Prácticas de laboratorio de Telemática II

Práctica 4

Departamento de Ingeniería Telemática

(ENTEL)

Mónica Aguilar
Juanjo Alins
Oscar Esparza
Jose L. Muñóz
Marcos Postigo
Antoni X. Valverde

La composición de este manual de prácticas ha sido realizada con el programa LaTeX 2ε. Agradecer a DONALD KNUTH la idea y el programa TeX y a LESLIE LAMPORT sus magníficas macros que han derivado en LaTeX.

Barcelona a 5 de Febrero de 2003

• Revision 1.4 tcp.lyx

Edición electrónica de este documento. Este manual de prácticas se puede conseguir en http://labtel2.upc.es

Índice de las prácticas

Índice de figuras

PRÁCTICA 4

Transmission Control Protocol (TCP)

4.1. Introducción

Dos de los protocolos de transporte estandarizados en la torre TCP/IP son TCP (*Transmission Control Protocol*) y UDP (*User Datagram Protocol*). Previamente a describir TCP y UDP, veremos una clasificación de los protocolos en cuanto a cómo establecen la conexión y a su fiabilidad.

Un protocolo puede clasificarse en función de cómo realiza la conexión en:

Orientado a conexión. En este caso, para realizar un intercambio de información entre dos entidades se pueden distinguir las siguientes fases:

1. Establecimiento de la conexión
2. Intercambio de información
3. Cierre de la conexión

Una de las características de un servicio orientado a conexión es que una vez establecida la conexión, los mensajes llegan al receptor en el mismo orden en que fueron enviados por el emisor.

Un ejemplo de este procedimiento es la comunicación a través del teléfono:

1. Establecimiento de la conexión: La persona llamante, inicia el establecimiento de la conexión descolgando el teléfono y marcando el número del abonado llamado. El abonado llamado descuelga el teléfono y la conexión ya está realizada.
2. Intercambio de información: Ambos abonados, llamante y llamado, conversan a través del teléfono.
3. Cierre de la conexión: Los abonados cuelgan el teléfono dando por finalizada la llamada.

No orientado a conexión. En este caso, el intercambio de información no necesita de las fases de establecimiento y cierre de la conexión. La entidad transmisora envía el mensaje sin que la receptora sepa que va a recibir ese mensaje.

Un ejemplo de un servicio no orientado a conexión es el sistema postal. En este caso, el remitente envía una carta al destinatario. El destinatario sabe que el remitente le envió una carta en el momento en que la recibe.

Otro parámetro que se utiliza para clasificar un protocolo es la **fiabilidad** del protocolo en cuanto a la entrega de la información.

Se dice que un protocolo es **fiable** cuando implementa los mecanismos necesarios para que la información que transporta llegue al otro extremo libre de errores. Se dice que un protocolo es **no fiable** si no nos asegura la entrega de los mensajes al extremo receptor.

Como ejemplo, Internet Protocol (IP) es un protocolo no orientado a conexión y no fiable. Es no orientado a conexión porque un datagrama IP se entrega a la red sin un establecimiento previo de la conexión y es no fiable porque no

incorpora ningún mecanismo que nos informe de si el datagrama ha sido entregado al extremo receptor con éxito o si no ha podido ser entregado.

Dado que IP es un protocolo no orientado a conexión, no fiable, los protocolos que estén por encima de IP (Nivel de Transporte) se van a tener que enfrentar con las siguientes situaciones de error:

1. Pérdida de una trama
2. Llegada de una trama fuera de orden
3. Duplicación de una trama
4. Modificación de los datos de la trama

TCP (*Transmission Control Protocol*) es un protocolo orientado a conexión y fiable. Esto nos indica que TCP va a tener que resolver las cuatro posibles situaciones de error que hemos visto para proporcionar un servicio orientado a conexión y fiable, a sus propios usuarios (por usuarios se entiende aquellos protocolos o aplicaciones que utilicen TCP como protocolo de transporte).

UDP (*User Datagram Protocol*) es un protocolo no orientado a conexión y no fiable. Si una aplicación desea un servicio fiable, o bien utiliza TCP o si utiliza UDP deberá ser la aplicación quien implemente los mecanismos de fiabilidad.

Obsérvese que el uso de TCP o UDP dependerá de varios aspectos. Entre ellos, y además de la fiabilidad, la **eficiencia**:

> Supongamos que vamos a realizar una conferencia telefónica durante la que estaremos hablando 10 minutos. El tiempo invertido en marcar el teléfono del abonado llamado y que éste descuelgue es de 30 segundos y el tiempo de "liberar la llamada" (despedirse y colgar el teléfono) de 5 segundos. El tiempo total invertido para completar toda la llamada será de 10*60+30+5=635 segundos y el tiempo invertido en el intercambio de información son 600 segundos. La **eficiencia** obtenida es de 600/635, 94.5 %. Si el tiempo invertido en el intercambio de información hubiese sido de 5 segundos (decir solo una palabra) la **eficiencia** hubiera bajado al 12.5 %.

Un protocolo orientado a conexión (como TCP) utiliza un cierto tiempo en establecer y liberar la conexión.

La fiabilidad de un protocolo puede ser un factor determinante, pero no siempre. Cuando se transmite voz, es más importante que las muestras de voz estén disponibles en los instantes requeridos en el receptor, que el hecho de que haya alguna muestra errónea, perdida o duplicada. Por esta razón, la mayoría de sistemas de transmisión multimedia sobre IP utilizan el protocolo UDP.

> Un ejemplo de protocolo no orientado a conexión y no fiable es el sistema postal de correos. En este sistema el remitente escribe una carta que posteriormente “encapsula” (introduce) en un sobre con la dirección del destinatario. Dicho sobre se entrega al servicio postal de correos, cuya misión es hacer llegar el sobre a la dirección del destinatario. El destinatario, a priori, no sabe que el remitente ha entregado el sobre al servicio postal de correos. Si durante el tranporte del sobre, éste se pierde, ni el destinatario, ni el remitente serán informados de este hecho (no fiable).
>
> Si el remitente desea enviar otra carta al mismo destinatario, debería introducirla de nuevo en un sobre con la dirección pertinente y entregarlo al servicio postal de correos. Si el transporte de este segundo sobre se realiza de forma más rápida que el primero (por ejemplo por avión), es posible que los sobres (y las cartas) lleguen “desordenados” (problema inherente a un servicio no orientado a conexión).

Una de las características de un servicio no orientado a conexión es que las unidades de datos (los sobres) se manejan de forma independiente aunque la información transportada (las cartas) formen parte de una misma comunicación. Por esta razón, cada sobre debe llevar la dirección del destinatario. Observese que en el ejemplo de la llamada telefónica (servicio orientado a conexión), la dirección (número de teléfono del abonado receptor) sólo se proporciona en el establecimiento de la conexión.

No es difícil notar, que a pesar de tener un servicio no orientado a conexión y no fiable, nosotros (el escritor de cartas) puede utilizar mecanismos para obtener, finalmente, un servicio fiable. Estos mecanismos serían:

1. Para solucionar el problema de la fiabilidad, el remitente de la carta incluye una postdata pidiendo al destinatario una confirmación antes de 14 días. El remitente está suponiendo que en el peor de los casos, el servicio postal de

correos no tardará más de 7 días en entregar la carta al remintente y 7 días más en recibir la confirmación. Si en 14 días no ha recibido confirmación puede suponer que la carta que envío o la confirmación se ha perdido. En este caso vuelve a enviar la carta.

2. Para solucionar el problema de "ordenamiento", las cartas se numeran de forma consecutiva. De esta forma el remitente sabe leerlas de forma "ordenada" o bien detectar qué carta se ha perdido.

Finalmente, el siguiente ejemplo, muestra un servicio no orientado a conexión y fiable. Este caso es el de las empresas de transporte de paquetes (estilo "packet express"). Se deja como ejercicio al lector determinar porque es un servicio no orientado a conexión y fiable.

4.2. Puertos y Sockets

Para explicar qué son los puertos y los *sockets* lo realizaremos a través de un ejemplo. Supongamos que tenemos en un misma máquina (Maq-A) varios procesos comunicándose simultáneamente con otros tantos procesos en diversas maquinas (Maq-B, Maq-C ...). Cada vez que un datagrama IP llega a Maq-A, el nivel IP extrae la información y la pasa al nivel de transporte (UDP ó TCP). El nivel de transporte, realizará las funciones específicas según sea UDP o TCP y posteriormente debe entregar la información al proceso al cual va dirigida. El nivel de transporte en la torre TCP/IP, utiliza el concepto de puerto para demultiplexar la información recibida y entregarla a la aplicación correspondiente. El puerto es un número de 16 bits sin signo (0-65535) que se asigna a un proceso que quiere comunicarse a través de TCP/IP y ese número de puerto va encapsulado en los paquetes TCP y UDP. La Figura 4.1 muestra un ejemplo de lo que se acaba de describir.

