ÍNDICE GENERAL

CAPÍTULO 3

CAPA DE TRANSPORTE EN INTERNET

3.1 Introducción

En capa de transporte utilizamos dos protocolos **extremo a extremo**: UDP y TCP. No tienen nada que ver con los routers (en teoría, en la práctica saben de capa de transporte si tienen un servidor dhcp). Ambos **se encapsulan en datagramas IP**.

> **Definición 3.1.1 Puerto**
> Número de 16 bits que se utiliza para determinar a qué servicio le debe llegar la información una vez recibido el mensaje. Hay hasta 65535 puertos, son números de 16 bits (2 bytes). Los puertos superiores a 1024 son de **libre disposición**.

3.2 Protocolo de datagrama de usuario (UDP)

Es uno de los protocolos más antiguos.

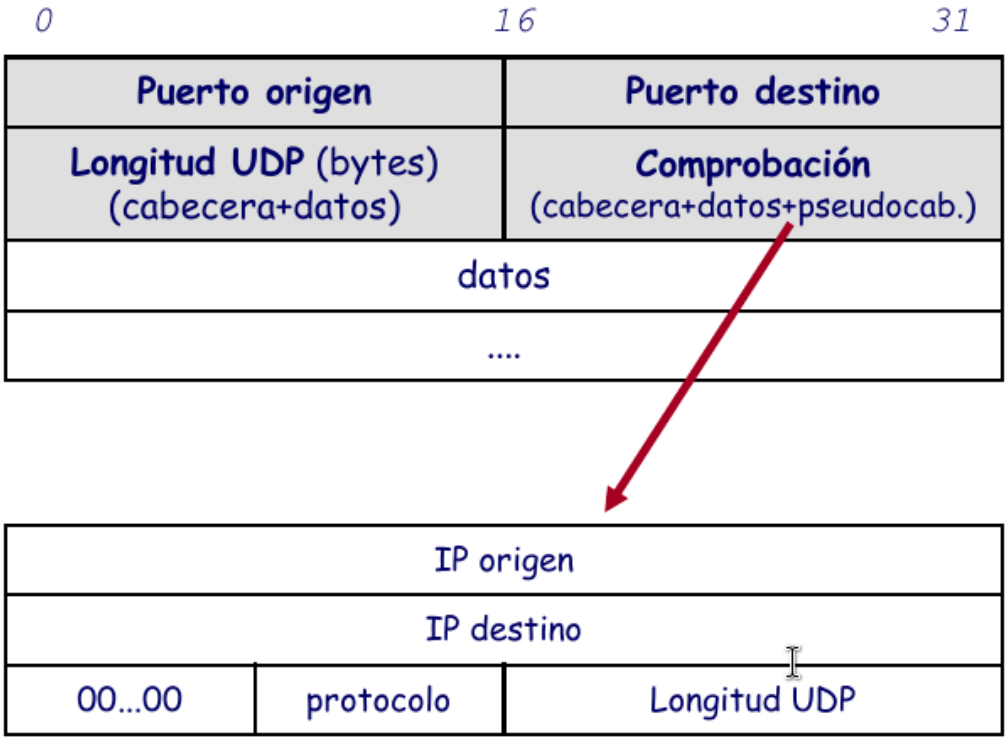

Figura 3.1: Qué datos se añaden al datagrama al usar el protocolo UDP

Características UDP

- Protocolo *best-effort*, él asume que las cosas van a funcionar bien.
- **No orientado a conexión**: no hay confirmación de conexión, no hay retardos para establecerla y cada mensaje es independiente.
- **No es fiable**: se pueden perder los paquetes.
- No sabes si llegarán **desordenados**.
- **No controlas congestión ni flujo**, quieres ser lo más rápido posible.
- Realiza **multiplexación/demultiplexación**: de quién viene, a quién va. Se usan los puertos para determinar la aplicación destino y origen.
- Se suele usar para **aplicaciones multimedia** porque manda la información de forma liviana y rápida, siendo tolerantes a fallos pero no a retardos.

