

Redes y comunicaciones

Resumen teorías - Capa de transporte

Clase 1 - Introducción a la capa de transporte	3
Funcionalidad de la capa de transporte	4
PDUs	5
Multiplexación del transporte	6
Selección del Protocolo de Transporte	6
UDP	7
Headers y servicios	7
Datagrama	8
Calculo del checksum	9
TCP	10
Headers y servicios	10
Segmento	11
Funcionamiento	12
Establecimiento de la conexión	12
3 way handshake	13
Cierre de la conexión	13
Orientado a streams	14
Números de secuencia/ACK	15
Control de errores	15
Diagrama de estados	16
Clase 2 - Control de errores	19
Escenarios	19
Errores posibles	19
Stop & Wait (No lo usa TCP)	20
Sin errores	21
Segmentos perdidos:	22
Errores checksum/CRC	22
Segmentos duplicados	23
ACK duplicados	24
Conclusiones S&W	25
Pipelining/Sliding Window	25
Notas sobre buffers Tx y Rx	26
Go-back N	28
Emisor	29
Receptor	30

Ejemplo en acción 😎	31
Ventana inicial	31
Envíos	32
Ventana actualizada	33
Segmento perdido	34
ACK acumulados	35
Buffer fuera de orden	36
ACK duplicados, NAK	37
RTO	38
Selective repeat	38
Emisor	40
Receptor	41
Control de errores en TCP	41
RTT dinámico	44
Timestamp	45
Clase 3 - Control de flujo en TCP	47
Servicios de TCP	47
Control de errores y de flujo	47
Control de flujo	47
Ejemplo ventana deslizante	49
Escalado de Ventana	53
Clase 4 - Control de congestión	56
Causas de congestión en la red	57
Modelo end-to-end	57
Versión vieja de TCP	57
Tahoe	58
Slow Start	58
Congestion Avoidance	59
Fast Retransmit	59
Old Tahoe vs Tahoe	60
TCP Reno	60
Fast retransmit	61
Fast Recovery	61
AIMD (Additive Increase/Multiplicative Decrease)	65
New Reno	66
Control de congestión por la red	67
ECN (Explicit Congestion Notification)	67
Funcionamiento	68

Clase 1 - Introducción a la capa de transporte

Transporte de Internet, TCP/IP, usa IP como servicio

IP provee un servicio “débil”, pero eficiente: “Best-effort” ([acá lo explica nuestro amigo ChadGPT 😎](#))

En IP los paquetes/datagramas pueden ser descartados, des-ordenados, retardados duplicados o corrompidos

Paquetes IP tienen solo dirección DST y SRC, ¿Cómo se elige la aplicación?

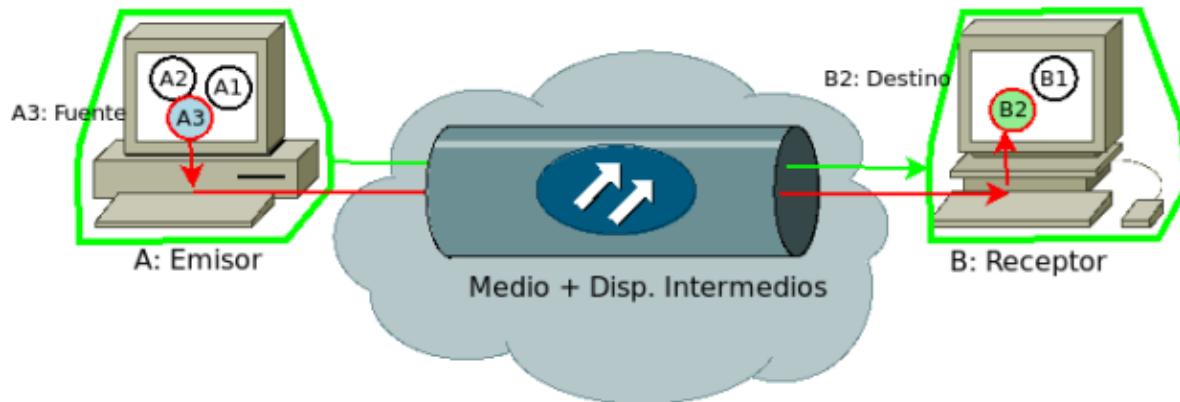
IP corre en “todos” los nodos de la red (internet) (routers y host), protocolos de transporte solo necesario en end-points (hosts)

IP: comunicación lógica HOP-BY-HOP (porque va dando saltos entre los nodos intermedios (switches/routers)), comunica hosts

Transporte: comunicación lógica HOST-TO-HOST (END-TO-END), comunica procesos

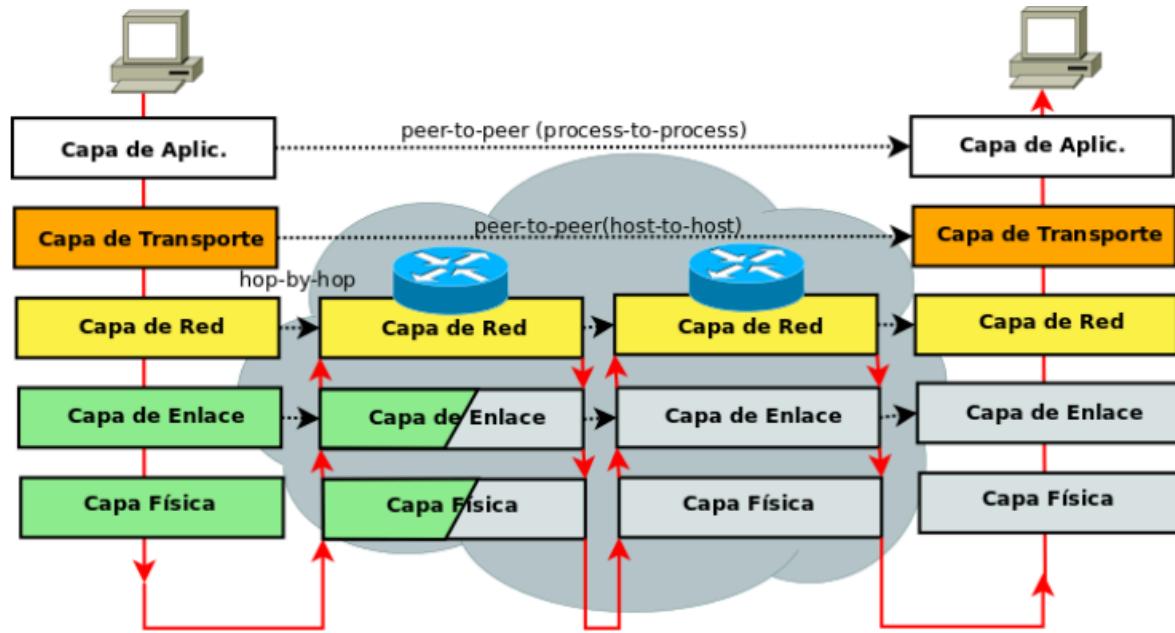
Aplicación: comunicación lógica PROCESS-TO-PROCESS (END-TO-END) comunica usuarios, agentes, etc