Un socket[1] identifica de forma univoca una conexión entre dos procesos en TCP/IP. Refiriéndonos a la Figura 4.1, la conexión entre los procesos *a* y *a'* está identificada por el socket (IP_1, P_1, IP_2, P_4), es decir, dirección IP y puerto de origen y dirección IP y puerto de destino. Obsérvese que el número de puerto es local a la máquina; así dos procesos en máquinas diferentes pueden tener el mismo número de puerto.

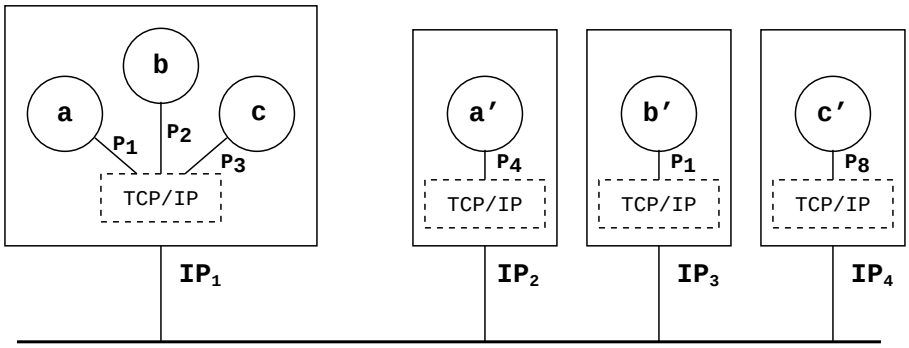

Figura 4.1: Puertos y sockets

4.3. Arquitectura cliente-servidor

En el mundo de TCP/IP las comunicaciones entre procesos siguen una arquitectura denominada cliente-servidor. La idea es que un proceso actúa como un servidor (proporciona servicios a los clientes) y espera que algún cliente se conecte a él en demanda de un servicio. Otro proceso actúa como cliente y se conecta al servidor para obtener un servicio.

[1] La palabra socket se utiliza con acepciones diferentes según el entorno. Cuando se habla del protocolo TCP/IP, la definición es la que se ha dado. Si se está hablando del API de programación de TCP/IP su significado es diferente.

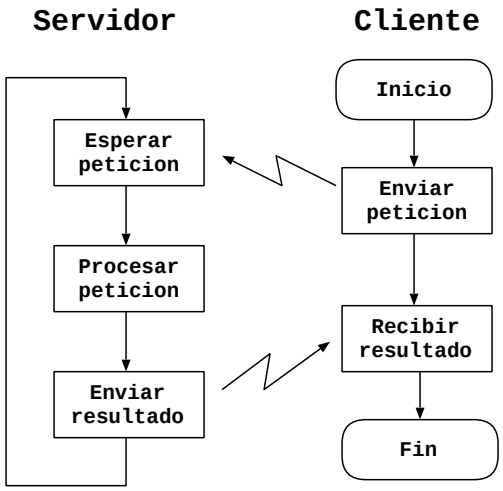

Figura 4.2: Arquitectura Cliente Servidor

Un ejemplo del funcionamiento es la conexión desde un navegador a un servidor de WEB. El servidor de WEB debe estar "esperando" que algún navegador se conecte a él con el fin de proporcionar una pagina HTML. Desde el punto de vista del usuario, primero arranca el navegador, luego se realiza la conexión y cuando el usuario se "ha cansado" de navegar, cierra el navegador, mientras que el servidor de WEB continúa "esperando" otras conexiones. La Figura 4.2 muestra el flujograma que describe este comportamiento.

De esta arquitectura se deduce que un cliente debe conocer la IP y el puerto del servidor. Siguiendo con el ejemplo del navegador, si el usuario introduce la URL *http://www.upc.es*, el navegador se conectará al puerto 80. Esto es debido a que hay una serie de puertos conocidos con el nombre de "*well known ports*" en los que se aconseja qué tipo de servicio deben proporcionar. Así el puerto 21 es para el servidor de ftp, 23 es para el servidor de telnet, el 25 para el servidor de SMTP (servicio de correo electrónico), el 80 para el servidor de http, etc. Esto no significa que todos los servidores WEB "escuchen" en el puerto 80, ya que es potestad del administrador del sistema decidir qué puerto utilizará su servidor WEB. Sin embargo, la mayoría de clientes tienen establecido un puerto por defecto para el servidor (como en el caso del navegador y el puerto 80), de forma que si se cambia el puerto de un servicio bien definido, solo se logrará confundir al cliente (tanto el cliente software como al usuario que está utilizando ese cliente). A modo de ejemplo, si se desea utilizar un navegador para conectarnos a un puerto diferente del de defecto (por ejemplo el 8080), la forma de introducir la URL es *http://nombre_servidor:8080*. Si el servidor de WEB de la UPC estuviese en el puerto 8080, la URL sería *http://www.upc.es:8080*.

4.4. *User Datagram Protocol*

Tal como se ha comentado UDP (*User Datagram Protocol*) es un protocolo no orientado a conexión y no fiable. La especificación oficial de UDP se encuentra en el RFC768.

El formato y los campos de un datagrama UDP se muestran en la Figura 4.3.

El campo de longitud se refiere a la longitud total, en bytes, de la cabecera y los datos. La aplicación que utiliza UDP, debe controlar el tamaño del paquete UDP (la cantidad de bytes enviados en un mismo paquete UDP) si quiere evitar que el nivel IP fragmente el paquete UDP porque se ha excedido el MTU (*Maximum Transmission Unit*) del medio físico. Por ejemplo, si el medio físico es una *Ethernet* con tramas *Ethernet II*, el campo de datos de la trama *Ethernet* tiene una longitud máxima de 1500 bytes. En este caso, la longitud máxima del datagrama IP encapsulado en una trama *Ethernet* sera de 1500 bytes. Si utilizamos UDP como protocolo de transporte, la cantidad de bytes de usuario (campo *data* del datagrama UDP) será como máximo $1500 - 20 - 8 = 1472\,bytes$ (a 1500 se le resta la cabecera IP, usualmente 20 bytes, y la cabecera UDP, 8 bytes)

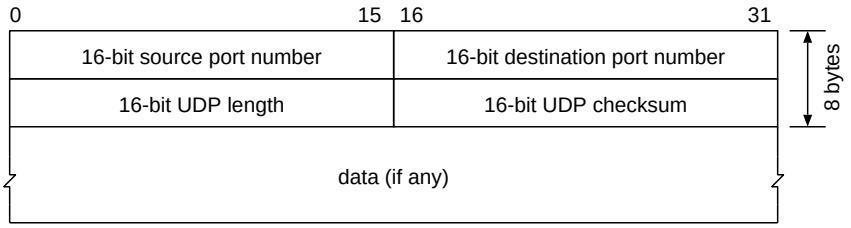

Figura 4.3: Datagrama UDP

El cálculo del *checksum* se especifica en la sección 4.6. El *checksum* incluye tanto la cabecera UDP como el campo de datos, (recordemos que el checksum de la cabecera de IP solo incluye la cabecera IP y no los datos del datagrama IP). En UDP el el cálculo del *checksum* es opcional. Si el emisor no realiza el cálculo del *checksum*, envía este campo con ceros. Si el emisor realiza el cálculo del *checksum* y obtiene un valor 0, envia todo unos (65535) que es equivalente en aritmética de complemento a uno. Si el emisor calcula el *checksum* y el receptor detecta un *checksum* erróneo, el datagrama UDP se descarta silenciosamente (esta condición de error no se indica ni al emisor ni a los protocolos de nivel superior del receptor).

4.5. *Transmission Control Protocol*

TCP es un protocolo orientado a conexión, fiable. Específicamente, se dice que TCP proporciona un servicio de transporte de un flujo de bytes, orientado a conexión, fiable.

El concepto de transporte de un *flujo de bytes*, indica que la aplicación entrega a TCP los bytes que desea transmitir y que TCP no interpretará esos bytes, únicamente los hará llegar a la aplicación receptora en el orden en que se los entregaron. En otras palabras, si una aplicación entrega 20 bytes a TCP, seguido de una entrega de 30 bytes, seguido de una entrega de 60 bytes, la aplicación receptora no sabe cuántas entregas se hicieron al TCP, sino que puede ocurrir que al leer los bytes recibidos, lo haga en cinco lecturas de 22 bytes.