La cabecera UDP de añade sobre el datagrama (se llaman **datagramas de usuario** porque añaden muy poco a ip), tiene las siguientes partes:

- **Puerto origen y destino**: Usamos puertos porque son universales y estandarizados.
- **Un checksum adicional**: (cabecera + datos + pseudocabecera) Incluye cosas que se incluían en el checksum de antes para cuando hay NAT.
- **Longitud de UDP**: (cabecera + datos)

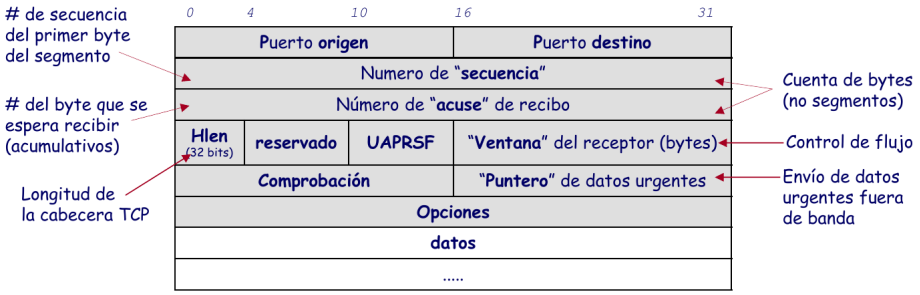

Figura 3.2: Qué datos se añaden al datagrama al usar el protocolo TCP. Se llaman **segmentos** TCP.

3.3 Protocolo de control de transmisión (TCP)

TCP es el otro protocolo de la capa de transporte, que es mucho más complejo de UDP por su interés en control de errores y fiabilidad. Tiene las siguientes características:

Características TCP

- Es **punto a punto**. (NO sirve para uno a muchos)
- **Orientado a conexión**: Exige un *hand-shaking* (extremo a extremo los dos hosts se sincronizan antes de enviar los datos). Tienes **tres fases**:
  1. Establecimiento.
  2. Intercambio de datos.
  3. Cierre.
- Garantiza **entrega ordenada**: El primero que entra es el primero que sale y hay entrada garantizada.
- **Full duplex**: Bidireccional. Ambos puntos pueden enviar y recibir.
- Tiene **detección y recuperación de errores** con confirmaciones (*ACKs*) y *timeouts* (mando un ack y si no vuelve después de un tiempo reenvío el segmento, el tiempo es el timeout).
- Tiene **dos formas de arreglar errores**: **repetición** y **FEC**(forward error corrector). EN FEC sé de antemano el error y meto información reudndante que pueda arreglarlo antes de ejecutar. No se usa en internet, solo en móviles.
- Es **fiable**, controla **congestión** y **flujo** con el uso de ventanas deslizantes.
- **Incorpora confirmaciones con los datos** (*piggybacking*)
- Se **adapta** a las condiciones de red **dinámicamente**.

Entonces, en resumidas cuentas, las **responsabilidades** de TCP son:

1. Multiplexación/demultiplexación de aplicaciones (a quién va de quién viene).
2. Control de conexión (establecimiento y cierre).
3. Control de errores y flujo.
4. Control de congestión.

UAPRSF, explicación de imagen

1. U: Urgent
2. A: ACK
3. P: Push
4. R: Reset
5. S: Syncronize (establecer)
6. F: Fin (cerrar)

3.3.1. Multiplexación/demultiplexación

Para la **multiplexación y demultiplexación** de TCP (transportar segementos a las aplicaciones correctas) se usan los puertos. La explicación que he puesto en UDP se aplica aquí también. Usa también **puerto origen y destino**.

3.3.2. Control de la conexión

Pasos de la conexión

1. **Establecimiento**: Sincronizar los dos hosts y reservar recursos en ambos.
2. **Intercambio de datos**: en ambas direcciones.
3. **Cierre de conexión**: Liberación de recursos.

No es posible garantizar fiabilidad sin ambigüedad usando los recursos de un mecanismo que **no** es fiable como IP. Por eso hacen falta mecanismos para asegurar que van llegando los paquetes en su orden.

three-way handshake

1. Cliente manda un SYN=1 al servidor (Apertura activa)
2. El servidor recibe el SYN, reserva recursos y manda un ACK + SYN. Esta reserva de recursos es casi incondicional.
3. El cliente recibe el ACK y actualiza sus números de secuencia. Devuelve un ACK para el SYn del servidor

Números de secuencia

> **Definición 3.3.1 Número de secuencia**
>
> Campo de 32 bits que cuenta bytes en módulo 232 (el contador se da la vuelta cuando llega al valor máximo). No empieza normalmente en 0, sino en un valor denominado **ISN** (Initial Sequence Number) elegido “teóricamente” al azar; para evitar confusiones con solicitudes anteriores. El ISN es elegido por el sistema. Como son pseudoaleatorios, se puede hacer un **ataque de suplantación**.

