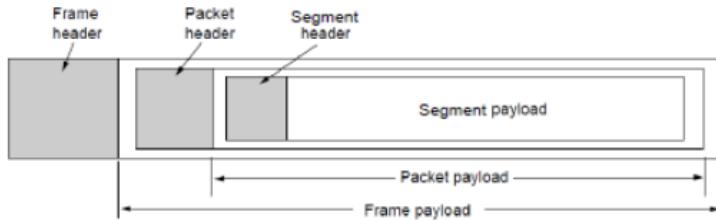


Capa de transporte

- Se ejecuta en las WMR, confia en la capa de Red
- Lo entiende el transporte en el hardware de la CT.
- Permite que las aplicaciones se ejecuten como si estuvieran directamente conectadas.

Intercambia Segmentos, que están contenidos en paquetes (CR) contenidos en frames (CD)



Se envían paquetes:
- de datos
- info de control.

Deben entregarse ordenadamente:

- **Solución 1:** Para la entrega ordenada de segmentos al host de destino se puede:

- Numerar los segmentos a enviar (usando **números de secuencia**) – respetando el orden del flujo de datos recibido de la capa de aplicación.
- Usar para cada número de segmento enviado un **temporizador de retransmisiones**.
- Mandar **confirmaciones de recepción (ACK)** para segmentos recibidos correctamente.
- Si expira el temporizador de un segmento sin recibir el ACK, retransmitir el segmento correspondiente.
- Los segmentos recibidos son **re-ensamblados en orden** y entregados a la capa de aplicación del receptor.

er responden móviles.
a TCP, conexiones
y terminos.
dicen temporizadores
7 reenvíos de fragmentos

Protocolo Seguro que garantiza envío correcto:
TCP: resuelve

- Retransmisión de paquetes:
 - uso de números de secuencia, confirmaciones de recepción y temporizadores.
- Fijar la duración de temporizadores de retransmisiones (algoritmo complejo)
- Manejo de conexiones entre pares de procesos
- Direccionamiento
- Control de congestión
- Control de flujo

Una ETCP acepta **flujos de datos** a transmitir de procesos locales,

- Cada flujo de datos se **divide en fragmentos** llamados segmentos que no exceden los 64 KB,
- y se envía cada segmento dentro de un datagrama IP.

Utilizan sockets entre hosts → se usan para las comunicaciones.

Socket: IP + puerto.

→ las conexiones se identifican mediante los identificadores de sockets (S_1, S_2).

Importante: Cada byte de un flujo de datos a enviar en una conexión TCP tiene su propio **número de secuencia** de 32 bits.

– Esto impone un límite en el tamaño de un flujo de datos.

¿Por qué se necesitan los números de secuencia?

– para confirmaciones de recepción y para otros asuntos según veremos.

La ETCP emisora y la receptora intercambian datos en forma de **segmentos**.

– Segmento = **encabezado TCP** ++ (0 o más bytes) de datos.

– Cada segmento, debe caber en la carga útil de 65.515 bytes del IP.

– Cada red tiene una **unidad máxima de transferencia (MTU)** y cada segmento debe caber en la MTU.

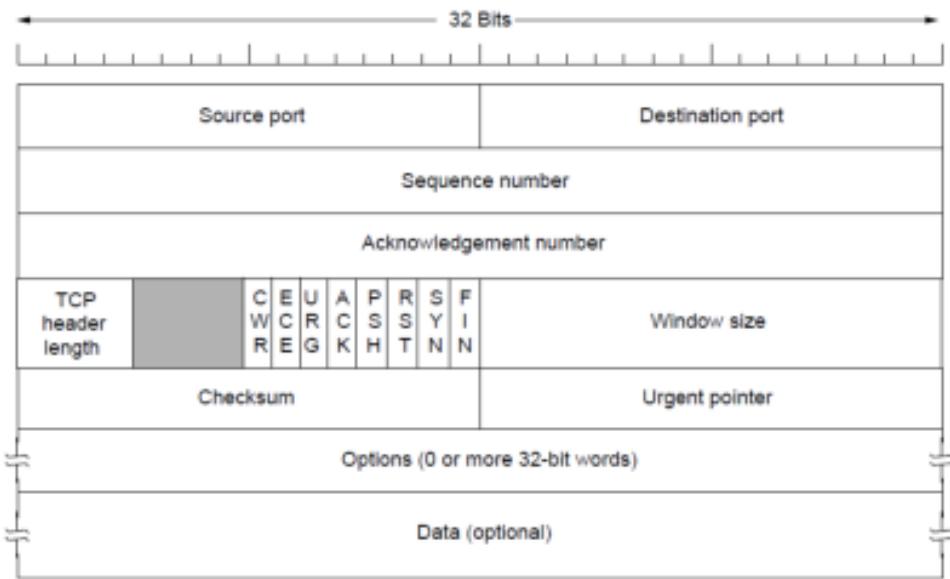
- En la práctica la MTU es usualmente de 1500 bytes (el tamaño de la carga útil de Ethernet).

TCP ~~funcionamiento~~ | las entregas siguen su número de secuencia.

Cuando un transmisor envía un segmento, también inicia un temporizador.

- Cuando llega el segmento a destino, la ETCP receptora devuelve un segmento (con datos si existen, sino sin ellos) que contiene un **número de confirmación de recepción** igual al *siguiente número de secuencia que espera recibir*.
- Si el temporizador expira antes de llegar el ack, el emisor envía de nuevo el segmento.

1. Encabezado fijo de 20 bytes
2. Opciones de encabezado en palabras de 32 bits
3. Datos opcionales



Los segmentos sin datos se usan para acks y mensajes de control.