La capa de transporte brinda servicio a capa de aplicación y usa servicios de capa de internet (red/IP)



Red (IP) direcciona hosts, transporta al mensaje del transporte hasta el host (dir IP)

Transporte (TCP, UDP, u otro) transporta al mensaje de aplicación hasta el proceso (puerto)



Aplicación “habla” de proceso a proceso, servicio provisto por la capa de transporte
 Transporte “habla” de host a host, donde los paquetes (segmentos) son llevados por la red

Funcionalidad de la capa de transporte

Encapsulación, define PDU donde se envía los mensajes de la aplicación

MUX/DEMUX process-to-process (puertos, Ports)

Soporte de datos de tamaños arbitrarios

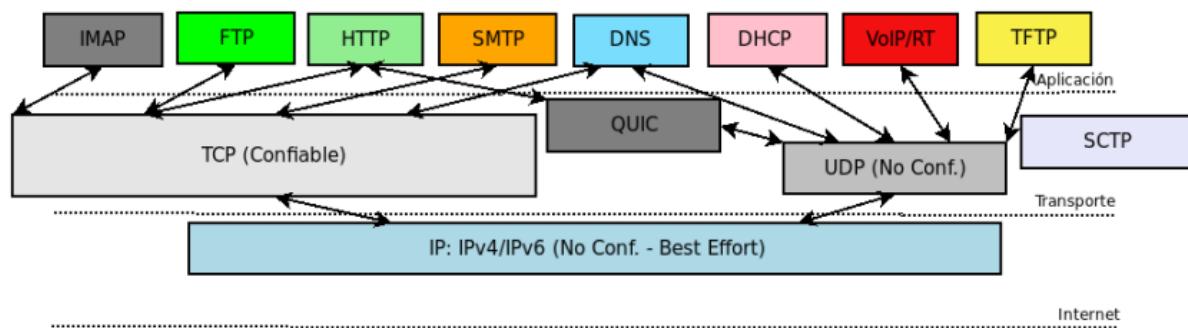
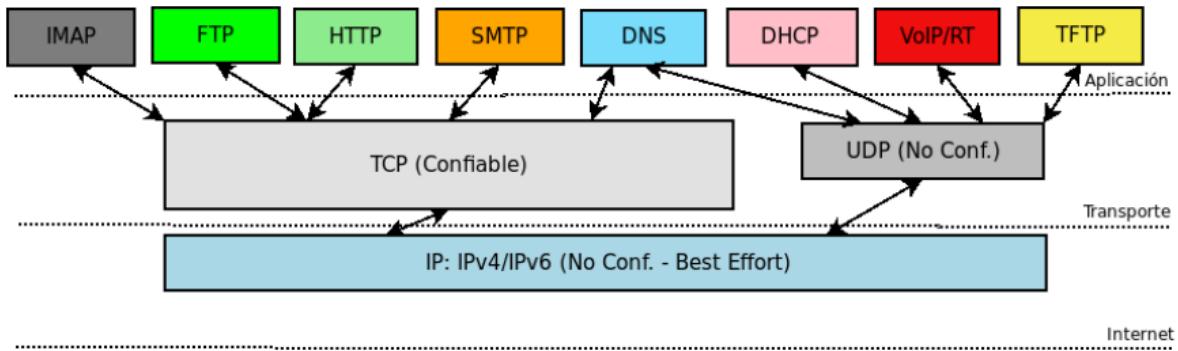
Control y Detección de Errores, pérdida, duplicación, se corrompen

¿Cómo enviar info sobre la red de acuerdo al estado de la misma? y ¿Cuándo y Cómo una aplic. debe enviar datos?:

- Control de flujo
- Control de congestión

Dos modelos básicos:

