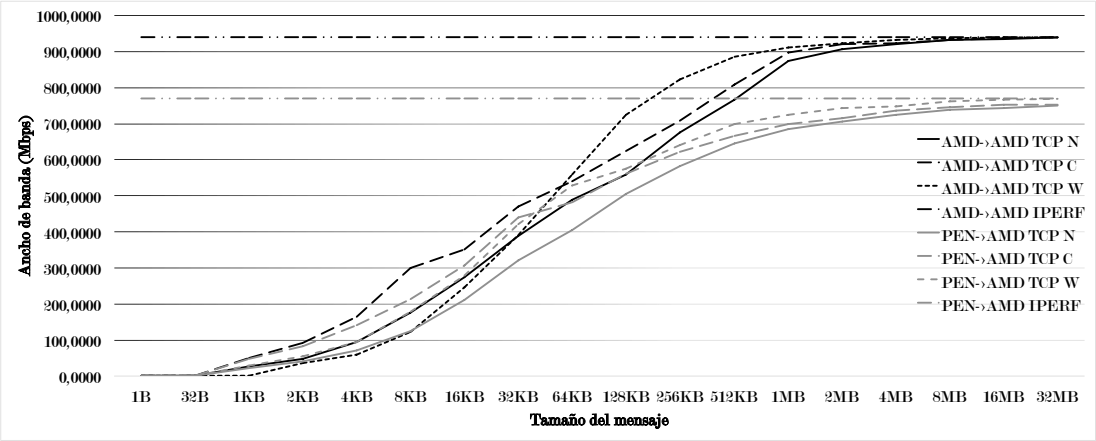

Figura 3.1: Prestaciones de TCP en la red de los AMD y el Pentium

también comprobar su coherencia con los datos que obtuvimos de Iperf en la Sección 2.2. En la Figura 3.1 y la Figura 3.2 se muestran las pruebas en la red que contiene los AMDs y el Pentium.

- De AMD a AMD: podemos observar que la versión *N* y la *C* son las versiones más lineales, la versión que ignora el tiempo de conexión se mantiene en todo momento por encima de la versión que tiene toda la sobrecarga de TCP en cuenta, aunque conforme se aumenta el tamaño del mensaje vemos que la ganancia obtenida por obviar el tiempo de conexión se vuelve insignificante.

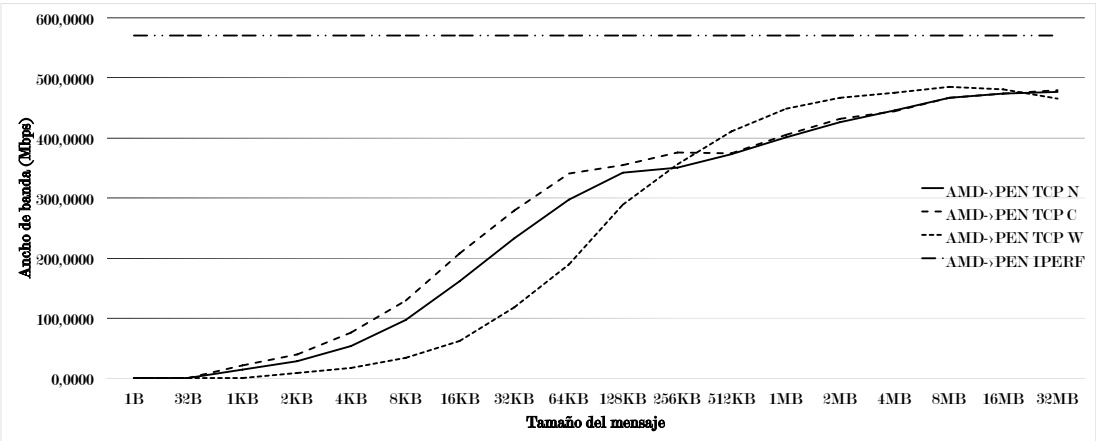

Figura 3.2: Prestaciones de TCP en la red de los AMD y el Pentium

Mientras tanto, la versión que trata de abrir la ventana de recepción antes de realizar el envío de los datos, la *W*, empieza por debajo de las otras dos versiones, dado que al encontrarse la ventana de recepción establecida a un número de bytes elevado, los reconocimientos si enviamos una cantidad pequeña de datos se retrasan mucho. Mientras que cuando la cantidad de datos a enviar es grande, vemos un aumento en ancho de banda que sobrepasa las prestaciones de la versión *C* y por ende de la *N*. En ambos casos todas las versiones tienden al valor que nos garantizó *iperf*.

- Del Pentium a uno de los AMD: observamos un comportamiento muy similar al del caso anterior. La versión *N*, en este caso, permanece en todo momento por debajo de las otras dos. Cuando el mensaje es lo suficientemente grande como para que abrir la ventana de congestión resulte de utilidad, vemos como la versión *W* sobrepasa en prestaciones a la *C*. Aunque en este caso la diferencia en prestaciones no es tan grande entre estas dos versiones.

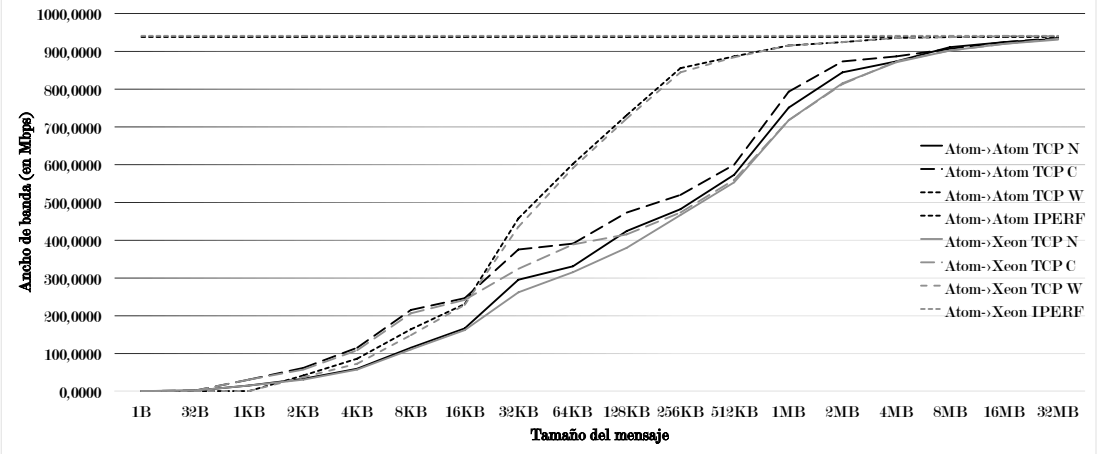

Figura 3.3: Prestaciones de TCP en la red de los Xeon y Atom

- De AMD a Pentium III: en este caso el comportamiento es similar al caso anterior pero mucho menos potente. Vemos que se necesita un mensaje mucho mayor para que abrir la ventana de congestión resulte práctico, enviar 512 KBytes en vez de 64. Además vemos que la diferencia con respecto a la cota superior es mucho mayor, que en los dos casos anteriores, siendo de casi 100 Mbits/s.

En la Figura 3.3 y la Figura 3.4 los resultados en la red de los Atoms y Xeones.

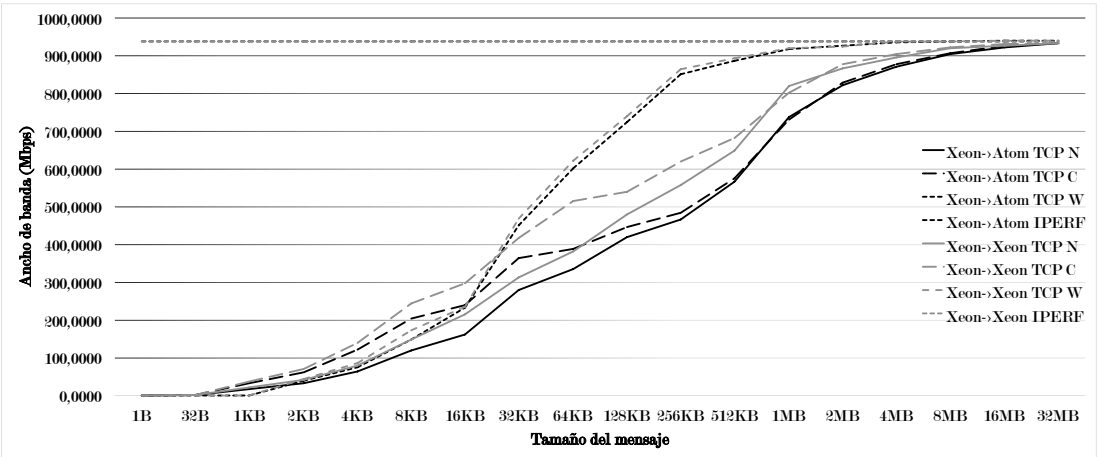

Figura 3.4: Prestaciones de TCP en la red de los Xeon y Atom

El comportamiento en las comunicaciones entre los Atoms y los Xeons sigue el mismo patrón que hemos observado cuando comunicábamos ambos AMDs. Pero se aprecia el aumento de potencia en que la abertura de la ventana de congestión resulta útil con paquetes mucho más pequeños. Solo con enviar 2 KBytes el rendimiento se iguala a la versión *N* y con 64 KBytes se obtienen en la mayoría de los casos 200 Megabits por segundo más de ancho de banda que la versión *C*. Como en los otros casos vemos que todas las pruebas tienden a la cota marcada por *iperf*.

Si bien los equipos con procesador Xeon aparentan ser más potentes que los ordenadores Atom dado que, en la comunicación Xeon a Xeon, las versiones *N* y *C* consiguen alcanzar un valor de ancho de banda notablemente superior al obtenido en los otros escenarios en el intervalo de 32 KB a 1 MB.

Capítulo 4

El protocolo implementado

En este capítulo describimos el protocolo de nivel de transporte desarrollado en este trabajo final de grado. Para ello, en la Sección 4.1 se describe el nuevo protocolo. Posteriormente, en la Sección 4.2 se describe la implementación del mismo sin entrar en detalles de si se utiliza UDP o directamente Ethernet para el envío y la transmisión de los datos.

A continuación, en la Sección 4.3 se detalla una primera implementación del mismo sobre UDP Nótese que este no es el objetivo final del trabajo (que es implementar el protocolo directamente sobre Ethernet) pero esta primera aproximación sobre UDP nos ayudará a validar la viabilidad del mismo de manera sencilla al tiempo que nos permitirá posteriormente comparar el ancho de banda de esta implementación intermedia con el resto de implementaciones. Finalmente, en la Sección 4.4 se describirá la implementación sobre Ethernet así como se estudiarán las prestaciones que consigue el protocolo implementado-

4.1. Descripción del protocolo

4.1.1. Contexto

El objetivo es crear un nuevo protocolo de transporte que se asiente directamente sobre Ethernet y que permita de forma fiable, garantizando la recepción de toda la información enviada por el cliente y su entrega a la aplicación receptora en orden, la comunicación entre dos ordenadores pertenecientes al mismo cluster, lo que se puede extender a, dos ordenadores que pertenezcan a la misma red local.

Actualmente, para el envío de datos en este contexto se utilizan o TCP o UDP dependiendo del servicio que se necesite. Pero planteémonos, ¿realmente necesitamos utilizar la arquitectura de protocolos de Internet para trabajar en una red local? A continuación vamos a analizar capa por capa si las funcionalidades que aportan a la comunicación de información a través de la red, son necesarias o no en el contexto en que nos encontramos:

- La capa de aplicación es externa al estudio que estamos desarrollando. Queremos diseñar un protocolo que se encargue del envío de los datos que le suministre esta capa y su posterior devolución en el nodo receptor.

- Con respecto a la capa de transporte, necesitamos un protocolo fiable, que incluya multiplexación y demultiplexación para permitir la comunicación proceso a proceso, con lo que si utilizásemos una tecnología ya existente en esta capa, debería ser TCP. Nótese que al realizar una comunicación entre dos ordenadores dentro de una red de área local, los paquetes no van a atravesar ningún router y, por tanto, éstos no podrán descartar paquetes. Podríamos pensar por tanto, que no se van a perder paquetes.

  No obstante, es posible que la tarjeta de red receptora descarte paquetes porque lleguen a mayor velocidad de la que se necesita para retirarlos. Este caso ya lo habíamos visto en la Sección 2.2 cuando utilizábamos el equipo con procesador Pentium III como servidor del programa "iperf". Por tanto, debemos suponer que puede haber perdida de paquetes y ser capaces de corregir esta perdida mediante retransmisión de los mismos.

- La capa de red, mediante el protocolo IP, ofrece direccionamiento lógico de las tramas por la red. Esta capa es la que permite que una trama salga de nuestra red a través del router correspondiente y se encamine hacia su destino fuera de nuestra red. En el contexto en el que nos encontramos sería ideal poder prescindir completamente de esta capa.

- La capa de enlace de datos, mediante Ethernet, nos ofrece direccionamiento físico, encaminamiento de las tramas dentro de la red local y se encarga del acceso al medio. Necesitamos estas prestaciones en el protocolo que estamos desarrollando y siendo que además Ethernet se encarga del manejo de la capa física, lo ideal sería utilizar Ethernet como base para reducir la sobrecarga.

- La capa física, que se encarga de trabajar con los bits tal y como viajan a través del enlace. No va a ser modificada en este trabajo final de grado y dejaremos que sea íntegramente manejada por Ethernet.

Visto esto, debemos separarnos de la arquitectura clásica de Internet. Eliminando la capa de red recuperamos espacio en la trama Ethernet que está ocupado con información que no vamos a necesitar y además eliminamos el coste de calcular esta información y el mantenimiento de la información de control que incorpora. Entonces utilizamos Ethernet, pues lo necesitamos para que los switches sepan encaminar las tramas así como para que los nodos reconozcan al destinatario y al emisor de la trama y dejamos que Ethernet se encargue de la capa física.

Ahora bien, la capa de enlace de datos no nos garantiza fiabilidad, y no existen protocolos de la capa de transporte que la garanticen siendo capaces de trabajar sin la capa de red. Aquí es donde entra este trabajo, en diseñar e implementar un protocolo en la capa de transporte que se comunique directamente con la capa de enlace de datos, que sea fiable y permita la comunicación proceso a proceso.

Las comunicaciones dentro de un *cluster* se suponen seguras, nadie a quien el administrador no le haya dado acceso puede acceder, con lo que el CRC de Ethernet debería ser suficiente para detectar errores en la trama debidos a interferencias en la red, y vamos a suponer que no se van a realizar ataques deliberados contra la información que vamos a enviar.

4.1.2. Cabecera del protocolo

La única información que se encuentra en la trama es la que nos aporta Ethernet, es decir, que tenemos las direcciones MAC origen y destino y un campo para especificar el protocolo que estamos utilizando. Planteémonos que información necesitamos, que sea imprescindible, de la cual necesitemos mantener un estado durante toda la ejecución del programa, y se deba de intercambiar entre cliente y servidor durante toda la comunicación.

Lo primero que necesitamos es poder ofrecer comunicaciones proceso a proceso, para esto vamos a introducir un único valor de cuatro bytes que realiza la misma función de los puertos de TCP y UDP. En nuestro caso vamos a llamar a este valor "Identificador de la conexión", lo estableceremos automáticamente en el emisor y se lo comunicaremos al receptor. De esta forma toda trama que se envíe con datos sobre esta comunicación llevará en su cabecera este identificador. Además, cuando el sistema operativo nos suministre una trama con identificador distinto al que estamos utilizando en la comunicación, obviaremos completamente la trama.

Aunque pequeña existe la posibilidad de que en una serie de comunicaciones paralelas que tengan lugar entre los mismos nodos seleccionen al azar el mismo identificador. Este caso no se va a contemplar en el prototipo de protocolo implementado. Primero se va a confiar en el hecho de que 2^{32} valores posibles sean suficientes para que no colisiones dos comunicaciones. Además, en una versión del protocolo diseñada de cara al uso por usuarios, sería ideal que le identificador pudiese establecerse por parte de los mismos.

Necesitamos, a parte, controlar la entrega de todas las tramas así como su reconstrucción en el orden correcto. Para esto introducimos otro valor entero (de cuatro bytes) en la cabecera que al igual que hace TCP denominaremos "Número de secuencia". Este valor nos informará de la posición que ocupan los datos contenidos en el paquete en el mensaje original.

Figura 4.1: Primera versión de la cabecera del protocolo

Por último vamos a introducir otro valor entero en la cabecera que indicará el tamaño en bytes de la zona de datos. Esta primera versión de la cabecera se puede ver en la Figura 4.1. Pero, estudiando en más profundidad este campo podemos ver que, al estar utilizando Ethernet por debajo de nuestro protocolo, estamos sujetos al valor de MTU de 1500 bytes. Dado esto no necesitamos un valor de longitud de cuatro bytes, nos basta con utilizar dos bytes para este campo, igual que hace UDP. Esta modificación nos permite introducir dos bytes más de datos por cada trama enviada, lo cual puede no parecer mucho pero siendo que podemos aprovechar y hacerlo, no sería correcto dejar pasar esta oportunidad. Pero aún podemos ir más allá, si en todo momento enviamos tramas completas excepto la última, no necesitamos incluir el tamaño en la

cabecera ya que es fácilmente deducible en el servidor. Puesto que será siempre el tamaño de datos que permitan la trama y las cabeceras a excepción de la última trama, que sabemos cual es gracias al número de secuencia, cuyo tamaño es el resto. La nueva cabecera se puede ver en la Figura 4.2.

