Capa de Transporte – Resumen

PDU ⇨ TCP Segmentos UDP Datagramas Protocol Data Unit

En IP los paquetes pueden ser descartados, des-ordenados, retardados duplicados o corrompidos. En estos también solo se tienen las direcciones IP destino y fuente. IP corre en todos los nodos de la red (routers y host), protocolos de transporte solo necesario en end-points (hosts).

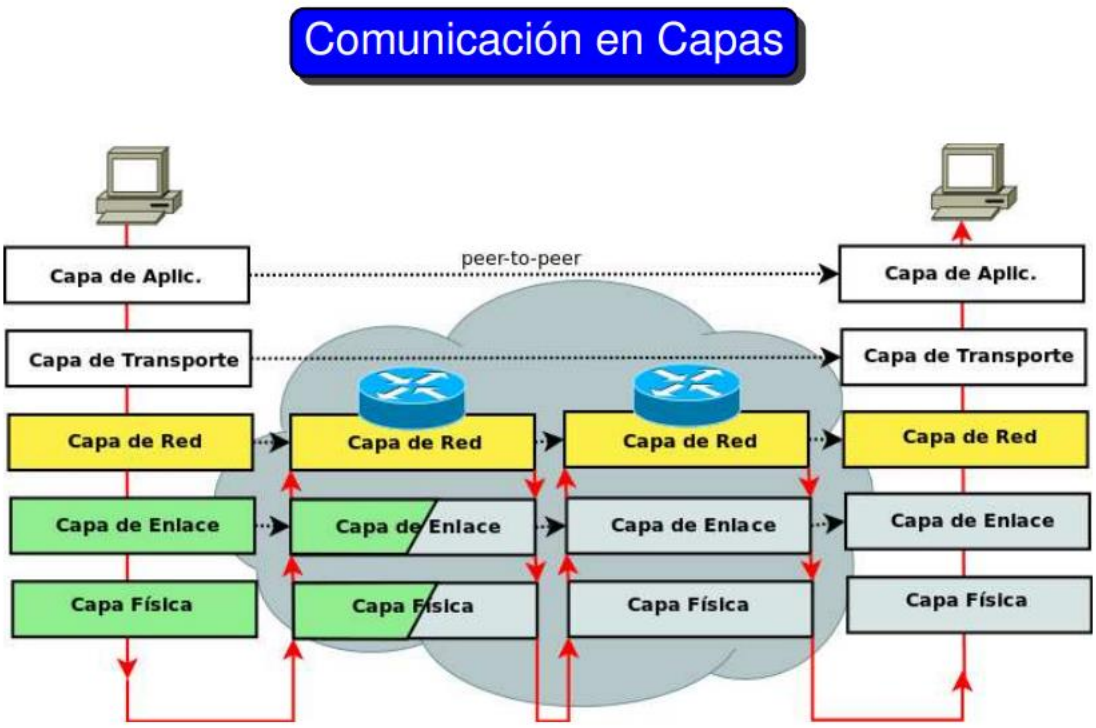
IP es el protocolo de transporte de la capa de transporte.

En IP es puerta a puerta (host-to-host) y en transporte es de proceso a proceso .

Básicamente, la función principal de la capa de transporte es proporcionar una comunicación lógica (desde la perspectiva de la aplicación, es como si los hosts que ejecutan los procesos estuvieran conectados directamente) entre procesos de aplicación que se ejecutan en dispositivos diferentes dentro de una red. Esta comunicación lógica permite que los procesos de aplicación se envíen mensajes entre sí sin preocuparse por los detalles de la infraestructura física subyacente.

Los protocolos de la capa de transporte se implementan en los sistemas terminales (hosts), y no en los routers de la red. En el lado emisor, la capa de transporte convierte los mensajes de la aplicación en segmentos de la capa de transporte, que luego se encapsulan en paquetes de la capa de red y se envían al destino. Los routers de la red solo actúan sobre los campos correspondientes a la capa de red del paquete, sin examinar los campos del segmento de la capa de transporte encapsulado.

En el lado receptor, la capa de transporte extrae el segmento de la capa de transporte del paquete y lo entrega a la aplicación receptora. Para las aplicaciones de red, existen varios protocolos de la capa de transporte disponibles, como TCP y UDP, cada uno ofreciendo un conjunto diferente de servicios a las aplicaciones que los utilizan.



Funciones

- Multiplexar y Demultiplexar (Ambos).
- Soporte de datos de tamaños arbitrarios (Ambos).
- Control y Detección de Errores, perdida, duplicación, corrupción (Mayormente TCP).
- Control de Flujo (TCP).
- Control de Congestión (TCP).

Puerto

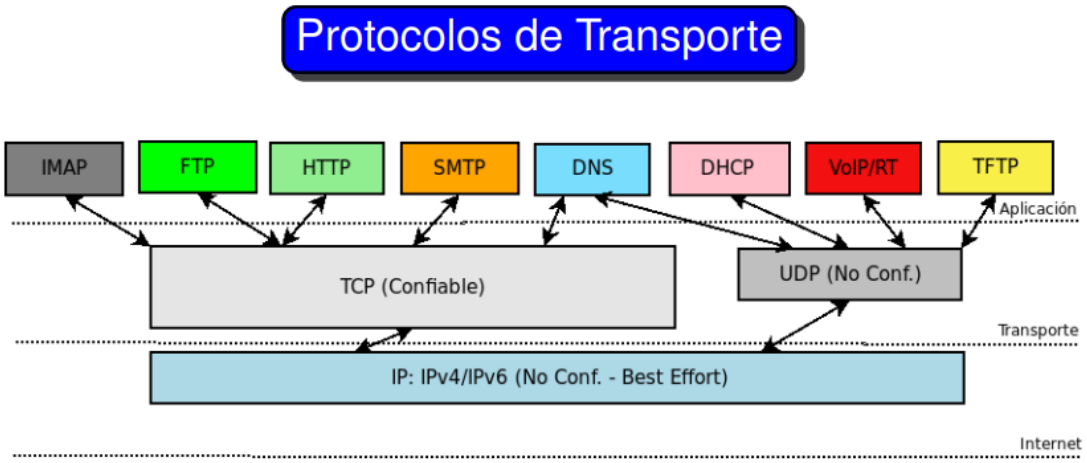
Se utilizan para distinguir las aplicaciones (y, por lo tanto, protocolos) que están enviando/recibiendo datos. Los puertos actúan como puntos finales en una comunicación y permiten que múltiples aplicaciones en una misma computadora o dispositivo se comuniquen simultáneamente a través de la red.

Modelo confiable vs Modelo no confiable

Modelo Confiable: TCP – Se encarga de ordenar si llegan los paquetes desordenados, reenviar si hay errores, etc.

Modelo NO Confiable: UDP – Solo se encarga de Multiplexar y Demultiplexar, si hay algún error no notifica. “Best Effort de IP`”

Están identificados por códigos (le sirve a IP para saber a que modelo sirve la información) UPD es 17 y TCP es 6.



Socket

La aplicación de acuerdo a como está programada selecciona el transporte.

El acceso a los servicios de transporte se hace mediante API: Network socket.

Un proceso puede tener uno o varios sockets, por tanto la capa de transporte entrega los datos al socket (y no directamente a la aplicación), para identificar los sockets, éstos tienen un identificador único. Cada segmento de la capa de transporte contiene un campo para poder entregar los datos al socket adecuado. En el receptor, la capa de transporte examina estos campos para identificar el socket receptor y lo envía (demultiplexación).

Denominamos multiplexación al trabajo de reunir los datos en el host origen desde diferentes sockets, encapsulando los fragmentos de datos con la información de cabecera (que se usará en la demultiplexación).

(Otra definición por las dudis)

- Multiplexación: Reúne los paquetes de datos de los diferentes sockets en el host de origen, encapsulando los encabezados de información (que será

después utilizada para desmultiplicarlos) para crear segmentos y pasar estos segmentos a la capa de red.

- Demultiplexación: En el extremo receptor, la capa de transporte examinas los campos para identificar el socket receptor, y entonces dirigir el segmento a ese socket.

  El socket TCP queda identificado por una tupla de cuatro elementos: dirección IP de origen, número de puerto de origen, dirección IP de destino, número de puerto de destino. Por lo tanto, cuando un segmento TCP llega a un host procedente de la red, el host emplea los cuatro valores para dirigir (demultiplexar) el segmento al socket apropiado.

  El socket UDP queda completamente identificado por una tupla que consta de una dirección IP de destino y un número de puerto de destino. En consecuencia, si dos segmentos UDP tienen diferentes direcciones IP y/o números de puerto de origen, pero la misma dirección IP de destino y el mismo número puerto de destino, entonces los dos segmentos se enviarán al mismo proceso de destino a través del mismo socket de destino.