Puerto de origen y puerto de destino:

- Son de 16 b cada uno
- La dirección de un puerto mas la dirección IP del host forman un punto terminal unico de 48 b
- Los puntos terminales de origen y de destino en conjunto identifican la conexión

El campo **numero de secuencia** de un segmento es un numero de byte en el flujo de bytes transmitido y corresponde al primer byte en el segmento. Tiene 32 b de longitud

El campo **numero de confirmacion de recepcion** indica el siguiente byte esperado del flujo de bytes a transmitir, Tiene 32 b de longitud

Longitud del encabezado TCP: N° de palabras de 32 bits en el encabezado TCP

Longitud del campo de opciones: variable

Direccionalamiento

Si el cliente no tiene un puerto entrante cuando recibe:

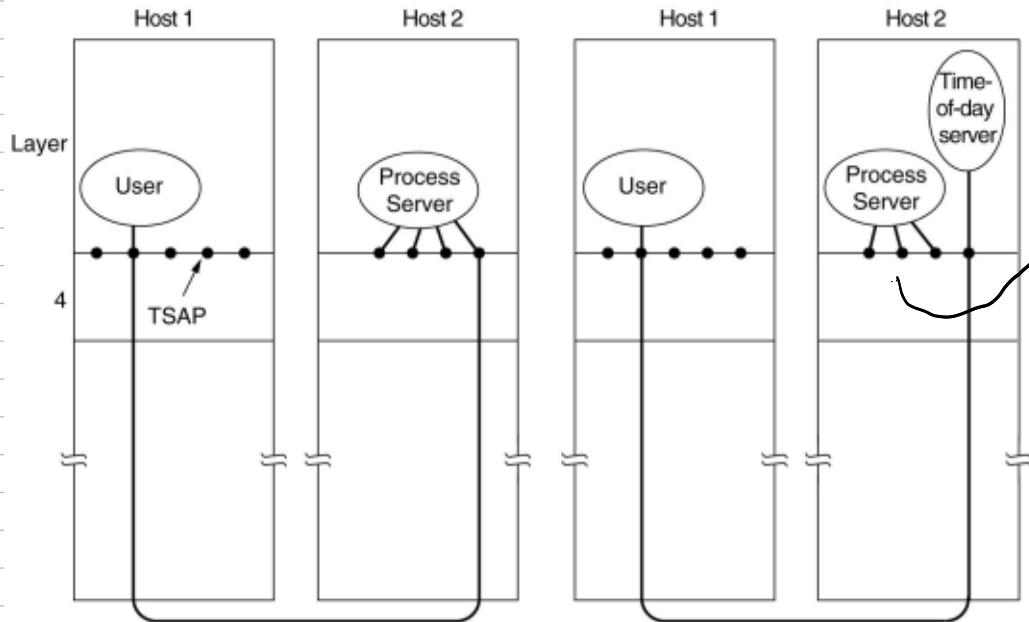
Solución: Existe un proceso especial llamado **servidor de directorio** que para cada tipo de servicio sabe cuáles son los puertos de los servidores que prestan ese tipo de servicio.

- Pasos seguidos:
 1. El usuario establece una conexión con el servidor de directorio (que escucha en un puerto bien conocido).
 2. El usuario envía un mensaje especificando el nombre del servicio.
 3. El servidor de directorio le devuelve la dirección puerto.
 4. El usuario libera la conexión con el servidor de directorio y establece una nueva con el servicio deseado.

¿Cómo se hace cuando se crea un servicio nuevo?

- El servicio nuevo debe registrarse en el servidor de directorio, dando su nombre de servicio como la dirección de su puerto.
- El servidor de directorio registra esta información en su base de datos.

Como no todos los procesos están siempre activos entonces el **SERVICIO DE PROCESOS** se encarga de encuadrar a los procesos en **inactivos** y "despertarlos" cuando hace falta.



Cuál es la LIF con la que lleva la ejecución → procesos?

Puertos bien conocidos

- N° puertos bien conocidos, son los números menores a 1024
- Tabla de puertos bien conocidos (ver abajo).
- **Demonios** = procesos servidores que atienden en un puerto
 - P. ej. que el *demonio FTP* se conecte a sí mismo al puerto 21 en el tiempo de arranque.

Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

Problema: Se podría llenar la memoria con demonios que están inactivos la mayor parte del tiempo.

Solución: Un solo demonio llamado **inetd (demonio de internet)**, escucha un conjunto de puertos al mismo tiempo y espera por un pedido de conexión.

- Usuarios potenciales de un servicio comienzan a hacer **pedido CONNECT** especificando el puerto del servicio que quieren.
- Si no hay ningún servidor esperando por ellos, *inetd* bifurca un nuevo proceso y ejecuta el demonio apropiado en él, y ese demonio maneja la solicitud.

- Inetd aprende qué puertos va a usar de un **archivo de configuración**.

- Los demonios asociados a los puertos de este archivo **solo** están activos si hay trabajo para hacer.

- Se puede tener **demonios permanentes** en los puertos más ocupados e inetd en los demás.

- Esto lo fija el administrador de sistema.

Entrega de datos confiable.

problema: duplicar.

sol: N° de secuencia, algo en número pred ver si: es un nuevo o duplicado.

Protocolo de Parada y Espera

■ **Suposición:** el canal de comunicaciones subyacente puede perder paquetes (de datos, de ACKs)

- Los paquetes tienen N° de secuencias.
 - Con 1 bit es suficiente.
- Se trabaja con Ack's
 - El receptor debe especificar N° de secuencia del paquete siendo confirmado.
- Se usan retransmisiones de paquetes.
 - Para esto se requiere de uso de temporizadores.

Comportamiento del emisor:

1. El emisor envía paquete P y **para** de enviar.
2. **Espera:** El emisor espera una cantidad "razonable" de tiempo para el ACK
3. Si llega el ACK a tiempo, se envía siguiente paquete. Goto 2.
4. Sino se retransmite paquete P. Goto 2.
- Si hay paquete o ACK demorado pero no perdido:
 - La retransmisión va a ser un duplicado con igual número de secuencia ; luego se descarta en el receptor.

Protocolo de tubería.

Mismo tiempo.

- Emisor tiene un buffer donde guarda los paquetes hasta que recibe la confirmación.

Retroceso-N.

receptor envía ACK acumulativo: mayor N a rec tq todos los que anteriores se retransmiten pierden.

- Emisor tiene timer por el paquete ms visto no confirmado.

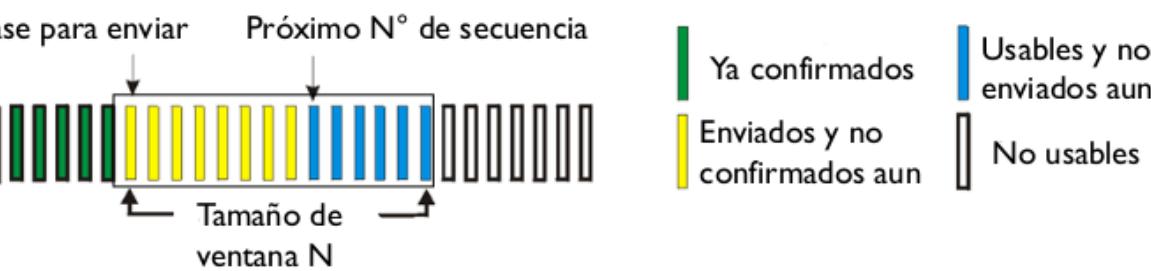
- El receptor descarta todos los paquetes subsecuentes al paquete perdido.