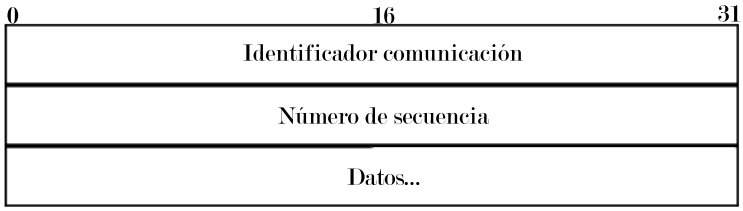

Figura 4.2: Estructura final de la cabecera del protocolo

Podríamos decir que esta es la cabecera fija del protocolo, que utilizarán las tramas que contengan datos. Las tramas que utilicemos para sincronizar cliente y servidor, cuyo funcionamiento se explicará en la sección siguiente, se caracterizan por tener siempre el número de secuencia a cero, lo que las identifica como tramas de sincronía. Además a continuación de este valor hay un byte que identifica el tipo de la sincronía: inicial, final o de reconocimiento.

4.1.3. Garantizando la fiabilidad

Como ya hemos visto, TCP es un protocolo fiable y consigue esto estableciendo una conexión entre cliente y servidor. Así que vamos a crear nuestro protocolo orientado a la conexión, una conexión más simple que TCP intentando que sea menos pesado, reduciendo la sobrecarga tanto temporal como espacial, aprovechándonos del hecho de que nuestro protocolo se ejecutará dentro de una red de área local.

Para conseguir esto se ha diseñado un sistema que se encarga de sincronizar ambos terminales. Para esto vamos a utilizar lo que hemos definido como tramas de sincronía, como TCP cuando pone el byte de SYN a uno para indicar que la trama se utiliza para sincronizar la conexión, nosotros vamos a hacer lo mismo pero utilizando el valor almacenado en el número de secuencia. Vamos a reservar todas las tramas con secuencia igual a cero para utilizarlas como tramas de sincronía. Estas tramas no van a contener información de la capa de aplicación, llevarán otra información para indicar su propósito.

TCP, cuando envía sincronías espera hasta recibir una respuesta y si el tiempo de espera excede de un máximo, reenvía el segmento de sincronización. Nosotros vamos a implementarlo de forma distinta, vamos a enviar las tramas de sincronía un número fijo de veces, el suficiente para garantizar que estás tramas siempre se reciben, a pesar de posibles interferencias en el enlace o descartes en la tarjeta de red receptora.

De esta forma, en el momento en que el computador emisor pasa datos a nuestro protocolo, este envía primero un número determinado de veces una trama de sincronía inicial en la que informa al receptor del inicio de la comunicación así como del número de bytes que ocupa la información que quiere enviar. De esta forma la comunicación está establecida y comienza el envío de los datos de la aplicación y una vez se ha completado el envío, se envía de nuevo una sincronía indicando el fin de la comunicación.

En este momento caben dos posibilidades, que el receptor haya recibido toda la información, en cuyo caso envía una sincronía al emisor indicándolo y termina la ejecución del programa como se puede ver en la Figura 4.3. La otra posibilidad es que no toda la información se haya recibido. Esto puede deberse a interferencias en la red o a la incapacidad del buffer de la tarjeta de red del servidor de almacenar todas las tramas que se reciben mientras no se hayan tratado aun.

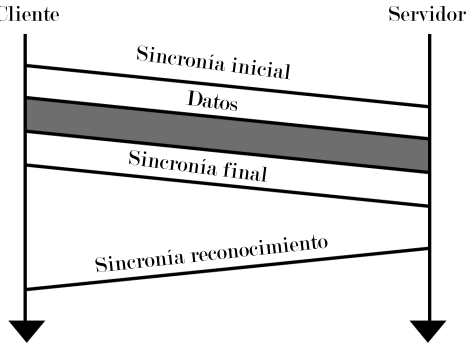

Figura 4.3: Diagrama de funcionamiento si no se pierden tramas

Pero ahora no nos vamos a plantear los motivos por los cuales algunas tramas no han sido recibidas. El caso es que el servidor no tiene toda la información de la aplicación y se le debe reenviar la información que le falte. Lo que hace el servidor es construir un vector con los números de secuencia de las tramas que no ha recibido y envía esta información al cliente de la misma forma que el cliente ha enviado los datos: empieza con una trama de sincronía inicial, envía el vector, repartido en tantas tramas como se deba para respetar el tamaño máximo que permite la red y cierra la comunicación con una sincronía final. El emisor, llegados a este punto, recoge el vector que contiene los números de secuencia de las tramas que no se han recibido y repite el proceso inicial, pero esta vez enviando solo las tramas que faltan y no la totalidad de estas.

En la Figura 4.4 se puede ver el funcionamiento descrito en el párrafo anterior en caso de que no se reciban todas las tramas. Por supuesto la petición de tramas que no se han recibido y la respuesta se realiza tantas veces como sea necesario hasta que el servidor tenga toda la información de la aplicación. Lo único es que el programa siempre termina con una sincronía de reconocimiento.

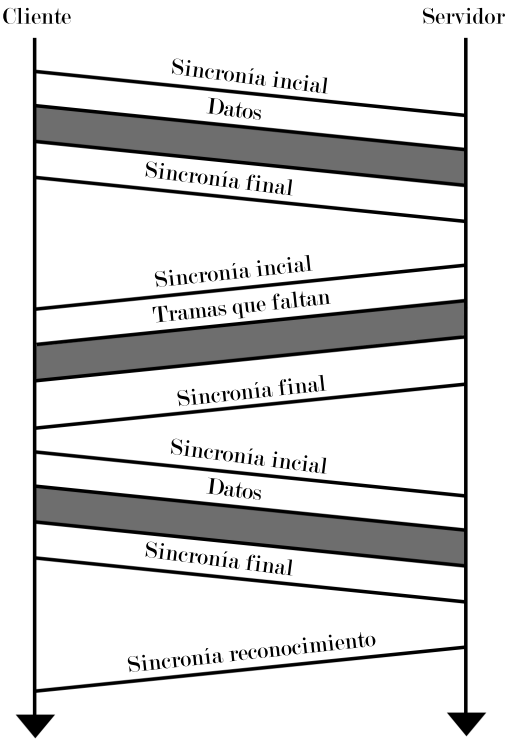

Figura 4.4: Diagrama de funcionamiento si se pierden tramas

4.2. Implementación

Como ya se ha mencionado el protocolo se va a implementar en C, dado que de ser eficiente nos gustaría que se pudiera incluir en las distribuciones de Linux, siendo lo más amigable con el sistema operativo que se pueda. Además es un lenguaje cuyo código es muy eficiente y permite gran cantidad de operaciones de bajo nivel, ideal para nuestro objetivo de enviar tramas sin utilizar TCP, UDP ni siquiera IP. Evidentemente, se van a diseñar dos programas, el cliente y el servidor.

Ambos extremos del protocolo necesitan del envío de tramas de sincronía. Como se ve en el fragmento de código que aparece a continuación, lo primero que se hace es ordenar los datos de la cabecera del protocolo que, como ya se ha explicado, es distinta en las tramas de sincronía que en las de datos. El primer parámetro es el identificador de la comunicación, después introducimos el número de secuencia a cero para indicar que esta trama es de sincronía. El siguiente parámetro que introducimos es el valor del *flag*, o etiqueta, qué indica de que tipo es la trama de sincronía (inicial, final o reconocimiento). Por último, si la trama es inicial necesitamos enviar en el mensaje el tamaño del mensaje para comunicárselo al servidor.

```
1 void envioTramaSincronia(uint flag, uint length){
2   uint i = 0, tam_total = TAM_HEAD_ETH + 3 * sizeof(int);
3
4   cabecera[0] = id_com;
```

```
5     cabecera[1] = 0;
6     cabecera[2] = flag;
7     if (flag == FLAG_INICIAL){
8       tam_total += sizeof(int);
9       cabecera[3] = length;
10    }
11
12    while(i < NUMTRAMASESPECIALES){
13      if (sendto(sockfd, /*[...]*/) == -1) {
14        perror("sendto");
15      }
16      i++;
17    }
18  }
```

Este ejemplo de código se corresponde con la implementación en Ethernet, donde se tiene un vector de tamaño igual a la longitud de una trama Ethernet, y la variable "cabecera" es un puntero a entero sin signo que apunta a 14 bytes de separación del inicio del vector, indicando donde se sitúa la cabecera del protocolo implementado. La macro "TAM_HEAD_ETH" es igual a 14 bytes, el tamaño de la cabecera de IEEE 802.3.

4.2.1. Cliente

El programa cliente inicia la comunicación con el servidor y le envía la información de la aplicación. Para esto, necesita como parámetros la información de como alcanzar al servidor a través de la red, e información del mensaje. Se requiere de un puntero a la zona de memoria en la que se encuentra la información a enviar, así como un entero para indicar la cantidad de bytes que ocupa esa zona de memoria, o por lo menos la cantidad que se ha de enviar. El primer paso es crear el *socket*, proceso que se explica en la Sección 4.3 y la Sección 4.4, para *sockets* UDP y *RAW* Ethernet respectivamente.

Tras crear el socket, el siguiente paso es calcular el número de tramas que se deben enviar, dada una macro *TAM_DATOS_PROTO* cuyo valor es el tamaño máximo de la zona de datos de la aplicación en cada trama. El resultado de la división entera del tamaño del mensaje a enviar y esta constante da como resultado el número de tramas completas que se necesitan para enviar toda la información. Además, el módulo del tamaño del mensaje y el tamaño antes mencionado da como resultado el tamaño que ha de tener una trama incompleta con los últimos datos del mensaje, en caso de ser necesaria.

Ahora ya lo que se debe hacer es enviar la información, en el fragmento de código siguiente se observa el cuerpo del método principal del cliente. Como mencionamos al explicar el diseño, el primer paso es enviar la trama de sincronía inicial, tras esto se envía el mensaje y después la trama de sincronía final. Hecho esto, entramos en un bucle que terminará cuando recibamos la trama de sincronía de reconocimiento. En este bucle, lo primero que hacemos es recibir información del servidor. Los datos que recibamos serán o bien una sincronía de reconocimiento, terminando con el programa; o bien información de tramas perdidas en cuyo caso, procedere-

mos al envío de la información faltante. Una vez salimos del bucle, debemos liberar las zonas de memoria que hayamos reservado y cerrar el *socket*.

```
1  void envio(/*[...]*/, ubyte *mensaje, uint length){
2    crearSocket(/*[...]*/);
3
4    resto = length % TAM_DATOS_PROTO;
5    num_tramas = (uint) (length / TAM_DATOS_PROTO) + (resto?1:0);
6
7    envioTramaSincronia(FLAG_INICIAL, length);
8
9    if(num_tramas == 1){
10     envioMensajeIoU(resto);
11   }else{
12     envioMensajeIoU(TAM_DATOS_PROTO);
13     envioMensaje(num_tramas, resto);
14   }
15
16   envioTramaSincronia(FLAG_FINAL, 0);
17
18   tramas_faltan = (ubyte *)mi_malloc(num_tramas * sizeof(ubyte)
       , "tramasFaltan");
19   for (i = 0; i < num_tramas; i++){
20     tramas_faltan[i] = 1;
21   }
22
23   while(!ack){
24     recepcion(&ack, num_tramas);
25
26     if(!ack){
27       envioTramaSincronia(FLAG_INICIAL2, 0);
28
29       if (tramas_faltan[0] == 0){
30         total_tramas_fallidas++;
31         if(num_tramas == 1){
32           envioMensajeIoU(resto);
33         }else{
34           envioMensajeIoU(TAM_DATOS_PROTO);
35         }
36         tramas_faltan[0] = 1;
37       }
38       for(i = 1; i < num_tramas; i++){
39         if(tramas_faltan[i] == 0){
40           envioMensajePorNum(i + 1, num_tramas, resto);
41           tramas_faltan[i] = 1;
42         }
```

```
43          }
44
45          envioTramaSincronia(FLAG_FINAL, 0);
46        }
47    }
48
49    mi_free((void **)&tramas_faltan, "tramasFaltan");
50    close(sockfd);
51  }
```

Cuando el protocolo envía una zona de los datos de la aplicación, coloca un puntero apuntando al primer byte de mensaje que debemos enviar y desde este punto retrocede el puntero el tamaño de la cabecera del protocolo. Guardamos en una zona de memoria separada la información que hay entre la nueva posición del puntero y la original y colocamos en su lugar la cabecera, actualizando el número de secuencia por el valor que deba tener. Una vez enviada la trama volvemos a dejar el mensaje como estaba. Esto no se puede hacer con los primeros bytes del mensaje dado que al retroceder el puntero estaríamos entrando en una zona de memoria que no hemos reservado, lo que provocaría una error de pila. Así pues el procedimiento a seguir es: reservar espacio en memoria del tamaño de cuantos datos podamos enviar, colocar la cabecera del protocolo, y copiar tantos bytes como queden disponibles del mensaje a la trama.

```
1  void envioMensajeIoU(uint bytes){
2    ushort tam_total = TAM_HEADS + bytes;
3
4    cabecera[0] = id_com;
5    cabecera[1] = 1;
6
7    memcpy(datos, mens, bytes);
8
9    if (sendto(sockfd, /*[...]*/) == -1) {
10     perror("sendto");
11   }
12 }
```

En el código que aparece unas líneas atrás, se muestra el método que envía la primera trama de datos de la aplicación. Como se debe hacer siempre se fija el identificador de la comunicación en la cabecera del protocolo y en este método se fija siempre el valor del número de secuencia a uno, y se copian los primeros bytes del mensaje a otra zona de memoria. En su lugar para el envío de cualquier otra sección de los datos de la aplicación debemos tener en cuenta si es o no la última trama para establecer su longitud. Establecemos un puntero en la zona en la que se encuentra el mensaje de la aplicación, después desplazamos este puntero hasta el primer byte que se debe de enviar, y por último hacemos retroceder el puntero tantos bytes como el tamaño de la cabecera del protocolo. Ahora con el puntero a memoria apuntando a la dirección que queremos, guardamos en un *buffer* los datos que hay desde esta posición al primer byte de mensaje a enviar en esta trama y ponemos en su lugar la cabecera del protocolo actualizando el número de secuencia. Realizamos el envío y devolvemos la información del mensaje que

habíamos sustituido con la cabecera por la información original.

```
void envioMensajePorNum(uint num_trama, uint num_tramas, uint
    resto){
  uint tam_total = TAM_HEADS + ((num_trama == num_tramas)?resto
     :TAM_DATOS_PROTO);
  ubyte *punt_aux = mens + (TAM_DATOS_PROTO * (num_trama - 1))
     - TAM_HEADS;

  memcpy(buffer, punt_aux, TAM_HEADS);
  cabecera[1] = num_trama;
  memcpy(punt_aux, trama, TAM_HEADS);

  if (sendto(sockfd, /*[...]*/) == -1) {
    perror("sendto");
  }

  memcpy(punt_aux, buffer, TAM_HEADS);
}
```

El método *envioMensaje(uint num_tramas, uint resto);* es igual que el método que se muestra en el fragmento de código anterior con la salvedad de que en vez de indicársele una trama a enviar, este realiza el envío desde la segunda trama hasta la última. Enviando todas las tramas con un tamaño igual al máximo que permite la trama Ethernet (descontándole la longitud de las cabeceras), excepto la última cuyo tamaño lo indica el parámetro "resto".

4.2.2. Servidor

El primer paso en el servidor, al igual que en el cliente es crear el *socket*. Una vez creado, esperamos a recibir la trama de sincronía inicial, que lleva la información relevante al tamaño del mensaje, con esto calculamos el número de tramas que vamos a recibir así como el tamaño de la última. Reservamos espacio para dos vectores, un vector de bytes que utilizaremos para almacenar si la trama indicada por el índice del vector está recibida o no. En el otro, guardaremos los punteros de memoria a las zonas auxiliares en las que se guardará cada trama, antes de realizar la ordenación una vez las recibamos todas.

```
void *recepcion(char *InterfaceName, uint *length){
    crearSocket(InterfaceName);

  recibirTramaInicial(&tamanio);
  num_tramas_totales = tamanio / TAM_DATOS_PROTO;
  if( tamanio %TAM_DATOS_PROTO ){
    num_tramas_totales++;
  }
  num_tramas_faltan = num_tramas_totales;

```

```
11    datos = (struct datos *)mi_malloc(num_tramas_totales * sizeof
          (struct datos), "datos");
12    recibida = (ubyte *)mi_malloc(num_tramas_totales * sizeof(
          ubyte), "recibida");
13    for(i = 0; i < num_tramas_totales; i++){
14      recibida[i] = 0;
15    }
16    recibirTramas(&num_tramas_faltan);
17
18    while(num_tramas_faltan){
19      pedirTramas();
20      recibirTramaInicial2();
21      recibirTramas(&num_tramas_faltan);
22    }
23
24    envioTramaSincronia(FLAG_ACK);
25  }
```

Con esta información preparada, ya se puede recibir tramas hasta que nos llegue la trama de sincronía final. Si se han recibido todas las tramas, lo único que queda es enviar la sincronía de reconocimiento y realizar la ordenación de los datos de la aplicación así como liberar los punteros de memoria y cerrar los *sockets*. El método que se encarga de la recepción de tramas se muestra a continuación.

```
1  void recibirTramas(uint *num_tramas_faltan){
2    uint num_secuencia;
3    ubyte final = 0, do_malloc = 1;
4
5    while(!final){
6      if (do_malloc){
7        mens = (char *)mi_malloc(TAM_TRAMA_ETH, "mens");
8        do_malloc = 0;
9      }
10
11     if (recvfrom(sockfd, /*[...]*/) == -1) {
12       perror("recvfrom");
13     }
14
15     cabecera = (int *)(mens + TAM_HEAD_ETH);
16
17     if (cabecera[0] == id_com){
18       if(cabecera[1]){
19         num_secuencia = cabecera[1] - 1;
20         if (recibida[num_secuencia]){
21           continue;
22         }
```

```
23            recibida[num_secuencia] = 1;
24            datos[num_secuencia].direccion = (ubyte *)mens;
25            (*num_tramas_faltan)--;
26            do_malloc = 1;
27          } else if(cabecera[2] == FLAG_FINAL){
28            final = 1;
29            mi_free((void **)&mens, "mens");
30          }
31        }
32      }
33  }
```

Básicamente, se reciben tramas hasta que llega una trama de sincronía final. En el momento en que se recibe una trama se comprueba: primero, el identificador de la comunicación, si es distinto del nuestro ignoramos la trama; segundo, el número de secuencia si es igual a cero, solo hacemos caso a la trama si es una sincronía final, si es distinto de cero comprobamos que la trama no haya llegado duplicada y si no es así nos guardamos el puntero a la zona de memoria donde se encuentra y actualizamos el contador de tramas pendientes de recibir así como el vector de tramas recibidas.