- Modelo Confiable: TCP
- Modelo NO Confiable: UDP



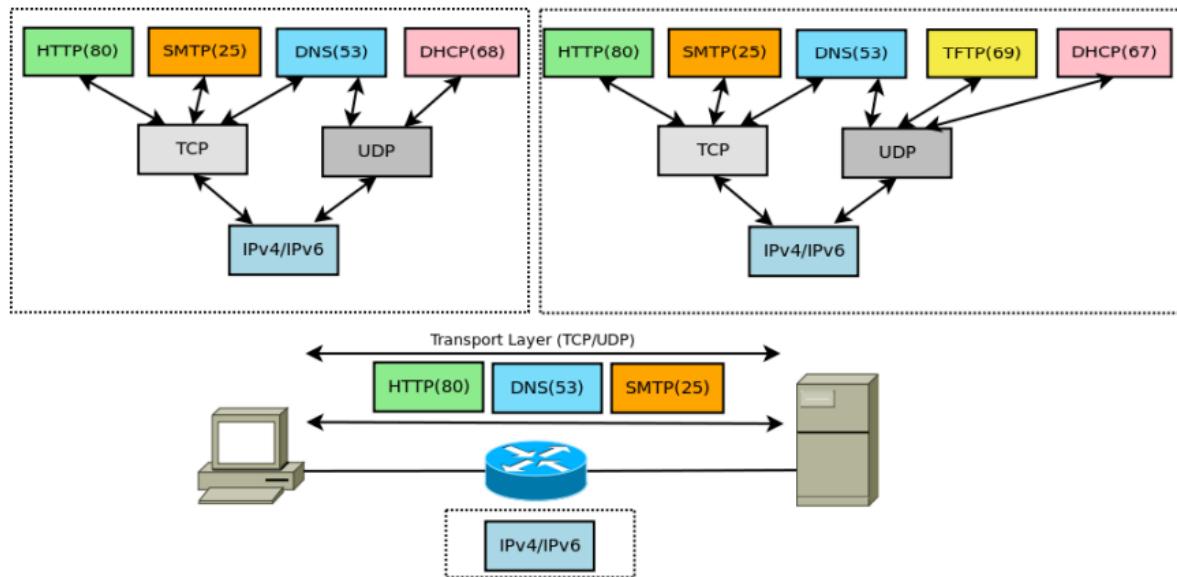
PDUs

IP (red): paquete o datagrama

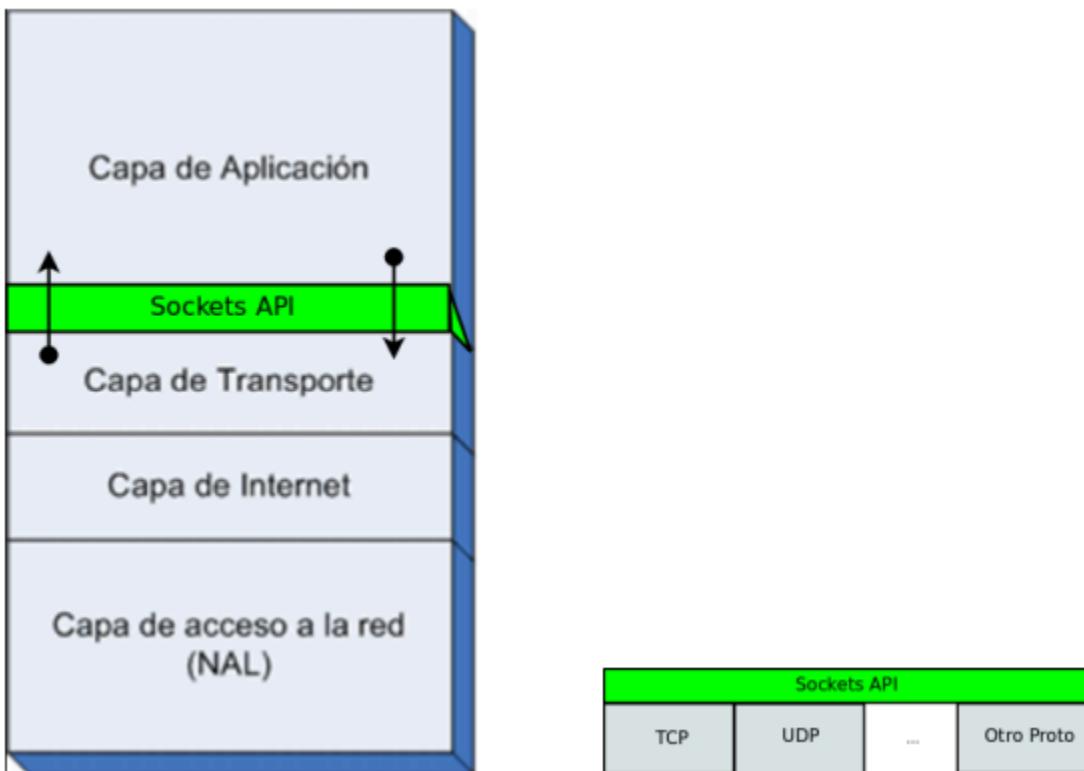
Transporte:

- De forma general: segmento
- UDP: datagrama

Multiplexación del transporte



Selección del Protocolo de Transporte



La aplicación de acuerdo a como esta programada selecciona el transporte
El acceso a los servicios de transporte se hace mediante la API Network socket

UDP

User Datagram Protocol (RFC-768)

Protocolo Minimalista. Menor Overhead

Características de IP: best-effort

Orientado a Packets/Datagramas (mensajes auto-contenidos)

PDU: Datagrama (Por coherencia con nivel Transporte se suele llamar Segmento)

Solo provee MUX/DEMUX (a través de los números de puerto) y detección de algunos errores(UDP4, puede desact.)

No incrementa Overhead end-to-end

No requiere establecimiento de conexión

Servicio FDX (full-duplex)

Aplicaciones: video/voz streaming/TFTP/DNS/Bcast/Mcast, transporte de transporte (QUIC)

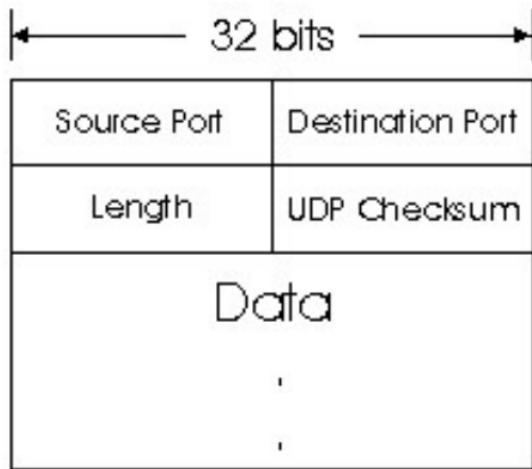
Headers y servicios

El encabezado IP provee: Ruteo, Fragmentación, Detección de algunos errores

El encabezado UDP provee: MUX/DEMUX de aplicación, Detección de errores (no obligatorio para UDP4, UDP sobre IPv4)

IP indica que lleva UDP con el código de protocolo 17

Datagrama



Campos:

- Puertos: MUX/DEMUX
- Longitud: UDP HDR + Payload
- Checksum
 - Cálculo usando Ca1(complemento a 1)
 - Opcional (0 = Sin checksum)
 - Calculado HDR + PseudoHDR + Payload
 - PseudoHDR: IP.SRC + IP.DST + Zero + IP.PROTO + UDP.LENGTH
 - PseudoHDR: protección contra paquetes mal enrutados
 - Aplicaciones de LAN por eficiencia lo podrían deshabilitar
 - Si tiene error se descarta silenciosamente