■ Comportamiento del receptor:

- Receptor envía **ack acumulativo**
 - mayor número de secuencia tal que todos los segmentos anteriores se recibieron bien.
- Asumir que el receptor recibió un paquete n .
- Si n está en orden (todos los paquetes anteriores llegaron) y está correcto (sin errores):
 - manda ack para n y entrega parte de datos de paquete n a capa superior.
- Sino:
 - el receptor descarta el paquete n y manda ACK del paquete más reciente recibido en orden.

Comportamiento del emisor:

- El emisor tiene un solo temporizador para el paquete más viejo no confirmado.
- Al expirar el temporizador (del segmento más viejo no confirmado),
 - retransmite todos los segmentos no confirmados.
- Si llega ACK nuevo y hay segmentos enviados no confirmados,
 - el temporizador es reiniciado.
- Si llega ACK nuevo y no hay segmentos sin confirmar,
 - el temporizador es detenido.
- **ventana emisora** = tramas enviadas sin ack positivo o tramas listas para ser enviadas.



- **timeout(n):** retransmite paquete n y todos los paquetes de mayor N° de secuencia en la ventana.

La ventana emisora no puede superar MAX_SEQ cuando $n \geq MAX_SEQ + 1$ número de secuencia.

Retroceso-N: No se retransmite el segmento perdido o demorado.

problema:
nunca un timeout
se lo que
pasa es que
en el buffer.
primero han
nada que
sin confirmar.

Repetición - Selectiva

- los paquetes en buen orden dopo de un paquete perdido se reenvian en buen orden cuando llega el ACK, se manejan todos en orden a la capa de aplicación.

- El receptor confirma individualmente todos los paquetes recibidos correctamente.

- Hay búferes para paquetes según se necesiten para su entrega eventual en orden a la capa de aplicación.

- El emisor solo reenvía paquetes para los cuales el ACK no fue recibido o se recibió un NAK.

- Hay un temporizador del emisor para cada paquete no confirmado.

- **Ventana del emisor**

- Contiene N N° de secuencias consecutivos
 - Limita N° de secuencias a enviar a paquetes no confirmados.

- **¿Qué tipos de paquetes puede haber en la ventana del emisor? (ayuda considerar que estamos en repetición selectiva)**

- Como se confirman todos los paquetes que llegan y puede haber paquetes perdidos:

- Paquetes enviados y confirmados porque antes hay paquetes no confirmados
 - Paquetes enviados y no confirmados
 - Paquetes listos para enviarse en búfer

Como el receptor no descarta los que no llegan en orden, debe guardarlos en un búfer los paquetes hasta que lleguen todos y se procederán tanto paquetes a la capa de aplicación.

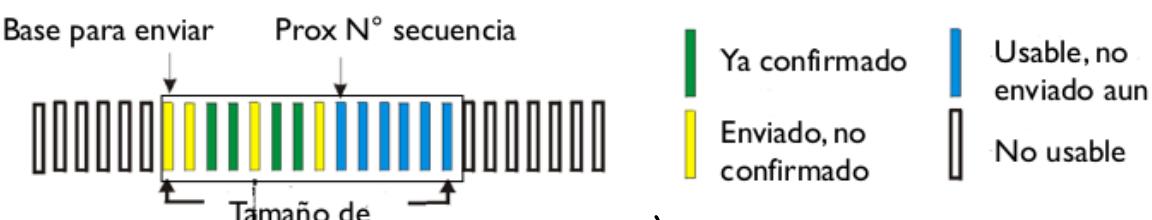
Receptor tiene una **VENTANA COBERTA**, en un intervalo dentro de tiempo en espacio de N° de sec.

Tipos de paquetes que puede haber en la ventana del receptor:

- Paquetes esperados y no recibidos
- Paquetes recibidos fuera de orden
- Paquetes aceptables en la ventana que no han llegado aun

Se mantiene en búfer un paquete aceptado por la ventana receptora

- hasta que todos los que le preceden hayan sido pasados a la capa de aplicación.



(a) Visión de números de secuencia del emisor



(b) Visión de números de secuencia del receptor

Cuando llega un paquete se pregunta si pertenece dentro de la ventana cuando se envía paquete → la top de opp, la ventana avanza, por lo que si llega paquete esperado avanzar, si es ignorar, pq caer fuera de la ventana.

Emisor

Datos vienen de arriba:

- Si el próximo N° secuencia a enviar de la ventana está disponible, almacenar y enviar paquete

timeout(n):

- Reenviar paquete n , reiniciar timer

ACK(n) en [sendbase, sendbase+N]:

- marcar paquete n como recibido
- Si n es paquete más pequeño no confirmado, **avanzar base de ventana** al siguiente N° secuencia no confirmado.

Receptor

pkt n en [base rcv, base rcv +N-1]

- Enviar ACK(n)
- Fuera de orden: almacenarlo
- En orden: entregar (también entregar paquetes en bufer en orden), avanzar ventana al siguiente paquete que no ha sido recibido aun.

pkt n en [base rcv-N, base rcv-1]

- Enviar ACK(n)

Sino:

- ignorar

Tamaño ventana

Receptor:

(MAX_SEQ -1) / 2

Súposiciones: N es tamaño de ventana del receptor. Algoritmo sin NAKs

Para hacer más efectivo este protocolo, los ACK's se mandan a "colisión" a otro paquete de datos, para así no mandar muchos paquetes SOLO con ACK's. De una forma se puede tener un buen intercambio de datos bidireccionales.

La CT para mandar un ACK, debe esperar por un paquete de con superponer un ACK

• Solución: método que usa temporizador auxiliar

- tras llegar un paquete de datos en secuencia, se arranca un temporizador auxiliar mediante *start_ack_timer*.
- Si no se ha presentado tráfico de regreso antes de que termine este temporizador, se envía un paquete de ack independiente.

temp-dur <<< temp-fctnmoner

Desequilibrio de protocolos de entrega de datos confiable

$$- D_{envio} = \text{demora en enviar paquete} = \frac{L}{R} \rightarrow \begin{array}{l} \text{tamaño del} \\ \text{paquete} \\ R \rightarrow \text{vel. transmision} \end{array}$$

- $U_{redes} = \text{utilización de la linea - fracción de tiempo en que el enlace estuvo ocupado en todo:} \Rightarrow \frac{D_{envio}}{RTT + D_{envio}}$

- RTT = tiempo ida y vuelta de un bit.

S: se envían muchos paquetes seguidos (pipeline) en la utilización se multiplica por la const de paquetes enviados.

Control de flujo

- evitar que un receptor rápido desborde un receptor lento.