4.3. Implementación en UDP

Vamos a realizar una primera implementación del protocolo a partir de UDP, esta versión nos servirá para comparar prestaciones con las que tenga la versión directamente en Ethernet, así como para comprobar si el protocolo resultaría eficiente.

Utilizando UDP, el tamaño máximo de trama es de 1500 bytes, puesto que utiliza Ethernet. A esta cantidad hay que restarle el tamaño de las cabeceras TCP y UDP, veintiocho bytes respectivamente, lo que nos deja 1472 bytes disponibles para el protocolo. Dado que utilizamos dos valores enteros en nuestra cabecera, son otros ocho bytes reservados, lo que deja un total de 1464 bytes para datos de aplicación disponibles para las tramas completas.

4.3.1. Creación y uso del socket

Como se puede observar en el método que aparece a continuación, para crear un *socket* UDP en el cliente, creamos un socket con los parámetros *AF_INET*: que indica que utilizamos direcciones de Internet, direcciones IP; *SOCK_DGRAM*: es decir, un socket de datagramas UDP, y un último parámetro de *flags* a cero pues no necesitamos establecer un parámetro concreto.

```
1 void crearSocket(int puerto){
2   int sockfd;
3
4   if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1){
5     perror("socket");
6     exit(1);
```

```
7    }
8
9    struct sockaddr_in my_addr, his_addr;
10   my_addr.sin_family = AF_INET;
11   my_addr.sin_port = htons(puerto);
12   my_addr.sin_addr.s_addr = htonl(INADDR_ANY);
13
14   his_addr.sin_family = AF_INET;
15   his_addr.sin_port = htons(Port);
16   his_addr.sin_addr.s_addr = inet_addr(IP);
17
18   if (bind(sockfd, (struct sockaddr *)&my_addr, addr_len) ==
         -1) {
19     perror("bind");
20     exit(1);
21   }
22 }
```

A continuación, creamos una estructura que contendrá la información IP del host. Establecemos la familia de direcciones a Internet, fijamos el puerto en el que escuchará UDP e indicamos que se escuche en todas las direcciones IP del *socket* y por último, unimos la información de la estructura con el *socket* creado. La otra estructura contiene la información del servidor y se diferencia en que especificamos su dirección IP, en el campo consecuente de la estructura.

En el servidor el proceso es el mismo, con la salvedad de que no se utiliza la estructura *his_addr*, la información relevante al cliente no la conocemos a priori y la obtendremos cuando recibamos la primera trama.

El envío de datos mediante un *socket* UDP se realiza mediante la función *sendto*, y la recepción mediante *recvfrom*. Ambas funciones reciben los mismos parámetros: el primero es el *socket*, el segundo es el puntero a la zona de datos que se desea enviar, el siguiente es la cantidad de bytes que se deben enviar. El cuarto sirve para especificar el campo de *flags* de la cabecera UDP. Los dos últimos parámetros se utilizan para especificar el destinatario del mensaje, en el penúltimo se introduce la estructura que contiene la dirección IP y el puerto del servidor; el último parámetro es simplemente el tamaño de esta estructura.

```
1 sendto(sockfd, trama, 4*sizeof(int), 0, (struct sockaddr *)&
      his_addr, addr_len));
2 recvfrom(sockfd, trama, 4*sizeof(int), 0, (struct sockaddr *)&
      his_addr, (socklen_t *)&addr_len));
```

4.3.2. Pruebas

Una vez que tenemos el protocolo sobre UDP funcionando, el siguiente paso es empezar a probar y ver cuanto ancho de banda conseguimos. Para medir el tiempo de ejecución vamos a seguir el mismo método que con el programa que hicimos en TCP. Ejecutamos *gettimeofday()*

justo antes de enviar la primera trama de sincronía en el emisor, y volvemos a ejecutarlo justo después de recibir el ACK del receptor.

En la Figura 4.5 vemos una gráfica de las prestaciones que se obtienen ejecutando el protocolo entre los AMDs y el Pentium III, variando el número de tramas de sincronía. Se ha probado utilizando una, diez, veinticinco y cincuenta tramas de sincronía. En la gráfica también hay una línea para cada uno de los valores que obtuvimos con *ipref*.

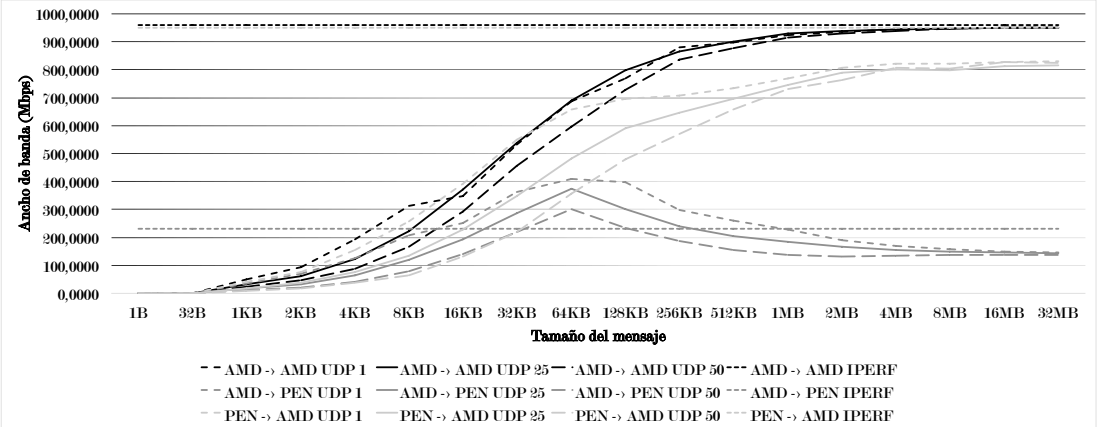

Figura 4.5: Prestaciones de UDP en la red de los AMD y el Pentium

Observemos primero la comunicación entre los dos AMDs que es la que más ancho de banda consigue, llegando virtualmente al valor de cota superior que tenemos. Vemos que en la mayoría de los casos la versión que consigue mayores prestaciones es la que utiliza diez tramas de sincronía, ¿pero no sería más lógico que fuese la que solo utiliza una trama? Lo que está pasando es que el host es más rápido que la red, entonces recibe la trama de sincronía y se prepara para recibir el resto, lo que deja libre el procesador para el sistema operativo hasta que se lleguen tramas con datos y vuelva a tocarnos turno en el procesador. Cuando utilizamos más tramas de sincronía tenemos un flujo casi continuo de tramas mientras esperamos que lleguen los datos, aunque estas tramas extras de sincronía se ignoran en el código, son suministradas por el *socket*.

La situación cuando contemplamos el Pentium como emisor y uno de los AMDs como receptor, sigue el mismo patrón que la comunicación anterior pero en este caso el ancho de banda es mucho menor al que nos ofrecía iperf como valor de cota, lo que nos demuestra que efectivamente el Pentium no es un equipo potente, pero aun tenemos que comparar este contexto con lo que obteníamos en TCP. A parte de esto, ahora observamos que contra menos tramas de sincronía mejores prestaciones obtenemos. Dado que, en esta ocasión el cuello de botella es siempre el Pentium, la organización del procesador no nos afecta negativamente cuando enviamos pocas tramas.

El patrón difiere cuando ponemos el ordenador poco potente como servidor. Vemos que el crecimiento de las versiones es bueno al principio, alcanzando el máximo en los 64 KBytes. Pero a partir de este momento las prestaciones empiezan a disminuir. La razón de esto es que por primera vez en todas las pruebas experimentamos perdida de tramas debida a que la tarjeta de red del Pentium no es capaz de almacenar todas las tramas que recibe hasta que son gestionadas

y tiene que ir machacando cuando llegan nuevas. El coste de ignorar la congestión es, en este contexto, demasiado grande y se estudiará una forma de abordarla en el Capítulo 6. La falta de potencia en el servidor también hace que aparezca otro problema, que es el bloqueo del protocolo cuando las tramas de sincronía se pierden, siendo que ni siquiera con 50 tramas de sincronía podemos garantizar la fiabilidad, en el Capítulo 5 estudiaremos como mejorar esta característica del protocolo.

Habiendo estudiado las prestaciones de la versión del protocolo sobre UDP, vamos a pasar a compararlas con los datos que habíamos obtenido en TCP. La Figura 4.6 muestra la tabla en la que se resumen todos estos datos. Para que la gráfica no contenga tanta información que resulte imposible de discernir hemos utilizado dos de las mediciones de UDP, las de diez y veinticinco tramas de sincronía; y de TCP hemos obviado la versión en la que se tenía en cuenta el tiempo de conexión y dejado las otras dos que eran por lo general más potentes.

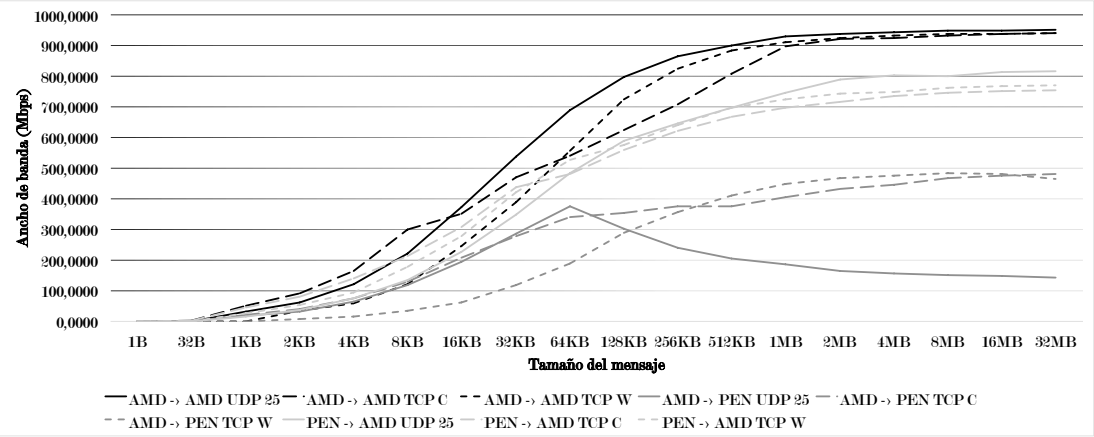

Figura 4.6: Comapración de UDP y TCP en la red de los AMD y el Pentium

Si seguimos el mismo orden al estudiar los casos que antes, vemos que en la comunicación entre los AMD, para un tamaño de mensaje pequeño, la versión de TCP que ignora el tiempo de establecimiento de la conexión, obtiene mejores prestaciones que la versión de UDP con 25 tramas de sincronía y se acerca mucho a la versión con diez. Pero, conforme aumentamos el tamaño del mensaje la separación entre ambos protocolos se hace más diferenciada. Al final, ambos se estabilizan, UDP en los 950 Mbits/s y TCP en 940.

El caso siendo el Pentium el emisor es muy similar al anterior, aquí TCP, cuando ignora el tiempo de establecer la conexión, es mejor que ambas versiones de UDP para tamaños de mensaje pequeños. Si el tamaño del mensaje es grande, el protocolo obtiene unas prestaciones considerablemente mejores que TPC, así como en el caso anterior la diferencia era pequeña, ahora UDP obtiene, de media entre ambas versiones, 830 Mbits/s mientras que, TCP alcanza los 760 Mbits/s. El caso que queda, ya se ha comentado que queda trabajo por hacer para que cuando el servidor es un equipo poco potente, el protocolo consiga buenas prestaciones.

Cambiemos ahora de red, vamos a estudiar las prestaciones entre los equipos Atom y Xeon. El estudio del número de tramas de sincronía en esta red es muy simple pues como muestra la Figura 4.7, en el caso de la comunicación entre dos equipos Atom, las prestaciones del protocolo son siempre crecientes y acordes al hecho de, contra menos tramas de sincronía mayor ancho de

banda. Este esquema se repite en los otros tres escenarios.

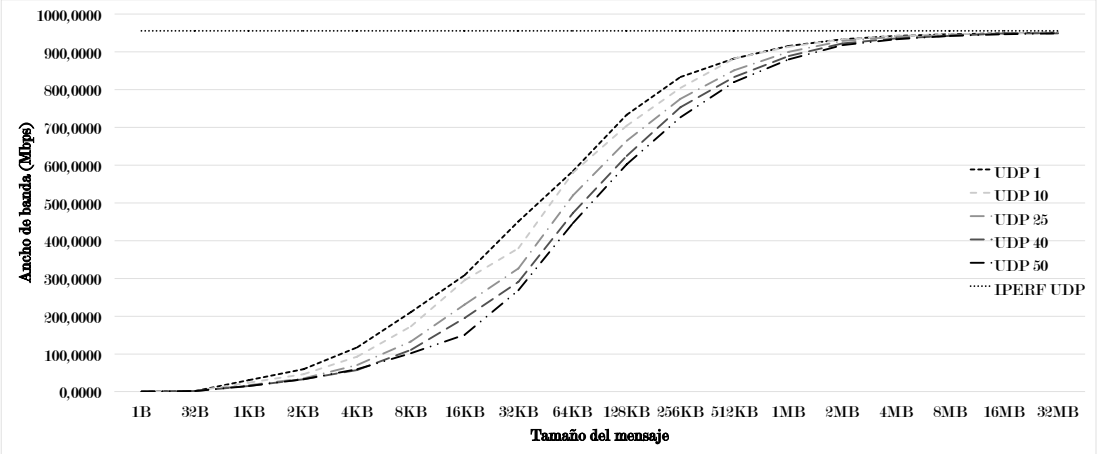

Figura 4.7: Prestaciones de UDP entre dos Atoms variando el número de tramas de sincronía

Pasemos a comparar UDP y TCP en esta red, basándonos a los datos de la gráfica de la Figura 4.8 y la Figura 4.9. Lo primero que vemos es que las versiones de TCP que no tienen en cuenta en el tiempo de conexión, ofrecen en todos los casos menores prestaciones que el protocolo, pero a su vez la versión que abre la ventana de TCP resulta más potente que nuestro protocolo.

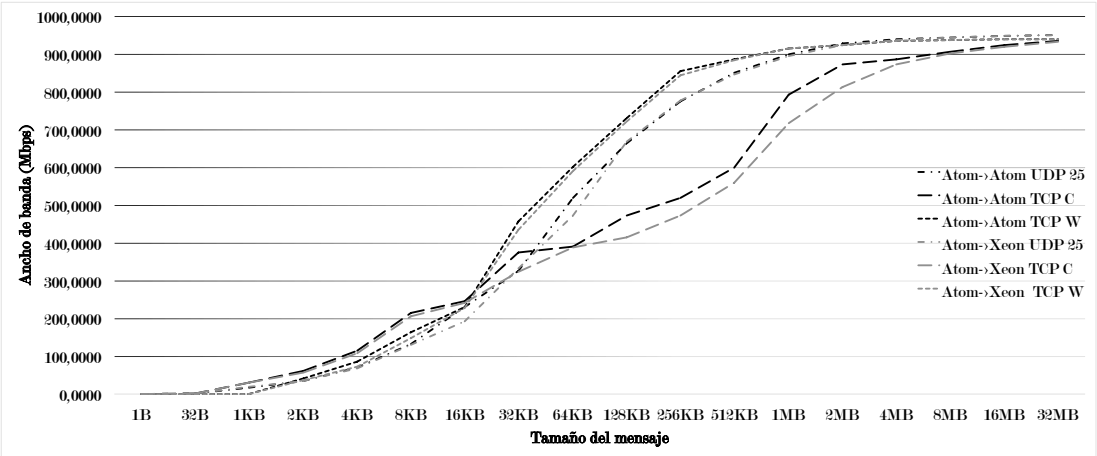

Figura 4.8: Contraste de prestaciones de UDP y TCP en la red de los Atom y Xeon

En estos escenarios nos encontramos entre la espada y la pared. El protocolo funciona bien, si el tamaño del mensaje es muy grande funciona mejor que cualquier versión de TCP ya que estas alcanzan su cota en los 940 Mbits/s, mientras que, utilizando UDP alcanzamos los 950 Mbits/s, el problema es que en la mayoría de los casos nos encontramos consiguiendo menor ancho de banda que TCP, siendo que le hemos dejado tiempo para prepararse y abrir la ventana de recepción. Si nos comparamos con una versión que sí tiene en cuenta todo el sobrecoste de TCP, aun excluyendo el establecimiento de la conexión, vemos que en la mayoría de los casos

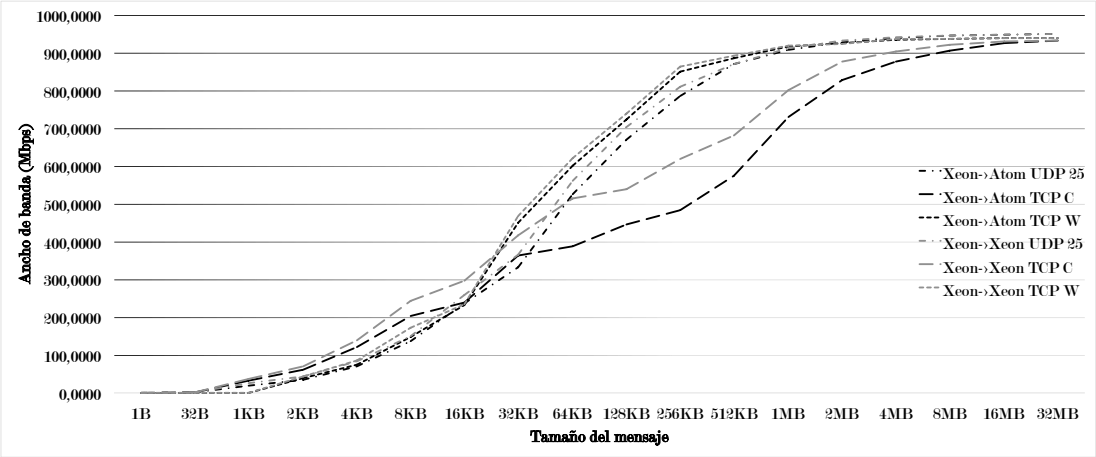

Figura 4.9: Contraste de prestaciones de UDP y TCP en la red de los Atom y Xeon

el protocolo obtiene mejores prestaciones.