TCP proporciona fiabilidad utilizando los siguientes mecanismos:

- Los datos de la aplicación son fragmentados en lo que TCP considera que es el mejor tamaño para ser transmitidos.
- Cuando TCP envía un segmento, inicia un temporizador, esperando que el otro extremo reconozca la recepción correcta del segmento enviado. Si el reconocimiento no ha sido recibido cuando el temporizador ya ha expirado, el extremo transmisor vuelve a enviar el mismo segmento.
- Cuando TCP recibe un segmento de datos, envía un reconocimiento al extremo que lo ha enviado[2].
- TCP incluye un *checksum* de la cabecera y los datos cuyo propósito es detectar cualquier modificación de los datos en tránsito. Si el *checksum* recibido es incorrecto, el segmento TCP se descarta silenciosamente y no se reconoce. Se espera que el temporizador del TCP del extremo opuesto expire y se lo vuelva a enviar.
- Dado que los segmentos TCP viajan encapsulados en datagramas IP, y dado que los datagramas IP pueden llegar desordenados, los segmentos TCP pueden llegar desordenados. El TCP del extremo receptor reordena, si es necesario, los segmentos recibidos y entrega, a la aplicación, los datos en el orden correcto.
- Dado que los datagramas IP pueden llegar duplicados, TCP debe descartar los segmentos TCP duplicados.
- TCP también proporciona control de flujo. Los extremos TCP tienen una cantidad finita de espacio de almacenamiento. El extremo TCP receptor indica al extremo TCP emisor cuánto espacio de almacenamiento tiene para los datos recibidos. De esta forma se previene que un ordenador cuyo ritmo de generación de datos es alto pueda superar la capacidad de recibir y procesar datos del ordenador receptor.

[2] Este reconocimiento se denomina ACK (*acknowledge*).

El sistema que utiliza TCP para proporcionar fiabilidad se engloba en los llamados protocolos de ventana deslizante.

4.5.1. Protocolos de Ventana Deslizante

La pregunta que nos hacemos es, ¿cómo puede un protocolo proporcionar un servicio de transferencia fiable sobre un sistema de comunicaciones no fiable?. Una de las técnicas más utilizadas es el ARQ (*Automatic Repeat Request*) o petición de retransmisión automática. En ARQ, el receptor informa al emisor del éxito o fracaso de la última transmisión, a través de una trama de reconocimiento. Los reconocimientos pueden ser positivos (reconocimiento de éxito) y negativos (reconocimiento de errores en la trama recibida). No todos los protocolos utilizan simultáneamente ambos tipos de reconocimiento. Por ejemplo TCP sólo utiliza reconocimientos positivos y cuando hay errores no envía nada, utilizando temporizadores para las retransmisiones.

En la Figura 4.4 se muestra la transmisión de dos tramas con reconocimiento. Este sistema se conoce con el nombre de *Stop & Wait* porque después de enviar un paquete, el emisor debe parar de transmitir y esperar la recepción del reconocimiento. Obsérvese que el tiempo efectivo utilizado para transmitir una trama es desde el inicio de la transmisión de la trama hasta que se recibe el reconocimiento. En este caso la eficiencia de transmisión es baja, puesto que el hecho de que el emisor deba esperar el reconocimiento le imposibilita para seguir transmitiendo.

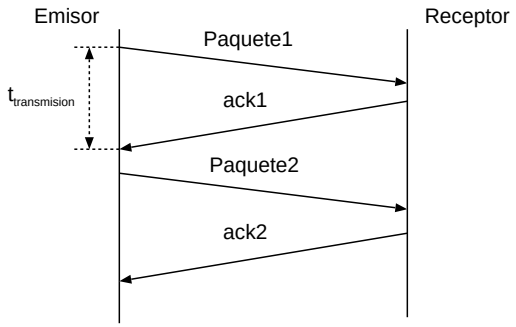

Figura 4.4: Stop & Wait

Supongamos que modificamos el algoritmo de manera que el emisor pueda tener como máximo F tramas no reconocidas. El emisor podrá enviar como máximo F tramas sin recibir reconocimiento de ellas. F se denomina el tamaño de la ventana y en cuanto reciba el reconocimiento de la primera trama (*ACK(1)*), podrá enviar una nueva trama, y así sucesivamente. Este es el caso representado en la Figura 4.5, para un tamaño de ventana $F = 6$.

El hecho de utilizar ventana deslizante implica que las tramas de información deben ir numeradas para que los reconocimientos indiquen mediante este número qué trama ha sido recibida correcta o incorrectamente.

Es importante destacar que en un sistema de ventana deslizante, no es necesario enviar una trama de reconocimiento por cada trama de datos. Se puede reconocer con una sola trama de reconocimiento varias tramas de datos teniendo en cuenta que si se han enviado 3 tramas de datos y se reconoce la trama 3, entonces, de forma automática, quedan reconocidas la 1 y la 2.

Es habitual que en una comunicación entre dos ordenadores, cada uno de ellos actúe simultáneamente de emisor y receptor, ya que la información suele fluir tanto en un sentido como en el otro. Para aumentar la eficiencia, el paquete de información incluye un campo utilizado para reconocimiento, evitándose el tener que enviar tramas de reconocimiento y de información diferentes cuando ambas tramas viajan en el mismo sentido. A esta técnica se le conoce con el nombre de *piggybacking*.

4.5.2. TCP y la ventana deslizante

TCP utiliza un mecanismo especializado de ventana deslizante con objeto de mejorar la eficiencia de transmisión y realizar un control de flujo.

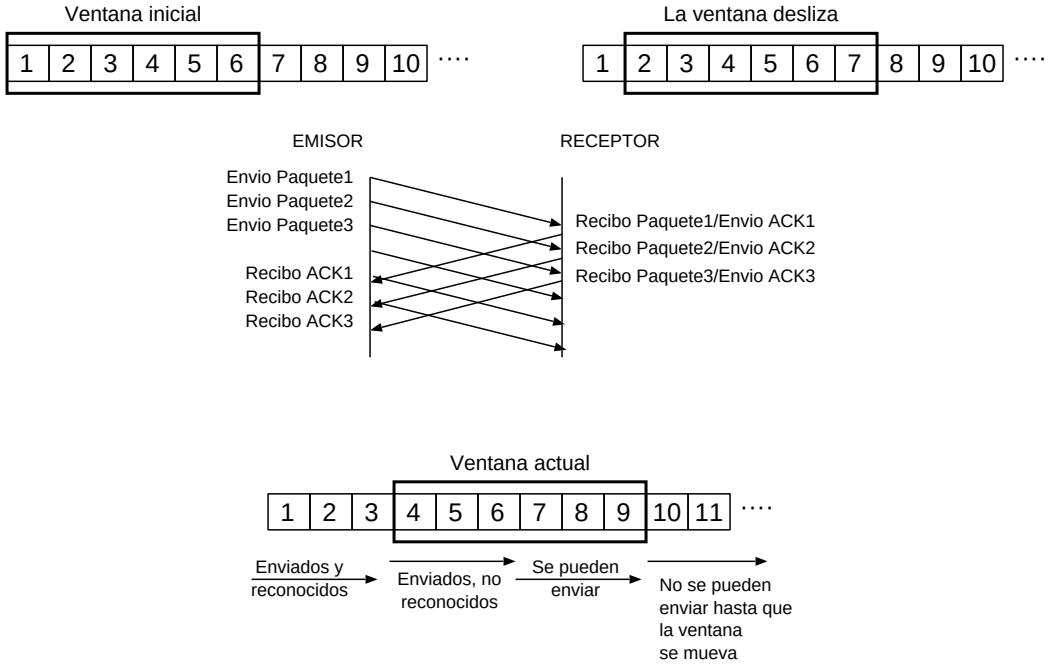

Figura 4.5: Ventana Deslizante

Como se ha comentado anteriormente, TCP ve los datos de usuario como un flujo de bytes que divide en segmentos para ser transmitidos. Por esta razón TCP no numera los segmentos sino los octetos transmitidos y a su vez, los reconocimientos reconocen octetos recibidos y no segmentos recibidos.

En TCP el tamaño de ventana indica el número de octetos que pueden ser recibidos y el receptor indica al emisor, en cada trama enviada, qué tamaño de ventana tiene disponible para recibir el siguiente segmento, realizando, de esta manera, el control de flujo. Si se indica un tamaño de ventana 0, el emisor deja de enviar segmentos hasta que recibe otro paquete indicando un tamaño de ventana distinto de 0.

4.5.3. El segmento TCP

La Figura 4.6 muestra cómo es el segmento TCP.

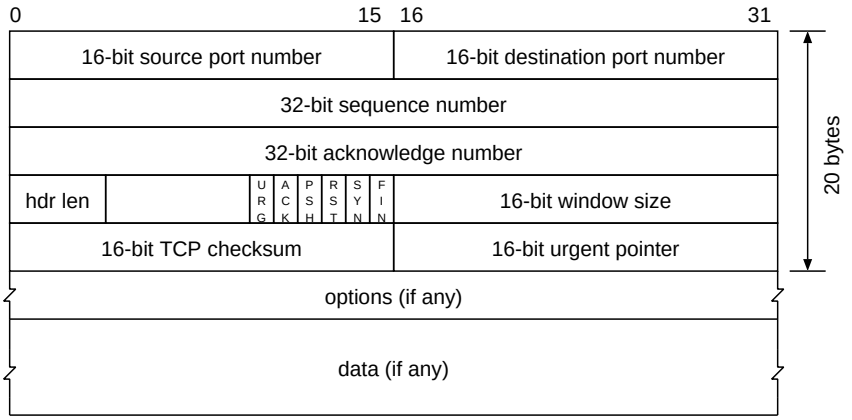

Figura 4.6: Segmento TCP

Los primeros 32 bits indican los puertos origen y destino. Los siguientes 32 bits son el número de secuencia. El número de secuencia identifica el byte del flujo de datos que está siendo enviado y que ocupa la primera posición en el campo de datos del segmento. El primer byte de un flujo en un segmento TCP no lleva el número 1 como número de secuencia, sino que se elige un número aleatorio (*Inital Sequence Number*) en cada conexión realizada y este número representa el número de secuencia del primer byte a transmitir durante esa conexión. Si el ISN de una conexión es el 1415531521, el primer byte estaría identificado por este mismo número y el segundo sería $1415531521+1$, y así sucesivamente. De esta forma si un segmento TCP llega a una conexión equivocada (cosa bastante inverosímil), los números de secuencia de ambas conexiones no podrían a llegar a confundirse nunca, ya que ambas conexiones utilizaron un ISN muy diferente y los datos que llegaron a la conexión equivocada nunca serán pasados a la aplicación equivocada (el TCP generaría un error al recibir un número de secuencia no esperado).