TCP incrementa el número de secuencia de cada segmento **según los bytes** que tenía el **segmento anterior**, con una sola excepción: los flags **SYN y FIN** incrementan en **1** el número de secuencia.

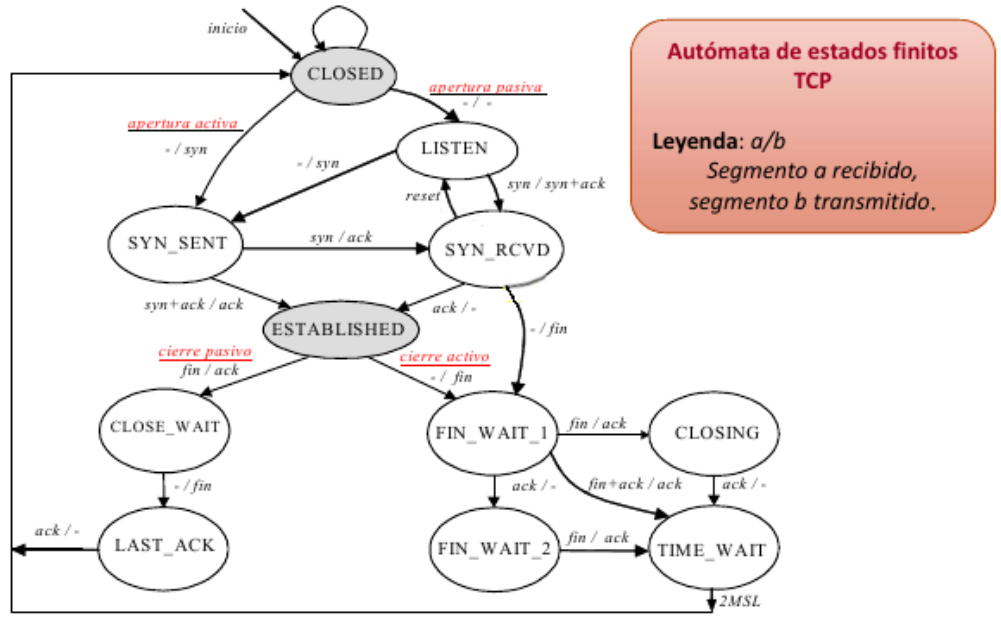

Figura 3.3: Protocolo tcp y sus cambios en un autómata. SYN_RCVD es SYN recibido

Los segmentos **ACK** (sin datos) **no incrementan el número de secuencia**.

EL campo **ACK** también **va numerado**, indican una cuenta en bytes y si pongo ahí un número está **todo confirmado hasta esa cuenta de bytes**.

Algunos posibles problemas

1. **Conexión simultánea**: Mandan un SYN a la vez y luego mandan SYN+ACK a la vez, dando información redundante que gasta recursos.
2. **SYN retrasados y duplicados**: Pueden perderse syn, o tardar más que lo siguientes, causando ACKs fuera de orden y paquetes perdidos.

Control de conexión

1. Un host manda una señal de cierre FIN=1 (**cierre activo**)
2. El otro lo recibe, lo confirma y pone el bit FIN=1. (**Cierre pasivo**)

3.3.3. Control de errores y de flujo

Cuando TCP manda un segmento arranca un **timer**, cuando el timer alcanza un valor, si no ha llegado un ack, entonces manda una interrupcion del sistema (**TIMEOUT**) y el segmento se vuelve a repetir. Tantas veces haga falta para que el segmento llegue a su destino. Es ineficaz porque solo aprovecho un tiempo mínimo de mi velocidad de transmisión hasta el próximo timeout.
Se van **mandando segmentos y creando relojes por cada uno** para que no haya parones en la transmisión y aprovechemos los recursos. Cualquiera de ellos se puede perder y cualquiera puede