• Tipo de control de flujo del que se ocupa la capa de enlace de datos:

- Control de flujo entre dos máquinas directamente conectadas entre sí (pueden ser enrutador o host).

• ¿Por qué puede necesitarse control de flujo en la capa de transporte si la capa de enlace de datos lo hace?

• El receptor puede demorarse en procesar mensajes debido a los problemas de la red:

- pérdida de segmentos,
- no se pueden procesar segmentos porque faltan anteriores.

- Podemos asumir que el receptor maneja búferes para los mensajes que llegan.

- **Esto es necesario porque:**

- Si la llegada de segmentos del emisor es mucho más rápido que el receptor para procesar los segmentos recibidos,
 - entonces el receptor necesitará poder almacenar segmentos antes de procesarlos.
- El receptor puede acumular una cantidad de segmentos suficientes antes de pasarlo a la capa de aplicación para que los procese.
- Los segmentos pueden llegar desordenados;
 - por lo tanto si llegan un grupo de segmentos y faltan segmentos previos a ellos, habrá que almacenarlos segmentos de ese grupo en buffer.

Problema: ¿Qué hace el receptor con los búferes si tiene varias conexiones?

Solución 1: se usan los búferes a medida que llegan segmentos.

Solución 2: se dedican conjuntos de búferes específicos a conexiones específicas.

Solución 1:

- Cuando entra un segmento el receptor intenta adquirir un búfer nuevo;
- si hay uno disponible, se acepta el segmento; de otro modo se lo descarta.

Suposición: cambia el patrón de tráfico de la red; se abren y cierran varias conexiones en el receptor.

Consecuencias:

- El receptor y el emisor deben ajustar dinámicamente sus alojamientos de búferes.
 - Esto significa ventanas de tamaños variables.
- Ahora el emisor no sabe cuántos datos puede mandar en un momento dado, pero sí sabe cuántos datos le gustaría mandar.

Solución 2:

Solución: El host emisor **solicita espacio en búfer en el otro extremo.**

- Para estar seguro de no enviar de más y sobrecargar al receptor.
- Porque sabe cuánto necesita.

➤ **Cuando el receptor recibe este pedido:**

- Sabe cuál es su situación y cuánto espacio puede otorgar.
- Aquí el receptor reserva una cierta cantidad de búferes al emisor.

➤ Los búferes podrían repartirse por conexión, o no.

➤ **Si los búferes se reparten por conexión y aumenta la cantidad de conexiones abiertas:**

- El receptor necesita ajustar dinámicamente sus reservas de búferes.

1. Inicialmente el emisor solicita una cierta cantidad de búferes, con base en sus necesidades percibidas.

2. El receptor otorga entonces tantos búferes como puede.

3. El receptor, sabiendo su capacidad de manejo de búferes podría indicar al emisor "**te he reservado X búferes**".

• **¿Cómo hace el receptor con las confirmaciones de recepción?**

• **El receptor puede incorporar tanto las ack como las reservas de búfer al en el mismo segmento.**

– El emisor lleva la cuenta de su **asignación de búferes** con el receptor.

– **Cada vez que el emisor envía un segmento:**

- Debe disminuir su asignación de búferes disponibles.

– **Cuando la asignación de búferes (disponibles) llega a 0:**

- El emisor debe detenerse por completo

para evitar situaciones donde la información de reserva de búferes se pierda, y se devuelva un ACKLOST, los paquetes con una info se devuelven contontamente.

Control de flujo en TCP

No se requiere:

- que los emisores envíen datos tan pronto como llegan de la aplicación.
- que los receptores envíen confirmaciones de recepción tan pronto como sea posible.
- que los receptores entreguen datos a la aplicación apenas los reciben.
 - Esta libertad puede explotarse para mejorar el desempeño.

No se puede usar el protocolo de control de flujo anterior para TCP.

- Porque en TCP los números de secuencia no significan número de paquete.
- Antes cada búfer ocupado tenía un número de paquete.
- Ahora los números de secuencia son posiciones en el flujo de datos a enviar.
- El receptor a lo más puede saber qué rangos de números de secuencia de bytes recibidos tiene en búfer.

Algunas mejoras que se pueden hacer en relación al protocolo anterior:

- Los encabezados de los segmentos recibidos ocupan espacio y no hace falta almacenarlos en búfer.
 - En su lugar se pueden almacenar datos recibidos del flujo de datos.
- No es necesario que el emisor solicite espacio de búfer al receptor.
 - El receptor sabe de cuánto espacio dispone y cuánto espacio puede otorgar.

Para esto se usa un búfer de recepción
circular en el receptor (guarda solo datos)

Como TCP usa un búfer circular único, el receptor no le puede decir al emisor: 'te he reservado x búferes'.

Para anunciar al emisor la reserva de espacio en búfer:

- El receptor puede indicar al emisor la cantidad de bytes consecutivos que se pueden enviar; comenzando por el byte cuya recepción se ha confirmado.
- A esto se le llama en TCP **tamaño de ventana**.
- En el encabezado TCP un **campo de tamaño de ventana** (de 16 bits) se usa para indicar esta información.

- El emisor también tiene un búfer circular para los datos a enviar.
- Los bytes que pueden enviar dependen de

- = el tamaño del búfer del receptor > 12 ventana
- = cant. de bytes NO en mayor que ninguno < 12 mayores anteriores.

La fórmula para calcular el tamaño de ventana el receptor es:

$$\text{Tamaño de ventana} = \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$$

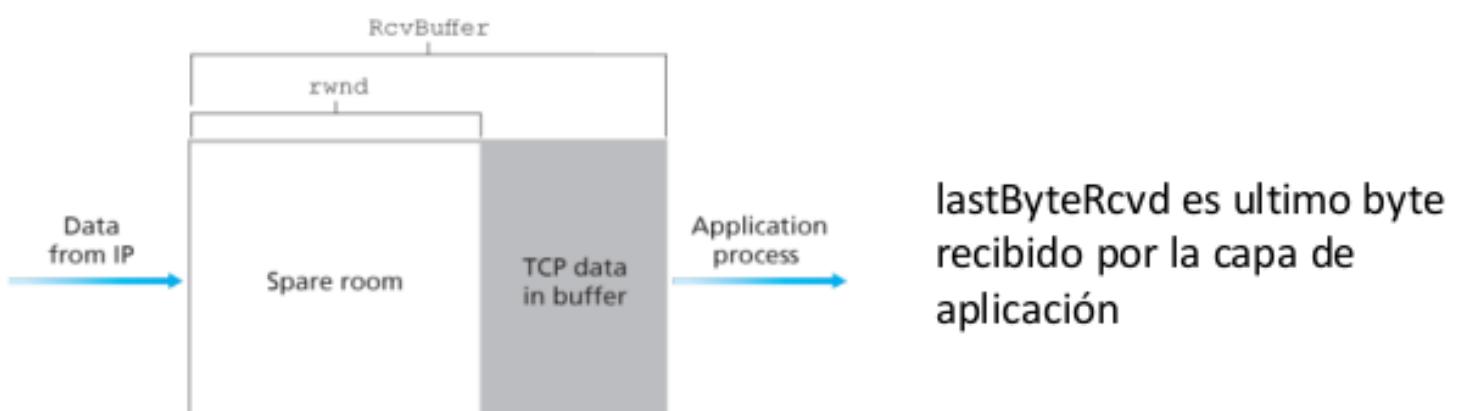


Figure 3.38 The receive window (*rwnd*) and the receive buffer (*RcvBuffer*)