UDP – User Datagram Protocol

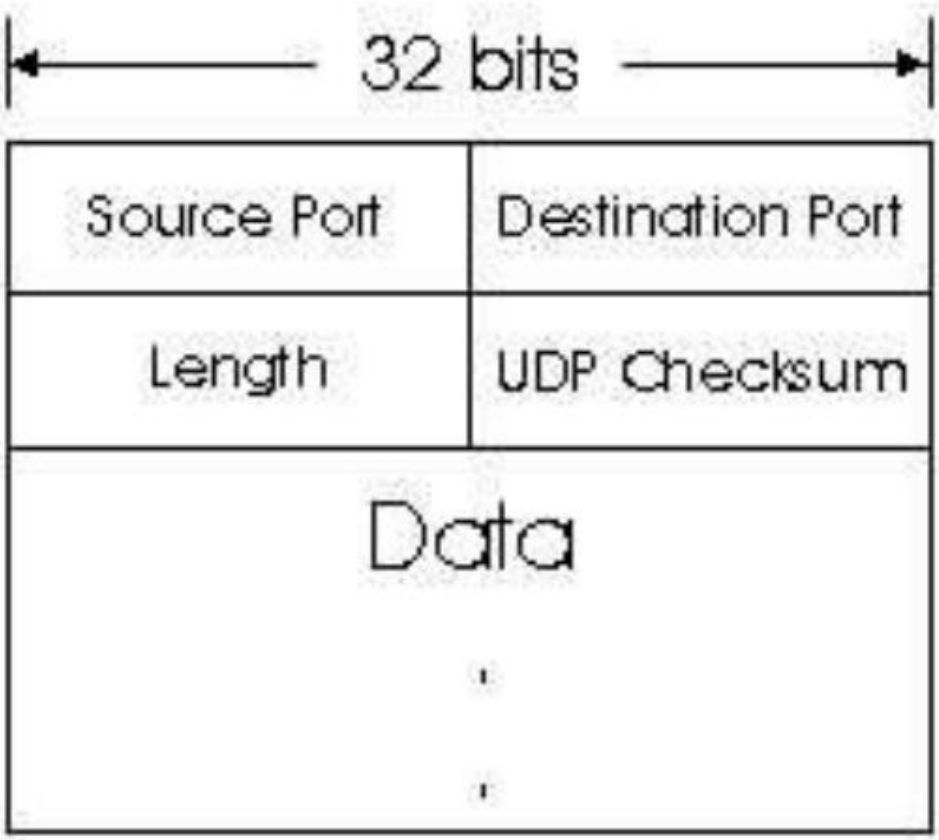
- Es un protocolo minimalista.
- Tiene menos overhead.
- Solo provee Mux/Demux
- No requiere establecimiento de conexión
    - *El multicast es una técnica que permite enviar un mensaje a un grupo de destinatarios de forma simultánea. A diferencia del broadcast, que envía un mensaje a todos los dispositivos de una red, el multicast solo envía el mensaje a los dispositivos que están interesados en recibirlo.*
      La técnica del multicast funciona sobre UDP, ya que no necesita establecer una conexión y se podría usar un mismo socket (un proceso tiene asociado un socket) para recibir datos de varios procesos que se quieren comunicar con un proceso a la vez.
      Teóricamente podría intentarse adaptar multicast sobre TCP, pero sería demasiado complejo e iría en contra de la naturaleza del modelo ya TCP establece una conexión punto a punto entre un único emisor y receptor.
- Orientado a Packets/Datagramas (mensajes auto-contenidos)
- Aplicaciones: video/voz streaming/TFTP/DNS/Broadcast/Multicast.

- Si no hay proceso en estado LISTEN en el puerto destino se responderá con un ICMP “Destination Unreachable” (esto es otro protocolo de IP). Este mensaje ICMP indica que el puerto o el host destino no están disponibles. Sin embargo, este paquete ICMP se puede perder y UDP no generará ninguna notificación al remitente ya que se trata de un protocolo sin conexión y no confiable.

Header

El encabezado UDP provee: MUX/DEMUX de aplicación, detección de errores (no obligatorio).

UDP	
Source Port	Especifica el número de puerto del proceso que envía el datagrama UDP
Destination Port	Este campo indica el número de puerto del proceso de destino al que se debe entregar el datagrama UDP
Length	Este campo especifica la longitud total del datagrama UDP, incluyendo tanto el encabezado como los datos. La longitud se mide en bytes y permite al receptor conocer la cantidad de información que debe procesar en el datagrama.
UDP Checksum	La suma de verificación es un valor calculado que se utiliza para detectar errores en el datagrama UDP durante la transmisión. Se calcula en función del contenido del datagrama, incluyendo el encabezado y los datos. El receptor verifica esta suma de verificación para determinar si el datagrama UDP ha llegado intacto o si ha sufrido algún tipo de corrupción durante la transmisión.



Checksum

Detecta errores simples, puede deshabilitarse. Se calcula en origin y se chequea en destino. No solo se considera contenido de datagrama para el checksum, si no también partes de información dentro de IP (pseudo-header). Si el checksum da error el datagrama se descarta.

El pseudo-header está dentro del datagreama IP. Esto sirve para protección contra paquetes mal enrutados. IP.SRC + IP.DST + Zero + IP.PROTO + UDP.LENGTH.

Básicamente lo que se realiza es la suma de los headers (menos el checksum obvi), los pseudo-headers de IP y un payload (vaya uno a saber que es) y luego se pasa a complemento a uno. En el destino lo que se puede hacer básicamente es calcular el checksum de nuevo (realizo la suma) y si a esa suma le sumo el ca1 realizado por el origen y da todos unos entonces todo esta bien. Si hay un accareo se suma al final.

```
Pseudo header SRC=10.0.2.10, DST=10.0.4.10, PROTO=17
              0A00 020A 0A00 040A 0011

UDP header SRCP=9000, DSTP=7, LEN=15, LEN-PH=15, CKSUM=0
           2328 0007 000F 000F

DATA       5445 5354 0A00 FF[00] [00] = v. padding

   00001010 00000000 = 10.0
   00000010 00001010 = 2.10
   00001010 00000000 = 10.0
   00000100 00001010 = 4.10
   00000000 00010001 =   17
   00100011 00101000 = 9000
   00000000 00000111 =    7
   00000000 00001111 =   15
   00000000 00001111 =   15
   01010100 01000101
   01010011 01010100
   00001010 00000000
   11111111 00000000
  ------------------
{1}11101110 00010111
                  {1}
  ------------------
  ~11101110 00011000
   00010001 11100111       EE18 ~EE18 = 11E7
```

TCP – Transport Control Protocol

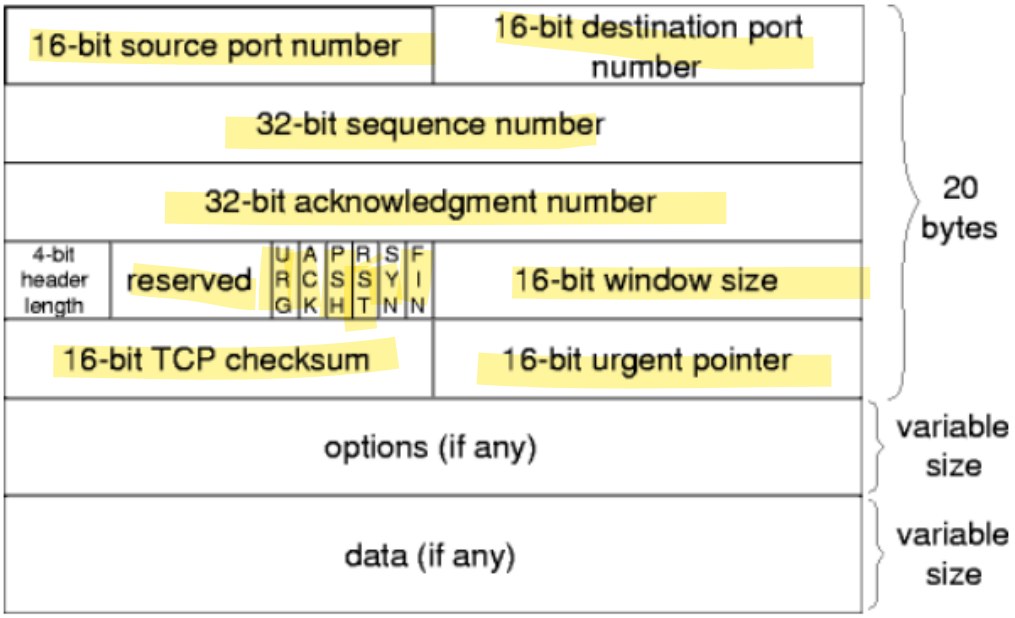
- Protocolo confiable, ordenado, buffering, control de flujo y de congestión.
- Orientado a Streams (secuencia de bytes, ≡ archivo, ordeno si vienen desordenados).
- Provee MUX/DEMUX.
- Incrementa Overhead end-to-end para ofrecer confiabilidad
- Requiere establecimiento de conexión (y cierre)
- Aplicaciones: FTP/HTTP/SMTP/acceso remoto(SSH, telnet, ...)/Unicast.
- Necesidad de manejar Timers, RTO (tmout. por cada segmento). (implementaciones lo manejan más eficiente).

Header

El encabezado TCP provee: MUX/DEMUX, detección de errores, sesiones, control de errores, control de flujo y control de congestión.

TCP	
16-bit source port number	Indica el número de puerto de origen de la aplicación que está enviando los datos.
16-bit destination port number	Especifica el número de puerto de destino de la aplicación receptora.
32-bit sequence number	Este campo se utiliza para mantener un seguimiento del orden de los segmentos TCP en una comunicación. Cada segmento TCP se etiqueta con un número de secuencia único.
32-bit acknowledgment number	Indica el número de secuencia que espera recibir el emisor del siguiente segmento. Ayuda a establecer que los datos se han recibido de manera confiable.
4-bit header length	Este campo especifica la longitud del encabezado TCP en palabras de 32 bits. Se utiliza para identificar dónde comienza la carga útil de datos en el segmento. Indica la longitud de los variables.
reserved	Este campo se reserva para uso futuro y debe establecerse en cero.
Flags	Son bits que se utilizan para controlar y gestionar la comunicación entre los dispositivos.
16-bit window size	Indica el tamaño de la ventana de recepción que el receptor tiene disponible para aceptar datos. Ayuda

	a controlar el flujo de datos en la conexión.
16-bit TCP checksum	Proporciona una suma de verificación para verificar la integridad de los datos en el segmento TCP y detectar errores de transmisión.
16-bit urgent pointer	Se utiliza en la comunicación para indicar la posición de datos urgentes dentro del segmento, si es necesario.
options	Este campo opcional permite la inclusión de información adicional en el encabezado TCP, como máximas segment size (MSS), ventana de escala, timestamp, entre otros. Las opciones se utilizan para ajustar y optimizar la comunicación según las necesidades de la aplicación.



Otros campos

TCP entrega y envía lo datos agrupados o separados de forma dis-asociada de la aplicación

Datos Urgentes: URG.

- Urgent Pointer valido su URG=1. ´
- Indica: offset positivo + #Seq = last Data Urgent byte.
- Indicar a la App. datos urgentes, debe leer.
- Debería combinarse con PSH. Habitualmente llamado

OOB data (TCP no soporta OOB!!!).

Pushear datos: PSH.

- Fuerza a TCP a pasar datos a la App.
- No lo deja “Bufferear” los datos recibidos (Input).