Calculo del checksum

```
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=13, LEN-PH=13, CKSUM=0
 2328 0007 000D 000D

DATA      5445 5354 0A00

 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 00001101 = 13
 00000000 00001101 = 13
 00101010 01000101
 00101001 01010100
 00001010 00000000
-----
~11101111 00000111      EF07 ~EF07 = 10F8
 00010000 11111000

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 (RFC-793)

Protocolo confiable, ordenado, con buffering, control de errores, de flujo y de congestión

Orientado a streams (secuencia de bytes, \equiv archivo)

PDU: Segmento, una porción del stream de bytes

Provee MUX/DEMUX

Incrementa Overhead end-to-end para ofrecer confiabilidad

Requiere establecimiento de conexión (y cierre)

Servicio FDX

Aplicaciones: transferencia de archivos, FTP/HTTP/SMTP/acceso remoto(SSH, telnet,...)/Unicast

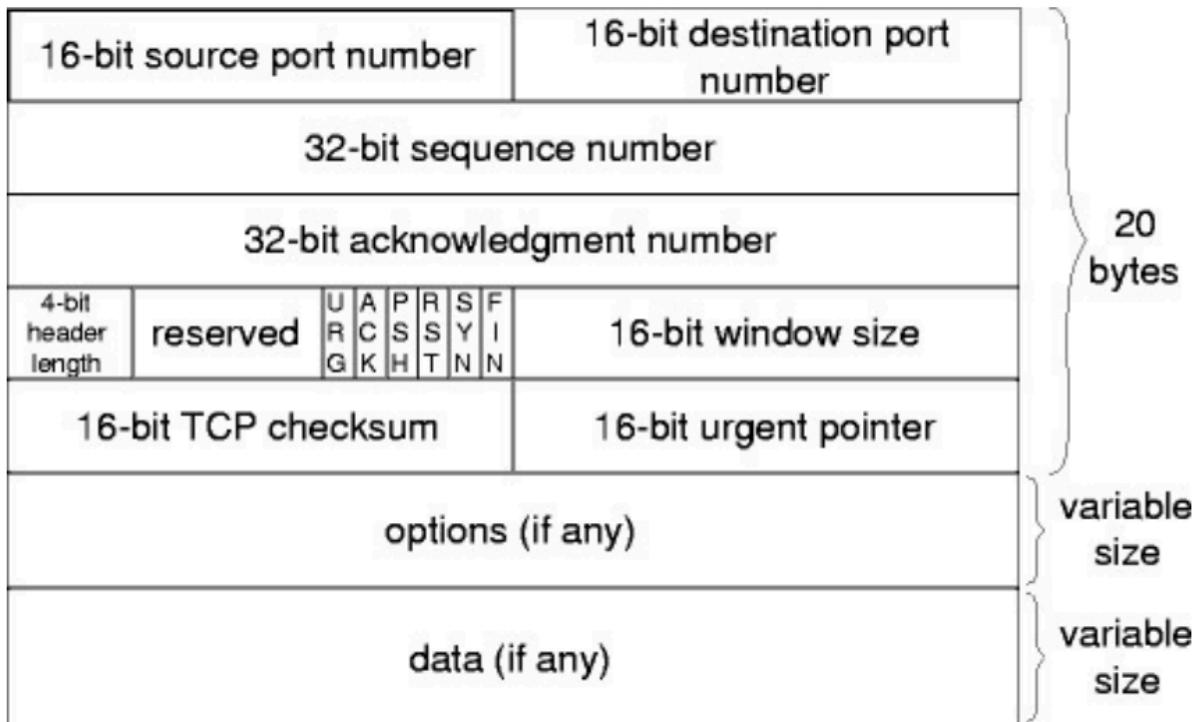
Headers y servicios

El encabezado IP provee: Ruteo, Fragmentación, Detección de algunos errores, pero no hace controles

El encabezado TCP provee: MUX/DEMUX de aplicación, detección de errores (obligatorio) y controles para hacerlo confiable

IP indica que lleva TCP con el código de protocolo 6

Segmento



Campos:

- Puertos: MUX/DEMUX
- No tiene Longitud total, si de HDR LEN (variable, max 60B Unit=4B)
- Total LEN se computa para PseudoHDR, no viaja en el segmento
- El tamaño del segmento se calcula dentro del datagrama IP
- Checksum:
 - Cálculo con Ca1(complemento a 1)
 - Obligatorio, calculado, igual que UDP
 - Si tiene error podría pedir retransmisión, implementación de TCP descarta y espera RTO (Retransmission Timer)
- Necesidad de manejar Timers, RTO (tmout por cada segmento) (implementaciones lo manejan más eficientemente)
- Campos de Sesiones: Flags: SYN(Synchronize), FIN(Finish), RST (Reset)
- Campo de Detección de Errores: Checksum