El receptor:

- Cuando la conexión TCP recibe bytes en el orden correcto y en secuencia, coloca los datos en el buffer de recepción.
- El receptor puede confirmar llegada de datos nuevos y anunciar nuevo tamaño de ventana al emisor.
- Si búfer de recepción está lleno, avisar tamaño de ventana de cero.
- Una vez que el receptor entrega a la capa de aplicación X datos de búfer de recepción lleno, puede avisar al emisor de un tamaño de ventana de X.

El emisor:

- Si el tamaño de ventana anunciado es cero el emisor no podrá enviar datos.
- El emisor envía segmentos cumpliendo la siguiente propiedad:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{tamaño de ventana}.$$

Pérdida de Segmentos en TCP

Solución 1: el receptor solicita segmento/s específico/s mediante segmento especial llamado NAK.

- Tras recibir segmento/s faltante/s, el receptor puede enviar una confirmación de recepción de todos los datos que tiene en búfer.
- Cuando el receptor nota una brecha entre el número de secuencia esperado y el número de secuencia del paquete recibido, el receptor envía un NAK en un campo de opciones.

Solución 2: (acks selectivos) el receptor le dice al emisor que piezas recibió.

- El emisor puede así reenviar los datos no confirmados que ya envió.
- Se usan dos campos de opciones:
 - **Sack permitted option:** se envía en segmento SYN para indicar que se usarán acks selectivos.
 - **Sack option:** Con lista de rangos de números de secuencia recibidos.

Cuando la ventana es de 0, el emisor no puede enviar segmentos, salvo en dos situaciones:

1. pueden enviarse **datos urgentes** (p.ej. Para que el usuario elimine el proceso en ejecución en la máquina remota),
2. el emisor puede enviar un segmento de 1 B para hacer que el receptor **re-anuncie** el siguiente byte esperado y el tamaño de la ventana.
 - TCP proporciona esta opción para evitar un bloqueo irreversible si llega a perderse un anuncio de ventana.

Solución (opción de escala de ventana): permitir al emisor y al receptor negociar un factor de escala de ventana.

- Ambos lados pueden desplazar el tamaño del campo de ventana hasta 14 bits a la izquierda,
- permitiendo por lo tanto ventanas de hasta 2^{30} bytes.
- La mayoría de las implementaciones actuales de TCP manejan esta opción.

Control de Congestión

S: un emisor manda más de lo que la red puede soportar se congestiona. Este control de congestión se aplica al emisor.

Para controlar la congestión:

- En TCP algunos hosts disminuirán la tasa de datos.

Para llevar la cuenta de cuántos datos un host puede enviar por la red:

- TCP maneja una **ventana para la congestión (VC)** - cuyo tamaño es el número de bytes que el emisor puede tener en la red en un momento dado.

En TCP el host tiene una forma de **detectar congestión**.

En TCP cuando un host detecta congestión:

- El host ajusta el tamaño de la VC.

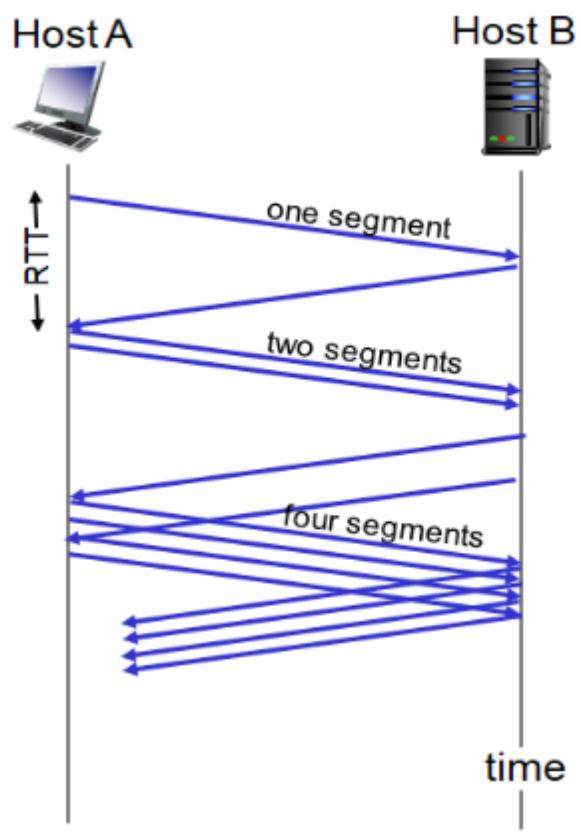
EXPLICACIÓN de temporizadores.

TCP assume que las expiraciones son por congestión.

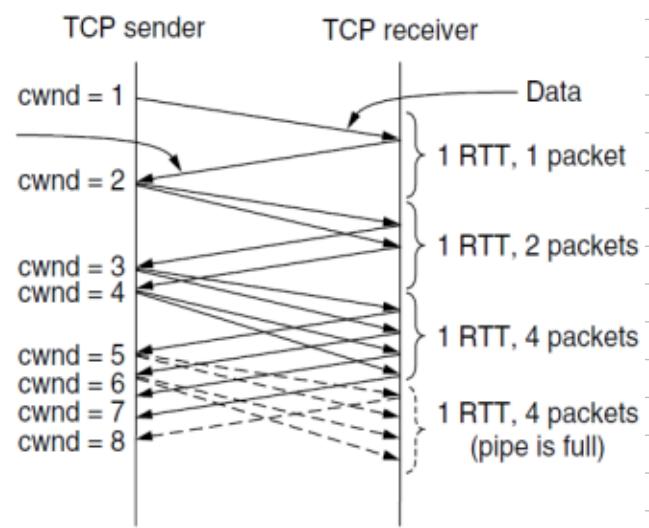
Como calcular la ventana de congestión?