Opciones

- Maximum Segment Size (MSS), min. recomendado 536B, RFC-879 (basado en MTU=576, TCP+IP=40). Aclaraciones en RFC-6691:
    - El tamaño de la ventana de recepción TCP es la cantidad de datos de recepción (en bytes) que se pueden almacenar en búfer durante una conexión.
      En lugar de usar un tamaño de ventana de recepción predeterminado codificado de forma rígida, TCP se ajusta a incrementos pares del tamaño máximo de segmento (MSS).
      Maximum Segment Size es un campo de los encabezados que indica el tamaño más grande de datos que puede tener un segmento sin ser fragmentado. El MSS mide la parte de un paquete que no tiene encabezado, lo que se conoce como carga útil. El MSS está determinado por otra métrica que tiene que ver con el tamaño de los paquetes: MTU, o la unidad máxima de transmisión, que sí incluye los encabezados TCP e IP (Protocolo de Internet).
      El MSS es igual a la MTU menos el tamaño de un encabezado TCP y un encabezado IP:
      MTU - (encabezado TCP + encabezado IP) = MSS

Una de las principales diferencias entre la MTU y el MSS es que si un paquete supera la MTU de un dispositivo, se divide en trozos más pequeños, o "se fragmenta." En cambio, si un paquete supera el MSS, se descarta y no se entrega.
El MSS se negocia durante la configuración de la conexión, es decir, durante el saludo de tres vías.

- Window Scaling. Selective Acknowledgements (SACK).
- Timestamps.
- NOP.
- Otras.

Checksum

Se calcula igual que en UDP, la única diferencia es que es obligatorio. Acá el total LEN se computa para PseudoHDR, no viaja en el segmento (TCP no tiene longitud total)

Negociación

- Permite Opciones y Negociación.
- TCP entrega y envía lo datos agrupados o separados de forma dis-asociada de la aplicación

Establecimiento de conexión (3WH)

Es un método de tres pasos que requiere que tanto el cliente como el servidor intercambien segmentos SYN y ACK antes de que comience la comunicación de datos real.

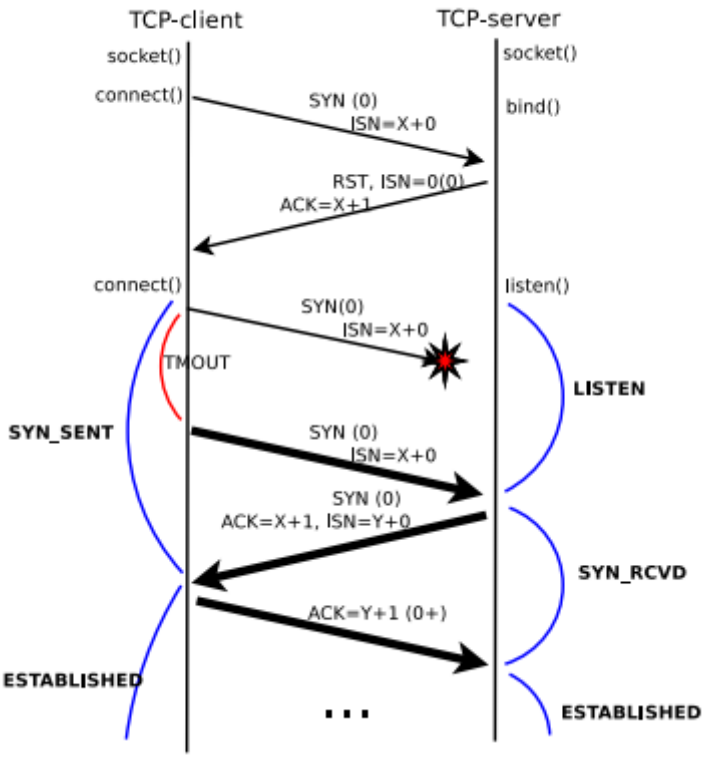
Paso 1 (SYN) – El cliente inicia el proceso enviando un segmento al servidor con el bit SYN establecido y un número de secuencia inicial (ISN) generado de manera pseudoaleatoria (un contador que se incrementa cada 4 mseg). Este es importante para identificar y ordenar los datos en la conexión.

Paso 2 (SYN/ACK) – El servidor recibe el segmento del cliente, reconoce el bit SYN y responde enviando un segmento de respuesta con los bits SYN y ACK establecidos. En este, el servidor incluye su propio número de secuencia inicial (ISN), que también es elegido de manera pseudoaleatoria.

Se envía RST si no hay proceso en estado LISTEN.

Paso 3 (ACK) – El cliente recibe la respuesta del servidor, reconociendo el ISN del servidor. El cliente responde enviando un segmento de confirmación con el bit ACK establecido, confirmando que ha recibido correctamente la respuesta del servidor. Ambos establecen una conexión confiable con la cual iniciarán la transferencia de datos real. En este ya se puede enviar información.

En UPD no se utiliza nada similar ya no se establece ninguna conexión



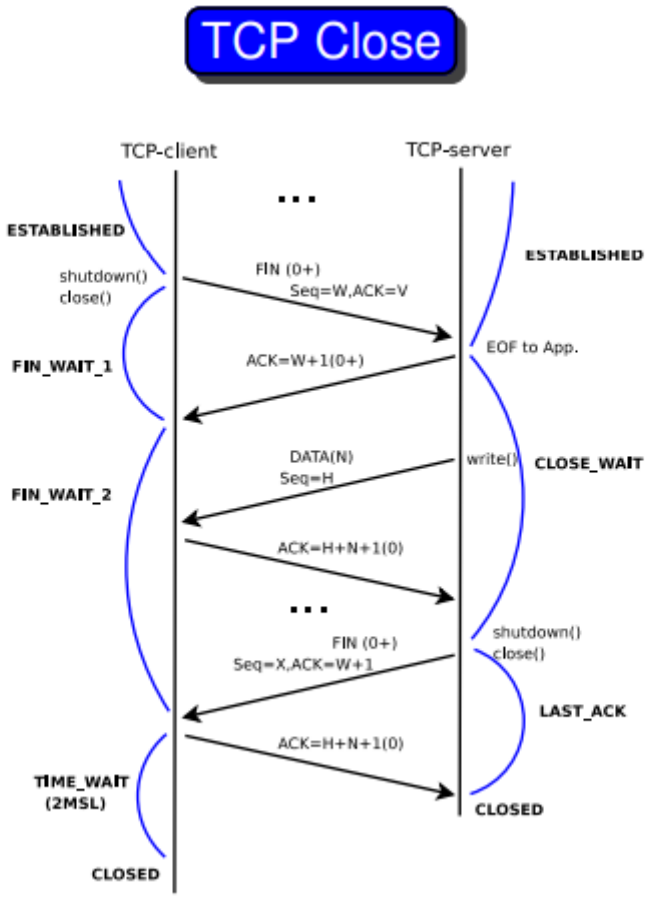
ACK

El ACK es la información que estoy esperando recibir del otro extremo.

Acá salen los ISN relativos (0, 1, 2, …). Los absolutos son con un contador de SO (ya lo mencioné) El número de secuencia se incrementa en el establecimiento de la conexión, en el cierre y cuando se envían datos.

Cierre de conexión (4WH)

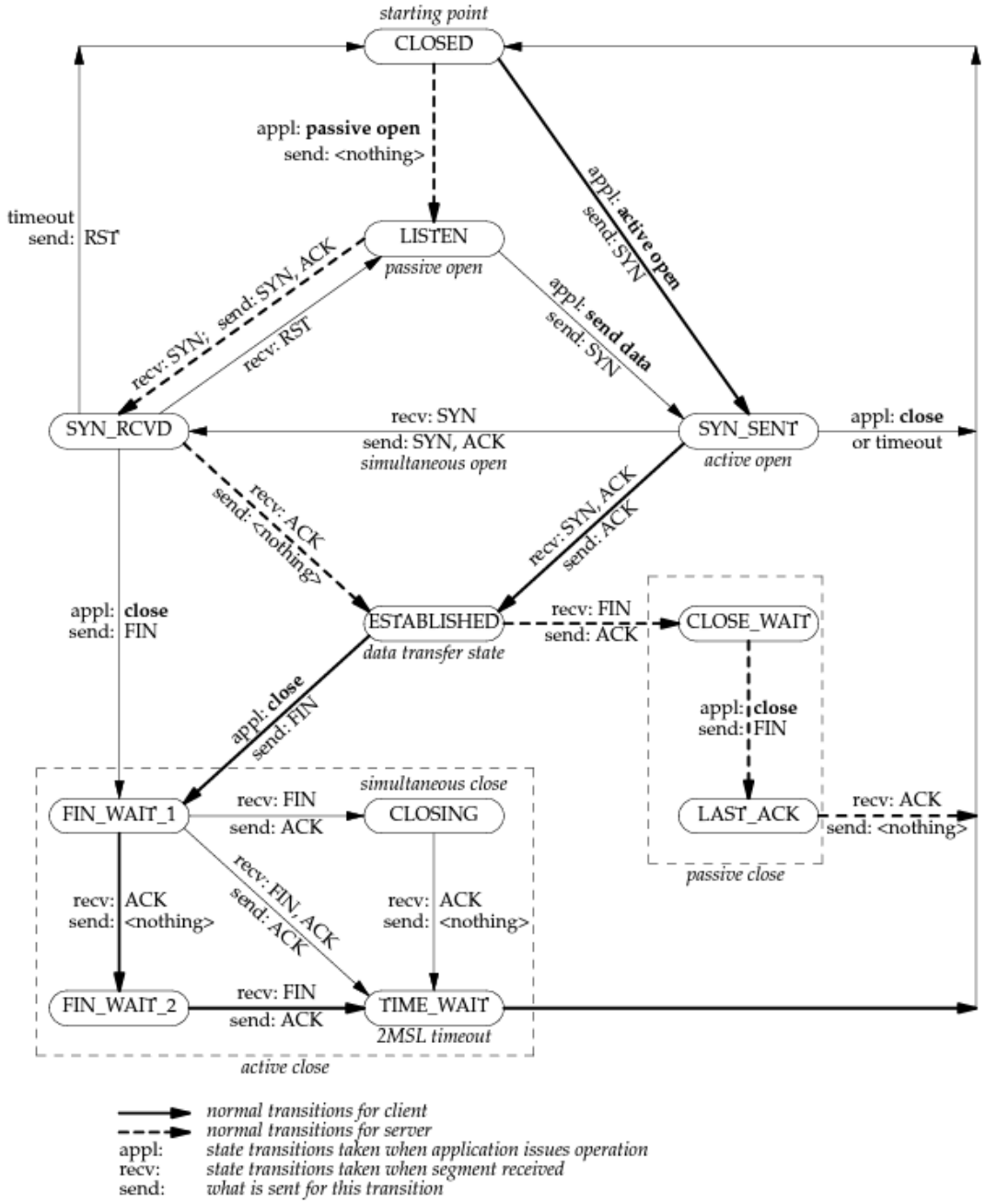
- Flags: FIN (Finish), ACK y RST.
- 4Way-Close (4WC).
- Posibilidad de Half-Close.
- Podría cerrarse en 3WC.
- Espera en TIME WAIT, 2MSL (aprox. 2*2min).
- Evitar con SO_REUSEADDR.
- Cierre incorrecto con RST.
- Close simultaneo.