A la vista de los datos que se han expuesto, el protocolo parece prometedor, pues si con UDP estamos consiguiendo, en general, mejores prestaciones que con TCP ofreciendo los mismos servicios. Cuando nos libremos del sobrecoste de IP y UDP, migrando el protocolo a Ethernet los resultados deberá ser aún mejores.

4.4. Implementación en Ethernet

Viendo que la versión en UDP resulta prometedora, pasamos a implementar el protocolo directamente en Ethernet. En este contexto contamos con la totalidad de la trama Ethernet y quitando los ocho bytes de la cabecera, contamos con 1492 bytes para utilizar para datos de la aplicación, 28 bytes más por trama que en la versión anterior y nos libramos de la sobrecarga de IP y UDP.

4.4.1. Creación y uso del socket

Para poder utilizar un *socket* que te permita establecer todos los datos de la trama, sin que el sistema operativo establezca las cabeceras de los protocolos que se suelen utilizar por defecto, se necesita de un *RAW Socket*.

```
1 void crearSocket(char *ifName, char *dest_mac){
2   struct ifreq if_idx, if_mac;
3
4   if ((sockfd = socket(AF_PACKET, SOCK_RAW, htons(PROTOCOLO)))
       == -1) {
5     perror("socket");
6   }
7
```

```
8   memset(&if_idx, 0, sizeof(struct ifreq));
9   strncpy(if_idx.ifr_name, ifName, IFNAMSIZ-1);
10  if (ioctl(sockfd, SIOCGIFINDEX, &if_idx) < 0)
11    perror("SIOCGIFINDEX");
12
13  memset(&if_mac, 0, sizeof(struct ifreq));
14  strncpy(if_mac.ifr_name, ifName, IFNAMSIZ-1);
15  if (ioctl(sockfd, SIOCGIFHWADDR, &if_mac) < 0)
16    perror("SIOCGIFHWADDR");
17
18  if (setsockopt(sockfd, SOL_SOCKET, SO_BINDTODEVICE, ifName,
      IFNAMSIZ-1) == -1)  {
19    perror("SO_BINDTODEVICE");
20    close(sockfd);
21    exit(EXIT_FAILURE);
22  }
23
24  eh->ether_shost[0] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
      [0];
25  eh->ether_shost[1] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
      [1];
26  eh->ether_shost[2] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
      [2];
27  eh->ether_shost[3] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
      [3];
28  eh->ether_shost[4] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
      [4];
29  eh->ether_shost[5] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
      [5];
30  eh->ether_dhost[0] = (uint8_t)dest_mac[0];
31  eh->ether_dhost[1] = (uint8_t)dest_mac[1];
32  eh->ether_dhost[2] = (uint8_t)dest_mac[2];
33  eh->ether_dhost[3] = (uint8_t)dest_mac[3];
34  eh->ether_dhost[4] = (uint8_t)dest_mac[4];
35  eh->ether_dhost[5] = (uint8_t)dest_mac[5];
36
37  eh->ether_type = htons(PROTOCOLO);
38
39  socket_address.sll_ifindex = if_idx.ifr_ifindex;
40  socket_address.sll_halen = ETH_ALEN;
41
42  socket_address.sll_addr[0] = (uint8_t)dest_mac[0];
43  socket_address.sll_addr[1] = (uint8_t)dest_mac[1];
44  socket_address.sll_addr[2] = (uint8_t)dest_mac[2];
45  socket_address.sll_addr[3] = (uint8_t)dest_mac[3];
```

```
46    socket_address.sll_addr[4] = (uint8_t)dest_mac[4];
47    socket_address.sll_addr[5] = (uint8_t)dest_mac[5];
48 }
```

En este caso el método *socket*, recibe como parámetros: primero, la macro *AF_PACKET* para indicar al socket que su dominio es de paquetes de bajo nivel, segundo, la macro *SOCK_RAW* para conseguir acceso "bruto" a los datos que se envían y, tercero, el parámetro del campo tipo de la cabecera Ethernet, que identifica el protocolo que se está utilizando.

El siguiente paso es, mediante el comando *ioctl()*, obtener para la interfaz de red por la que se deben enviar los datos, el índice numérico que utiliza el sistema operativo; y con el mismo comando, nuestra dirección MAC en esa interfaz. Mediante *setsockopt()* hacemos que el socket es establezca para escuchar en la interfaz, del mismo modo que se utiliza *bind()* para que el *socket* UDP escuche en una dirección IP.

El resto de instrucciones del método son para guardar los datos de la cabecera en las estructuras que se van a utilizar en el código. El puntero "eh" apunta al principio de la zona de memoria reservada para la trama ethernet, que también está apuntada por la variable "trama" y la variable socket_address se utiliza para la llamada a *sendto()*.

```
1 sendto(sockfd, trama, tam_total, 0, (struct sockaddr*)&
    socket_address, addr_len);
2 recvfrom(sockfd, buffer, TAM_TRAMA_ETH, 0, NULL, NULL);
```

La creación del socket en el servidor es muy similar al cliente, con la excepción de que al no conocer la dirección MAC del cliente, los parámetros destino se fijan una vez se a recibido una trama del cliente, lo que corresponde a las líneas de la 30 a la 47 obviando la 39.

4.4.2. Pruebas

En la Figura 4.10 se puede ver la gráfica que compara las prestaciones del protocolo ya implementado en Ethernet a veinticinco tramas de sincronía contra UDP, al mismo número de tramas de sincronía y TCP. El motivo por el cual no hay una gráfica con el comportamiento de las diferentes versiones de Ethernet cuando se hace variar el número de tramas de sincronía es que siguen en todos los casos el patrón análogo de UDP, que consiguiendo mayores prestaciones.

Como se puede ver en la imagen para tamaños pequeños de mensaje la versión de TCP que no tiene en cuenta el tiempo de conexión resulta ligeramente más eficiente que el protocolo implementado, pero conforme el tamaño del mensaje crece las prestaciones del protocolo suben, siendo mejores que UDP y cualquier versión de TCP. En la comunicación entre los AMD, el protocolo sobre UDP conseguía 950 Mbits/s de ancho de banda, ambas versiones de TCP se aproximan a los 940 Mbits/s y la versión del protocolo que opera directamente a partir de Ethernet alcanza los 970 Mbits/s. Esta diferencia se vuelve abismal cuando comparamos el caso en el que el Pentium es el emisor, UDP ofrece 816 Mbits/s, TCP abriendo la ventana de recepción roza los 770 Mbits/s, mientras que el protocolo sobre Ethernet consigue 966 Mbits/s; casi 200 Mbits/s más que TCP.

La comunicación siendo el Pentium el servidor, sufre los mismos problemas que ya había-

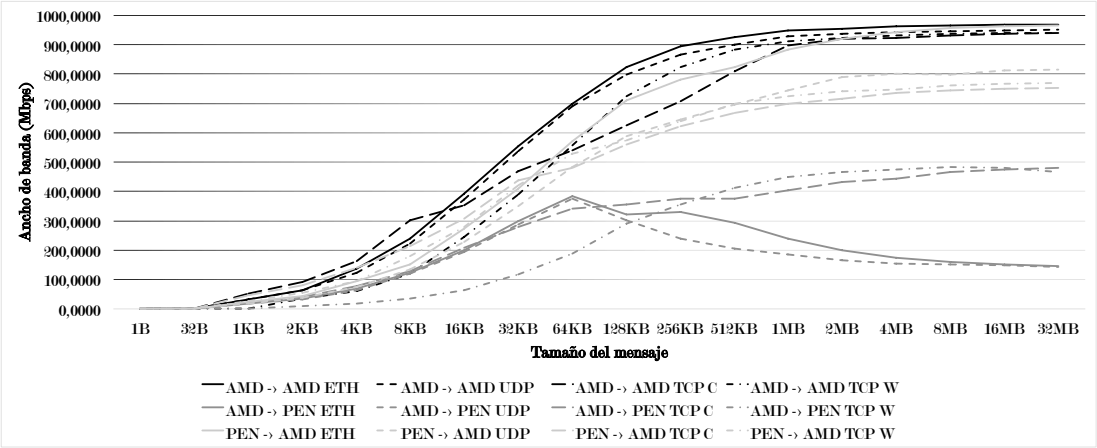

Figura 4.10: Comparación de Ethernet contra UDP y TCP en la red de los AMD y el Pentium

mos visto en la versión sobre UDP. Se pierden tramas, y el reenvío es demasiado costoso, aunque si que se nota un aumento de las prestaciones, debido a la ausencia del coste de IP y UDP y al espacio extra de que disponemos por cada trama.

Pasemos ahora a la otra red, en la Figura 4.11 y la Figura 4.12 se muestra una gráfica con las pruebas del protocolo sobre Ethernet y UDP a veinticinco tramas de sincronía y la versión de TCP que abre la ventana de recepción antes de ejecutar el envío. No se ha utilizado la versión de TCP que no tiene en cuenta el tiempo de conexión porque como se veía en la Figura 4.8 su rendimiento era muy bajo comparado con el de UDP, con lo que no aportaba nada el incluirla de nuevo.

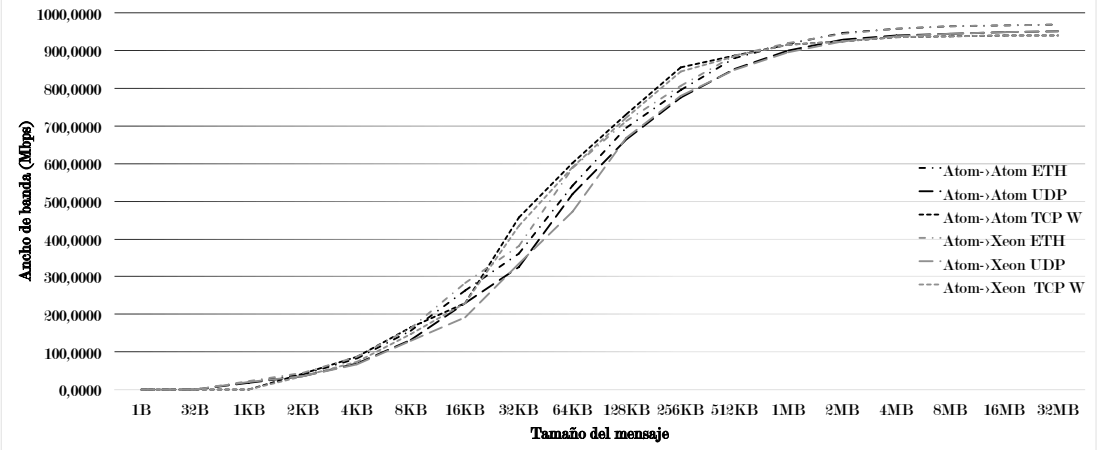

Figura 4.11: Prestaciones de Ethernet contra UDP y TCP en la red de los Atom y Xeon

Lo primero a destacar es la constante de que en todos los escenarios el protocolo implementado sobre Ethernet obtiene un mayor ancho de banda que el protocolo implementado sobre UDP. En el extremo cuando ya se estabiliza el ancho de banda, en Ethernet obtenemos 970 Mbits/s mientras que UDP se queda en 950 Mbits/s. Compararnos con TCP se complica, aunque TCP

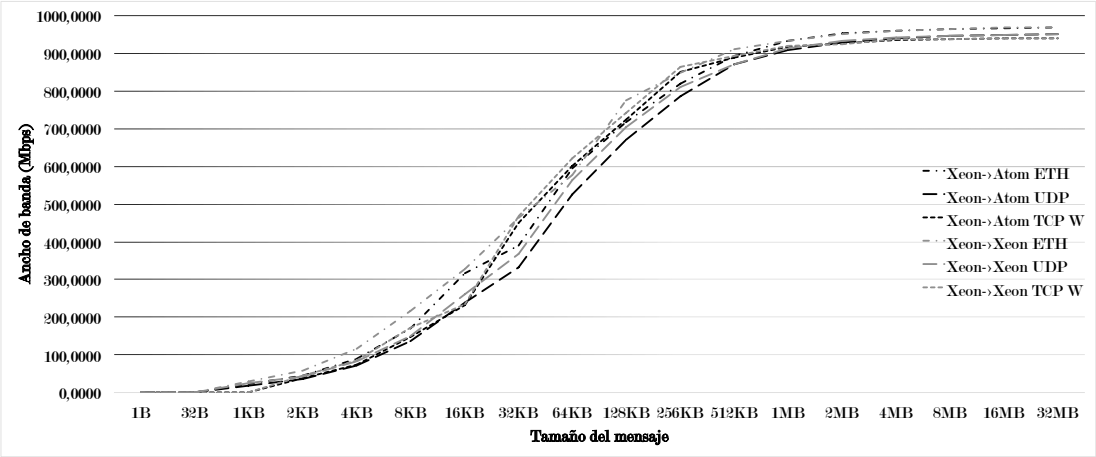

Figura 4.12: Prestaciones de Ethernet contra UDP y TCP en la red de los Atom y Xeon

encuentra su cota en los 940 Mbits/s, si que es cierto que hay algunos puntos intermedios en los que obtiene mayor ancho de banda que nuestro protocolo.

Cabe recordar que estos resultados de TCP los estamos obteniendo con una versión trucada para eliminar parte del sobrecoste que el protocolo impone, con la finalidad de poder afirmar hasta que punto nuestro protocolo es eficiente. Si en el mismo escenario que los mostrados en la gráfica anterior, comparáramos el protocolo implementado sobre Ethernet con la versión básica de TCP, la que tiene en cuenta todo el peso de este protocolo, veríamos la abismal diferencia que existe entre nuestra implementación y TCP. Pero no vamos a teorizar, sino que en la Figura 4.13 se puede ver esta comparación.

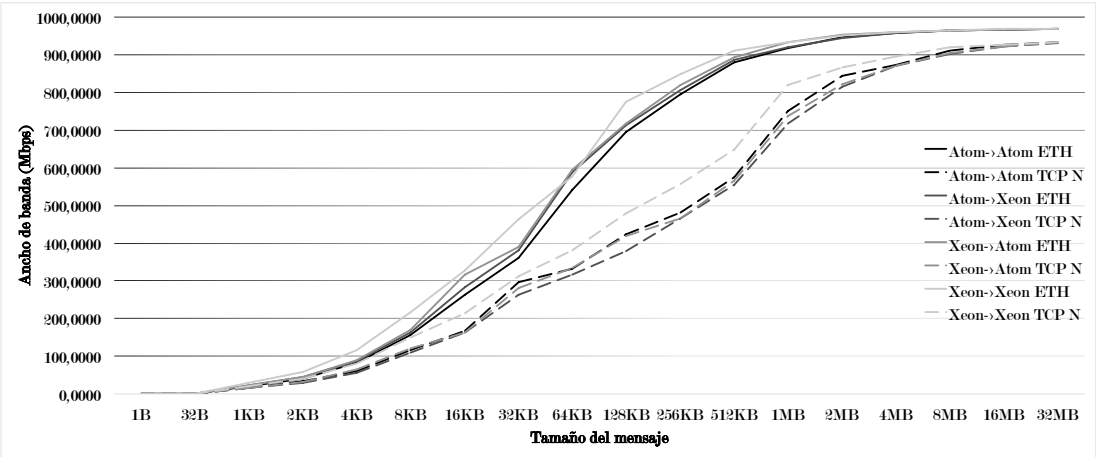

Figura 4.13: Comparación de Ethernet y la versión básica de TCP en la red de los Atom y Xeon

Los resultados que obtenemos muestran que existen tres escenarios claramente diferenciados en resultados cuando probamos cualquier protocolo de comunicación en red, basados en la potencia tanto de cliente como servidor:

- Cliente y servidor potentes, en este caso un protocolo de comunicaciónes funciona de manera fluida, las tramas son enviadas por un nodo y se reciben en el otro, el cual tiene potencia suficiente para manejarlas sin que sus *buffers* se saturen. En este contexto se sitúan tanto la comunicación entre los dos equipos AMD, como todos los escenarios entre los equipos con procesador Atom y Xeon.