El campo del número de reconocimiento (*acknowledge number*) se utiliza para almacenar el número del siguiente byte que se espera recibir. Este campo sólo se decodifica si el flag ACK está activado. Si se recibió un paquete de datos con el número de secuencia 1415531521 y con 100 bytes de datos (desde el 1415531521 hasta el 1415531620), el segmento TCP de respuesta llevaría activado el flag ACK y el número de reconocimiento sería el 1415531621, puesto que espera recibir el siguiente al 1415531620.

El campo *hdr len* (Header Length) indica la longitud de la cabecera que puede llevar opciones haciendo que la longitud de la cabecera varíe.

A continuación vienen 6 bits reservados (no se utilizan) y 6 bits que actúan de *flags*:

- urg: *Urgent Pointer* válido
- ack: Número de reconocimiento válido
- psh: Segmento con datos de usuario. Pasarlos a la aplicación tan pronto como sea posible.
- rst: Reinicio de la conexión
- syn: Inicio de conexión
- fin: Final de conexión

El campo *window size* es de 16 bits e indica el tamaño de ventana en bytes. Los 16 bits siguientes se utilizan para el *checksum* y los siguientes 16 bits para indicar un *urgent pointer* (el *urgent pointer* se explicará en la sección 4.5.5).

Existen varias opciones en TCP, y la más utilizada es el MSS (*Maximum Segment Size*). Cada extremo de una conexión indica al otro cuál es el tamaño máximo de segmento que desea recibir. Este valor se especifica dentro del campo de opciones al inicio de la conexión.

4.5.4. Establecimiento y cierre de una conexión TCP

4.5.4.1. Establecimiento de la conexión

El proceso que abre una conexión, lo hace enviando un segmento TCP con el *flag* SYN activado y un número de secuencia (ISN) que identificará los bytes transmitidos. No se transmiten datos de usuario. El *flag* ACK no está activado porque todavía no hay bytes que reconocer.

El proceso que recibe este segmento, contesta con un segmento TCP con el *flag* SYN activado, un número de secuencia aleatorio, el flag ACK activado y el número de reconocimiento igual al ISN recibido más uno (se supone que un segmento SYN consume un byte).

El proceso que inició la conexión responde al segmento recibido con un segmento con el *flag* ACK activado y el número de secuencia de reconocimiento igual al número de secuencia recibido más uno.

En este momento ya está establecida la conexión. Esta apertura de conexión se conoce con el nombre de “establecimiento de conexión a tres bandas”, porque se utilizan tres segmentos. La Figura 4.7 muestra el establecimiento de una conexión TCP.

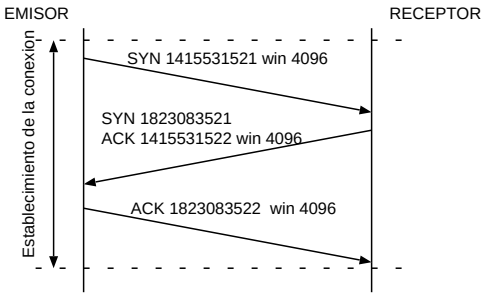

Figura 4.7: Establecimiento de la conexión

4.5.4.2. Timeout en el establecimiento de la conexión

Puede ocurrir que el extremo remoto no responda al segmento SYN de apertura de conexión. En ese caso se reintenta la apertura de la conexión varias veces. Por ejemplo, en un S.O. Linux con un kernel 2.4.10, el resultado obtenido al intentar establecer una conexión fue:

```
No Time        Source         Destination      Prot Info
1  0.000000    192.168.69.5   192.168.69.19    TCP  32773 > 23 [SYN]
                                                     Seq=4209576593 Ack=0 Win=5840
2  2.990898    192.168.69.5   192.168.69.19    TCP  32773 > 23 [SYN]
                                                     Seq=4209576593 Ack=0 Win=5840
3  8.990899    192.168.69.5   192.168.69.19    TCP  32773 > 23 [SYN]
                                                     Seq=4209576593 Ack=0 Win=5840
4  20.990900   192.168.69.5   192.168.69.19    TCP  32773 > 23 [SYN]
                                                     Seq=4209576593 Ack=0 Win=5840
5  44.990899   192.168.69.5   192.168.69.19    TCP  32773 > 23 [SYN]
                                                     Seq=4209576593 Ack=0 Win=5840
6  92.990900   192.168.69.5   192.168.69.19    TCP  32773 > 23 [SYN]
                                                     Seq=4209576593 Ack=0 Win=5840
```

El tiempo de espera entre intentos de conexión es de la forma $3 \cdot 2^n$, $n = 0, \ldots, 4$. El número de reintentos antes de decidir que la conexión no se puede realizar se controla a través de la variable */proc/sys/net/ipv4/tcp_syn_retries* cuyo valor durante la prueba realizada era de 5.

4.5.4.3. Maximum Segment Size (MSS)

Una de las opciones que se incluyen en los segmentos SYN es el *Maximum Segment Size*. Cada uno de los extremos informa al otro del tamaño máximo de segmento que desea recibir. Como regla general, cuanto mayor sea el MSS, sin que se produzca fragmentación a nivel IP, mayor será la eficiencia ya que se amortiza las cabeceras de IP y TCP. Si el nivel físico es *Ethernet* con tramas *Ethernet II*, el mayor datagrama IP será de 1500 bytes. Si restamos la longitud de las cabeceras IP y TCP tendremos el MSS: $1500 - 20 - 20 = 1460$ bytes. Si se utiliza un encapsulado IEEE 802.3 entonces el MSS tendría un valor de 1452 bytes.

4.5.4.4. Cierre de una conexión TCP.

El cierre de una conexión se realiza mediante segmentos TCP con el *flag* FIN activado. El cierre de una conexión TCP utiliza 4 segmentos. Esto es debido a que una conexión TCP es *full-duplex*, y cada una de las direcciones debe ser cerrada independientemente de la otra. La Figura 4.8 muestra el cierre de una conexión TCP.

La idea en un cierre de conexión TCP es que cualquiera de los extremos que haya enviado todos los datos puede realizar un cierre de su conexión (*half-close*), mientras que puede seguir recibiendo datos del extremo remoto.

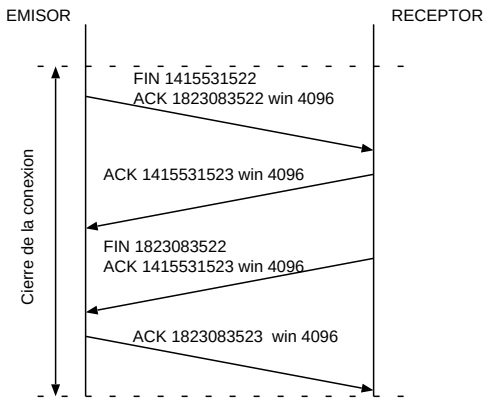

Figura 4.8: Cierre de una conexión TCP

4.5.4.5. Diagrama de transición de estados de TCP

La Figura 4.9 muestra el diagrama de transición de estados de TCP. Algunos estados son específicos del servidor mientras que otros lo son del cliente. Por ejemplo, como se ha comentado en la sección 4.3, un servidor puede estar "esperando" que los clientes se conecten a él. La espera corresponde a un estado de "LISTEN". En realidad se dice que el servidor está "escuchando", en un puerto, nuevas conexiones. Cuando se ejecuta un servidor pasa de estado "CLOSED" a estado "LISTEN" esperando que le lleguen segmentos TCP con el flag SYN activado (establecimiento de conexión). Sin embargo, cuando se ejecuta un cliente, éste iniciará directamente la conexión con el servidor enviándole un segmento SYN.

La Figura 4.10 es un ejemplo de los estados y los segmentos del cliente y servidor en la fase de establecimiento y cierre de la conexión para un modo de operación normal.