- Campos de Control de Errores: ACK, sequence number (#Seq), acknowledgement number (#Ack)

- Campo de Control de Flujo: a los de control de errores se agrega, Window size (dice el tamaño máximo de datos que puede recibir en una sola entrega)
- Campos de Control de Congestión: se agregan flags si participa la red

Funcionamiento

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

- La aplicación puede enviar 3000 bytes en un write y TCP lo podría enviar en 3 segmentos separados de 1000 bytes c/u
- La aplicación puede enviar 100 bytes y luego otros 200 y TCP esperar para enviarlos todos juntos en un segmento de 300 bytes
- La aplicación puede intentar leer 200 bytes del buffer y TCP solo entregar 150 bytes y luego el resto

Requiere mantener “recursos” en cada extremo para los controles:

- Buffers de Rx y Tx
- Timer(s) RTO
- Variables: datos enviados, no confirmados, retransmisiones, umbrales, RTT, etc
- Estado de la conexión

Establecimiento de la conexión

Flags: SYN (Synchronize), ACK (Acknowledge) y RST (Reset)

3Way-Handshake (3WH)

En el 3 segmento se puede enviar info

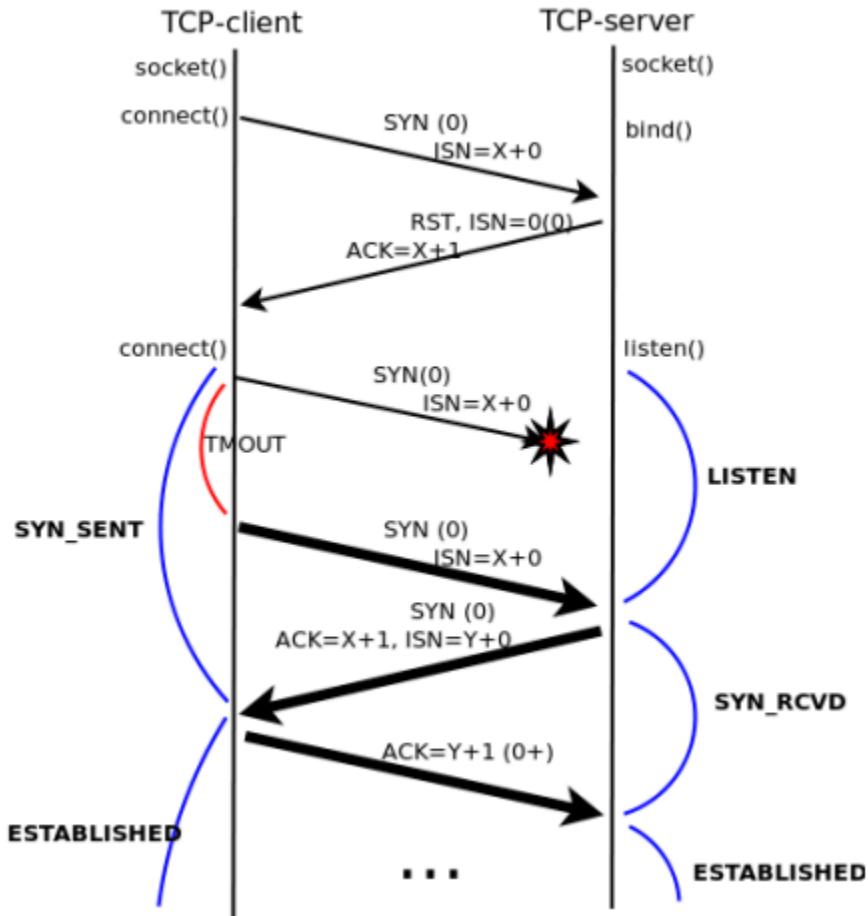
El ISN (Initial Sequence Number), se utiliza un contador que se incrementa cada 4 mseg

RST si no hay proceso en estado LISTEN

Open Pasivo (servidor) y Activo (cliente)

Open simultáneo

3 way handshake



Cierre de la conexión

Flags: FIN (Finish), ACK y RST

4Way-Close (4WC)

Posibilidad de Half-Close (si un extremo manda un segmento con FIN, no podrá enviar más datos, pero sí recibir)

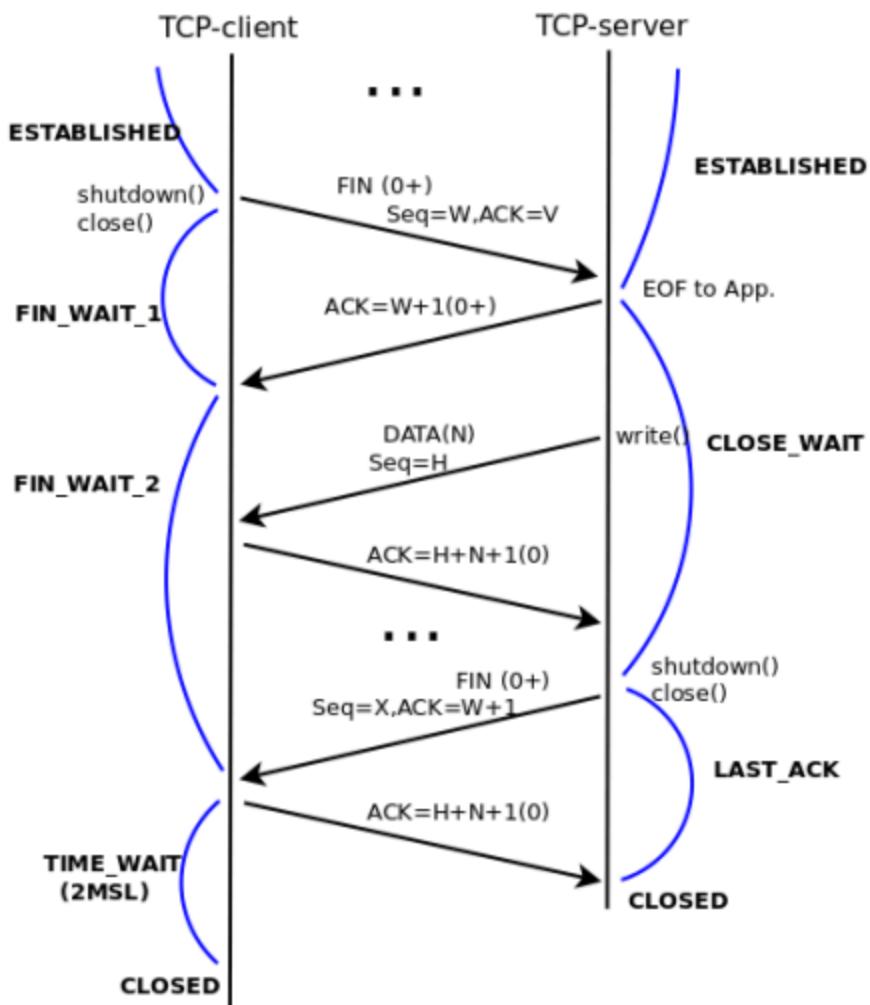
Podría cerrarse en 3WC

Espera en TIME WAIT, 2MSL (aprox. 2*2min)

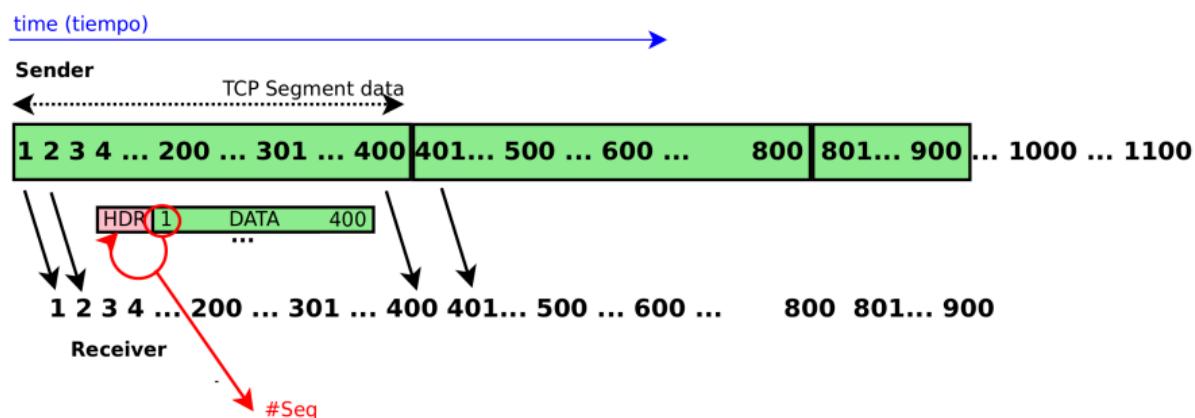
Evitar con `SO_REUSEADDR`

Cierre incorrecto con RST

Close simultáneo



Orientado a streams



Números de secuencia/ACK

Time	172.20.1.1	172.20.1.100	Comment
0.000	(41749) SYN	→ (11111)	Seq = 0