- Servidor potente y cliente lento, en este escenario el cliente se convierte en el cuello de botella del sistema de forma que la comunicación fluye despacio pero de forma satisfactoria, el servidor tiene tiempo de sobra para manejar las tramas que le van llegando, y una vez ha terminado solo necesita comunicarse con el cliente para indicarle que la recepción del mensaje se ha realizado correctamente, en este contexto se sitá la comunicación con un AMD como servidor y el Pentium III como cliente.

- Servidor lento y cliente rápido, este es el escenario que más problemas plantea. Cuando el servidor es poco potente existe la posibilidad de que los *buffers* de la tarjeta de red y del *socket* se saturen, de forma que ciertas tramas se pierden irremediablemente. En el protocolo que hemos implementado si esta perdida afecta a todas las tramas de sincronía tanto cliente como servidor se bloquean. En este escenario se sitúa la comunicación entre el Pentium III como servidor y uno de los AMDs como cliente.

Un cuarto escenario sería aquel en el que tanto cliente como servidor son equipos poco potentes, pero como ya hemos visto la penalización sufrida por tener un cliente lento es menor que la de por un servidor lento y, lo que se obtendría en realidad es una mejora en la calidad de la comunicación comparado con el escenario de cliente rápido y servidor lento, puesto que al disminuir la potencia del cliente, este tarda más en enviar cada trama con lo que el servidor cuenta con más tiempo para manejarlas. Así pues, el agobio de los *buffers* sería menor y la cantidad de tramas recibidas por iteración aumentaría.

Visto esto, y con la finalidad de reducir la cantidad de datos en las gráficas en los capítulos siguientes se mostrarán los resultados de únicamente el escenario de cliente y servidor rápidos, concretamente cliente y servidor AMD, con la finalidad de ver como afectan las modificaciones realizadas en el protocolo a las prestaciones de la versión presentada en este capítulo así como seguir comparándolas con TCP, las proporciones en las que las prestaciones se vean modificadas son extendibles al resto de escenarios excepto al otro del cual se van a incluir datos, el de servidor lento y cliente rápido.

Capítulo 5

Conseguir robustez en la fiabilidad

5.1. Planteamiento del problema y propuesta de solución

Uno de los problemas que encontramos al realizar las pruebas con un servidor poco potente fue que, además de poder perderse tramas de datos, cosa para la cual el protocolo es capaz de adaptarse, podían perderse todas las tramas de sincronía, haciendo que cliente y servidor bloqueen irremediablemente sus procesos. Por ejemplo, esto puede pasar cuando el emisor ha enviado todos los datos de la aplicación y envía las tramas de sincronía final, acabando el envío y ejecutando *recvfrom()* para escuchar en el socket, momento en el cual el proceso se suspende hasta que reciba información; el receptor, por su parte, no recibe ninguna de las tramas de sincronía finales con lo que también se queda suspendido a la espera de las sincronías.

Para solucionar el problema que se nos plantea deberíamos evitar bloquear los procesos. C ofrece una serie de funciones con las cuales se puede "preguntar" al *socket* si hay información disponible para leer sin bloquear el hilo o bloqueándolo hasta que transcurre un determinado tiempo de espera, de esta forma no necesitamos ejecutar *recvfrom()* a menos que sepamos seguro que hay tramas en la tarjeta de red y el proceso no se va a bloquear. Entre estas funciones se encuentran:

- *ioctl()*[5]: si se ejecuta esta función pasándole como parámetros: el socket, la macro *FIONREAD* que indica que queremos saber si el socket tiene disponibles datos para lectura, y un puntero a un entero; el valor de retorno de la función es un entero que valdría -1 si hubiese habido un error interno de la función, pero el hecho de que la función se haya ejecutado correctamente, el valor de retorno es cero, no indica que haya datos disponibles para lectura. Se debe consultar el valor que se haya escrito en el tercer parámetro. Esta variable indica cuantos bytes ocupa la primera trama disponible para lectura en el socket.

  El problema principal de esta función es que obtiene la información de si, en el preciso momento en que se ejecuta, hay datos para leer. Si queremos utilizarla en nuestro protocolo deberíamos hacer la "pregunta" en bucle hasta que hayan datos. Pero esto no es nada eficiente, podríamos suspender el hilo con un *sleep()* entre pregunta y pregunta, ¿pero cuantos micro segundos serían aceptables?

```
1 /* Librer\'ia necesaria */
2 #include <sys/ioctl.h>
```

```
/* Llamda a la funci\'on */
int ioctl(int socket, unsigned long request, void *argp);
```

- *select()*[8]: esta función recibe cinco argumentos: primero el número de descriptores de archivo a los que tiene que atender como máximo, o la macro *FD_SETSIZE*. Los tres parámetros siguientes reciben una estructura *fd_set* que no es sino un conjunto de todos los descriptores de los que se quiere saber si están disponibles para lectura, escritura o excepciones respectivamente; el quinto argumento es una estructura de tipo *timeval* para especificar el tiempo máximo de espera que si se supera el sistema operativo despierta el proceso aunque no se hayan modificado los descriptores.

  El valor de retorno de *select()* es el número de descriptores de archivo que están disponibles: cero si se ha alcanzado el tiempo máximo y -1 si ha ocurrido algún error. Esta función es más eficiente que la anterior pues no tenemos que estar "preguntandole" al *socket* constantemente si tiene datos para leer, sino que suspendemos el hilo hasta que el sistema operativo nos despierte. El problema es que seguimos teniendo que indicar un valor temporal máximo en el cual suponemos que el otro extremo de la comunicación se ha bloqueado. Además *select()* puede modificar los valores del conjunto y de la estructura temporal durante su ejecución, con lo que cada vez que se va a utilizar la función se deben construir sus argumentos.

```
/* Librer\'ia necesaria */
#include <sys/select.h>
/* Cracion del set */
fd_set set;
/* Borrar la informaci\'on en el set */
void FD_ZERO(&set);
/* Introducir un socket en el set */
void FD_SET(int socket, fd_set *set);
/* LLamada a la funci\'on */
int select(int nfds, fd_set *readfds, fd_set *writefds,
    fd_set *exceptfds, struct timeval *timeout);
```

- *pselect()*[7]: es una evolución de la función *select()*. La estructura temporal que recibe es un *timespec* que permite indicar el tiempo máximo con una precisión de nanosegundos, mientras que la estructura que utiliza *select()* tiene una precisión máxima de microsegundos, además tiene un sexto argumento, que permite modificar la *sigmask* del proceso. El mayor punto a favor de utilizar esta función en lugar de su hermana menor, es que no modifica los argumentos durante su ejecución.

```
/* Librer\'ia necesaria */
#include <sys/select.h>
/* Cracion del set */
fd_set set;
/* Introducir un socket en el set */
void FD_SET(int socket, fd_set *set);
/* Uso de la funci\'on */
```

```
8 int pselect(int nfds, fd_set *readfds, fd_set *writefds,
      fd_set *exceptfds, const struct timespec *timeout, const
       sigset_t *sigmask);
```

- *poll()[6]*: la última de las funciones que vamos a explicar. Recibe tres parámetros, un puntero a una estructura *pollfd* con los descriptores de fichero y una máscara de bits indicando qué evento se quiere atender (disponible para lectura, escritura...), el número de descriptores y un entero que indica el tiempo de espera máximo en milisegundos. El valor de retorno es equivalente al de *select()*.