Estados conexión

- LISTEN representa la espera de una solicitud de conexión procedente de cualquier TCP remoto y puerto.
- SYN-SENT representa la espera de una solicitud de conexión coincidente después de haber enviado una solicitud de conexión.
- SYN-RECEIVED representa la espera de una confirmación de solicitud de conexión después de haber recibido y enviado una solicitud de conexión.
- ESTABLISHED representa una conexión abierta, lo que permite que los datos recibidos sean entregados al usuario. Es el estado normal durante la fase de transferencia de datos de la conexión.
- FIN-WAIT-1 representa la espera de una solicitud de terminación de conexión procedente del TCP remoto, o la confirmación de la solicitud de terminación de conexión previamente enviada.
- FIN-WAIT-2 representa la espera de una solicitud de terminación de conexión procedente del TCP remoto.
- CLOSE-WAIT representa la espera de una solicitud de terminación de conexión procedente del usuario local.

- CLOSING representa la espera de una confirmación de la solicitud de terminación de conexión procedente del TCP remoto.
- LAST-ACK representa la espera de una confirmación de la solicitud de terminación de conexión previamente enviada al TCP remoto (la cual incluye una confirmación de su solicitud de terminación de conexión).
- TIME-WAIT representa la espera de suficiente tiempo para asegurar que el TCP remoto ha recibido la confirmación de su solicitud de terminación de conexión.
- CLOSED representa la ausencia total de una conexión activa.



Control de errores

- Mecanismo protocolar, algoritmo, que permite ordenar los segmentos que llegan fuera de orden y recuperarse mediante solicitudes y/o retransmisiones de aquellos segmentos perdidos o con errores.
- Objetivo: recuperarse de los efectos del re-ordenamiento, la pérdida o la corrupción de los paquetes en la red.
- Se realiza por cada conexión: End-to-End, App-to-App.

Va a haber básicamente un timer de confirmación (RTO). Si paso el tiempo, reenvío el mensaje (retransmito)

TCP utiliza el RTT para estimar el tiempo que tarda un paquete en viajar desde el remitente hasta el destinatario y de vuelta.

El RTT estimado se utiliza para determinar el tiempo de espera (timeout) para recibir una confirmación (ACK) del destinatario.

RTT: se calcula a medida que se envían datos y se reciben ACK. Es el tiempo transcurrido entre el envío de un segmento y la recepción del ACK correspondiente.

RTO: se calcula cada vez que se van mandando segmentos. Se manda segmento, se inicia timer (del RTO) y si se pasa ese timer, se retransmite. TCP mantiene por RTO por el más viejo. Básicamente se suman los tiempos, se hace un promedio pesado, se calcula la desviación para suavizar la medición y ahí se obtiene el RTO. El RTO se va ajustando. El RTO es el tiempo que se espera antes de retransmitir un mensaje.

En TCP los timestamps son opcionales:

La opción de marcas de tiempo en TCP permite a los endpoints mantener una medición más precisa del tiempo de ida y vuelta (RTT) de la red entre ellos. Este valor ayuda a cada pila TCP a configurar y ajustar su temporizador de retransmisión. Hay otros beneficios, pero la medición RTT es el principal.

Para ello se incluye un Timestamp Value TSval en cada segmento que se envía. Los valores TSval se repiten en el lado opuesto de la conexión en el campo Timestamp Echo Reply TSecr . Entonces, cuando se confirma un segmento, el remitente de ese segmento puede simplemente restar su marca de tiempo actual del valor TSecr para calcular una medición precisa del tiempo de ida y vuelta (RTT).

RTT=TSecr − TSval

S&W

El remitente envía un paquete y luego espera una confirmación del receptor antes de enviar el siguiente. No se envía el próximo mensaje hasta que no se confirma el que se envió.

Cuando se envía un segmento se inicia un timer (similar al RTO). Si se termina ese timer y no se recibe ACK, se reenvía.

Es sencillo pero ineficiente. No se aprovecha el ancho de banda. Aun así es mejor enviado mas datos en segmentos (no da esto)

Se envía un dato por vez, ventana de transmisión/recepción: K = 1, W = 1.

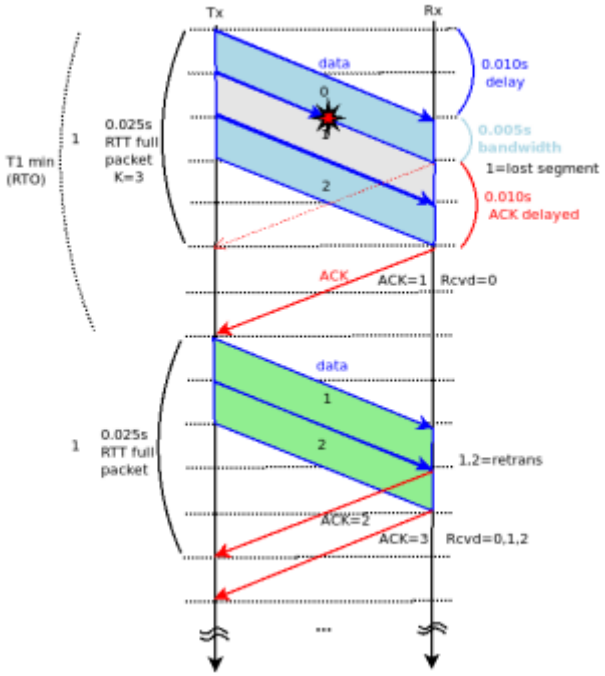
Ventana desplegable.

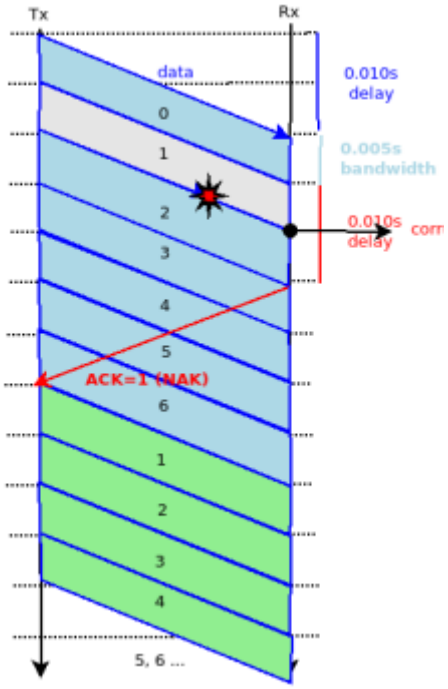
- Usa pipelining: permitir enviar múltiples segmentos/paquetes sin aun haber recibido confirmaciones, paquetes “in-flight”.
- La cantidad de segmentos que se puede enviar sin aun recibir confirmación se llama Ventana, notado como K o W , K = n, donde n > 1.
- Requiere ampliar los números de segmentos y de las confirmaciones además del buffering entre capa usuaria-RDT en ambos procesos: Sender, Receiver.
- Por cada mensaje enviado se inicia un timer de retransmisión: T1 o RTO
- Por cada confirmación se descarta/re-inicia el timer RTO. Si no se recibe confirmación vence RTO (timeout) y se retransmite, nuevo timer

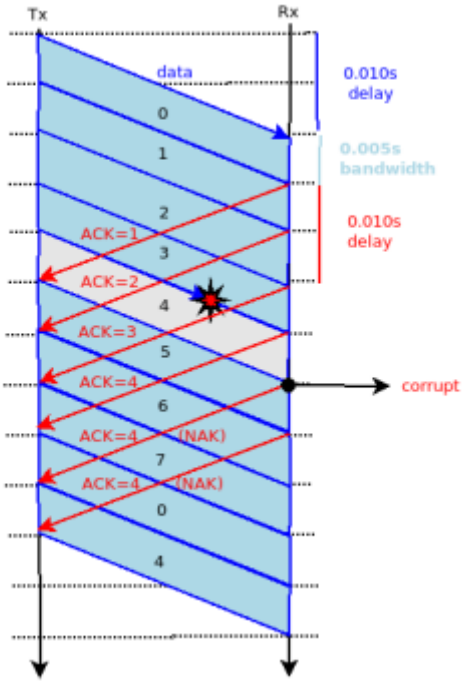
Go-Back N

- Se tiene una ventana ”estática” de tamaño K = n, n > 1.
- Numeración de segmentos, se realiza en modulo M, K <= (M − 1). Si K = M puede haber problemas.
- No admite segmentos fuera de orden, ni confirmaciones fuera de orden. Solo se confirman por la positiva los segmentos que se pudieron colocar en el buffer en orden.
- Se puede confirmar desde N hacia atrás (ACK acumulativos). No necesariamente se confirma cada segmento individualmente.