0.001	← (41749) SYN, ACK	(11111)	Seq = 0 Ack = 1
0.001	(41749) ACK	→ (11111)	Seq = 1 Ack = 1
90.730	(41749) PSH, ACK - Len: 5	→ (11111)	Seq = 1 Ack = 1
90.730	(41749) ACK	→ (11111)	Seq = 1 Ack = 6
100.15	(41749) PSH, ACK - Len: 16	→ (11111)	Seq = 1 Ack = 6
100.15	← (41749) ACK	(11111)	Seq = 6 Ack = 17
104.58	(41749) PSH, ACK - Len: 5	→ (11111)	Seq = 6 Ack = 17
104.58	(41749) ACK	→ (11111)	Seq = 17 Ack = 11
112.29	(41749) PSH, ACK - Len: 6	→ (11111)	Seq = 17 Ack = 11
112.29	← (41749) ACK	(11111)	Seq = 11 Ack = 23
114.89	(41749) PSH, ACK - Len: 6	→ (11111)	Seq = 23 Ack = 11
114.89	← (41749) ACK	(11111)	Seq = 11 Ack = 29
120.62	(41749) FIN, ACK	→ (11111)	Seq = 29 Ack = 11
120.62	← (41749) FIN, ACK	(11111)	Seq = 11 Ack = 30
120.62	(41749) ACK	→ (11111)	Seq = 30 Ack = 12

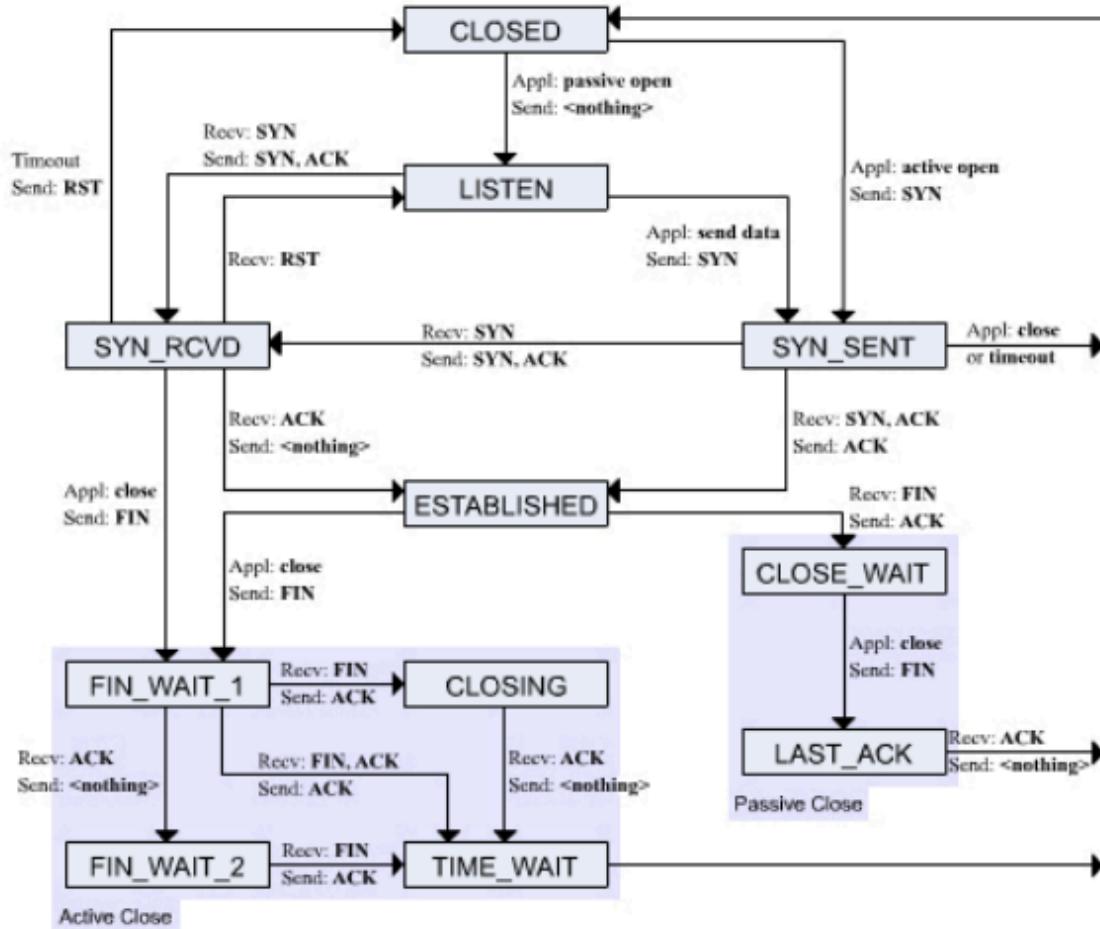
Control de errores

Errores que pueden existir en IP:

- Pérdida de paquetes: descartados
- Des-ordenados, retardados
- Duplicados
- Corrompidos

TCP intenta solucionarlos con el control de errores que implementa

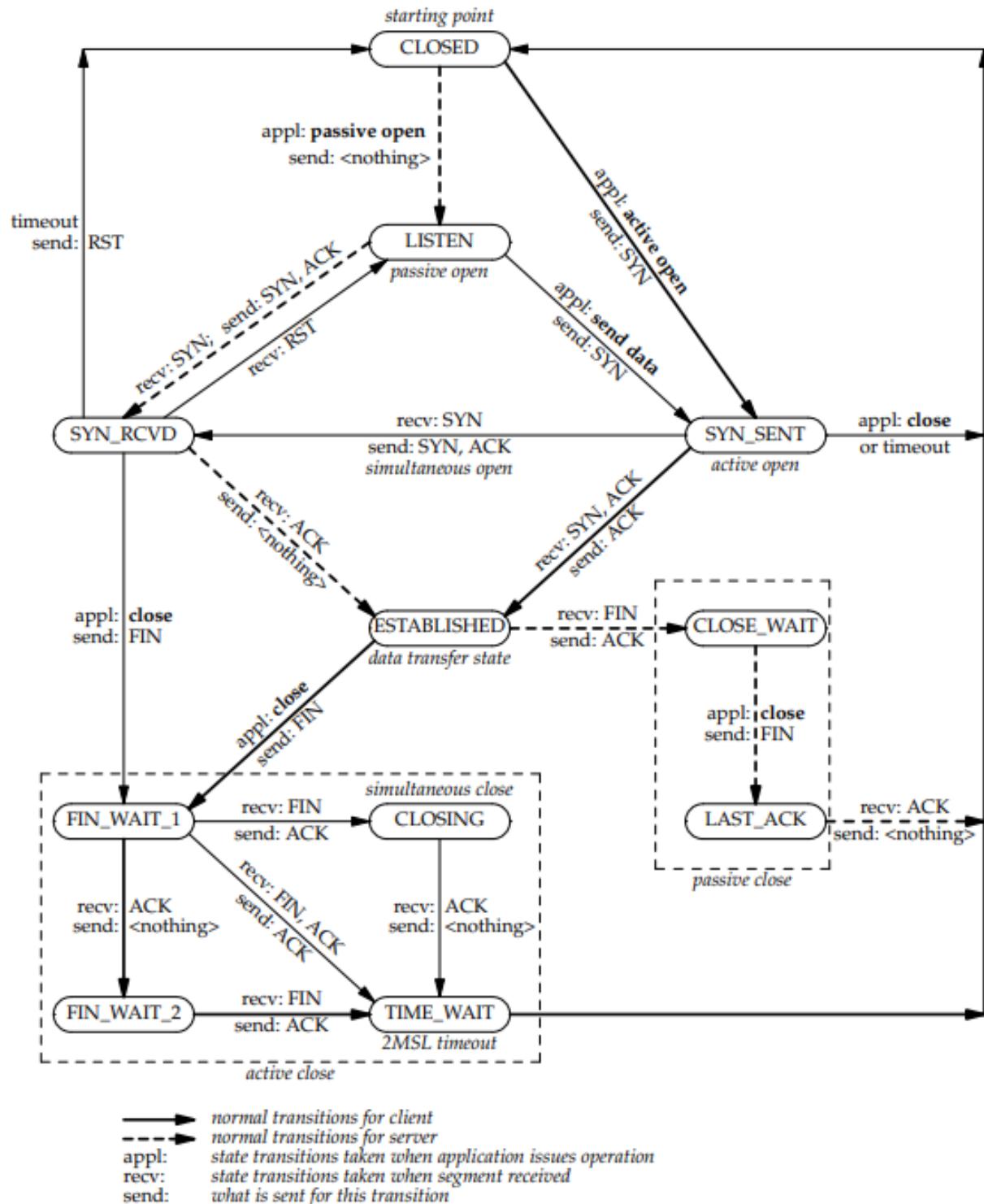
Diagrama de estados



A connection progresses through a series of states during its lifetime. The states are: LISTEN, SYN-SENT, SYN- RECEIVED, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT, and the fictional state CLOSED. CLOSED is fictional because it represents the state when there is no TCB, and therefore, no connection. Briefly the meanings of the states are:

- LISTEN** represents waiting for a connection request from any remote TCP and port.
- SYN-SENT** represents waiting for a matching connection request after having sent a connection request.
- SYN- RECEIVED** represents waiting for a confirming connection request acknowledgment after having both received and sent a connection request.