Algoritmo de arranque lento (Jacobson 1988).

- El emisor asigna a la VC el segmento de tamaño máximo (STM) usado por la conexión; entonces envía 1 STM.
 - Emisor y receptor se ponen de acuerdo en el tamaño del STM.
- Si se recibe el ack de este segmento antes que expire el temporizador, el emisor agrega el equivalente en bytes de un segmento a la VC para hacerla de 2 STM y envía dos segmentos.



- Cuando la VC es de n segmentos, si de todos los n se reciben acks a tiempo, se aumenta la VC en la cuenta de bytes correspondiente a n segmentos.
- La VC sigue creciendo exponencialmente hasta expiración temporizador (timeout) o **alcanzar el tamaño de la ventana receptora**.
- Si ocurre timeout se recorta la VC a tamaño $VC/2$, o sea no se enviarán ráfagas de segmentos mayores a $VC/2$.
- Esta es la solución de la edición ante penúltima del libro de Tanembaum.



→ Se duplica
capacidad de la red
al hacer esto.

→ Esperar a ese timeout
puede resultar en esperar largo.

¿Cómo reconoces las perdidas más rápidamente?

Control en TCP.

– Cuando se pierde un segmento y otros segmentos luego del segmento perdido llegan al receptor:

- El receptor genera acks que confirman lo mismo (i.e. siguiente byte esperado).
- Se llaman **acks duplicados**.

– ¿Qué significa que el emisor recibió un ack duplicado?

- Es probable que llegó otro segmento al receptor y el segmento perdido no dio señales de vida.

TCP presume que
3 ACK's duplicados
significo que el
paquete se perdió.

Luego se puede retransmitir inmediatamente y
antes de que expire su temporizador.

Algoritmo TCP Tahoe:

- Usa un **umbral** además de las ventanas de recepción y congestión.
- Al ocurrir una expiración del temporizador o detectarse 3 acks duplicados, se fija el umbral en la mitad de la ventana de congestión actual, y la ventana de congestión se restablece a un segmento máximo.
- Luego se usa el **arranque lento** para determinar lo que puede manejar la red, excepto que el crecimiento exponencial termina al alcanzar el umbral.
- A partir del punto en el que se alcanza el umbral las transmisiones exitosas aumentan linealmente la ventana de congestión (en un segmento máximo por ráfaga).
- Recomenzar con una ventana de congestión de un paquete toma un RTT (para todos los datos previamente transmitidos que dejen la red y para ser confirmados, incluyendo el paquete retransmitido).
- Si no ocurren más expiraciones de temporizador/3 acks duplicados, la ventana de congestión continuará creciendo hasta el tamaño de la ventana del receptor.
 - En ese punto dejará de crecer y permanecerá constante mientras no ocurran más expiraciones de temporizador y la ventana del receptor no cambie de tamaño.

Comenzando con arranque lento cada vez que se pierde un paquete es muy lento.

Solución: Algoritmo de TCP Reno (1990).

- **Idea:** Evitar arranque lento (excepto cuando la conexión es comenzada) cuando expira el temporizador de re-envíos.
- **Funcionamiento:**
 1. Luego de iniciada la conexión se comienza con arranque lento.
 2. A continuación la ventana de congestión crece linealmente hasta que se detecta una pérdida de paquete.
 - Se cuentan acks duplicados
 - Se considera pérdida de paquete 3 acks duplicados

3. El paquete perdido es retransmitido (usando retransmisión rápida).

4. Recuperación rápida:

- Se manda un paquete por cada ack duplicado recibido.
- Un RTT luego de la retransmisión rápida el paquete perdido es confirmado.
- La recuperación rápida termina con esa confirmación de recepción.

5. Luego de recibir el nuevo ack:

- la ventana de congestión de una conexión se achica a la mitad de lo que era cuando se encontraron 3 duplicados (**decrecimiento multiplicativo**).
- El conteo de ack duplicados se pone en 0.

6. Luego la ventana de congestión va incrementando de a un segmento por cada RTT (**crecimiento aditivo**).

7. Este comportamiento continua indefinidamente.

Segmentos duplicados.

- Si se pierde un ACK y se retransmite el segmento - por congestión en red se demora y se envía por temporizadores.
- para ello se componen los segmentos a ver si ya entran en el receptor o no
- n^o se devuelven en caso segmento

hay un problema h^o nec pvcde ser reutilizables pq en finito y tiene un largo determinado.

los duplicados que dan vueltas por ahí llegan hasta al receptor.

Idea: Asegurar que ningún paquete viva más allá de T sec. (**tiempo de vida de paquete**)

- Esto se refiere a paquetes de datos, retransmisiones de ellos y a confirmaciones de recepción.
- Eliminar paquetes viejos que andan dando vueltas por ahí.

Sed + el tiempo que vive un segmento, de crearse un segmento viejo en un n° de secuencia no se vaya a usar dentro de T segundos.

Para lograr que al regresar al principio de los n° de secuencia, los segmentos viejos con el mismo n° de secuencia hayan desaparecido hace mucho tiempo:

El espacio de secuencia debe ser lo suficientemente grande para garantizar esto.

Ese espacio se calcula según el tamaño de los segmentos, T y la velocidad de la red.

peño, la conexión se immobiliza usando segmentos que no llevan los primeros segmentos.

Idea de solución: hay que escoger como número inicial de secuencia de la conexión nueva un n° de secuencia que haga imposible o improbable que el duplicado retrasado de n° de secuencia genere problemas.

- Además se mantiene dentro de una conexión que el origen etiqueta los segmentos con n° de secuencia que no van a reutilizarse dentro de T sec (tiempo de vida del paquete).

- **Implementación 1** (en libro de Comer): al crear una nueva conexión cada extremo genera un n° de secuencia de 32 bits aleatorio que pasa a ser el número inicial de secuencia para los datos enviados.

- Alguna implementación de TCP usa esta solución.

- **¿Por qué tiende a funcionar?**

- La probabilidad de que un paquete duplicado retrasado genere problemas en una conexión siguiente es baja debido a la elección aleatoria del número inicial de secuencia de la conexión siguiente.

- **Implementación 2** (en libro de Tanembaum): vincular n° de secuencia de algún modo al tiempo y para medir el tiempo usar un reloj.
 - Cada host tiene un **reloj de hora del día**.
 - Los relojes de los hosts no necesitan ser sincronizados;
 - se supone que cada reloj es un contador binario que se incrementa a si mismo en intervalos uniformes.
 - **El reloj continua operando aun ante la caída del host**
- Cuando se establece una conexión los k bits de orden mayor del reloj = **número inicial de secuencia**.