  Una estructura *pollfd* contiene tres argumentos, uno para indicar el descriptor de fichero, otro para que el usuario indique, en forma de máscara de bits, los eventos que se quieren atender, si lectura, escritura o excepciones. El tercer argumento es equivalente al anterior pero lo modifica la función, en vez del usuario, y se ha de comprobar para poder saber cuál de los eventos se ha producido en el descriptor.

```
1 /* Librer\'ia necesaria */
2 #include <poll.h>
3 /* Creaci\'on de la estructura */
4 struct pollfd fds[1];
5 /* Introducci\'on del socket en la estrucctura */
6 fds[0].fd = socket;
7 fds[0].events = POLLIN;
8 /* Uso de la funci\'on */
9 int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

Todas las funciones que hemos visto, plantean el problema de utilizar un valor como tiempo de espera máximo. Teniendo en cuenta que, el envío de 32 MBytes entre dos equipos Atom tarda en ejecutarse 0,277076 segundos, el valor que podemos colocar de espera ha de ser muy pequeño, pero las precisiones de milisegundos que ofrecen *select()* y *poll()* son suficientes.

Ahora que tenemos una forma de no bloquear el proceso al recibir información debemos buscar la forma más eficiente de asegurar la fiabilidad. Ya que existen muchas condiciones que pueden bloquear ambos procesos de forma más o memos perjudicial. Supongamos que el servidor únicamente ha fallado en la recepción de todas las tramas de sincronía final. Sería fácil despertarlo con una nueva trama de sincronía final, y esto puede hacerlo el cliente si en vez de bloquear el proceso, escucha en el *socket* y si se excede el tiempo máximo de espera reenvía la trama de sincronía final.

Por otro lado, en el caso poco probable de que el servidor no reciba la primera trama de sincronía inicial del cliente, ya no se trata solo de reenviársela para que se despierte, sino de que todo todas las tramas de datos que hemos enviado han sido ignoradas, pues el servidor no ha podido reservar espacio para el mensaje y además no reconoce al cliente. En este caso, conseguir que se sincronicen el emisor y receptor se complica, porque además el cliente tampoco sabe exactamente qué es lo que no se ha recibido. ¿Enviamos las sincronías finales y si tras otro tiempo de espera no obtenemos respuesta enviamos iniciales y finales?

En resumen, en la manera en que está hecha la comunicación ahora no se puede garantizar la fiabilidad, o el coste de garantizarla es muy grande. Con el fin de conseguir un protocolo que realmente sea fiable, vamos a implementar unas nuevas versiones y poner a prueba sus prestaciones aunque más importante de si el ancho de banda sigue siendo el mismo que con la versión original, debemos tener en cuenta que las distintas ejecuciones del protocolo siempre terminen su ejecución correctamente.

Utilizando un socket TCP para las sincronías

En esta versión vamos a despreocuparnos de si las tramas de sincronía llegan o no, y vamos a trasmitirlas a través de TCP. Ambos cliente y servidor ahora van a tener dos *sockets*, el *raw socket* que utilizamos para mandar las tramas utilizando solo Ethernet y la cabecera del protocolo y un segundo socket que utilizará TCP y utilizaremos únicamente para el envío de las tramas de sincronía.

Como utilizamos TCP sabemos que la información que enviemos va a ser recibida, así que cuando uno de los extremos esté recibiendo tramas se servirá de la función *ioctl()* para saber si alguno de los sockets tiene información disponible para lectura. Siempre se consultará primero el *raw socket* y solo en caso de que este no tenga información, consultaremos el de TCP, así nos aseguramos de que hemos recibido todas las tramas de datos posibles antes de ver si las tenemos todas o no.

Reenviando las sincronías si se excede el tiempo de espera

En esta implementación vamos a mantener solo el socket Ethernet y vamos a utilizar algunas de las funciones antes explicadas para crear un cliente que, si no recibe información del servidor tras un tiempo de espera, reenvíe las sincronías. A parte de esto, como estamos desarrollando una versión fiable, vamos a eliminar el envío de "n" tramas de sincronía, sino que vamos a enviar únicamente una cada vez que sea necesario.

Además de esto, vamos a modificar cómo se establece la comunicación: se van a eliminar todas las sincronías iniciales excepto la primera pues le proporciona al servidor el tamaño del mensaje. El comportamiento en el servidor es el mismo de siempre, pero el cliente se va a modificar. Ahora, una vez realizado el envío de todas las tramas, bloqueamos el hilo con una de las funciones vistas anteriormente, y si no recibimos nada antes de que se agote el tiempo de espera, reenviamos únicamente la trama de sincronía inicial y final, no todo el mensaje.

La otra modificación es que, a la hora de bloquear el proceso esperando el reconocimiento final o la petición de tramas en el cliente, vamos a escuchar en el socket sin bloquearlo y si no recibimos ninguna trama de sincronía en un tiempo máximo reenviamos la sincronía final, con el objetivo de despertar al servidor. Vamos a implementar esta propuesta utilizando tanto *select()* como *poll()*, para comprobar si una de ellas tiene mayor sobre coste que otra siendo que su utilidad es la misma.

5.2. Pruebas

Vamos a comparar el comportamiento de las nuevas versiones que hemos implementado entre dos terminales potentes como son los AMDs. En este contexto no vamos a aprovechar las nuevas prestaciones para garantizar la fiabilidad pues esta no se veía vulnerada; pero nos sirve para saber cuánto hemos perdido por garantizar la fiabilidad y asegurarnos de que no nos hemos vuelto peores que TCP. Las prestaciones de cada uno se pueden ver en la gráfica de la Figura 5.1.

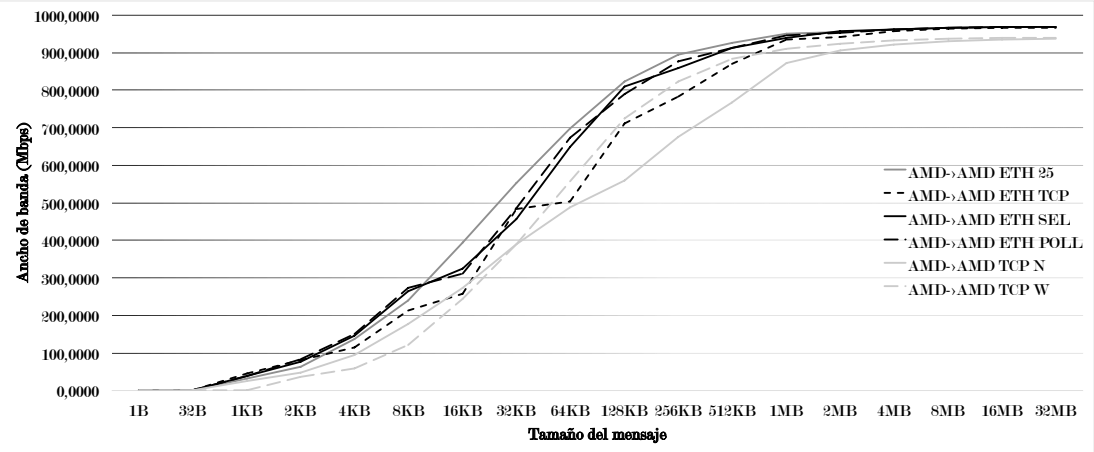

Figura 5.1: Comparación de las versiones fiables de Ethernet, la no fiable y TCP entre los AMD

Lo primero que se observa es que los programas que solo utilizan TCP ofrecen menor ancho de banda en todos los casos que las versiones del protocolo. Si pasamos a comparar las distintas versiones, vemos que la que utiliza dos sockets, uno para Ethernet y otro para TCP, es la versión menos potente de las que hemos implementado, poniéndose en algunos casos por debajo de la versión de TCP que realiza una primera apertura de la ventana de recepción.

Pasemos ahora a comparar las versiones que únicamente utilizan Ethernet. Aunque hay algunos picos debido a que hay muchos factores en la red que pueden hacer que de una prueba a otra se vayan algunos milisegundos que al pasar a mega bytes por segundo influyen mucho, vemos que la diferencia entre utilizar poll y select es nimia. Ambas versiones son realmente fiables y, aunque pierden un poco de ancho de banda comparadas con la versión original, el valor de cota de las tres versiones tienden a establecerse en el mismo valor.

Pasemos ahora a estudiar el escenario para el cual se ha realizado específicamente este capítulo, pues es en el que pudimos observar que con un servidor lento la perdida de tramas posibilitaba el bloqueo de los procesos. En la Figura 5.2 podemos observar los resultados de ancho de banda de cada una de las nuevas versiones, lo más importante que no se puede apreciar en la gráfica es que las pruebas son satisfactorias y las tres versiones implementadas han resultado fiables en cada una de las pruebas realizadas.

Observamos que la versión que utiliza el protocolo TCP para comunicar las sincronías obtiene menores prestaciones cuando lo comparamos con el resto de versiones en el pico de ancho de banda que se produce al enviar un mensaje de 64KB, que es el tamaño de mensaje más grande que se envía sin perdida de tramas.

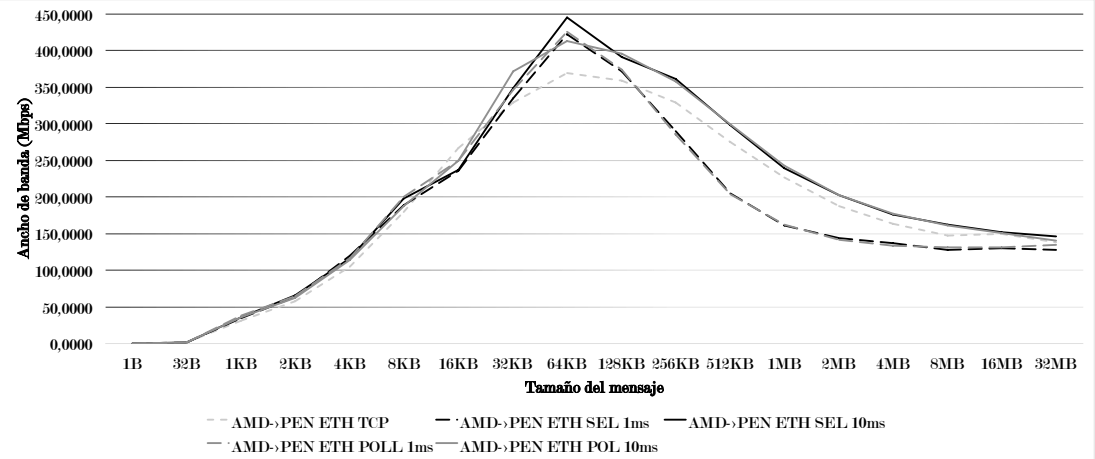

Figura 5.2: Comparación de las versiones fiables de Ethernet con el Pentium como servidor

Con respecto a las versiones que utilizan funciones para escuchar en el *socket* se han realizado pruebas empleando un tiempo máximo de espera de uno y diez milisegundos. En las pruebas realizadas con 1 ms de tiempo de espera observamos una perdida enorme de ancho de banda debida a que el cliente se adelanta demasiado a la hora de suponer que el servidor no ha recibido la sincronía de finalización de envío y le pide un nuevo reenvío de las tramas que faltan. Esto hace que el servidor reciba los números de secuencia por duplicado, realizando un doble envío sin dar tiempo a que las tramas de la tarjeta de red receptora se manejen en su totalidad.

Al contrario vemos que emplear 10 ms resulta muy eficiente y se consigue un ancho de banda mayor que con la versión que confía en TCP para el envío de las sincronías, aunque en el valor extremo las tres implementaciones tienden al mismo valor. Dado que no hay gran diferencia entre las funciones *poll* y *pselect*, se ha decido utilizar la segundo para incluirla en la versión estable del protocolo.

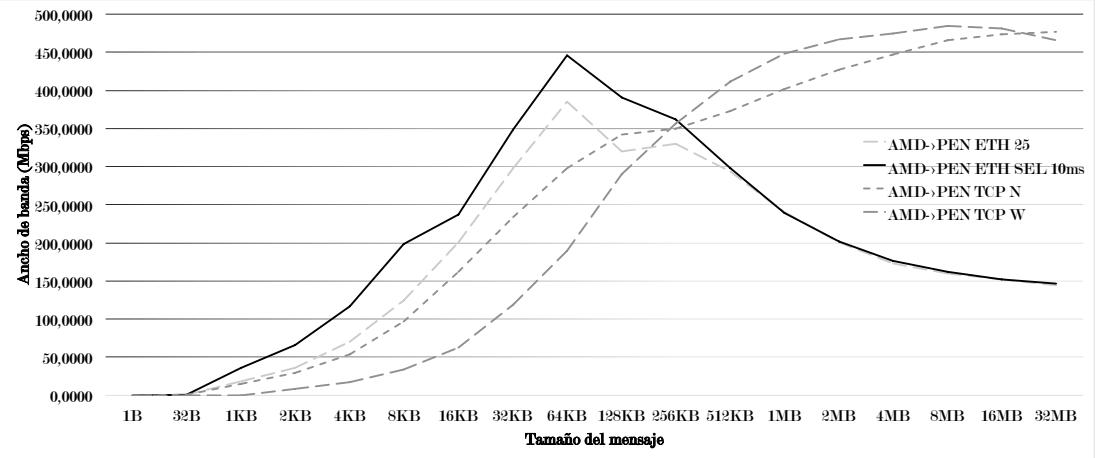

Figura 5.3: Comparación de la versión fiable de Ethernet, la no fiable y TCP con el Pentium como servidor

En la Figura 5.3 se muestra la comparación de esta nueva implementación con las anteriores.

Se aprecia un aumento notable en las prestaciones de la nueva versión, consiguiendo un ancho de banda mucho mayor para un mensaje de pequeño tamaño así como manteniéndose por encima de TCP hasta los 256KB de mensaje. Aunque conforme seguimos aumentando el tamaño del mensaje la nueva versión se equipara con la antigua.

Al final, se ha conseguido alcanzar el objetivo del capítulo, se ha implementado el protocolo de forma que resulte fiable en el caso de que la comunicación se produzca con un servidor lento, pero tenemos que confiar en el uso de un parámetro de tiempo de espera establecido a priori que puede, si es demasiado pequeño, suponer un lastre para el protocolo. Solo queda tratar de mejorar las prestaciones del protocolo para tamaños de mensaje grandes, cosa que trataremos de conseguir en el próximo capítulo, introduciendo control de congestión para distanciar temporalmente la recepción de tramas en el receptor de forma que sea capaz de manejar la mayor cantidad posible, teniendo que realizar pocas sobreescrituras en la tarjeta de red.

Capítulo 6

Implementando control de congestión

6.1. Planteamiento del problema y propuesta de solución

Uno de los problemas que tiene el protocolo cuando se ejecuta con un servidor poco potente en relación al cliente, es que se pierden tramas, aunque también puede ocurrir entre dos equipos de igual potencia si la red es poco potente o tiene mucho tráfico. Actualmente el protocolo es capaz de recuperarse de la pérdida de tramas, pero de una forma poco eficiente como hemos podido observar cada vez que lo probábamos con uno de los AMD como cliente y el Pentium III como servidor.

En las siguientes gráficas se puede observar el proceso completo de recepción de tramas en una comunicación en el contexto mencionado con un mensaje de 256 KB. Vamos a utilizar el término "iteración" para definir cada ronda de envíos. Así pues, la iteración cero es el estado del servidor antes de recibir ningún dato, la primera iteración es cuando el cliente envía la sincronía inicial con el tamaño del mensaje y el mensaje, y las iteraciones posteriores incluyen unicamente el envío de las tramas que el servidor notifica que le faltan.

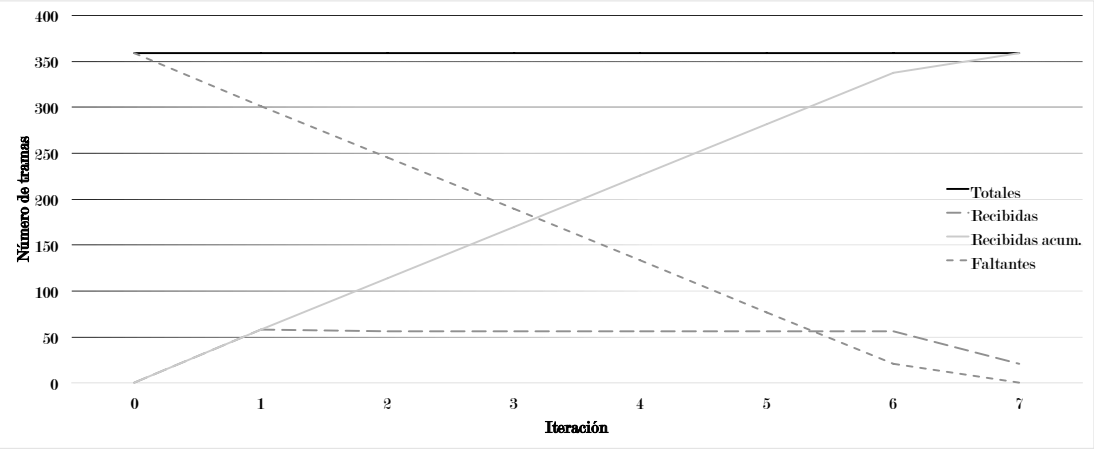

Figura 6.1: Resumen de una ejecución con perdidas para 256KB

La gráfica de la Figura 6.1, muestra el estado del servidor en cada iteración. Aunque los datos fueron extraídos de una ejecución del protocolo utilizando UDP, nos sirven para ilustrar

que el número de tramas recibidas en cada iteración parece ser constante. En cada iteración el servidor es capaz de leer de la tarjeta de red cincuenta y seis tramas y se leen en bloque. Esto quiere decir que las nuevas tramas que van llegando no sustituyen a las que ya se encuentran en el buffer del *socket* sino que son ignoradas por el kernel.

La gráfica de la Figura 6.2 aporta más información al funcionamiento del buffer del *socket*: en la primera iteración se han recibido cincuenta y siete tramas. Esta discrepancia de una trama se debe a que el número de tramas es lo bastante grande como para que otra trama, en este caso la doscientos ochenta y tres entre en el buffer del *socket*.

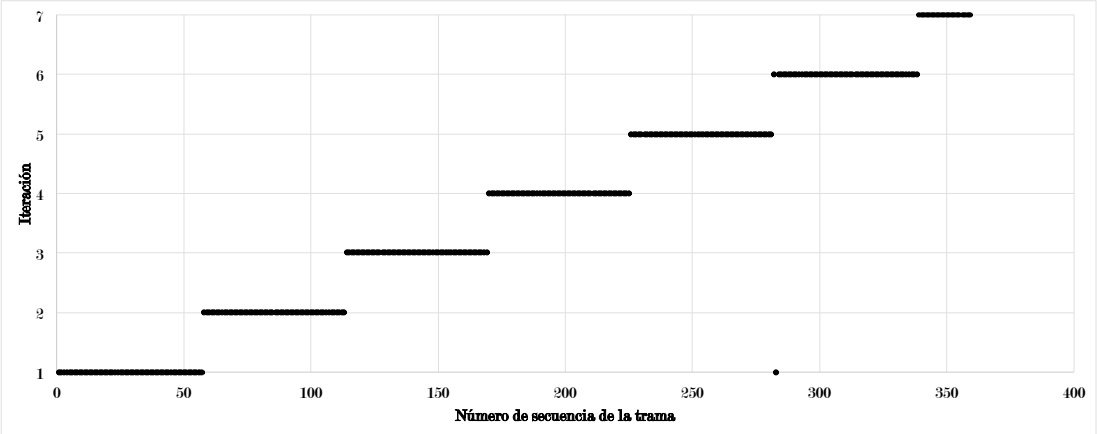

Figura 6.2: Recepción de cada trama por cada iteración para 256KB

Vamos a mejorar el protocolo y vamos a hacer que envíe las tramas fijándose en este tamaño de bloque que el buffer es capaz de cargar. Vamos a modificar únicamente el envío de las tramas que faltan, introduciendo un *sleep()* cada vez que se han enviado el número de tramas especificadas por el bloque. El tamaño de bloque solo lo conocemos cuando el servidor nos notifica las tramas faltantes y no es igual al número de tramas que sí ha recibido. Pues como ya hemos visto en la primera iteración con un mensaje de 256 KB el número de tramas recibidas aumenta en uno, pero este aumento crece conforme aumenta el tamaño del mensaje. Por lo tanto, el tamaño de bloque va a ser igual al número de secuencia de la primera trama que falta menos uno.

6.2. Pruebas

Vamos a continuación, una vez implementada esta versión a probarla. La Figura 6.3 muestra la ejecución de la versión normal del protocolo sobre Ethernet con 25 tramas de sincronía, la versión del protocolo utilizando *select()* para escuchar en el *socket* sin bloquearlo, la nueva versión con control de congestión que tiene todas las características de la anterior y la ejecución de dos de las versiones de TCP, la que tiene todo el sobrecoste en cuenta y la que abre la ventana de recepción.

Podemos observar que la nueva versión tiene unas prestaciones equivalentes a la fiable, pues las modificaciones realizadas no afectan en este contexto puesto que no se pierden tramas. Mientras tanto la Figura 6.4 resume las pruebas con el Pentium como servidor. En esta gráfica se

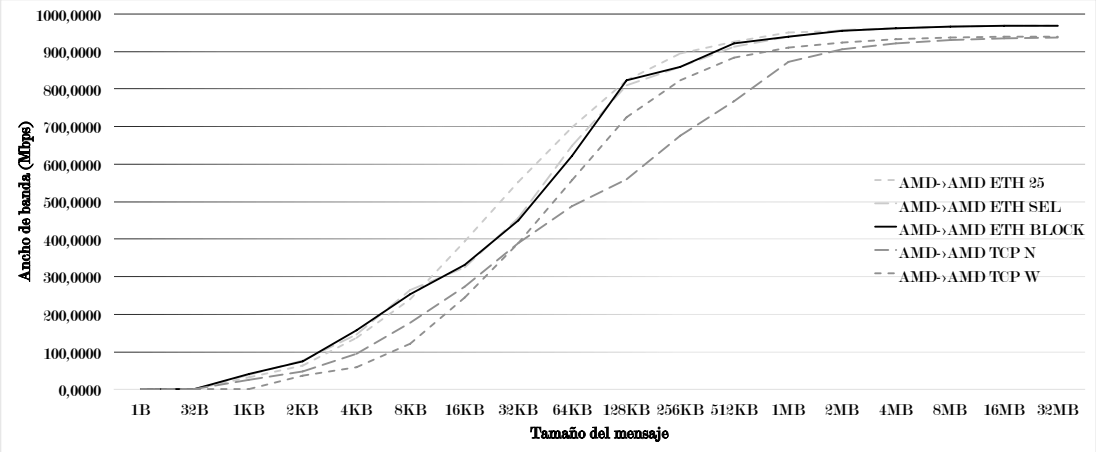

Figura 6.3: Prueba de la versión con control de congestión entre los AMD

han realizado dos ejecuciones de la versión con control de congestión: en la primera se utiliza un tiempo de espera máximo de un milisegundo tanto en las instrucciones *select()* como en el *sleep()* entre en envío de un bloque de tramas y el envío del siguiente; en la otra se ha introducido en el *select()* un tiempo de espera de diez milisegundos.

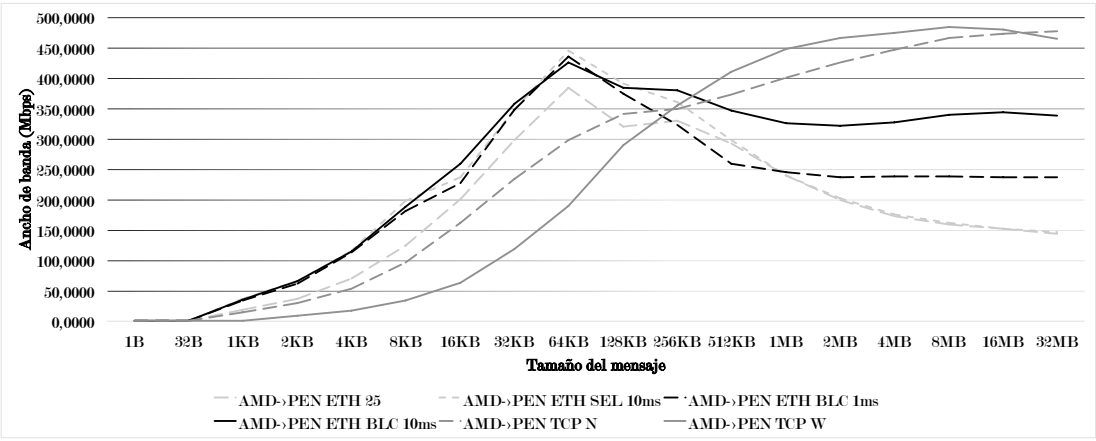

Figura 6.4: Prueba de la versión con control de congestión con el Pentium como servidor

Se aprecia una gran mejoría en las prestaciones del protocolo. Hemos pasado de, en el extremo, rondar los 150 Mbits/s de las versiones que no tienen en cuenta la congestión, a 330 Mbits/s en la versión que utiliza un tiempo de espera de 10 ms, y de nuevo vemos que utilizar solo 1 ms de en el *select()* es excesivamente poco y lleva a que haya mayor número de iteraciones de las que debería. Aunque seguimos por debajo de TCP en este contexto, hemos duplicado el ancho de banda conseguido, acercándonos a los 470 Mbits/s que alcanza TCP.

Capítulo 7

Conclusiones

El protocolo TCP presenta una gran sobrecarga, necesaria para poder ofrecer fiabilidad en sus comunicaciones; mientras que UDP es un protocolo que no garantiza la entrega de todos los datos. Además ambos protocolos operan necesariamente a partir del protocolo IP.

El trabajo presentado en esta memoria ha mostrado el diseño y la implementación de un nuevo protocolo, pensado para operar únicamente en un escenario en el cual las características que ofrece el protocolo IP son completamente innecesarias. Por este motivo se plantea la posibilidad de implementar un nuevo protocolo de comunicaciones a nivel de transporte que no utilice la capa de red sino que se asiente directamente sobre la capa de enlace de datos, concretamente el protocolo Ethernet.

En este trabajo final de grado, hemos diseñado un protocolo que solo utiliza ocho bytes de la zona de datos de la trama Ethernet como cabecera, que garantiza la fiabilidad y que incorpora un concepto muy básico, en comparación con TCP, de control de congestión.

Los resultados que hemos obtenido al probar el protocolo implementado contra el protocolo TCP han sido en general muy satisfactorios, aunque cuando el escenario de prueba consiste en un servidor poco potente recibiendo datos de un cliente muy potente, hemos necesitado replantear algunos de los conceptos de la versión original para adaptarlo a mejorar las prestaciones del protocolo en este escenario, hemos mejorado la robustez de la fiabilidad del protocolo, ya que la primera versión no era fiable, así como incluido control de congestión.

En gran parte de los casos, las pruebas realizadas comparando el ancho de banda del protocolo implementado con el que ofrece TCP, demuestran que hemos implementado un protocolo que garantiza las mismas características de una comunicación TCP, pero consiguiendo una mayor cota superior de ancho de banda, además de mejores prestaciones con paquetes de pequeño tamaño.

7.1. Líneas futuras

Hemos visto que el protocolo que hemos implementado resulta poco eficiente cuando tenemos un servidor poco potente. Entre las mejoras que podrían implementarse para aliviar esta pérdida de prestaciones se encuentran:

- **Implementación multihilo**: para conseguir que el servidor sea más rápido realizando la lectura de las tramas entrantes podríamos utilizar hilos para que el hilo principal solo realice las lecturas en el *socket* y se pase el resto del trabajo a otro hilo.

- **Incluir permanencia**: una vez se ha realizado una comunicación con un host, se debería almacenar en memoria persistente, un fichero que almacene la dirección MAC del servidor así como el número de tramas que puede recibir antes de que el buffer de su tarjeta de red se bloquee. La información de este fichero debería ser almacenada mediante algún tipo de función hash que permite sobretodo la recuperación de la información de forma eficiente.

  De esta forma, al realizar una posterior comunicación en vez de enviar todas las tramas a la máxima potencia que consiga el cliente, se realizaría el envío por bloques. En caso de que se produjese una modificación en el hardware de algún equipo del cluster o tanto en el ancho de banda bruto como en la latencia de la red, se deberían realizar las modificaciones pertinentes en el fichero creado por el protocolo, para borrar cualquier estimación que se pueda ver afectada por la modificación realizada.

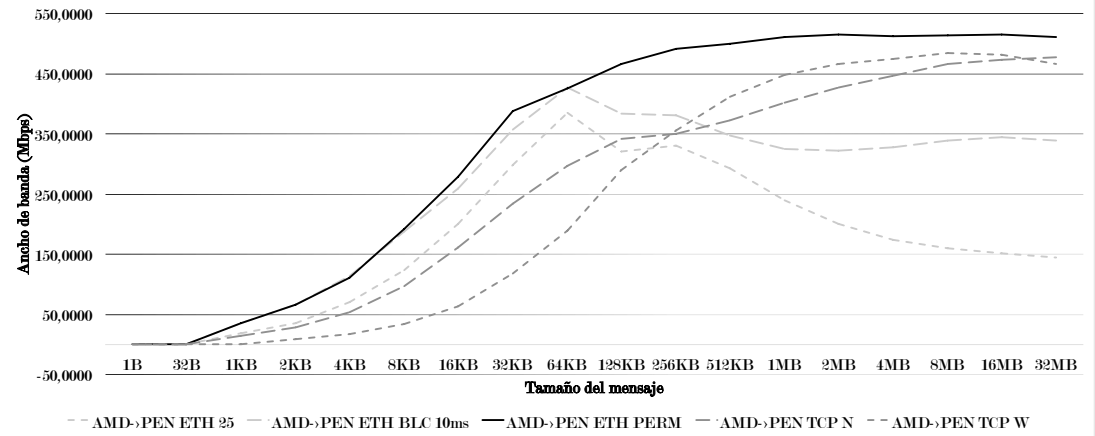

Figura 7.1: Prestaciones de una versión con permanencia

  Vemos en la Figura 7.1 las prestaciones que conseguiría una versión que conociese a priori el número de tramas que la tarjeta de red del servidor puede recibir sin comenzar a sobreescribir. Vemos que las prestaciones son muy similares a las versiones anteriores del protocolo hasta que superamos los 64 KB de mensaje, momento en el cual las prestaciones de las otras versiones comienzan a descender, mientras que el ancho de banda conseguido por la nueva versión sigue aumentando, manteniéndose en todo momento por encima de todas las implementaciones de TCP.

- **Migrar a redes de altas prestaciones**: una vez implementado el protocolo para Ethernet, queda la puerta abierta para realizar el mismo estudio en redes de altas prestaciones como *InfiniBand*. Este bus de comunicaciones presente en los *clusters*, en su versión básica ofrece una velocidad bruta de 2,5 Gbps llegando a ser eficaz solo en 2 Gbps y su versión más potente ofrece 100 Gbps de velocidad bruta y con un caudal eficaz de 96 Gbps [13].

  Esta red también tiene un modo en bruto para el envío y la recepción de información, habría que estudiar como se implementa, modificar el tamaño máximo de una trama así

como el de las cabeceras. Excepto por esto, el código del protocolo implementado debería ser altamente reutilizable.

- **Probar en otras versiones de Ethernet**: Además de realizar un estudio similar en redes de altas prestaciones, también se podría estudiar el comportamiento de este protocolo en redes que utilicen versiones más potentes de Ethernet como 40 o 100 Gb Ethernet.

Bibliografía

[1] HERRIN, GLENN: «Linux IP Networking: A Guide to the Implementation and Modification of the Linux Protocol Stack», 2000. [Online; accessed 30-06-2015].
http://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html

[2] JAMES F. KUROSE, KEITH W. ROSS: Redes de computadores: un enfoque descendente. Traducido por Vuelapluma S.L.U. PEARSON EDUCACIÓN, S. A., 2010.

[3] KERRISK, MICHAEL: Linux Programmer's Manual 2nd: gettimeofday, 2012.
http://man7.org/linux/man-pages/man2/gettimeofday.2.html

[4] ——: Linux Programmer's Manual 2nd: clock_gettime, 2013. [Online; accessed 30-06-2015].
http://man7.org/linux/man-pages/man2/clock_gettime.2.html

[5] ——: Linux Programmer's Manual 2nd: ioctl, 2015. [Online; accessed 30-06-2015].
http://man7.org/linux/man-pages/man2/ioctl.2.html

[6] ——: Linux Programmer's Manual 2nd: poll, 2015. [Online; accessed 30-06-2015].
http://man7.org/linux/man-pages/man2/poll.2.html

[7] ——: Linux Programmer's Manual 2nd: pselect, 2015. [Online; accessed 30-06-2015].
http://man7.org/linux/man-pages/man2/pselect.2.html

[8] ——: Linux Programmer's Manual 2nd: select, 2015. [Online; accessed 30-06-2015].
http://man7.org/linux/man-pages/man2/select.2.html

[9] MAH, BRUCE A.: iperf3, 2015. [Online; accessed 30-06-2015].
https://github.com/esnet/iperf

[10] MARK GATES, AJAY TIRUMALA: iperf, 2014. [Online; accessed 30-06-2015].
https://iperf.fr

[11] MARTON, AUSTIN: «Sending raw Ethernet packets from a specific interface in C on Linux», 2011. [Online; accessed 30-06-2015].
https://austinmarton.wordpress.com/2011/09/14/sending-raw-ethernet-packets-from-a-specific-interface-in-c-on-linux/

[12] SCHAUFLER, ANDREAS:. «Linux Network Performance: RAW ethernet vs. UDP». [Online; accessed 30-06-2015].
http://aschauf.landshut.org/fh/linux/udp_vs_raw/index.html

[13] WIKIPEDIA: «InfiniBand — Wikipedia, The Free Encyclopedia», 2015. [Online; accessed 30-06-2015].
https://en.wikipedia.org/wiki/InfiniBand

Lista de Abreviaturas

ETH BLOCK n ms Nombre de la versión del protocolo implementado sobre Ethernet que introduce control de congestión, con un tiempo máximo de espera de "n" mili segundos en la instrucción *select*, por defecto el tiempo entre envíos de bloques es de 1 ms.

ETH n Versión del protocolo implementado directamente sobre Ethernet utiliando "n" tramas de sincronía, si se obvia el número se suponen 25 tramas.

ETH POLL n ms Nombre de la versión del protocolo que utiliza la instrucción *poll()* para escuchar en el *socket*, con un tiempo de espera máximo de "n" mili segundos

ETH SEL n ms Nombre de la versión del protocolo que utiliza la instrucción *select()* para escuchar en el *socket*, con un tiempo de espera máximo de "n" mili segundos

ETH TCP Nombre de la versión del protocolo que utiliza TCP para el envío de las tramas de sincronía.

ICMP Internet Control Message Protocol

IP Internet Protocol

Mbps, Mbits/s, Mb/s Megabit por segundo

TCP Transmission Control Protocol

TCP C Versión de TCP que no tiene en cuenta el tiempo de establecimiento de la conexión

TCP N Versión de TCP que tiene en cuenta todo el sobre coste de este.

TCP W Versión de TCP que no tiene en cuenta el tiempo de establecimiento de la conexión y además realiza un primer envío con la finalidad de abrir la ventana de recepción

UDP User Datagram Protocol

UDP n Versión del protocolo implementado sobre UDP utilizando "n" tramas de sincronía, si se obvia el número se suponen 25 tramas.

Apéndice A

Parametros.h