- Se puede re-transmitir ante un timeout o un NAK. Requiere buffering extra en el emisor, no se pueden descartar del buffer de com. con la capa usuaria. Si se re-transmite ante un timeout se hace desde N hacia adelante, los que ya se enviaron.
- Emisor puede mantener solo un timer para el segmento más viejo enviado y aun no confirmado, si este expira retransmite todos los no confirmados.
- Si llega una confirmación en orden arranca un nuevo timer.
- Si llega un segmento de datos en orden se puede/debe confirmar por la positiva indicando el próximo.
- Si llega un segmento fuera de orden o este corrupto se puede confirmar por la negativa indicando que se espera el próximo al último recibido de forma adecuada.
- Ante el error se puede esperar la retransmisión.
- Puedo mandar y confirmar datos al mismo tiempo (Piggy Backing)
- Si se pierde uno segmento es como “vaciar” el pipe y volver atrás
- Las confirmaciones por la negativa pueden generar ACK duplicados (NAK).







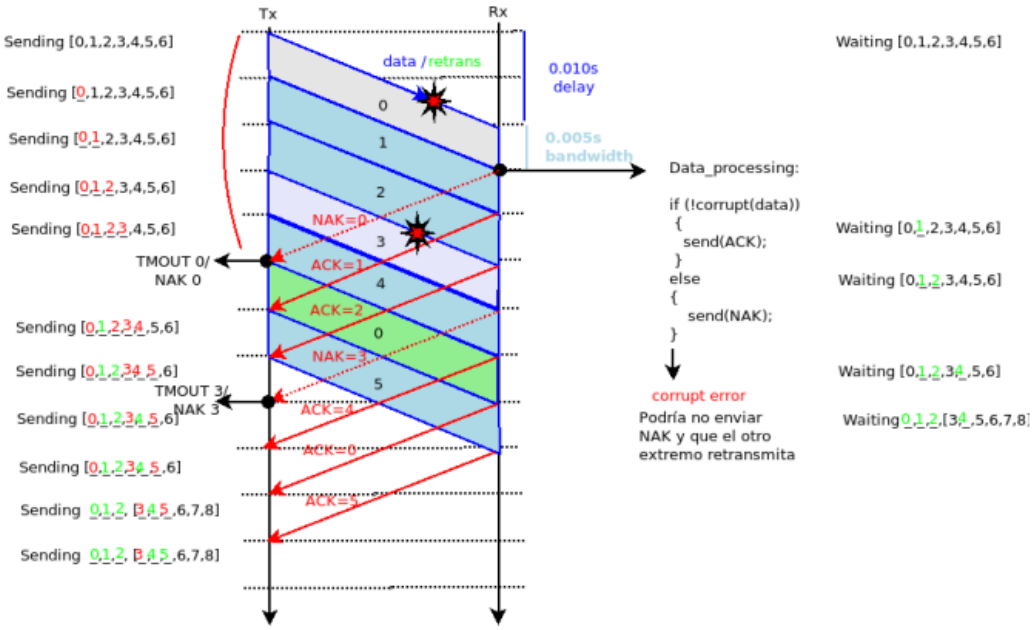
Select N Repeat

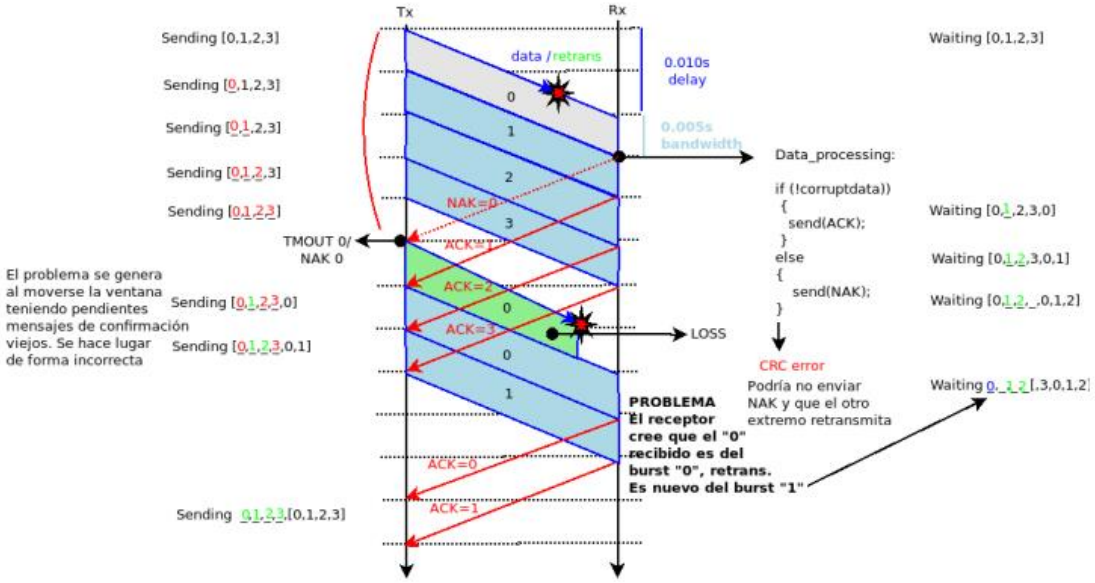
- En TCP es opcional
- Solo retransmite los que no se confirmaron.
- El receptor puede confirmar de a uno o usar bit vectors/intervalos de confirmaciones.
- No se puede usar confirmaciones acumulativas.

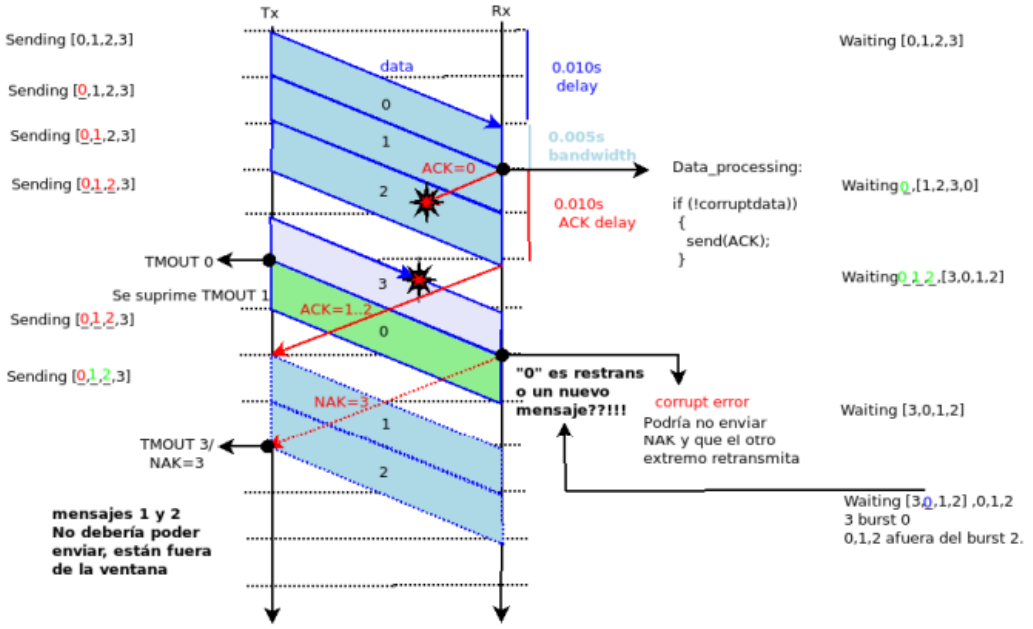
- No se deben confundir los segmentos de diferentes ráfagas. No se deben reusar #ID/SEQ hasta asegurarse que tiene todos los mensajes previos o estos no están en la red. Para esto el tamaño de la ventana no debe exceder la mitad del tamaño total del espacio de números de secuencia. La razón detrás de esta restricción es evitar la posibilidad de que un número de secuencia se reutilice antes de que el ACK correspondiente haya llegado, ya que la ventana se implementa como un buffer circular, entonces si fuese más grande podría haber paquetes representados por la misma posición en el buffer lo que podría llevar a confusiones en la correcta interpretación de los frames.

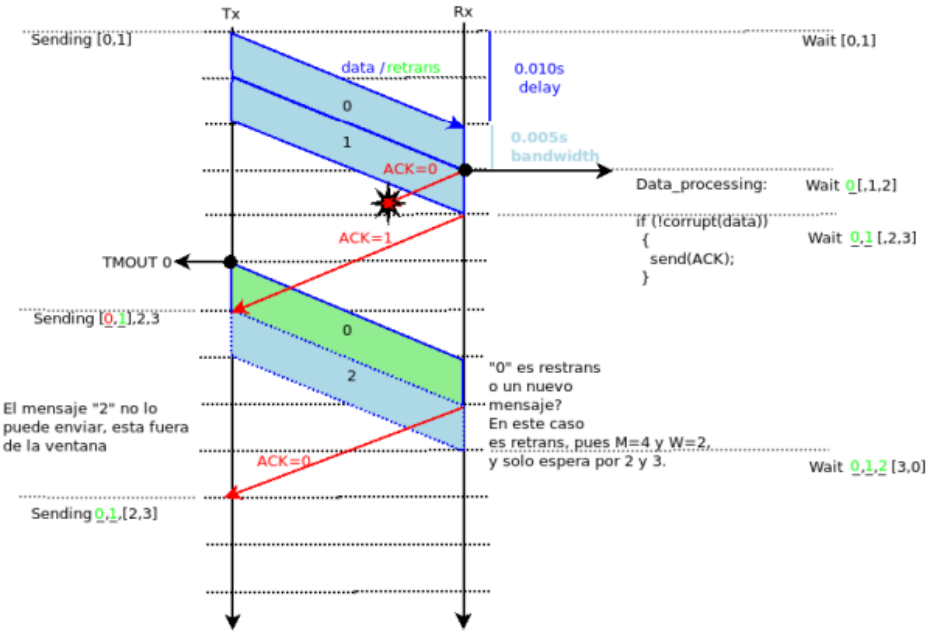
$$K <= \frac{(M-1)}{2}$$

- La ventana se desliza sin dejar huecos, desde los confirmados más viejos (confirmación, corro ventana, solo si no deja hueco)