Establecimiento de conexiones

- **Problema:** Cuando un host se cae, al reactivarse sus ET no saben dónde estaban en el espacio de secuencia.
- Este es un problema porque para el siguiente segmento a enviar no se sabe qué números de secuencia generar;
 - si se genera mal, entonces el nuevo segmento podría tener el mismo número de secuencia que otro segmento distinto circulando por la red.
- **Solución:** requerir que las ET estén inactivas durante T segundos tras una recuperación para permitir que todos los segmentos viejos expiren (entonces no vamos a tener dos segmentos diferentes con el mismo número de secuencia).

Como establecer conexión entre 2 hosts
 EMISO ENVIÓ $S = CR\ N, P$ A Aceptador Y
 Receptor CONFIRMO CON m CA N
 N = numero de rec.

Caso: S se demora demasiado en llegar a host 2, vence timer en host 1 y host 1 manda un duplicado $S' = CR, N, P$ al host 2.

- Luego puede pasar que host 2 reciba S' y un buen tiempo después S .

Situación: No se recuerda en el destino n° de secuencias para conexiones.

Problema: No tenemos forma de saber si un segmento CR conteniendo un n° de secuencia inicial es un duplicado de una conexión reciente o una conexión nueva.

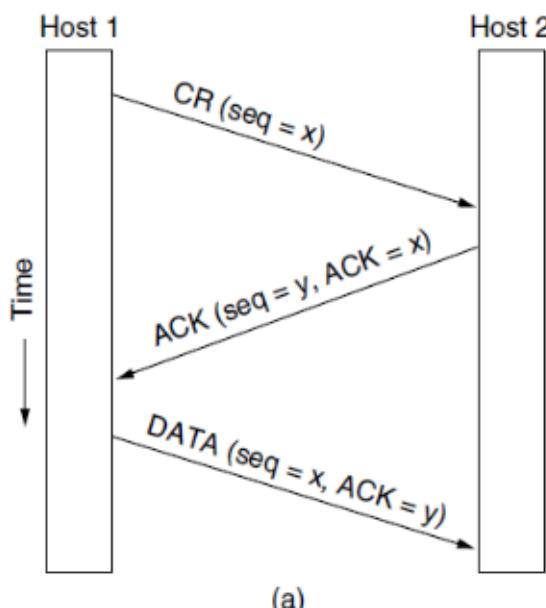
- No sabe si mandar un segmento CA o no.

Solución: acuerdo de 3 vías.

Solución: Acuerdo de tres vías

de Tomlinson de 1975.

- Caso de operación Normal
- **Fijarse** en el número de secuencia del segmento de datos enviado.
- ¿Cómo sería el caso que llega un segmento CR duplicado al host 2?



(a)

en caso de aparecer duplicados, los hosts utilizan la N de rec pgo para dar cuenta que no duplicados.

en TCP

El nº de secuencia inicial de una conexión **no es 0**.

- Se usa un **esquema basado en reloj** con un pulso de reloj cada **4 µsec**.
- Al caerse un host, no podrá reiniciarse durante el **tiempo máximo de paquete** (120 seg),
- para asegurar que no haya paquetes de conexiones previas vagando por Internet.

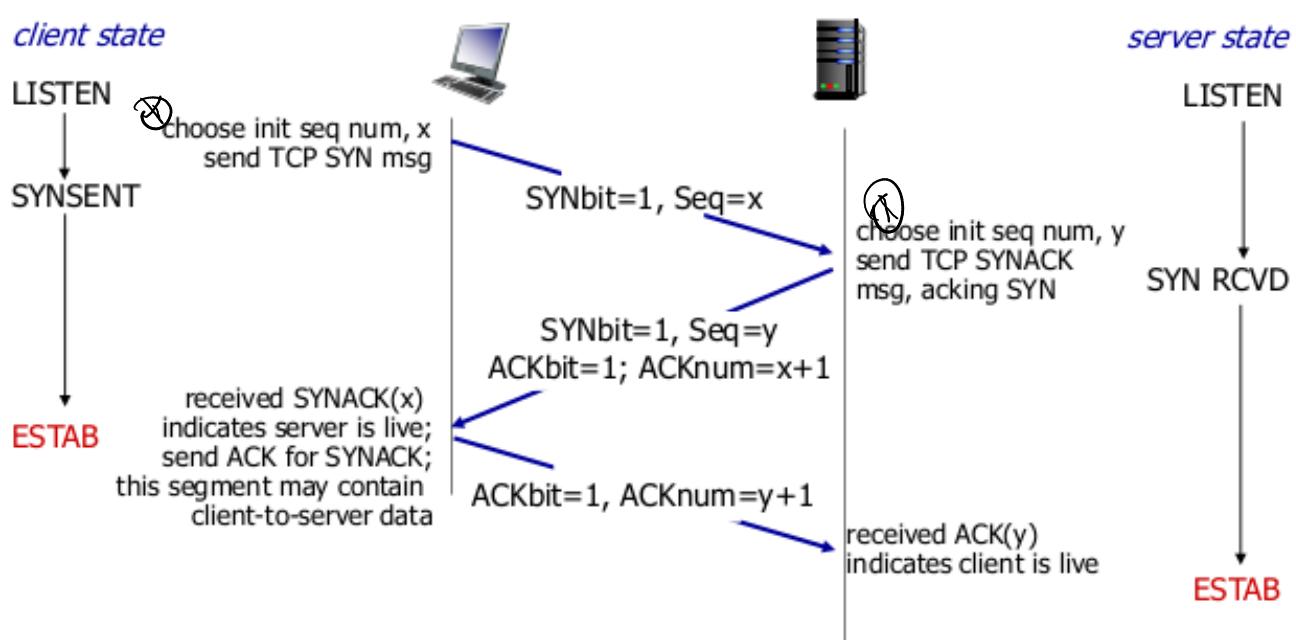
Campos del encabezado TCP para el establecimiento de conexiones

SYN se usa para establecer conexiones.

- Solicitud de conexión: $SYN = 1$ y $ACK = 0$.
- La respuesta de conexión sí lleva una confirmación de recepción, por lo que tiene $SYN = 1$ y $ACK = 1$.
 - Recordar que además hay campo con N° de secuencia confirmado.