- ESTABLISHED** represents an open connection, data received can be delivered to the user. The normal state for the data transfer phase of the connection.
- FIN-WAIT-1** represents waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.
- FIN-WAIT-2** represents waiting for a connection termination request from the remote TCP.
- CLOSE-WAIT** represents waiting for a connection termination request from the local user.
- CLOSING** represents waiting for a connection termination request acknowledgment from the remote TCP.
- LAST-ACK** represents waiting for an acknowledgment of the connection termination request previously sent to the remote TCP (which includes an acknowledgment of its connection termination request).
- TIME-WAIT** represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request.
- CLOSED** represents no connection state at all.

A TCP connection progresses from one state to another in response to events. The events are the user calls, OPEN, SEND, RECEIVE, CLOSE, ABORT, and STATUS; the incoming segments, particularly those containing the SYN, ACK, RST and FIN flags; and timeouts.



Clase 2 - Control de errores

Se requiere un mecanismo de control sobre un canal no confiable, por ejemplo IP (best-effort)

Se realiza con ARQ: Automatic Repeat reQuest/Automatic Repeat Query, End-to-End

Utiliza números de secuencia y confirmaciones para validar que los datos se recibieron OK (en orden y sin errores)

Se confirman los datos que se reciben (ACK)

Requiere mantener timer: RTO (Retransmission Timeout) o TMOUT

Requiere mantener buffer de segmentos a transmitir TxBuf y posiblemente para segmentos recibidos RxBuf

Puede corroborar mediante checksum/CRC errores de bits en los datos recibidos

Escenarios

Para simplificar, en varios ejemplos se supone solo un emisor y un receptor, solo se transmite en un sentido datos y no se requiere elegir origen ni destino