Tx
Rx
Sending [0,1,2,3]
Sending [0,1,2,3]
Sending [0,1,2,3]
Sending [0,1,2,3]
Sending [0,1,2,3]
Sending [0,1,2,3,0]
Sending [0,1,2,3,0,1]
Sending 0,1,2,3,[0,1,2,3]
data /retrans
0.010s
delay
0.005s
bandwidth
0
1
2
3
NAK=0
ACK=1
ACK=2
ACK=3
0
1
ACK=0
ACK=1
TMOUT 0/
NAK 0
LOSS
Data_processing:
if (!corruptdata))
{
send(ACK);
}
else
{
send(NAK);
}
CRC error
Podría no enviar
NAK y que el otro
extremo retransmita
PROBLEMA
El receptor
cree que el "0"
recibido es del
burst "0", retrans.
Es nuevo del burst "1"
Waiting [0,1,2,3]
Waiting [0,1,2,3,0]
Waiting [0,1,2,3,0,1]
Waiting [0,1,2,_,0,1,2]
Waiting 0,_,1,2,[,3,0,1,2]
El problema se genera
al moverse la ventana
teniendo pendientes
mensajes de confirmación
viejos. Se hace lugar
de forma incorrecta


Tx
Rx
Sending [0,1,2,3]
Sending [0,1,2,3]
Sending [0,1,2,3]
Sending [0,1,2,3]
Sending [0,1,2,3]
Sending [0,1,2,3]
data
0.010s
delay
0.005s
bandwidth
0
1
2
ACK=0
0.010s
ACK delay
TMOUT 0
Se suprime TMOUT 1
3
ACK=1, 2
0
NAK=3
1
2
TMOUT 3/
NAK=3
"0" es restrans
o un nuevo
mensaje??!!!
corrupt error
Podría no enviar
NAK y que el otro
extremo retransmita
Data_processing:
if (!corruptdata))
{
send(ACK);
}
mensajes 1 y 2
No debería poder
enviar, están fuera
de la ventana
Waiting [0,1,2,3]
Waiting 0,[1,2,3,0]
Waiting 0,1,2,[3,0,1,2]
Waiting [3,0,1,2]
Waiting [3,0,1,2] ,0,1,2
3 burst 0
0,1,2 afuera del burst 2.



TCP

- TCP hace el control de errores por bytes (byte oriented), no por segmentos.
- Los segmentos se numeran de acuerdo a bytes enviados (nro. Del primer byte).
- Los números se negocian al establecer la sesión y cada implementación los elige libremente (ISN).
- Las confirmaciones son “anticipativas”, indican el nro. de byte que esperan.
- Utiliza Go-back-N con ventana dinámica (flow-control), utiliza piggy-backing y permite negociar Ventana Selectiva con Opciones.
- Para control de errores TCP utiliza los campos: #SEQ, #ACK, flag ACK más timer y algunas opciones.

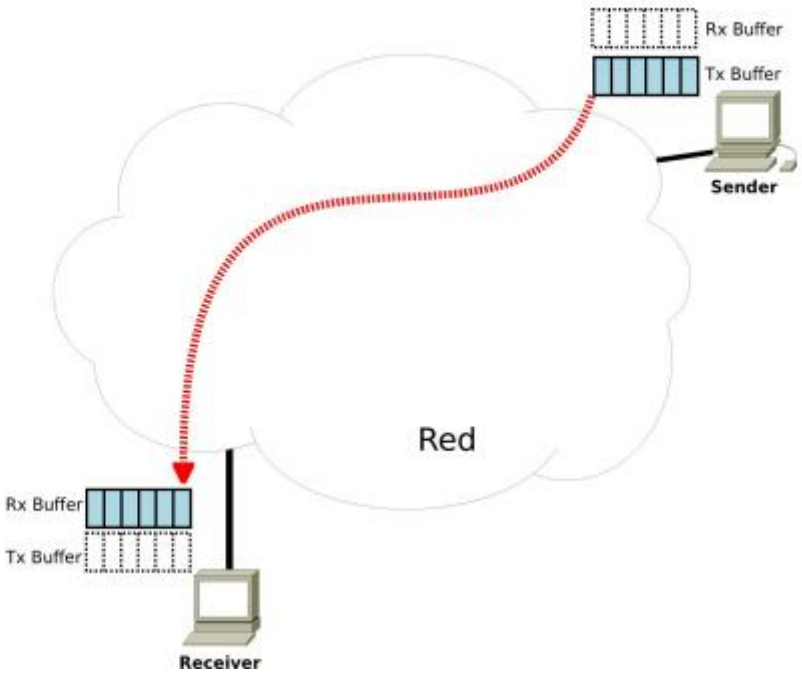
- Segmentos ACked no indica leído por aplicación, sí recibido por TCP (RFC-793) (ubicado en el **Rx Buffer** del receptor).
- Si el receptor detecta error en el segmento simplemente descarta y espera que expire *RTO* en el emisor (podría envía un NAK, re-enviar ACK para el último recibido en orden, forma de solicitar lo que falta).
- Receptor con segmentos fuera de orden descarta directamente y podrá re-enviar ACK (podría dejar en **Rx Buffer** pero no entregar a la aplicación, tiene huecos).
- Se puede confirmar con ACK acumulativos.

- TCP NO arrancar un *RTO* por cada segmento, solo mantiene un por el más viejo enviado y no ACked y arranca uno nuevo solo si no hay *RTO* activo.
- Si se confirman (ACked) datos, se inicia un nuevo *RTO* (RFC-6298) recomendado.
- El nuevo *RTO* le esta dando más tiempo al segmento más viejo aún no confirmado.
- Si vence un RTO se debe retransmitir el segmento más viejo no ACKed y se debe doblar: Back-off timer $RTO = RTO * 2$ $RTO_{MAX} = 60s$ (RFC-6298) recomendado.
- TCP calcula el RTO de forma dinámica. RFC-6298(2011), ayudado por Timestamp Option.

Control de Flujo

- Mecanismo protocolar, algoritmo, que permite al receptor controlar la tasa a la que le envía datos el transmisor.
- Control cuanto puede enviar una aplicación sabiendo que la receptora tiene capacidad de recibirlo y procesarlo.
- Objetivo: prevenir que el emisor sobrecargue al receptor con datos evitando un mal uso de la red.
- Sincronización de buffers para que no haya sobrecarga en los extremos.

- De Extremo a Extremo, principio end-to-end.



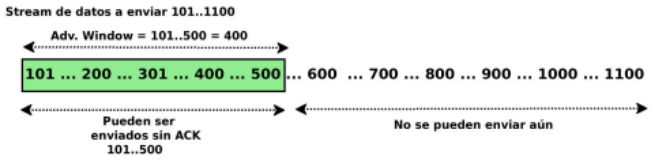
- El receptor (cada extremo puede recibir, es FDX) indica el espacio del buffer de recepción, Rx Buffer, en el campo del segmento: Window (de datos o ACK) Advertised Window (Ventana Anunciada).
- Por cada segmento que envía indica el tamaño del buffer de recepción Rx Buffer. Cada conexión mantiene su propio buffer en espacio del kernel (TCP).
- Window (Ventana) indica la cantidad de datos en bytes que el emisor le puede enviar sin esperar confirmación (mejora notablemente contra Stop & Wait).
    - El Window Size refleja el espacio libre disponible. Si la aplicación lee se agranda.
- La ventana de recepción de cada extremo es independiente.
- Cada vez que llega un segmento es puesto por TCP en el Rx Buffer, TCP lo debe confirmar.
- Cada vez que la aplicación lee se hace espacio en el Rx Buffer. Se va modificando el tamaño de la ventana.

- Cada vez que llega un ACK en orden se mueve la ventana en el Transmisor, se descartan segmento confirmado de Tx Buffer
- Se tiene una ventana deslizante y dinámica

En la ventana considero los que envíe y no me confirmaron y los que tengo por enviar. Se puede mantener un buffer para write (pero no implica mucho cambio)

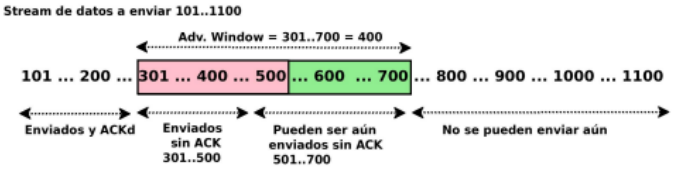
Ejemplo

- Se establece la conexión, se indica $WIN = 400$.

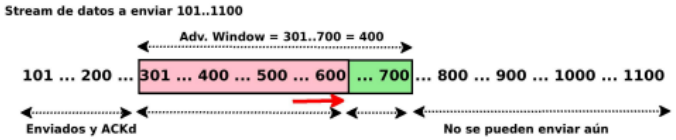


- Luego, la aplicación que envía escribe, `write()`, y se envían 400 bytes (los 400 bytes se pueden enviar en múltiples segmentos).
- Se recibe un segmento con $ACK = 301$ y $WIN = 400$.
- Se desliza ventana.

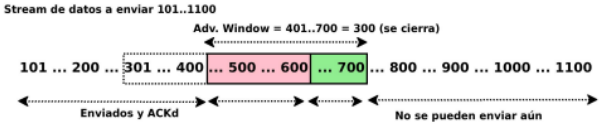
- 101..300 en ningún buffer, enviados y leídos.
- 301..500 en Tx Buffer y "en vuelo" o entrando a Rx Buffer.
- 501..700 en Tx Buffer, aún no han sido enviados.
- 701..1100 en la aplicación que envía, bloquea en caso de `write()`, depende de Tx Buffer.



- Se envía un segmento con los bytes 501..600.
- No se recibe confirmación aún, el último segmento recibido $WIN = 400$.
- 301..600 en Tx Buffer y "en vuelo" o llegando a Rx Buffer.
- 601..700 en Tx Buffer, aún no han sido enviados.

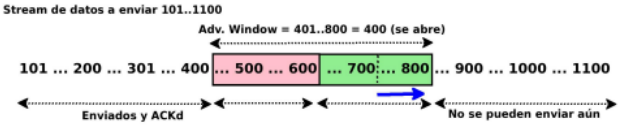


- Se recibe un segmento $ACK = 401$, $WIN = 300$.
- 101..300 ya estaban procesados, 301..400 en Rx Buffer, no se han leído aún.
- 401..600 en Tx Buffer, "en vuelo".
- 601..700 en Tx Buffer, aún no han sido enviados.
- Ventana se cierra, la aplicación receptora no lee, no llama a `read()`.