4.5.4.6. 2MSL Timeout

En la Figura 4.9, debajo del estado TIME_WAIT, puede leerse "2MSL timeout". Esto quiere decir que cuando el TCP que realiza un cierre activo (active close) llega al estado TIME_WAIT debe esperar un cierto tiempo antes de dar por finalizada la conexión. Ese tiempo es $2 \cdot MSL$ donde MSL[3] (Maximum Segment Lifetime) es el tiempo máximo de vida de un segmento. Dado que un segmento TCP viaja encapsulado en un datagrama IP, y éste tiene un campo TTL (time-to-live), un segmento TCP también tendrá un tiempo de vida en la red.

¿Por qué el TCP que realiza un cierre activo debe esperar un cierto tiempo?. El cierre de una conexión nunca es una cosa sencilla, ya que los paquetes de cierre pueden perderse y los extremos TCP pueden quedarse en un estado incorrecto. Refiriéndonos a la Figura 4.10, una vez que el cliente ha recibido el segmento FIN, ha pasado al estado TIME_WAIT y envía un segmento de reconocimiento. Puede suceder que ese segmento de reconocimiento se "pierda" y no llegue nunca a su destino. Si así fuera el caso, el servidor volvería a retransmitir el segmento FIN. Por tanto, el cliente espera un tiempo prudencial ($2 \cdot MSL$) de manera que pueda recibir esa retransmisión del segmento FIN si el reconocimiento se ha perdido.

Un efecto colateral de este estado de espera, es que durante ese tiempo el socket formado por la dirección IP del cliente, puerto del cliente, dirección IP del servidor, puerto del servidor, no puede ser usado para otra conexión. En realidad, la propia implementación del TCP implica que un puerto local que esté en un socket en estado TIME_WAIT no puede ser usado de nuevo hasta que expire ese estado.

[3]El RFC 793 especifica el MSL con una duración de 2 minutos. Los valores de MSL en diferentes implementaciones es de 30 segundos, 1 minuto o 2 minutos.

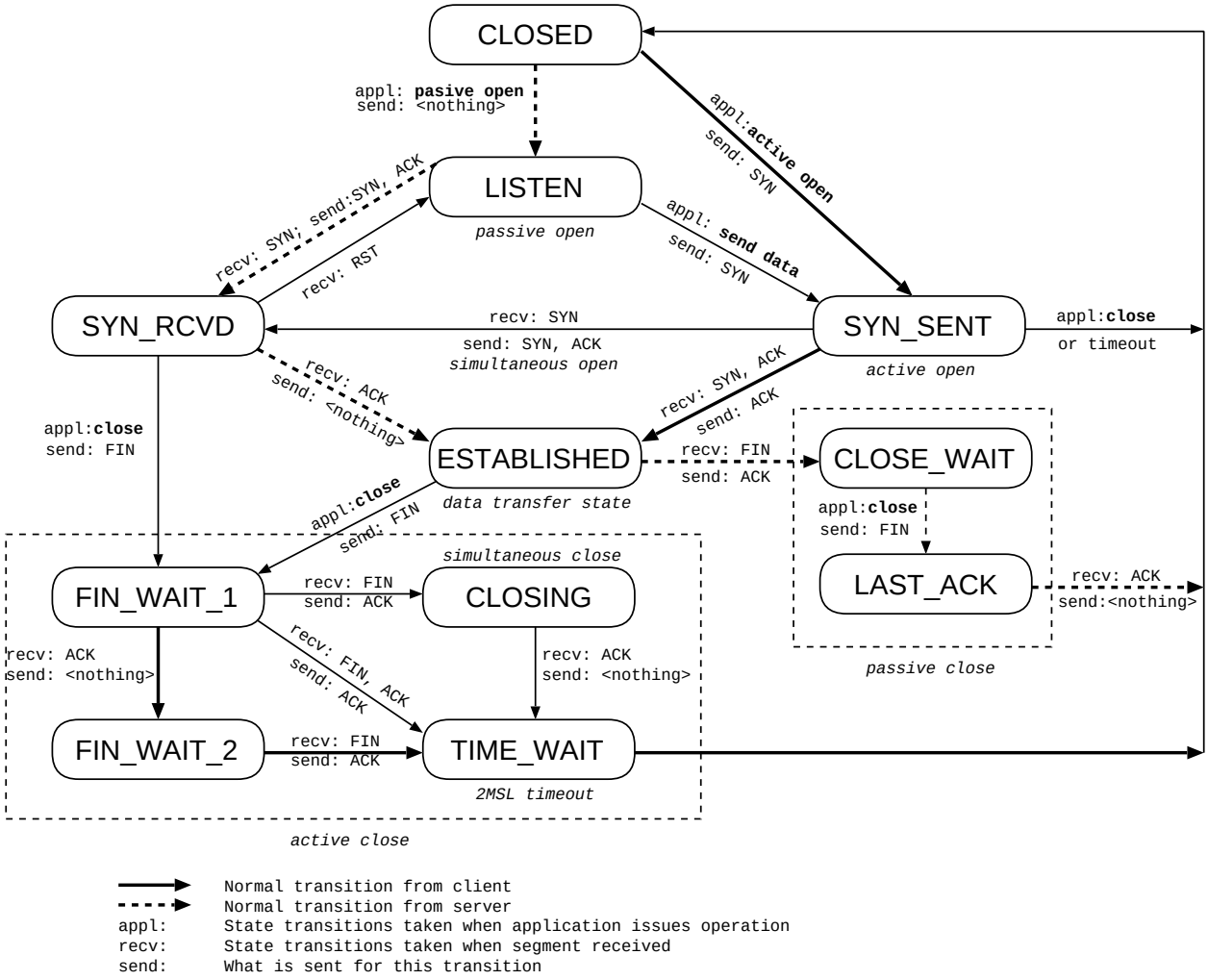

Figura 4.9: Diagrama de transición de estados de TCP

4.5.4.7. Segmentos RESET

En general, TCP envía un segmento RESET siempre que recibe un segmento que no parece ser correcto para la conexión que tenía establecida.

Un caso bien conocido en el que se envía un segmento RESET es cuando se intenta realizar una conexión a un puerto no activo de un servidor (un puerto no activo significa que no hay ninguna aplicación que esté “escuchando” en ese puerto de ese servidor). En este caso, el servidor responde al segmento SYN con un segmento RST.

4.5.5. Transferencia de información

4.5.5.1. El flag PSH

Los segmentos TCP que transportan datos de usuario suelen llevar activado el flag de *push* (PSH) aunque no es estrictamente necesario. Recordemos que el flag PSH indica que los datos de ese segmento y los datos que hayan sido almacenados anteriormente deben ser pasados a la aplicación receptora lo antes posible. Se puede dar el caso que varios segmentos que transportan datos no lleven activado el flag PSH; el TCP receptor almacenará esos datos pero no los entregará a la aplicación receptora hasta que reciba un segmento con el PSH activado.

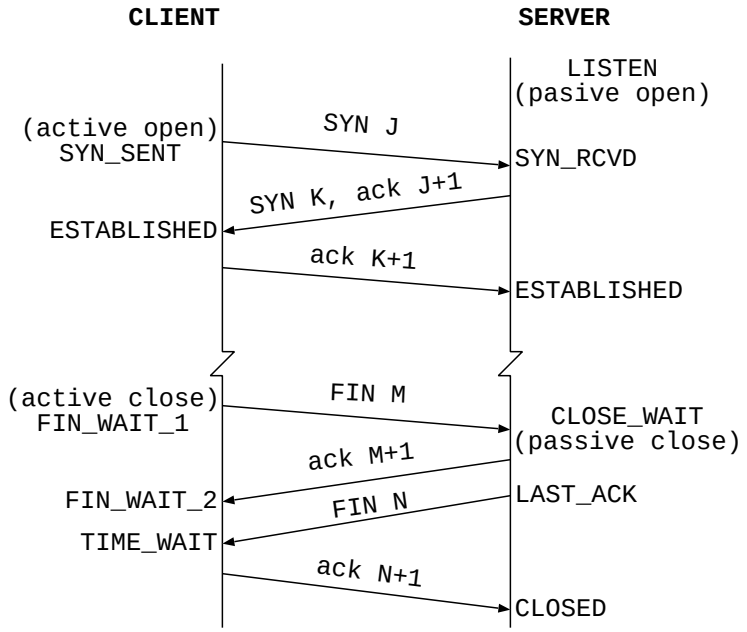

Figura 4.10: Estados de TCP correspondientes a un establecimiento y cierre de conexión

Sin embargo, la mayoría de implementaciones TCP activan el flag de PSH en cada segmento que transporta datos, haciéndolos disponibles a la aplicación inmediatamente.

4.5.5.2. El flag *URG y el campo* Urgent Pointer

El *flag* URG y el campo *Urgent Pointer* se utilizan cuando se desea enviar datos que se suponen que no pertenecen al flujo "normal" de bytes, sino que son datos urgentes. Estos bytes urgentes se insertan en medio del flujo "normal" de bytes, activando el *flag* URG y estableciendo el campo *Urgent Pointer* de la cabecera de TCP a un valor que sumado con el valor del campo de número de secuencia de la cabecera de TCP se obtiene el número de secuencia del último byte de los datos urgentes.