```
1  //
2  //   Parametros.h
3  //   TFG
4  //
5
6  #ifndef TFG_Parametros_h
7  #define TFG_Parametros_h
8
9  #define TAM_TRAMA_ETH 1514
10 #define TAM_HEAD_ETH 14
11 #define TAM_DATOS_ETH 1500
12 #define TAM_HEAD_PROTO 8
13 #define TAM_DATOS_PROTO 1492
14 #define TAM_HEADS 22
15 #define PROTOCOLO 0x08D0
16
17 #define FLAG_INICIAL 0
18 #define FLAG_FINAL 1
19 #define FLAG_ACK 2
20
21 #define TIMEOUT_S 0
22 #define TIMEOUT_1MS 1000000
23 #define TIMEOUT_10MS 10000000
24
25 typedef unsigned int uint;
26 typedef unsigned short ushort;
27 typedef unsigned char ubyte;
28
29 #endif
```

Apéndice B

envio.h

```
1  //
2  //  envio.h
3  //  TFG
4  //
5
6  #ifndef TFG_envio_h
7  #define TFG_envio_h
8
9  #include <stdio.h>
10 #include <unistd.h>
11 #include <stdlib.h>
12 #include <string.h>
13 #include <arpa/inet.h>
14 #include <linux/if_packet.h>
15 #include <sys/ioctl.h>
16 #include <netinet/ether.h>
17 #include <net/if.h>
18 #include <time.h>
19 #include <sys/time.h>
20
21 #include "Parametros.h"
22
23 int sockfd;
24 struct sockaddr_ll socket_address;
25 int addr_len = sizeof(struct sockaddr_ll);
26
27 ubyte trama[TAM_TRAMA_ETH];
28 struct ether_header *eh = (struct ether_header *) trama;
29 uint *cabecera = (uint *)(trama + sizeof(struct ether_header));
30 ubyte *datos = trama + sizeof(struct ether_header) +
      TAM_HEAD_PROTO;
31 ubyte buffer[TAM_TRAMA_ETH];
32
```

```
33 uint id_com = 0;
34 ubyte *tramas_faltan, *mens;
35 fd_set set;
36 struct timespec tv_sel, tv_blc;
37
38 void envio(char *ifName, char *dest_mac, ubyte *mensaje, uint
       length);
39 void crearSocket(char *ifName, char *dest_mac);
40 void envioTramaSincronia(uint flag, uint length);
41 void envioMensajeIoU(uint bytes);
42 void envioMensaje(uint num_tramas, uint resto);
43
44 void recepcion(ubyte *ack, uint num_tramas_t);
45 void envioMensajePorNum(uint num_trama, uint num_tramas, uint
       resto);
46
47 int timespec_subtract(struct timespec *result, struct timespec
       *t2, struct timespec *t1);
48
49 void *mi_malloc(size_t size, char *var);
50 void mi_free(void **ptr, char *var);
51
52 #endif
```

Apéndice C

envio.c

```
1  //
2  //   envio.c
3  //   TFG
4  //
5
6  #include "envio.h"
7
8  void envio(char *interface, char *dest_mac, ubyte *mensaje,
       uint length){
9    struct timespec tsStart, tsStop, tsDiff;
10   uint num_tramas, resto, i, *header, tam_bloque = 0,
         tramas_enviadas;
11   int res; ubyte ack = 0;
12
13   srand(time(NULL));
14
15   mens = mensaje;
16   tv_sel.tv_sec = TIMEOUT_S; tv_sel.tv_nsec = TIMEOUT_10MS;
17   tv_blc.tv_sec = TIMEOUT_S; tv_blc.tv_nsec = TIMEOUT_1MS;
18   id_com = rand();
19   resto = length % TAM_DATOS_PROTO;
20   num_tramas = (uint) (length / TAM_DATOS_PROTO) + (resto?1:0);
21
22   tramas_faltan = (ubyte *)mi_malloc(num_tramas * sizeof(ubyte)
         , "tramasFaltan");
23   for (i = 0; i < num_tramas; i++){
24     tramas_faltan[i] = 1;
25   }
26
27   crearSocket(interface, dest_mac);
28
29   if( clock_gettime(CLOCK_REALTIME, &tsStart) == -1 ) {
30     perror( "clock gettime" );
```

```
31      exit(EXIT_FAILURE);
32    }
33
34    envioTramaSincronia(FLAG_INICIAL, length);
35
36    if(num_tramas == 1){
37      envioMensajeIoU(resto);
38    }else{
39      envioMensajeIoU(TAM_DATOS_PROTO);
40      envioMensaje(num_tramas, resto);
41    }
42
43    envioTramaSincronia(FLAG_FINAL, 0);
44
45    while (1) {
46      FD_ZERO(&set);
47      FD_SET(sockfd, &set);
48
49      res = pselect(FD_SETSIZE, &set, NULL, NULL, &tv_sel, NULL);
50      if (res < 0 ) {
51        perror("pselect: ");
52      } else if (res == 0) {
53        envioTramaSincronia(FLAG_INICIAL, length);
54        envioTramaSincronia(FLAG_FINAL, 0);
55      } else {
56        break;
57      }
58    }
59
60    while(!ack){
61      recepcion(&ack, num_tramas);
62
63      if(!ack){
64        if (tam_bloque == 0){
65          for (i = 0; i < num_tramas; i++){
66            if (tramas_faltan[i] == 0){
67              tam_bloque = i;
68              break;
69            }
70          }
71        }
72
73        tramas_enviadas = 0;
74
75        if (tramas_faltan[0] == 0){
```

```
76          tramas_enviadas++;
77          if(num_tramas == 1)
78            envioMensajeIoU(resto);
79          else
80            envioMensajeIoU(TAM_DATOS_PROTO);
81          tramas_faltan[0] = 1;
82        }
83        for(i = 1; i < num_tramas; i++){
84          if(tramas_faltan[i] == 0){
85            tramas_enviadas++;
86            envioMensajePorNum(i + 1, num_tramas, resto);
87            tramas_faltan[i] = 1;
88            if (tramas_enviadas == tam_bloque){
89              nanosleep(&tv_blc, NULL);
90              tramas_enviadas = 0;
91            }
92          }
93        }
94        envioTramaSincronia(FLAG_FINAL, 0);
95      }
96    }
97
98    if( clock_gettime(CLOCK_REALTIME, &tsStop) == -1 ) {
99      perror( "clock gettime" );
100     exit(EXIT_FAILURE);
101   }
102
103   mi_free((void **)&tramas_faltan, "tramasFaltan");
104
105   close(sockfd);
106
107   timespec_subtract(&tsDiff, &tsStop, &tsStart);
108   printf("Tiempo de envio %d,%09ld segundos\n", tsDiff.tv_sec,
        tsDiff.tv_nsec);
109 }
110
111 void crearSocket(char *ifName, char *dest_mac){
112   struct ifreq if_idx;
113   struct ifreq if_mac;
114
115   if ((sockfd = socket(AF_PACKET, SOCK_RAW, htons(PROTOCOLO)))
        == -1) {
116     perror("socket");
117   }
118
```

```
119   /* Get the index of the interface to send on */
120   memset(&if_idx, 0, sizeof(struct ifreq));
121   strncpy(if_idx.ifr_name, ifName, IFNAMSIZ-1);
122   if (ioctl(sockfd, SIOCGIFINDEX, &if_idx) < 0)
123     perror("SIOCGIFINDEX");
124
125   /* Get the MAC address of the interface to send on */
126   memset(&if_mac, 0, sizeof(struct ifreq));
127   strncpy(if_mac.ifr_name, ifName, IFNAMSIZ-1);
128   if (ioctl(sockfd, SIOCGIFHWADDR, &if_mac) < 0)
129     perror("SIOCGIFHWADDR");
130
131   /* Bind to device */
132   if (setsockopt(sockfd, SOL_SOCKET, SO_BINDTODEVICE, ifName,
        IFNAMSIZ-1) == -1)  {
133     perror("SO_BINDTODEVICE");
134     close(sockfd);
135     exit(EXIT_FAILURE);
136   }
137
138   /* Ethernet header */
139   eh->ether_shost[0] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
        [0];
140   eh->ether_shost[1] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
        [1];
141   eh->ether_shost[2] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
        [2];
142   eh->ether_shost[3] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
        [3];
143   eh->ether_shost[4] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
        [4];
144   eh->ether_shost[5] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
        [5];
145   eh->ether_dhost[0] = (uint8_t)dest_mac[0];
146   eh->ether_dhost[1] = (uint8_t)dest_mac[1];
147   eh->ether_dhost[2] = (uint8_t)dest_mac[2];
148   eh->ether_dhost[3] = (uint8_t)dest_mac[3];
149   eh->ether_dhost[4] = (uint8_t)dest_mac[4];
150   eh->ether_dhost[5] = (uint8_t)dest_mac[5];
151   /* Ethertype field */
152   eh->ether_type = htons(PROTOCOLO);
153
154   /* Index of the network device */
155   socket_address.sll_ifindex = if_idx.ifr_ifindex;
156   /* Address length*/
```