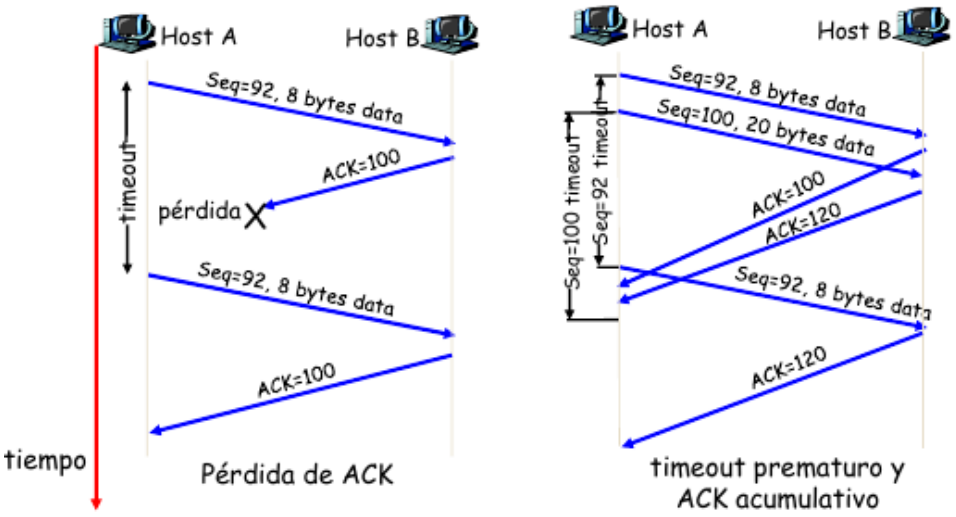

Figura 3.4: Control de errores en TCP. A la izquierda se pierde el ack y manda de nuevo el mismo paquete. A la derecha el timeout es demasiado rápido y manda paquetes innecesarios.

tener errores, así que en el emisor en la memoria, **por cada segmento que envía**, tiene que guardarse una **copia local** y si ocurre un timeout reenviar el paquete.

Necesito una zona de memoria donde voy **almacenando temporalmente los segmentos** hasta que reciba el ACK. **Cuando llegue el ACK** del segmento, **libero la memoria** reservada.
El **receptor** tiene que estar **sincronizado** para esperar el número de secuecia del segmento. Él tiene una zona de memoria para que, si llegan los segmentos desordenados, los puede **almacenar temporalmente para ordenarlos** y pasárselo a la aplicación cuando esté todo.

A estas zonas de memoria se les llaman **ventanas deslizantes**. Tenemos dos: Ventana deslizante del emisor (por si falla el envío) y del receptor (por si llega desordenado).

El timeout es **crítico** para la **eficacia** del protocolo.

Cómo estimar timeout

Tenemos que calcularlo **segmento a segmento**, porque la conexión varía en cada momento. Para eso, se usan **medias móviles**. El tiempo que se estima se llama RTT (Round Trip Time). Este tiempo depende de un parámetro α que determina cómo de sensible es a los grandes cambios (mayor alfa más sensible).

Control de flujo

Hay que **evitar** que el **emisor sature** al **receptor** por enviar demasiada información o demasiado rápido. Si yo te dicto a una velocidad superior a la que puedes transcribir o copiar entonces llega

Cuadro 3.1: Control de errores de TCP

Evento	Acción
Llega **ordenado**, **sin discontinuidad**, **todo** lo anterior **ya confirmado**	**Retrasa el ACK** con la esperanza de que el próximo segmento llegue pronto y los confirme los dos con el mismo mensaje.
Llega **ordenado**, **sin discontinuidad**, hay **pendiente un ACK**	**Enviar un ACK acumulativo** (con todo lo de antes)
Llega **desordenado**, con número de **secuencia mayor al esperado**, discontinuidad	Se **almacena temporalmente** y se **confirma** lo que teníamos **antes de la discontinuidad** para que nos envíen lo que falta. Si no está ordenado no puedo mandar el ACK más reciente.
LLega un segmento que **completa discontinuidad**.	Si completa el huequito que me faltaba, **se confirma (ACK) lo que puedas** (lo que tengas completo después de rellenar el huequito)

un momento en el que el buffer se te llena y pierdes paquetes. Para esto, el receptor envía cuántos créditos (bytes) puede recibir al emisor.

$$\textbf{ventana útil emisor} = \textbf{ventana ofertada receptor} - \textbf{bytes en transito}$$

Existe un temporizador (**timer de persistencia**)por si se pierde el ACK con el tamaño de ventana disponible, que manda 1 byte para ver si queda espacio o si el receptor responde. Si no responde, inicia el timer de nuevo o espera el próximo ACK.