En TCP las conexiones usan el **acuerdo de 3 vías**

1. Para establecer una conexión, el servidor, espera pasivamente una conexión entrante ejecutando LISTEN y ACCEPT y
 - especificando cierto origen o bien nadie en particular.
2. En el lado del cliente ejecuta CONNECT
 - la cual envía un segmento TCP con el **bit SYN encendido y el bit ACK apagado**, y espera una respuesta.
3. Al llegar el segmento al destino, la ETCP allí revisa si **hay un proceso** que haya ejecutado un LISTEN en el puerto indicado en el campo puerto de destino.
4. Si no lo hay envía una respuesta con el **bit RST encendido para rechazar la conexión**.
5. Si algún proceso está escuchando en el puerto ese proceso recibe el segmento TCP entrante y puede entonces aceptar o rechazar la conexión; si la acepta se envía un segmento de ack.
6. La secuencia de segmentos TCP enviados en el caso normal se muestra en la Figura siguiente.

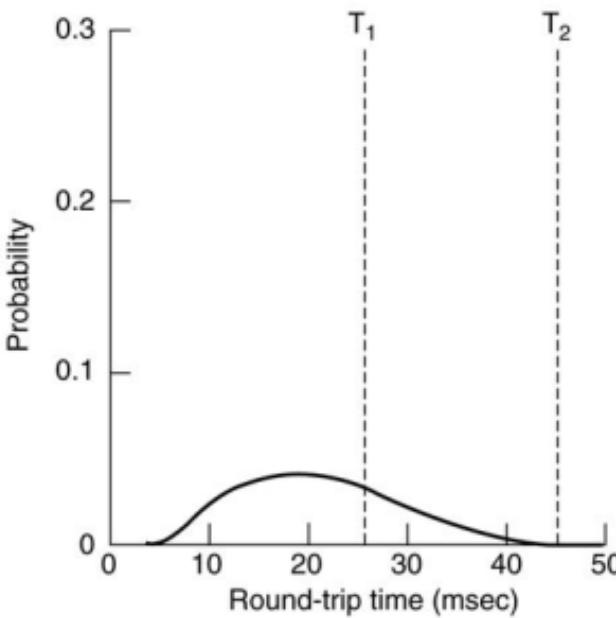


En este momento el nº elegido por el cliente es 2¹³² y luego cuando se manda el ACK, se pone el nº de respuesta 0.

Temporizadores de TCP

- **Problema:** ¿qué tan grande debe ser el intervalo de expiración del temporizador de retransmisión?

- Si se hace demasiado corto - digamos T_1 en la Figura:
- Ocurrirán retransmisiones innecesarias.
- Si se hace demasiado largo? - digamos T_2 :
- Sufrirá el desempeño por el gran retardo de retransmisión de cada paquete perdido



- **Situación:**

- La función de densidad de probabilidad del tiempo que tarda en regresar un ack TCP se parece a la Fig. anterior.
- La varianza y la media de la distribución de llegada de las ack pueden variar a medida que se generan y se resuelven congestionamientos.
- **Idea:** Ajustar constantemente el intervalo de expiración del temporizador, con base en mediciones continuas del desempeño de la red.

Solución: Algoritmo de Jacobson (1988) usado por TCP

- Por cada conexión el TCP mantiene una variable, **RTT (round trip time)**,
 - significa estimación actual del tiempo de ida y vuelta al destino.
- Al enviarse un segmento se inicia un temporizador,
 - para saber el tiempo que tarda el ack,
 - y para habilitar una retransmisión si se tarda demasiado.
- Si llega el ack antes de expiration el temporizador:
 - TCP mide el tiempo que tardó el ack , digamos M ,
 - entonces actualiza el RTT así:
$$RTT = \alpha RTT + (1-\alpha) M,$$
 - α es el peso que se le da al valor anterior. Por lo común $\alpha = 7/8$.

Un RTT inicial de 1 sec se aconseja en RFC 6298.

Problema: Dado RTT, hay que elegir una **expiración adecuada** del temporizador de retransmisión.

Solución 1: En las implementaciones iniciales:

$$\text{Expiración del temporizador} = 2 \times \text{RTT}.$$

Evaluación: Este valor es inflexible, pues falla en responder a la suba de la varianza de la función de densidad de probabilidad del tiempo de llegada de los ack.

Solución 2: (Jacobson 1988) hacer que el valor de timeout sea sensible tanto a la variación de RTT como a la varianza de la función de densidad de probabilidad del tiempo de llegada de los ack.

- Se mantiene una variable amortiguada D (**la desviación media**).
- Al llegar un ack, se calcula $|\text{RTT} - M|$.
- Se mantiene en D mediante:

$$D = \beta D + (1 - \beta) |\text{RTT} - M|,$$

- donde β típicamente es $\frac{3}{4}$.
- D es una aproximación bastante cercana a la desviación estándar.

¿Cómo estimar la expiración del temporizador? ¿De qué parámetros depende?

La mayoría de TCP usan
expiración del temporizadores: $\text{RTT} + 4D$
no es necesario calcularlo de nuevo si llega un ACK.

Solución: (algoritmo de Karn)

- No actualizar el RTT (cuando llega ack) de ninguno de los segmentos retransmitidos.
- Cuando ocurre un timeout se **duplica la expiración del temporizador**.
- Tan pronto se recibe un ack de segmento no retransmitido, el RTT estimado es actualizado y la expiración del temporizador se computa nuevamente usando la fórmula anterior.

El algoritmo de Karn lo usan la mayoría de las implementaciones TCP.

UDP

UDP (protocolo de datagramas de usuario)

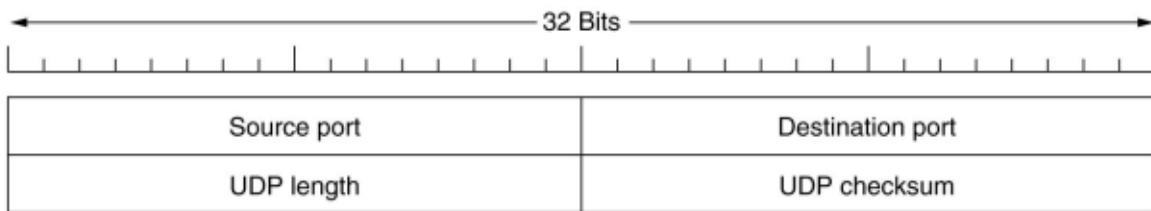
- Es no orientado a la conexión.

segmentos = encabezado de 8 B + carga útil.

- 2 puertos de 16b.

- El campo **longitud UDP** incluye el encabezado de 8 bytes y los datos.

The UDP header.



NO tiene:

- Control de flujo, congestión, retransmisión.

Util en situación cliente servicio.

- Si recibe la notificación de error, se vuelve a intentar.