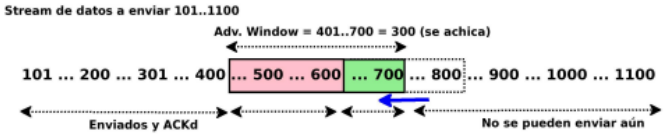


Ventana se cierra cuando la app del otro lado no lee.

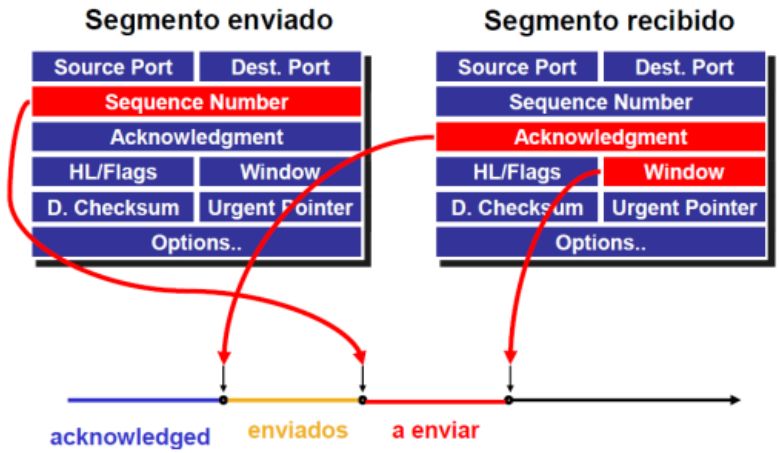
- Se recibe un segmento *ACK* = 401, *WIN* = 400.
- 401..600 en Tx Buffer, “en vuelo”.
- 601..800 en Tx Buffer, aún no han sido enviados.
- Ventana se abre, la aplicación receptora lee, llama a `read()`.
- 101..400 no están más en Rx Buffer, se procesaron.



- Se recibe un segmento *ACK* = 401, *WIN* = 300.
- 401..600 en Tx Buffer, “en vuelo”.
- 601..700 en Tx Buffer, aún no han sido enviados.
- Ventana se achica.
- No debería suceder, TCP achica el Rx Buffer.



Esto puede suceder porque el SO le saca memoria al proceso.



El emisor va calculando cuando puede enviar. Básicamente calcula lo que puede seguir enviando sin recibir confirmación. El “control de flujo” se activa cuando se va achicando la ventana.

a. ¿Quién lo activa? ¿De qué forma lo hace?

El control de flujo lo activa el receptor enviando ventanas más chicas. Esto deja en evidencia que el receptor tiene poco espacio (o no tiene más lugar) para seguir recibiendo datos. Esto se realiza a través del campo de tamaño de ventana en los encabezados de los segmentos TCP.

b. ¿Qué problema resuelve?

Resuelve el problema de la posible saturación o congestión de los buffers en los endpoints. Al indicar al emisor que reduzca la cantidad de datos que está enviando, evita que el receptor se sobrecargue.

c. ¿Cuánto tiempo dura activo y qué situación lo desactiva?

Cuanto tiempo dura activo depende del receptor (más que nada la velocidad en que lee la aplicación). El control de flujo está activo mientras el receptor envíe ventanas más pequeñas (indicando capacidad limitada). Durará activo hasta que el receptor envíe ventanas más grandes, lo que indica que tiene más capacidad para recibir datos.

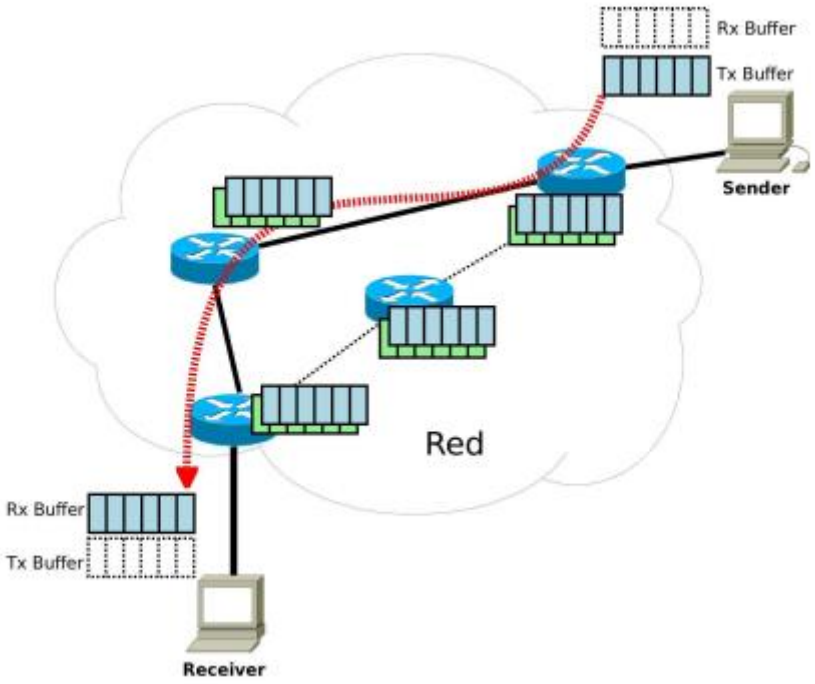
En todo momento ambos extremos están actualizando su propia ventana.

- Ventana de Recepción recibida: $Win = rwnd$.
- El receptor "ofrece/publica" la ventana *Win* en los segmentos TCP.
- El transmisor no puede enviar más de la cantidad de bytes en: $Win - Sent.No.ACKed$, $Effective\_Win = Win - (LastByteSent - LastByteAcked)$ si no se tiene en cuenta la congestión.
  - Al recibir ACKs de TCP (App. no lee aún) se cierra ventana.
  - Al recibir ACks y *Win* fijo desliza ventana (App. lee a rate fijo).
  - Al achicarse *Win* se reduce ventana (App, no lee).
  - Al agrandarse *Win* tiene posibilidad de enviar más (App. lee más rápido).
  - Tamaño de ventana seleccionado por el kernel o por aplicación `setsockopt()`.

Control de Congestión

- Se realiza por cada conexión: End-to-End, App-to-App.
- Permite que aplicaciones no saturen la capacidad de la red.

- Tiene en cuenta el estado de la red a diferencia del control de flujo que solo ve el receptor.
- **Objetivo**: Controlar el tráfico que se envía evitando que se colapse la red y se descarte teniendo que retransmitirse
- Me da una visión de la red



¿Por qué hay problemas de congestión?

- Límite de la capacidad de la red:
  - Velocidad de los Routers/Switches (CPU).
  - Capacidad de los Buffers de los Routers/Switches (Memoria).
  - Velocidad de los Enlaces (Interfaces)
- Demasiado tráfico en la red

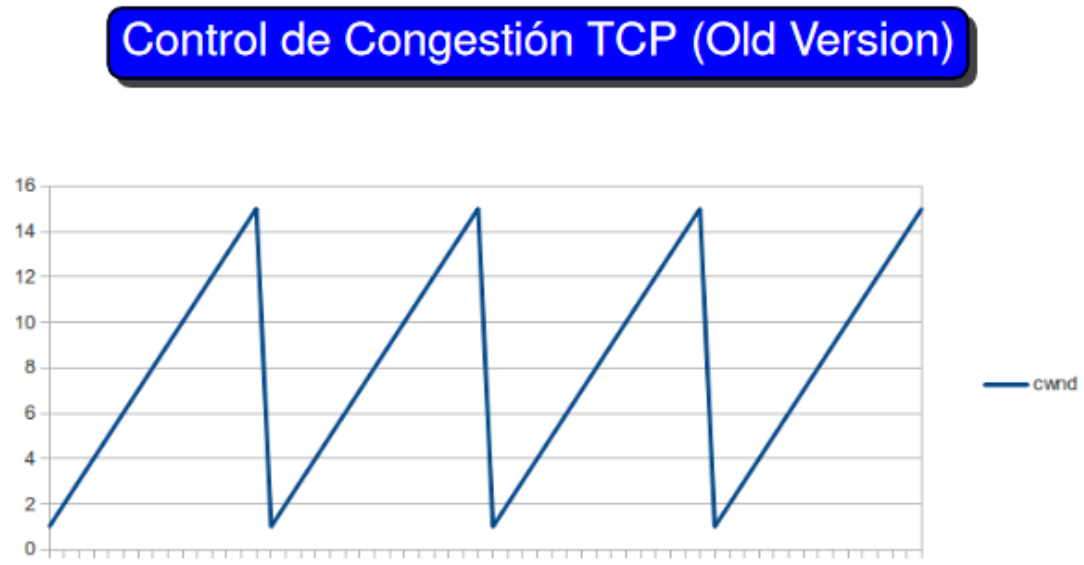
Modelo End-to-End

- Modelo en el cual no participa la red (más implementado).
- Se utilizan nuevos parámetros a los de Control de Flujo (variables locales):
  - cwnd Ventana de congestión. tiene en cuenta el estado de ´ la red
  - ssthresh Slow Start Threshold (Umbral).
  - Se calcula: MaxWin = Min(rwnd, cwnd). rwnd era la ventana de recepción, usada para el control de flujo

    - Básicamente la ventana efectiva es la ventana máxima (regida por la de flujo o congestión) – lo que ya se envió y no se confirmó (esto se llama FlightSize).
- Hay diferentes fases:
    - Si cwnd < ssthresh: Fase de crecimiento inicial: SS (Slow Start).
    - Si cwnd >= ssthresh Fase de mantenimiento: CA (Congestion Avoidance).