```
157   socket_address.sll_halen = ETH_ALEN;
158   /* Destination MAC */
159   socket_address.sll_addr[0] = (uint8_t)dest_mac[0];
160   socket_address.sll_addr[1] = (uint8_t)dest_mac[1];
161   socket_address.sll_addr[2] = (uint8_t)dest_mac[2];
162   socket_address.sll_addr[3] = (uint8_t)dest_mac[3];
163   socket_address.sll_addr[4] = (uint8_t)dest_mac[4];
164   socket_address.sll_addr[5] = (uint8_t)dest_mac[5];
165 }
166
167 void envioTramaSincronia(uint flag, uint length){
168   uint tam_total = TAM_HEAD_ETH + 3 * sizeof(int);
169
170   cabecera[0] = id_com;
171   cabecera[1] = 0;
172   cabecera[2] = flag;
173   if (flag == FLAG_INICIAL){
174     tam_total += sizeof(int);
175     cabecera[3] = length;
176   }
177
178   if (sendto(sockfd, trama, tam_total, 0, (struct sockaddr*)&
        socket_address, addr_len) == -1) {
179     perror("sendto");
180   }
181 }
182
183 void envioMensajeIoU(uint bytes){
184   ushort tam_total = TAM_HEADS + bytes;
185
186   cabecera[0] = id_com;
187   cabecera[1] = 1;
188
189   memcpy(datos, mens, bytes);
190
191   if (sendto(sockfd, trama, tam_total, 0, (struct sockaddr*)&
        socket_address, addr_len) == -1) {
192     perror("sendto");
193   }
194 }
195
196 void envioMensaje(uint num_tramas, uint resto){
197   uint i = 2, tam_total = TAM_TRAMA_ETH;
198   ubyte *punt_aux = mens + TAM_DATOS_PROTO - TAM_HEADS;
199
```

```
200   while(i < num_tramas){
201     memcpy(buffer, punt_aux, TAM_HEADS);
202     cabecera[1] = i;
203     memcpy(punt_aux, trama, TAM_HEADS);
204
205     if (sendto(sockfd, punt_aux, tam_total, 0, (struct sockaddr
          *)&socket_address, addr_len) == -1) {
206       perror("sendto");
207     }
208
209     memcpy(punt_aux, buffer, TAM_HEADS);
210
211     punt_aux = punt_aux + TAM_DATOS_PROTO;
212     i++;
213   }
214
215   if(resto){
216     tam_total = TAM_HEADS + resto;
217   }
218
219   memcpy(buffer, punt_aux, TAM_HEADS);
220   cabecera[1] = i;
221   memcpy(punt_aux, trama, TAM_HEADS);
222
223   if (sendto(sockfd, punt_aux, tam_total, 0, (struct sockaddr*)
       &socket_address, addr_len) == -1) {
224     perror("sendto");
225   }
226
227   memcpy(punt_aux, buffer, TAM_HEADS);
228 }
229
230 void recepcion(ubyte *ack, uint num_tramas_t){
231   uint valor_tam = 0, num_tramas, resto, *header;
232   int res;
233   ubyte final = 0, *tramas_aux;
234
235   resto = (num_tramas_t * sizeof(ubyte)) % TAM_DATOS_PROTO;
236   num_tramas = (num_tramas_t * sizeof(ubyte) / TAM_DATOS_PROTO)
        + (resto?1:0);
237
238   tramas_aux = tramas_faltan;
239
240   while(final == 0){
241     FD_ZERO(&set);
```

```
242     FD_SET(sockfd, &set);
243
244     res = pselect(FD_SETSIZE, &set, NULL, NULL, &tv_sel, NULL);
245     if (res < 0) {
246       perror("pselect: ");
247     } else if (res == 0) {
248       envioTramaSincronia(FLAG_FINAL, 0);
249     } else {
250       if (recvfrom(sockfd, buffer, TAM_TRAMA_ETH, 0, NULL, NULL
              ) == -1) {
251         perror("recvfrom");
252       }
253
254       header = (uint *)(buffer + TAM_HEAD_ETH);
255
256       if((header[0] == id_com) && (header[1] != 0) ){
257         valor_tam = (header[1] == num_tramas)?resto:
              TAM_DATOS_PROTO;
258         memcpy(tramas_aux + (header[1] - 1) * TAM_DATOS_PROTO,
              header + 2, valor_tam);
259       } else if( (header[0] == id_com) && (header[1] == 0)  &&
           (header[2] == FLAG_FINAL) ){
260         final = 1;
261       } else if( (header[0] == id_com) && (header[1] == 0) && (
           header[2] == FLAG_ACK) ){
262         *ack = 1;
263         final = 1;
264       }
265     }
266   }
267 }
268
269 void envioMensajePorNum(uint num_trama, uint num_tramas, uint
      resto){
270   uint tam_total = TAM_HEADS + ((num_trama == num_tramas)?resto
        :TAM_DATOS_PROTO);
271   ubyte *punt_aux = mens + (TAM_DATOS_PROTO * (num_trama - 1))
        - TAM_HEADS;
272
273   memcpy(buffer, punt_aux, TAM_HEADS);
274   cabecera[1] = num_trama;
275   memcpy(punt_aux, trama, TAM_HEADS);
276
277   if (sendto(sockfd, punt_aux, tam_total, 0, (struct sockaddr*)
        &socket_address, addr_len) == -1) {
```

```
278       perror("sendto");
279     }
280
281     memcpy(punt_aux, buffer, TAM_HEADS);
282 }
283
284 /* Return 1 if the difference is negative, otherwise 0.  */
285 int timespec_subtract(struct timespec *result, struct timespec
      *t2, struct timespec *t1) {
286     int nano = 1e9;
287
288     long int diff = (t2->tv_nsec + nano * t2->tv_sec) - (t1->
        tv_nsec + nano * t1->tv_sec);
289     result->tv_sec = diff / nano;
290     result->tv_nsec = diff % nano;
291
292     return (diff<0);
293 }
294
295 void *mi_malloc(size_t size, char *var){
296     void *v = malloc(size);
297     if(!v) {
298       fprintf(stderr, "Error al reservar memoria\n");
299       exit(EXIT_FAILURE);
300     }
301     return v;
302 }
303
304 void mi_free(void **ptr, char *var){
305     free(*ptr);
306     *ptr=NULL;
307 }
```

Apéndice D

recepcion.h

```
1  //
2  //  recepcion.h
3  //  TFG
4  //
5
6  #ifndef TFG_recepcion_h
7  #define TFG_recepcion_h
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h>
12 #include <arpa/inet.h>
13 #include <linux/if_packet.h>
14 #include <sys/ioctl.h>
15 #include <net/if.h>
16 #include <netinet/ether.h>
17
18 #include "Parametros.h"
19
20 int sockfd;
21 struct sockaddr_ll socket_address;
22 int addr_len = sizeof(struct sockaddr_ll);
23
24 ubyte trama[TAM_TRAMA_ETH];
25 struct ether_header *eh = (struct ether_header *) trama;
26 uint *cabecera_proto = (uint *)(trama + TAM_HEAD_ETH);
27 ubyte *datos_proto = trama + TAM_HEADS;
28 ubyte buffer[TAM_TRAMA_ETH];
29
30 struct datos{
31   ubyte *direccion;
32 };
33
```

```
34 uint id_com, num_tramas_totales;
35 uint *cabecera; /* Buffer de recepcion */
36 ubyte *mens, *recibida;
37
38 struct datos *datos;
39
40 void *recepcion(char *InterfaceName, uint *length);
41 void crearSocket(char *ifName);
42 void actualizarSocket(char *dest_mac);
43 void recibirTramaInicial(uint *length);
44 void recibirTramaInicial2();
45 void recibirTramas(uint *num_tramas_faltan);
46 void pedirTramas();
47 void envioTramaSincronia(uint flag);
48 void envioMensajeIoU(uint num_tramas, uint bytes);
49 void envioMensaje(uint num_tramas, uint resto);
50
51 void *mi_malloc(size_t size, char *var);
52 void mi_free(void **ptr, char *var);
53
54 #endif
```

Apéndice E

recepcion.c

```
1  //
2  //  recepcion.c
3  //  TFG
4  //
5
6  #include "recepcion.h"
7
8  void *recepcion(char *InterfaceName, uint *length){
9    uint i = 0, tamanio, num_tramas_faltan;
10   ubyte *puntCopia;
11
12   crearSocket(InterfaceName);
13
14   recibirTramaInicial(&tamanio);
15   *length = tamanio;
16
17   num_tramas_totales = tamanio / TAM_DATOS_PROTO;
18   if( tamanio % TAM_DATOS_PROTO )
19     num_tramas_totales++;
20
21   num_tramas_faltan = num_tramas_totales;
22
23   datos = (struct datos *)mi_malloc(num_tramas_totales * sizeof
       (struct datos), "datos");
24   recibida = (ubyte *)mi_malloc(num_tramas_totales * sizeof(
       ubyte), "recibida");
25   for(i = 0; i < num_tramas_totales; i++){
26     recibida[i] = 0;
27   }
28
29   recibirTramas(&num_tramas_faltan);
30
31   while(num_tramas_faltan){
```

```
32      pedirTramas();
33      recibirTramas(&num_tramas_faltan);
34    }
35
36    envioTramaSincronia(FLAG_ACK2);
37
38    puntCopia = (ubyte *)mi_malloc(tamanio, "puntCopia");
39    for(i = 0; i < num_tramas_totales - 1; i++){
40      memcpy(puntCopia + (TAM_DATOS_PROTO*i), datos[i].direccion
            + TAM_HEADS, TAM_DATOS_PROTO);
41    }
42    if(tamanio%TAM_DATOS_PROTO){
43      memcpy(puntCopia + (TAM_DATOS_PROTO*(num_tramas_totales -
            1)), datos[i].direccion + TAM_HEADS, tamanio%
            TAM_DATOS_PROTO);
44    }
45
46    for(i = 0; i < num_tramas_totales; i++){
47      mi_free((void **)&datos[i].direccion, "datos[i].direccion")
            ;
48    }
49
50    envioTramaSincronia(FLAG_ACK2);
51
52    close(sockfd);
53
54    mi_free((void **)&datos, "datos");
55    mi_free((void **)&recibida, "recibida");
56
57    return (int *)puntCopia;
58  }
59
60  void crearSocket(char *ifName){
61    struct ifreq if_idx;
62    struct ifreq if_mac;
63
64    if ((sockfd = socket(AF_PACKET, SOCK_RAW, htons(PROTOCOLO)))
          == -1) {
65      perror("listener: socket");
66    }
67
68    /* Get the index of the interface to send on */
69    memset(&if_idx, 0, sizeof(struct ifreq));
70    strncpy(if_idx.ifr_name, ifName, IFNAMSIZ-1);
71    if (ioctl(sockfd, SIOCGIFINDEX, &if_idx) < 0)
```

```
72      perror("SIOCGIFINDEX");
73
74    /* Get the MAC address of the interface to send on */
75    memset(&if_mac, 0, sizeof(struct ifreq));
76    strncpy(if_mac.ifr_name, ifName, IFNAMSIZ-1);
77    if (ioctl(sockfd, SIOCGIFHWADDR, &if_mac) < 0)
78      perror("SIOCGIFHWADDR");
79
80    /* Bind to device */
81    if (setsockopt(sockfd, SOL_SOCKET, SO_BINDTODEVICE, ifName,
          IFNAMSIZ-1) == -1)  {
82      perror("SO_BINDTODEVICE");
83      close(sockfd);
84      exit(EXIT_FAILURE);
85    }
86
87    /* Ethernet header */
88    eh->ether_shost[0] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
          [0];
89    eh->ether_shost[1] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
          [1];
90    eh->ether_shost[2] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
          [2];
91    eh->ether_shost[3] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
          [3];
92    eh->ether_shost[4] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
          [4];
93    eh->ether_shost[5] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)
          [5];
94
95    /* Index of the network device */
96    socket_address.sll_ifindex = if_idx.ifr_ifindex;
97  }
98
99  void actualizarSocket(char *dest_mac){
100   eh->ether_dhost[0] = (uint8_t)dest_mac[0];
101   eh->ether_dhost[1] = (uint8_t)dest_mac[1];
102   eh->ether_dhost[2] = (uint8_t)dest_mac[2];
103   eh->ether_dhost[3] = (uint8_t)dest_mac[3];
104   eh->ether_dhost[4] = (uint8_t)dest_mac[4];
105   eh->ether_dhost[5] = (uint8_t)dest_mac[5];
106   /* Ethertype field */
107   eh->ether_type = htons(PROTOCOLO);
108
109   /* Address length*/
```

```
110   socket_address.sll_halen = ETH_ALEN;
111   /* Destination MAC */
112   socket_address.sll_addr[0] = (uint8_t)dest_mac[0];
113   socket_address.sll_addr[1] = (uint8_t)dest_mac[1];
114   socket_address.sll_addr[2] = (uint8_t)dest_mac[2];
115   socket_address.sll_addr[3] = (uint8_t)dest_mac[3];
116   socket_address.sll_addr[4] = (uint8_t)dest_mac[4];
117   socket_address.sll_addr[5] = (uint8_t)dest_mac[5];
118 }
119
120 void recibirTramaInicial(uint *length){
121   ubyte recep = 1;
122   char dest_mac[ETH_ALEN];
123
124   while(recep){
125     if (recvfrom(sockfd, buffer, TAM_TRAMA_ETH, 0, NULL, NULL)
          == -1) {
126       perror("recvfrom");
127     }
128
129     cabecera = (uint *) (buffer + TAM_HEAD_ETH);
130
131     if( (cabecera[1] == 0) && (cabecera[2] == FLAG_INICIAL) ){
132       id_com = *cabecera;
133       *length = cabecera[3];
134
135       memcpy(dest_mac, &(buffer[ETH_ALEN]), ETH_ALEN);
136       actualizarSocket(dest_mac);
137
138       recep = 0; // Salida del while
139     }
140   }
141 }
142
143 void recibirTramas(uint *num_tramas_faltan){
144   uint num_secuencia;
145   ubyte final = 0, do_malloc = 1;
146   int res;
147
148   while(!final){
149     if (do_malloc){
150       mens = (ubyte *)mi_malloc(TAM_TRAMA_ETH, "mens");
151       do_malloc = 0;
152     }
153
```

```
154     if (recvfrom(sockfd, mens, TAM_TRAMA_ETH, 0, NULL, NULL) ==
            -1) {
155       perror("recvfrom");
156     }
157
158     cabecera = (uint *)(mens + TAM_HEAD_ETH);
159
160     if (cabecera[0] == id_com){
161       if(cabecera[1]){
162         num_secuencia = cabecera[1] - 1;
163         if (recibida[num_secuencia]){
164           num_repes++;
165           continue;
166         }
167         recibida[num_secuencia] = 1;
168         datos[num_secuencia].direccion = (ubyte *)mens;
169         (*num_tramas_faltan)--;
170         do_malloc = 1;
171       } else if(cabecera[1] == 0 && cabecera[2] == FLAG_FINAL){
172         final = 1;
173         mi_free((void **)&mens, "mens");
174       }
175     }
176   }
177 }
178
179 void pedirTramas(){
180   uint num_tramas, resto;
181
182   resto = (num_tramas_totales * sizeof(ubyte)) %
        TAM_DATOS_PROTO;
183   num_tramas = ((num_tramas_totales * sizeof(ubyte)) /
        TAM_DATOS_PROTO) + (resto?1:0);
184
185   if(num_tramas == 1){
186     envioMensajeIoU(num_tramas, resto);
187   }else{
188     envioMensajeIoU(num_tramas, TAM_DATOS_PROTO);
189     envioMensaje(num_tramas, resto);
190   }
191
192   envioTramaSincronia(FLAG_FINAL);
193 }
194
195 void envioTramaSincronia(uint flag){
```

```
196   int tam_total = TAM_HEAD_ETH + 3 * sizeof(int);
197
198   cabecera_proto[0] = id_com;
199   cabecera_proto[1] = 0; /* Secuencia */
200   cabecera_proto[2] = flag; /* Flags */
201
202   if (sendto(sockfd, trama, tam_total, 0, (struct sockaddr*)&
          socket_address, addr_len) == -1) {
203     perror("sendto");
204   }
205 }
206
207 void envioMensajeIoU(uint numTramas, uint bytes){
208   ushort valor_tam = bytes, tam_total = TAM_HEADS + valor_tam;
209
210   cabecera_proto[0] = id_com;
211   cabecera_proto[1] = 1;
212
213   memcpy(datos_proto, recibida, valor_tam);
214
215   if (sendto(sockfd, trama, tam_total, 0, (struct sockaddr*)&
          socket_address, addr_len) == -1) {
216     perror("sendto");
217   }
218 }
219
220 void envioMensaje(uint num_tramas, uint resto){
221   uint i = 2, tam_total = TAM_TRAMA_ETH;
222   ubyte *punt_aux;
223
224   punt_aux = (ubyte *) recibida;
225   punt_aux = punt_aux + TAM_DATOS_PROTO - TAM_HEADS;
226
227   while(i < num_tramas){
228     memcpy(buffer, punt_aux, TAM_HEADS);
229     cabecera_proto[1] = i;
230     memcpy(punt_aux, trama, TAM_HEADS);
231
232     if (sendto(sockfd, punt_aux, tam_total, 0, (struct sockaddr
          *)&socket_address, addr_len) == -1) {
233       perror("sendto");
234     }
235
236     memcpy(punt_aux, buffer, TAM_HEADS);
237
```

```
238       punt_aux = punt_aux + TAM_DATOS_PROTO;
239       i++;
240     }
241
242     if(resto){
243       tam_total = TAM_HEADS + resto;
244     }
245
246     memcpy(buffer, punt_aux, TAM_HEADS);
247     cabecera_proto[1] = i;
248     memcpy(punt_aux, trama, TAM_HEADS);
249
250     if (sendto(sockfd, punt_aux, tam_total, 0, (struct sockaddr*)
          &socket_address, addr_len) == -1) {
251       perror("sendto");
252     }
253
254       memcpy(punt_aux, buffer, TAM_HEADS);
255 }
256
257 void *mi_malloc(size_t size, char *var){
258     void *v = malloc(size);
259     if(!v) {
260       fprintf(stderr, "Error al reservar memoria\n");
261       exit(EXIT_FAILURE);
262     }
263     return v;
264 }
265
266 void mi_free(void **ptr, char *var){
267     free(*ptr);
268     *ptr=NULL;
269 }
```