La especificación original de TCP no deja claro si el *Urgent Pointer* debe apuntar al último byte de los datos urgentes o al byte que sigue al último byte urgente. Aunque documentos posteriores aclararon que el *Urgent Pointer* debe apuntar al último byte urgente, la mayoría de implementaciones de TCP (derivadas de la realizada en Berkeley) continúan utilizando la interpretación equivocada. En Linux es posible indicar en qué forma se desea que funcione el *Urgent Pointer* variando el valor de la variable */proc/sys/net/ipv4/tcp_stdurg*. Si el valor de esta variable es 0 se utiliza la interpretación de Berkeley, si vale 1 se utiliza la interpretación del estándar.

TCP no especifica mucho más sobre los datos urgentes, así no hay forma de saber cuándo se inician los bytes urgentes en el flujo de datos. Es la aplicación la que debe saber cómo extraer los datos urgentes.

4.5.6. RTT y *Timeouts*

En el funcionamiento de TCP y el mecanismo de ventana deslizante (sección 4.5.2) se indica que TCP realiza un reconocimiento positivo de los segmentos que ha recibido correctamente. Si el segmento se recibe con errores, el TCP receptor lo ignora. Al cabo de un cierto tiempo (*timeout*) el TCP emisor lo retransmite. El cálculo del *timeout* se realiza de forma dinámica para cada conexión TCP que se establece ya que las condiciones de transmisión pueden cambiar por diversas razones, como por ejemplo la congestión de la red o los diferentes caminos seguidos por los datagramas IP para diferentes conexiones remotas.

El cálculo del *timeout* se basa en el cálculo del RTT (*Round Trip Time*). El RTT, definido de forma genérica, es el tiempo que tarda un paquete en ir del emisor hasta el receptor sumado al tiempo que tarda en llegar la confirmación del receptor al emisor.

TCP calcula el RTT de dos formas diferentes (aunque complementarias):

1. TCP inicia un temporizador cuando se envía un byte (aleatorio) que viaja en un segmento. Cuando TCP recibe el ACK del segmento donde viajaba el byte detiene el temporizador y obtiene el RTT.
2. Una segunda forma de calcular el RTT es a través de una opción de TCP, el *timestamp*. El TCP emisor incluye una marca temporal (*timestamp*) como opción del TCP. El TCP receptor copia la marca temporal recibida en el segmento de reconocimiento enviado. De esta forma el TCP emisor puede calcular el RTT de forma más sencilla que el caso anterior.

El RTT que se utiliza para calcular el tiempo de *timeout* corresponde a un valor medio obtenido con los diferentes cálculos de RTT durante la misma conexión.

El *timeout* (RTO) se calcula de forma proporcional al RTT[4].

4.6. Cálculo del *checksum* de UDP y TCP

El campo de *checksum* de TCP y UDP cubre tanto la cabecera como los datos (El *checksum* de la cabecera IP sólo cubre la propia cabecera IP). Para realizar el cálculo del *checksum*, además de utilizar todo el paquete (TCP ó UDP) (cabecera + datos), se añade una pseudo-cabecera que incluye la dirección IP fuente, la dirección IP destino, el campo de protocolo de la cabecera de IP y la longitud del paquete (TCP ó UDP). Tal y como se indica en el RFC793, esta pseudo-cabecera proporciona protección adicional contra paquetes recibidos erróneamente por problemas de enrutado.

El *checksum* es el complemento a uno de 16 bits de la suma en complemento a uno de todas las palabras de 16 bits de la pseudo-cabecera, cabera y texto del paquete (TCP o UDP).

Si el paquete contiene un número impar de octetos, el último octeto del paquete se rellena con ceros por la derecha hasta obtener una palabra de 16 bits con propósito de calcular el *checksum*.

En la Figura 4.11 se muestran los campos utilizados para el cálculo del *checksum* para un segmento TCP.

4.7. Ejemplos

4.7.1. Ejemplo 1

El siguiente ejemplo corresponde a una conexión TCP entre las máquinas *darkstar* (192.168.0.1) y *falcon* (192.168.0.5). Aunque TCP permite transmisiones full-duplex, en este caso los datos se transmiten de *darkstar* a *falcon*. La captura de las tramas se ha realizado con el programa `tcpdump` desde *darkstar*. Cada linea corresponde a una trama transmitida. El primer número a la izquierda es el número de trama, el siguiente es el tiempo relativo a la primera trama en segundos, el tercer número es la diferencia de tiempos entre esa trama y la anterior, a continuación se indica el host origen, puerto origen, host destino, puerto destino con la notación *host_o.puerto_o >host_d.puerto_d:*. A continuación se indican los flags activados S (SYN), F (FIN), P (PSH), R (RST); un punto significa que ninguno de los anteriores está activado. Después de los flags, se indica el número de secuencia del segmento con el formato *primero:último (nbytes)* que se traduce en números de secuencia *primero* hasta, pero no incluyendo, *último*. Por facilidad de lectura, los números de secuencia se presentan de forma completa en los segmentos FIN y posteriormente se presentan relativos a éstos. Otros parámetros son el reconocimiento, el tamaño de ventana, y las opciones del TCP.

[4]Investigaciones más recientes han derivado en algoritmos más exactos del calculo del RTT medio, calculando también la desviación media del RTT y ajustando mejor el cálculo del RTO.

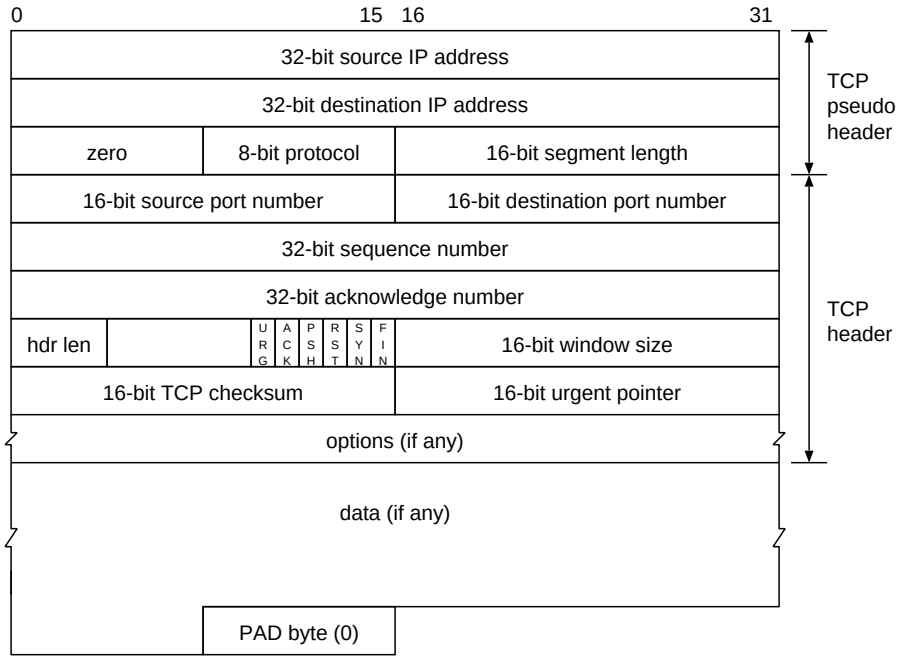

Figura 4.11: Campos para el cálculo del *checksum* de un segmento TCP

```
 1 0.000000            darkstar.32874 > falcon.7777: S 1485000412:1485000412(0) win 5840
                       <mss 1460,sackOK,timestamp 25460855 0,nop,wscale 0>
 2 0.100321 (0.100321) falcon.7777 > darkstar.32874: S 820456734:820456734(0) ack 1485000413
                       win 5792 <mss 1460,nop,nop,timestamp 2997742 25460855,nop,wscale 0>
 3 0.100434 (0.000113) darkstar.32874 > falcon.7777: . ack 1 win 5840
 4 0.100713 (0.000279) darkstar.32874 > falcon.7777: P 1:1025(1024) ack 1 win 5840
 5 0.100796 (0.000083) darkstar.32874 > falcon.7777: . 1025:2473(1448) ack 1 win 5840
 6 0.201881 (0.101085) falcon.7777 > darkstar.32874: . ack 1025 win 7168
 7 0.201988 (0.000107) darkstar.32874 > falcon.7777: . 2473:3921(1448) ack 1 win 5840
 8 0.202031 (0.000043) darkstar.32874 > falcon.7777: FP 3921:5121(1200) ack 1 win 5840
 9 0.204387 (0.002356) falcon.7777 > darkstar.32874: . ack 2473 win 10136
10 0.303457 (0.099070) falcon.7777 > darkstar.32874: . ack 3921 win 13032
11 0.304583 (0.001126) falcon.7777 > darkstar.32874: F 1:1(0) ack 5122 win 15928
12 0.304634 (0.000051) darkstar.32874 > falcon.7777: . ack 2 win 5840
```

En este ejemplo, *darkstar*, actuando como cliente, inicia una conexión desde el puerto 32874 con *falcon* al puerto 7777 enviando un segmento con el flag SYN activado y número de secuencia 1485000412, indicando un tamaño de ventana de 5840 bytes y en las opciones de TCP indica que desea recibir segmentos de tamaño máximo 1460 bytes (ver la sección 4.5.4.3) e incluye un *timestamp* para el cálculo del RTT.

falcon responde con un segmento SYN con un reconocimiento que indica que espera recibir un segmento cuyo número de secuencia sea 1485000413 (recordemos que el flag SYN activado “consume” un byte). Se puede observar en el campo de opciones del TCP, que *falcon* ha incluido su propia marca temporal (2997742) y devuelve el valor del *timestamp* que recibió en la trama anterior (2997742).