Old Version

- No considera control de congestión inteligente.
- “Ventana de congestión: cwnd” (no definida), trafico crece hasta que se resetea, buffer overflow o error.
- El destino podría limitarla con rwnd: valido para LAN.
- No utiliza Slow Start (SS), se supone: incrementa cwnd + + (en MSS) por cada RTT.
- ACK perdido o segmento erróneo deriva en arrancar de 0
- La congestión se detecta cuando expira el RTO.
- No valido para WAN

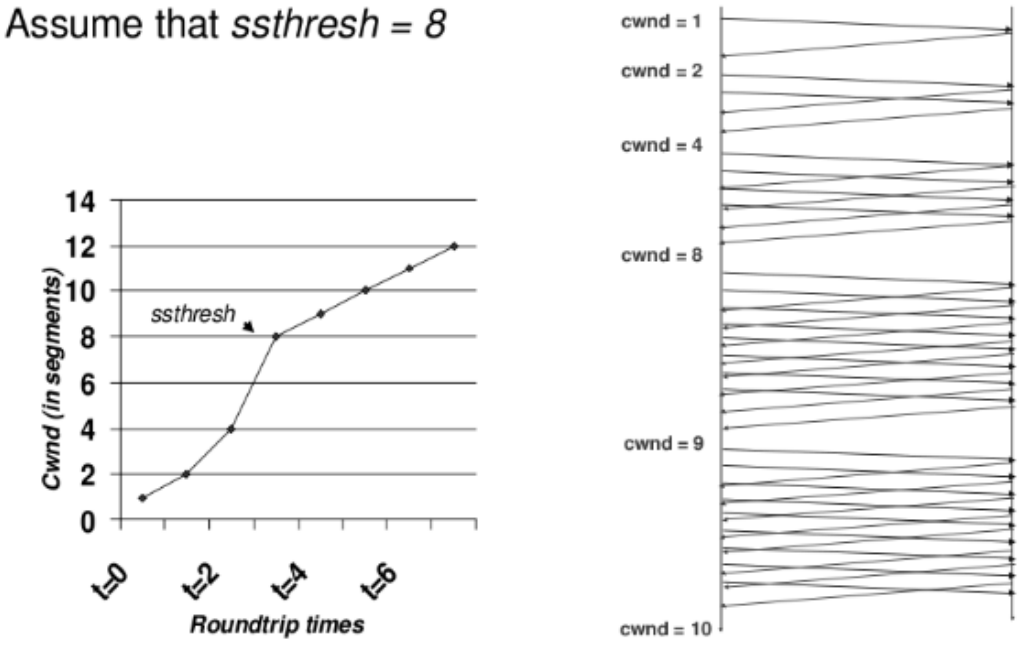


Old Tahoe/ Tahoe

- Hace uso de Slow Start. La ventana crece exponencialmente inicio cwnd = IW = 1 ∗ M SS, o similar, IW (Init Window) puede ser 2 o 3 segmentos (no mando tono en una).
- Se alcanza el umbral (sstresh) se trabaja con Congestión Avoidance (CA)

- El valor inicial del umbral es un valor alto (cualquiera)
- Old Tahoe no tiene fast retransmit y la congestión se detecta con timeout.
- Tahoe fast retransmit (pero no bien implementado)
- La congestión en Tahoe se detecta cuando:
    - RTO
    - 3ACK Duplicados (o más)
        - Ambos casos derivan en Slow Start:
            - Se establece umbral (sstresh): Min(cwnd/2, 2) $*$ MSS → El umbral pasa a la mitad de la ventana que habíamos alcanzado.
            - cwnd = LW = 1 $*$ MSS (Loss Window). → Vuelve a arrancar desde abajo
            - Puede suceder que MSS sea diferente entre emisor y receptor, para este caso se considera SM SS y RM SS. Los cálculos se hacen en base a SM SS.
- Slow Start:
    - Crece exponencialmente, de forma rápida, no es slow (lento).
    - Se le llama Slow Start porque comienza a probar con pocos paquetes.
    - Inicia cwnd = IW = 1 $*$ MSS (a veces se usa 2 o 3). Transmite y espera ACK.
    - ACK recibido, cwnd++: cwnd = 2 $*$ MSS, nuevos ACKs: cwnd = 4 $*$ M SS... (recibo ACK incremento exponencial)
    - Si destino retarda ACK no se cumple.
    - Incrementa cwnd ++ (en MSS) por cada ACK
- Congestion Avoidance:
    - Ante el primer evento de congestión se calcula el ssthresh.
    - Una vez que cwnd >= ssthresh crece de forma lineal.
    - Incrementa cwnd + + por cada RTT
- Fast Retransmit: objetivo, recuperarse más rápido que un timeout. En Tahoe, FRT seguido por Slow Start. Vuelve al inicio cwnd = 1 $*$ MSS.

Fases Control de Congestión TCP (SS y CA)



Reno

- Implementa de forma correcta Fast Retransmit y Fast Recovery (además de SS y CA)
- FlightSize = cwnd si siempre tuvo datos para enviar, aunque en general cwnd > FlightSize
- Fast retransmit:
    - Recuperase más rápido que un timeout
    - Si recibe 3 ACK duplicados (4 ACK para el mismo segmento) considera que se perdió (no es re-ordenamiento) y re-envía el segmento solicitado sin esperar RTO.
- Fast Recovery:
    - Luego de Fast Retransmit entra en fase Fast Recovery, crece lineal.
    - Agranda un poco la ventana (los 3 ACK que llegaron son segmentos que la red transporto y llegaron, da un espacio de al menos 3 segmentos más)
    - Incrementa de forma lineal la ventana por cada ACK recibido, considera que es un espacio nuevo en la red (incremento inicialmente en 3, por los 3ACK duplicados)

- Luego de Fast Recovery (se ACKed el segmento perdido) se realiza CA(Reno)

- Si hay timeout → Igual que Tahoe

Todo lo que dije pero en imagen

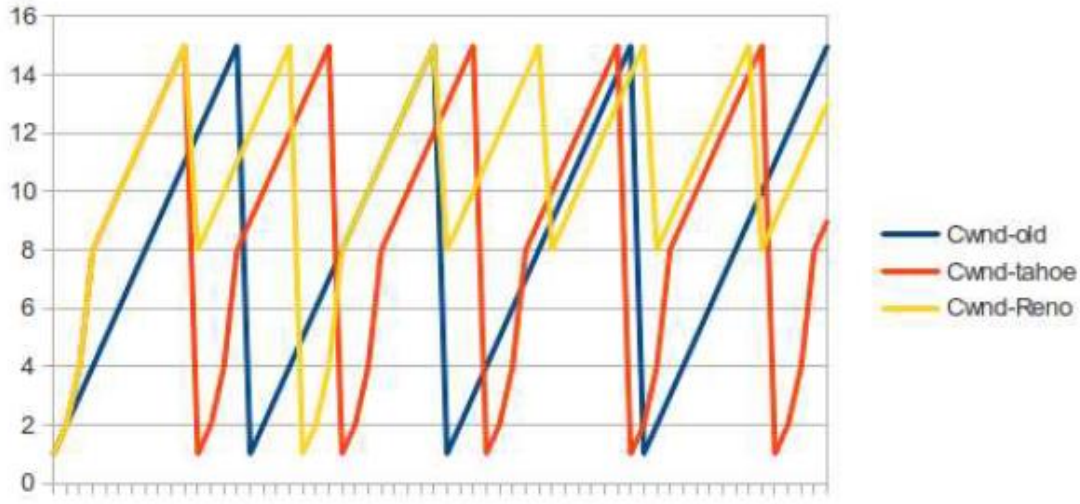
Control de Congestión TCP (Reno)

- Si se detectan 3DUP ACKs (Fast Retransmit):
  1. $ssthresh = Min(cwnd/2, 2) * MSS$, en RFC $ssthresh = Min(FlightSize/2, 2 * SMSS)$
  2. Fast Retransmit.
  3. Fast Recovery: $cwnd = (ssthresh + 3) * MSS$ (3 segments cached in receiver).
  4. Por cada ACK de segmento distinto que recibe incrementa la ventana $cwnd++$ (crece linealmente).
- Una vez recuperado (ACKed segmento perdido) vuelve "a la mitad", $cwnd = ssthresh$, y comienza con CA.
- Los incrementos lineales realizados en FR previos a la recuperación se vuelven atrás (se achica un poco $cwnd$).

Control de Congestión TCP (Reno)

- Si se timeout, RTO expira.
  1. Retransmite el segmento perdido.
  2. $ssthresh = Min(cwnd/2, 2) * MSS$, en RFC $ssthresh = Min(FlightSize/2, 2 * SMSS)$
  3. $cwnd = LW = 1 * MSS$, en RFC $LW = 1 * SMSS$.
  4. Inicia con Slow Start como en los algoritmos anteriores.
  5. Comienza a retransmitir como Go-Back-N a partir del segmento que dio timeout de acuerdo a lo que le permite la ventana.

3 ACK ARRANCA DESDE LA MITAD BASICAMENTE



- El modelo de reno es AIMD → Incremento aditivo (CA) , decremento multiplicativo (cuando vuelvo a la mitad)

New Reno

- Se recupera mas eficientemente. Mejora ante la ausencia de Selective ACK.
- Se recupera de ACK parciales y permite enviar varios segmentos perdidos
- Estándar de TCP

TCP Vegas otra implementación

ECN

- Participa la red
- Comunicación de TCP/IP
- Ambos extremos deben soportar el control de congestión manejado por la red
- La red marca los paquetes IP
- Source Quench no se termina utilizando tanto
- No se generan paquetes nuevos. Marco en los mismos mensajes.
- En TCP uso los flags CWR y ECE (se negocia y se notifica entre extremos con estos flags).
    - SYN el ECE = CW R = 1
    - SYN+ACK configuran ECE = 1 y CWR = 0.
- En datagramas IP se configuran las flags ECT (0) = 01 o ECT (1) = 10 (ECN-Capable Transport). Indica que los extremos soportan ECN si el router avisa.

- Si el router nota que viene un paquete para un buffer que se está por congestionar (para ello tiene un umbral) se fija en los bits de IP (los del punto de arriba) si alguno esta activado.
    - Si no tiene el flag, lo descarta
    - Si tiene la marca, marca el otro bit (hay uno que esta 0, ese lo pone en 1) y queda 11.
    - Este segmento lo sirve el receptor y le avisa al emisor (el que tiene que enviar menos) por medio de ECE
    - El emisor también configura a CWR en 1 para notificar que reacciono.

Control de Flujo, Control de Errores y Control de Congestión

- La capacidad de envío será MIN(Congestión, Flujo, Errores)