3.3.4. Control de congestión

La congestión se gestiona en el emisor **controlando el tamaño máximo de ventana de emisión**. Se origina porque los recursos de la red son limitados. Mientras que el flujo tiene que ver con la velocidad con la que mando y consumo info extremo a extremo, la **congestión es salto a salto**. Queremos **maximizar la eficacia**, que sea **imparcial** (no quiero que se prime a una fuente más que otra) y **estable**.

BandWidth-Delay te dice cuál debería ser tu ventana ideal. Es el **tamaño mínimo de la ventana de emisión** para que no hayan parones.

BW x Delay

Delay = RTT

BW = bits/seg x seg = bits

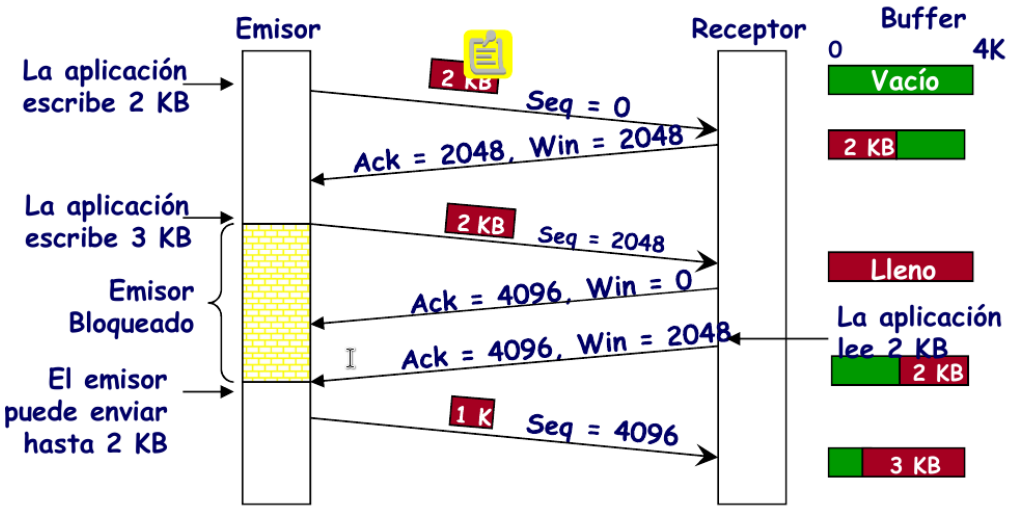

Figura 3.5: Control de flujo en TCP. Lo rojo son los datos, los ack van diciendo por dónde va el receptor. WIN es la ventana del receptor, que indica al emisor cuánto puede mandarle.

TCP-TAHOE

1. El límite del control de flujo está implícito.
2. Es el **mínimo entre el del control de flujo** (ventana receptor) y **el de congestión** (ventana congestión)
3. Inicialmente la ventana de congestión vale 1, **por cada ACK recibido o segmento confirmado aumenta 1** (1+1+2+4+8...). Va multiplicando por 2, la ventana crece **exponencialmente**.
4. Cuando supero un tamaño prefijado (**umbral**) cada vez que se reciben todos los ack pendientes **sumo 1** (lineal).
5. Cuando hay un **timeout** reacciono y **ventana congestión se vuelve 1** y el **umbral** es la **mitad** de **ventana congestión**. El umbral es un **parámetro de tcp**.

Existen **sabores** de TCP. Para el examen, solo tenemos que conocer **Tahoe** y **Reno**. En Reno, **si llegan 3 ack seguidos iguales al emisor,** es que hay congestión. **Parto la ventana por la mitad** y sigo **lineal**, en vez de ponerlo a 1.

3.4 Extensiones TCP

En linux se usa CuBIC, que usa una función cúbica del tiempo desde la última congestión. Pero en el examen solo pedirán Tahoe o Reno, así que esto nada más es una curiosidad teórica.