El segmento número 3, es el que cierra la fase de establecimiento de conexión a “tres bandas”. En este caso el número de reconocimiento es relativo al indicado en la linea 2. Es decir, que se indica que se espera recibir el segmento con número de secuencia 820456734 + 1.

Los segmentos 4 y 5 muestran cómo *darkstar* inicia la transferencia de datos con números de secuencia 1485000412 + 1 y 1485000412 + 1025; el segmento 4 tiene el flag PSH activado indicando que desea que la aplicación receptora pueda acceder a esos datos tan pronto como sea posible.

En el segmento 6 *falcon* reconoce positivamente los datos recibidos en el segmento número 4. El segmento 7 transporta 1448 bytes de datos de *darkstar* a *falcon.* El segmento 8 transporta 1200 bytes y tiene activados los flags de PSH (indicando que desea que la aplicación receptora pueda acceder a todos los datos recibidos tan pronto como sea posible) y el flag de FIN indicando que ese segmento inicia el cierre de la conexión.

Los segmentos 9 y 10, enviados desde *falcon* a *darkstar*, reconocen que los datos enviados en los segmentos 5 y 7 han sido recibidos satisfactoriamente.

En el segmento 11, *falcon* reconoce la recepción del segmento 8 y responde al cierre de conexión activando el flag de FIN. Finalmente *darkstar* reconoce el segmento 12 dando por finalizada la transmisión. Obsérvese que en este cierre se utilizan 3 segmentos (el 8, el 11 y el 12) en lugar de utilizar 4 segmentos tal y como se muestra en la figura 4.8

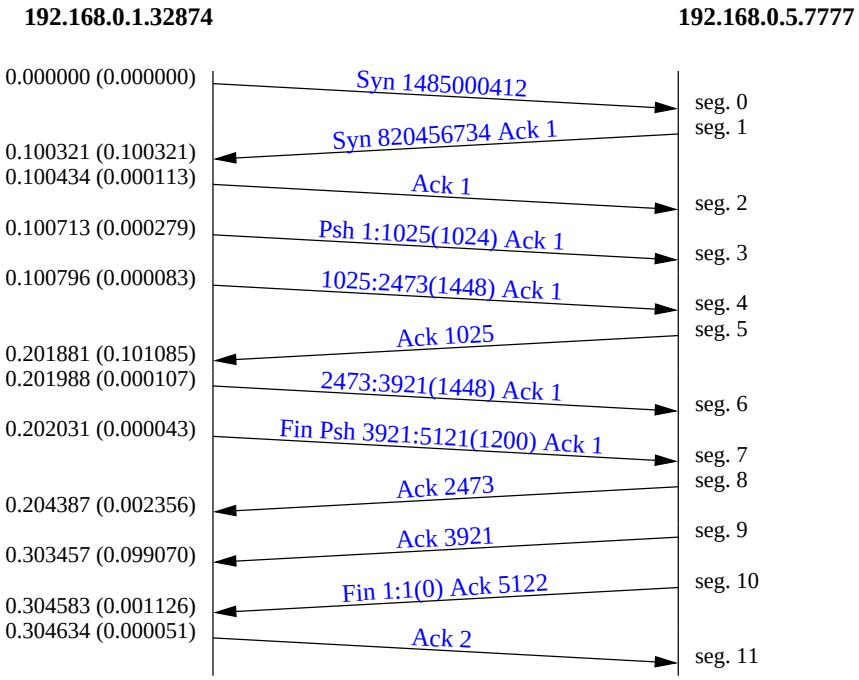

Figura 4.12: ICD (Inter Process Communication Diagram) del ejemplo 1

4.7.2. Ejemplo 2

En este ejemplo, se ha realizado una conexión entre *darkstar* y *falcon*. Sin embargo se ha utilizado un *driver* para simular pérdidas de paquetes y un retardo medio preestablecido (100 ms).

```
 1 0.000000           darkstar.32872 > falcon.7777: S 4166401040:4166401040(0) win 5840
                      <mss 1460,sackOK,timestamp 22866416 0,nop,wscale 0>
 2 0.100374 (0.100374) falcon.7777 > darkstar.32872: S 3485906442:3485906442(0) ack 4166401041
                      win 5792 <mss 1460,nop,nop,timestamp 403795 22866416,nop,wscale 0>
 3 0.100483 (0.000109) darkstar.32872 > falcon.7777: . ack 1 win 5840
 4 0.100768 (0.000285) darkstar.32872 > falcon.7777: P 1:1025(1024) ack 1 win 5840
 5 0.100850 (0.000082) darkstar.32872 > falcon.7777: . 1025:2473(1448) ack 1 win 5840
 6 0.201934 (0.101084) falcon.7777 > darkstar.32872: . ack 1025 win 7168
 7 0.202032 (0.000098) darkstar.32872 > falcon.7777: . 2473:3921(1448) ack 1 win 5840
 8 0.202074 (0.000042) darkstar.32872 > falcon.7777: P 3921:5369(1448) ack 1 win 5840
 9 0.303513 (0.101439) falcon.7777 > darkstar.32872: . ack 1025 win 7168
10 0.692975 (0.389462) darkstar.32872 > falcon.7777: . 1025:2473(1448) ack 1 win 5840
11 0.794419 (0.101444) falcon.7777 > darkstar.32872: . ack 3921 win 10136
12 0.794503 (0.000084) darkstar.32872 > falcon.7777: P 3921:5369(1448) ack 1 win 5840
```

```
13 0.795749 (0.001246) darkstar.32872 > falcon.7777: P 5369:6145(776) ack 1 win 5840
14 0.896720 (0.100971) falcon.7777 > darkstar.32872: . ack 3921 win 10136
15 1.252974 (0.356254) darkstar.32872 > falcon.7777: P 3921:5369(1448) ack 1 win 5840
16 1.354419 (0.101445) falcon.7777 > darkstar.32872: . ack 6145 win 13032
17 1.354519 (0.000100) darkstar.32872 > falcon.7777: . 6145:7593(1448) ack 1 win 5840
18 1.354561 (0.000042) darkstar.32872 > falcon.7777: P 7593:9041(1448) ack 1 win 5840
19 1.354835 (0.000274) darkstar.32872 > falcon.7777: FP 9041:10241(1200) ack 1 win 5840
20 1.455991 (0.101156) falcon.7777 > darkstar.32872: . ack 7593 win 15928
21 1.842980 (0.386989) darkstar.32872 > falcon.7777: P 7593:9041(1448) ack 1 win 5840
22 1.944446 (0.101466) falcon.7777 > darkstar.32872: . ack 9041 win 18824
23 1.944555 (0.000109) darkstar.32872 > falcon.7777: FP 9041:10241(1200)
24 2.045837 (0.101282) falcon.7777 > darkstar.32872: F 1:1(0) ack 10242 win 21720
25 2.045940 (0.000103) darkstar.32872 > falcon.7777: . ack 2 win 5840
```

Los tres primeros segmentos corresponden a la apertura de la conexión. Una vez establecida la conexión *darkstar* inicia la transmisión de datos en los segmentos 4 y 5, recibiendo un reconocimiento del segmento 4. *Darkstar* continúa transmitiendo datos en los segmentos 7 y 8. El segmento 8 tiene activado el flag PSH. *falcon* vuelve a reconocer, con el segmento 9, el segmento número 4 indicando al emisor que el siguiente (o siguientes) segmento no ha sido recibido. En el segmento número 10, *darkstar* retransmite los datos 1025 al 2472. *falcon* indica con el segmento 11, que los datos enviados en el segmento 7 fueron correctamente recibidos, pero no fue así con los datos del segmento 8. *Darkstar* reenvía, en el segmento 12, los datos que se perdierón en el segmento 8 (del 3921 al 5368) y en el segmento 13 envía nuevos datos. *falcon* vuelve a indicar que los últimos datos recibidos son los del segmento 10 y que espera recibir el byte 3921 y siguientes. En el segmento número 15, *darkstar* retransmite por tercera vez los bytes 3921 al 5368, que son reconocidos por *falcon*, junto con los datos enviados en el segmento 13, con el segmento 16. *darkstar* envia 3 nuevos segmentos (17, 18 y 19), activando el flag de cierre de conexión en el último de ellos. Los 6 últimos segmentos siguen el mismo procedimiento hasta finalizar, con exito, la transmisión.

Observese que el segmento número 23 vuelve a tener activado el flag de FIN como en el segmento número 19. Esto es lógico ya que *falcon* indicó que el segmento 19 no había sido recibido y por lo tanto desconocía el hecho de que *darkstar* estaba intentando cerrar la conexión. Por esta razón, *darkstar* vuelve a indicar el cierre de la conexión.

4.8. Ejercicios

Ejercicio 1

En este ejercicio se pretende visualizar los diferentes estados del cliente TCP a lo largo de una conexión. Para ello se va a utilizar el programa *sock* de R. STEVENS en modo cliente, para generar tráfico. El programa *sock* permite generar tráfico TCP y UDP actuando como cliente. Actuando como servidor, *sock* se convierte en un sumidero del trafico generado por el cliente. *sock* tiene varias opciones para configurar el funcionamiento, los tamaños de los buffers, el número de buffers a transmitir, etc (para ver todas las opciones ejecutar *sock* sin parámetros). El ordenador 147.83.40.50 tiene ejecutándose en el puerto 7777 un servidor *sock*. Entre los ordenadores del laboratorio y el puerto 7777 del ordenador 147.83.40.50 se ha añadido un retardo medio de 1.2 segundos.

El ejercicio se realiza desde la consola gráfica de los ordenadores, teniendo abiertos simultáneamente dos terminales ("*Konsole*"). Desde uno de ellos visualizaremos el estado de la conexión TCP de forma continua mediante el comando *netstat* con la opción -c para que se ejecute forma iterativa cada segundo. Específicamente el comando que se va a introducir es:

```
host> netstat -anc | grep 7777
```

Desde la otra consola, ejecutaremos el programa *sock* para que envíe 100 buffers de longitud 1024 bytes (longitud por defecto), al servidor (147.83.40.50), puerto 7777:

```
host> sock -n100 -i 147.83.40.50 7777
```

A medida que se establezca la conexión, se transmitan los buffers y se cierre la conexión, en la consola del *netstat* veremos los diferentes estados del TCP cliente.

Responda a las siguiente preguntas:

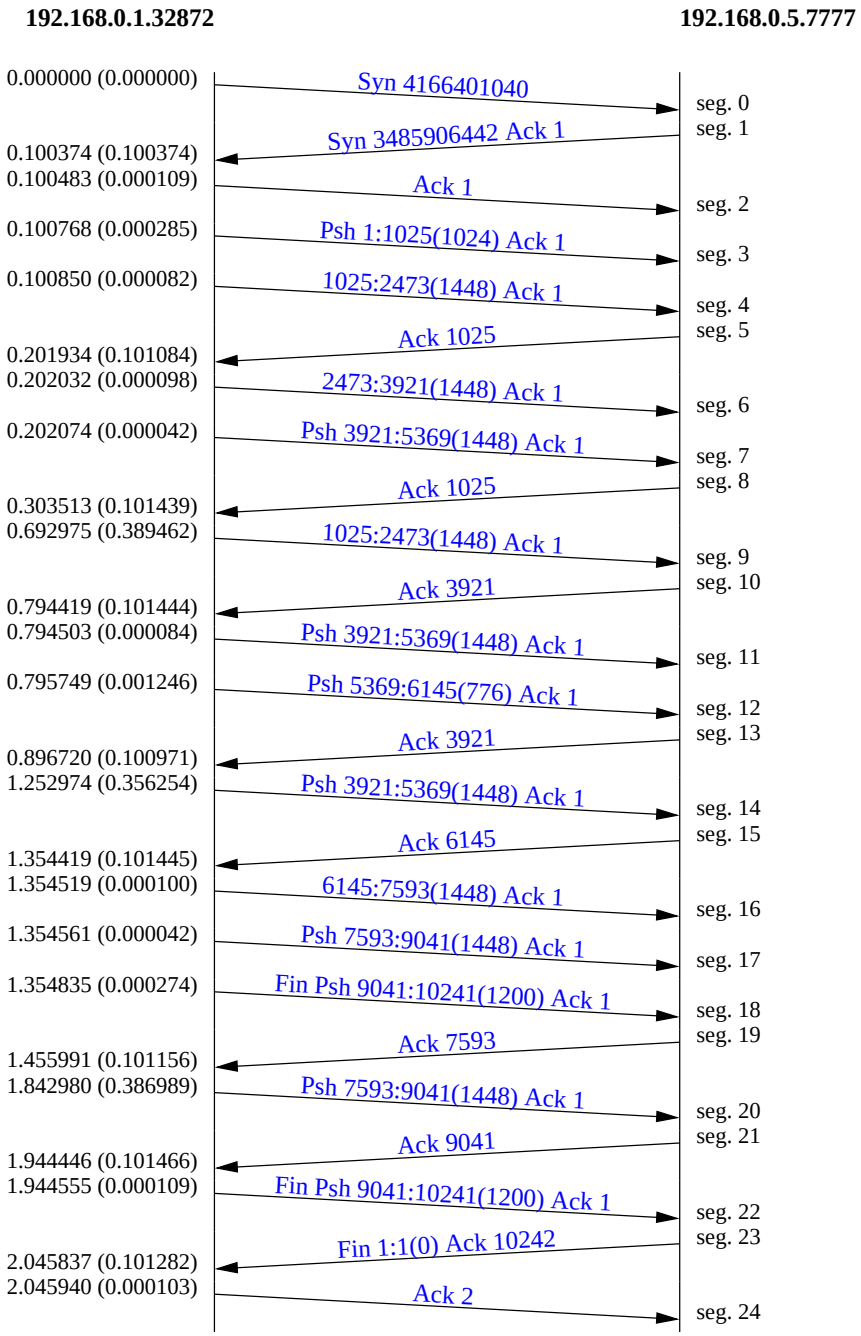

Figura 4.13: ICD del ejemplo 2

1. Ayudándose del la Figura 4.9, interprete cada uno de los estados que ha visualizado, indicando, además, cómo se ha llegado a cada uno de esos estados.

2. El servidor sock se ha ejecutado con las opciones:
   `sock -i -F -Q 5 -s 7777`.
   ¿Qué ocurriría si la opción -Q 5 no se hubiese utilizado?

3. ¿Para qué se ha utilizado un retardo tan elevado?

Ejercicio 2

En este ejercicio se comprobará que en estado TIME_WAIT, no es posible utilizar el mismo puerto para volvernos a conectar al servidor. Para ello ejecutamos:

```
host> sock -n8 -i -v 147.83.40.50 7777
```

Con la opción -v (verbose), se obtendrá, por pantalla, el puerto que está utilizando el cliente para conectarse al servidor. Una vez acabada la conexión pero todavía en estado TIME_WAIT, volvemos a ejecutar el cliente utilizando como puerto de salida el valor de puerto obtenido en la conexión anterior:

```
host> sock -n8 -i -b port 147.83.40.50 7777
```

La opción *-b port* obliga al cliente a utilizar el puerto *port* para la conexión.

Estimar la duración del estado TIME_WAIT.

Ejercicio 3

El objetivo de este ejercicio es la captura y análisis de tráfico UDP, con objeto de comparar este estudio con la captura y análisis de trafico TCP que se realizará en ejercicios posteriores. El servidor que se utilizará para conectarnos y enviar tráfico UDP está en el puerto 7778 del host 147.83.40.29. Como en los ejercicios anteriores, se utilizará el programa *sock* incluyendo la opción -u para indicar que deseamos tráfico UDP. Para poder observar el tráfico que vamos a generar utilizaremos *ethereal*. El filtro para que capture únicamente el trafico que deseamos será:

```
proto UDP and port 7778 and host 147.83.40.2x and 147.83.40.50
```

Una vez capturado el tráfico, analice los datagramas y responda a las siguientes preguntas:

1. ¿Cuántos segmentos se utilizan para la apertura de la conexión?
2. ¿Cual es el puerto que ha utilizado el cliente?
3. ¿Pueden coexistir simultáneamente dos clientes UDP con el mismo puerto de salida?
4. ¿Pueden coexistir simultáneamente un cliente UDP y otro TCP con el mismo puerto de salida?

Ejercicio 4

En este primer ejercicio de captura de tráfico TCP, se realizará un telnet al puerto de *echo* del servidor 147.83.40.50; (el puerto de *echo* de un servidor, devuelve los mismos datos que se han enviado). Realice la captura de una sesión y analice el tráfico de manera semejante al análisis realizado en la sección 4.7.1.

Ejercicio 5

En este segundo ejercicio de captura de tráfico TCP, la conexión debe realizarse al puerto *daytime* del servidor 147.83.40.50. (Como indica el nombre, el puerto *daytime* de un servidor proporciona la fecha y hora actual del sistema de ese servidor). Realice la captura de una sesión y analice el tráfico de manera semejante al análisis realizado en la sección 4.7.1.