Para simplificar se analiza primero un RTT (Round Trip Time) y BW (Bandwidth digital) fijos

Se comienza con un esquema S&W (Stop&Wait) hasta llegar al GBN (Go-Back-N) y SR (Selective-Repeat) como TCP

Esquema S&W solo a modo de introducción a los problemas y como se solucionan, aunque ineficiente

Control de errores End-to-End, abstracción de los nodos intermedios, la red no “colabora”

Errores posibles

En capa de red o inferior

Errores se pueden generar en los extremos o en la red

Se pierden datos y/o ACKs (por descartes o errores en la red)

Se duplican datos y/o ACK, por retransmisiones o por errores en equipos (extremos o intermedios, red)

Se desordenan por usar caminos múltiples o errores de procesamiento en equipos

Se corrompen datos y/o ACKs

Existe delay (retardos) mayores a 1 RTT y RTT podría ser variable

Stop & Wait (No lo usa TCP)

Emisor manda un segmento numerado y espera confirmación

Emisor arranca un RTO al enviar el segmento, si no recibe ACK (confirmación) re-envía

Receptor, cada vez que recibe segmento, confirma indicando (Num. por segmentos o bytes, en el ejemplo por bytes)

Se puede confirmar el actual o el que se espera (en el ejemplo el que se espera)

Receptor, si tiene errores, descarta o podría confirmar indicando qué Num.(#) espera.

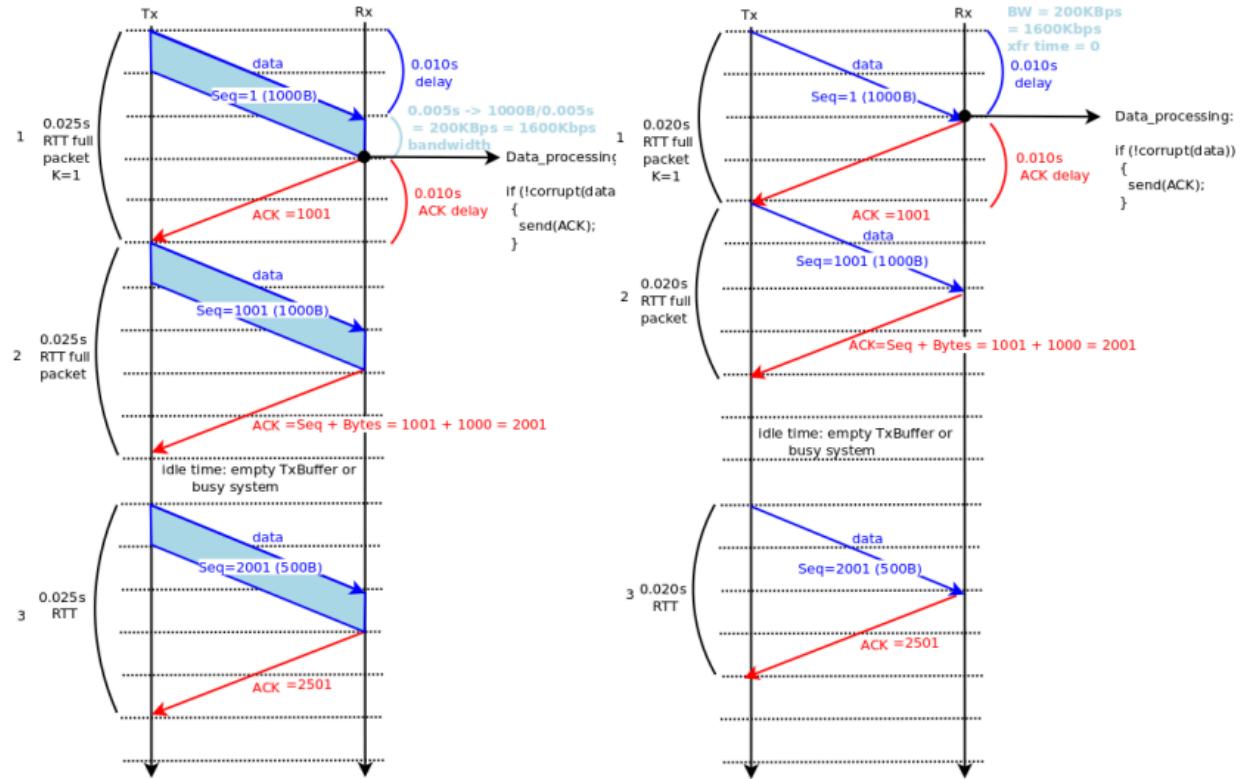
Funcionan como NAK (No Acknowledge)

Emisor si recibe confirmación envía nuevo segmento, si tiene en el TxBuf , e inicia nuevo RTO (en caso que la capa superior dejo datos para enviar en el buffer)

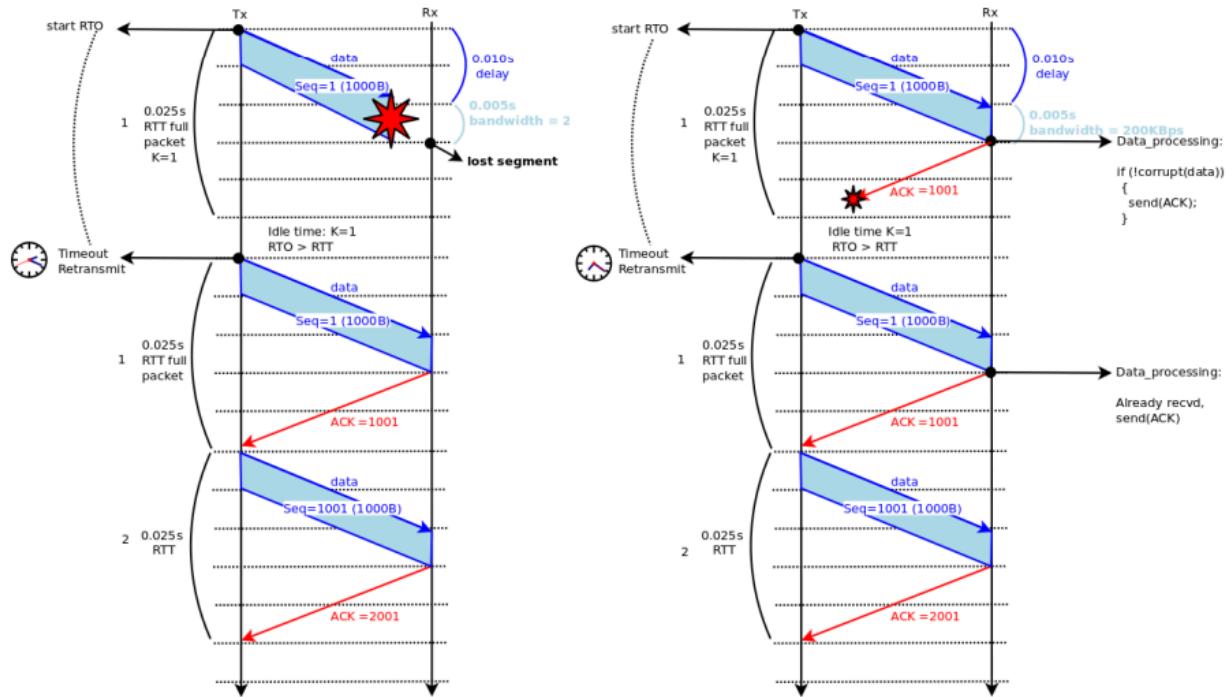
Emisor descarta confirmaciones fuera de secuencia (out-of-order) , e.g. atrasadas

Si recibe confirmación del anterior (similar NAK) podría re-enviar, en caso de ACKs > N,
 $N \geq 2$

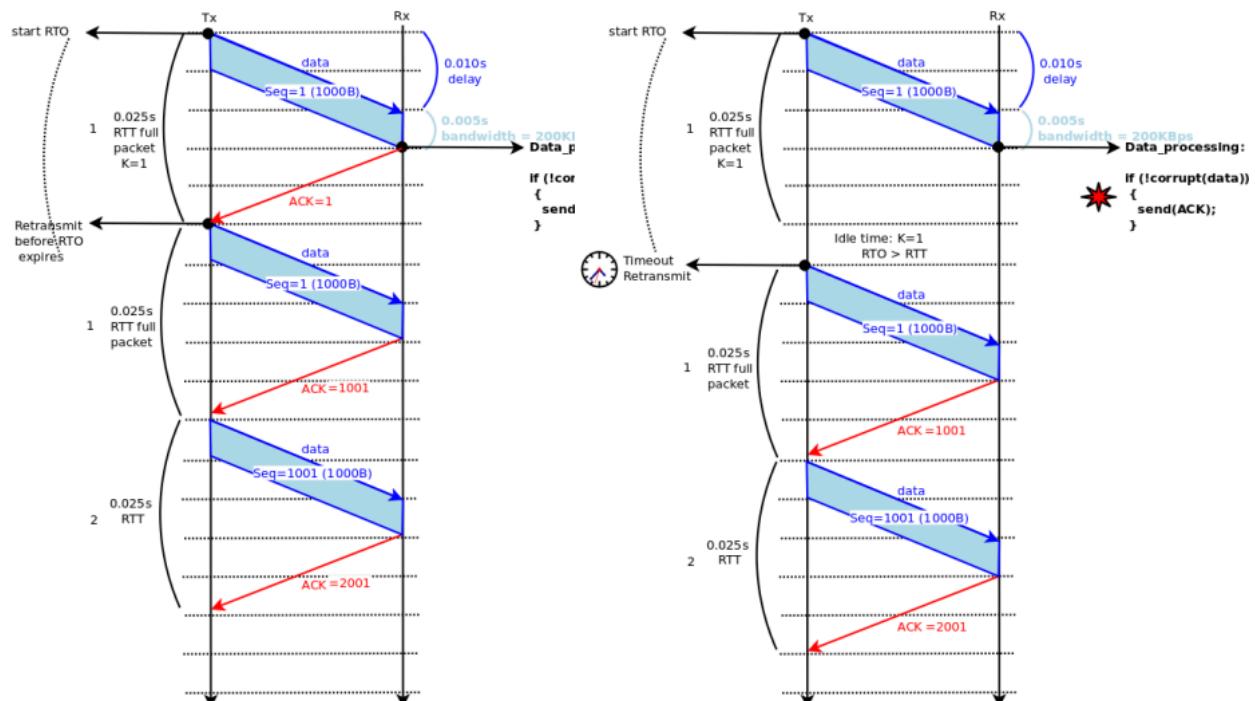
Sin errores



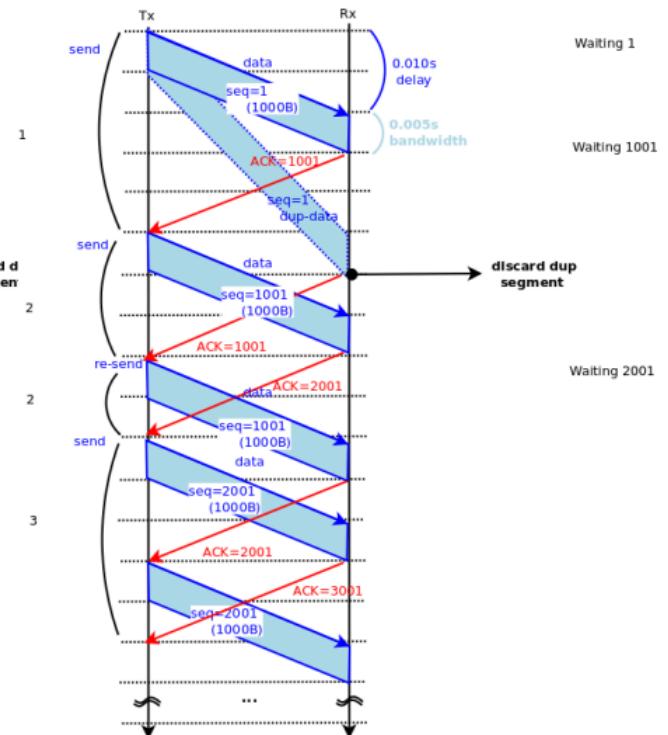
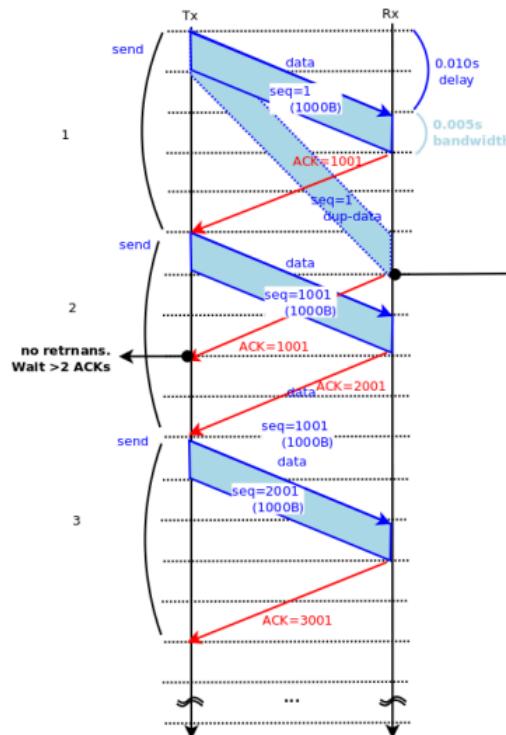
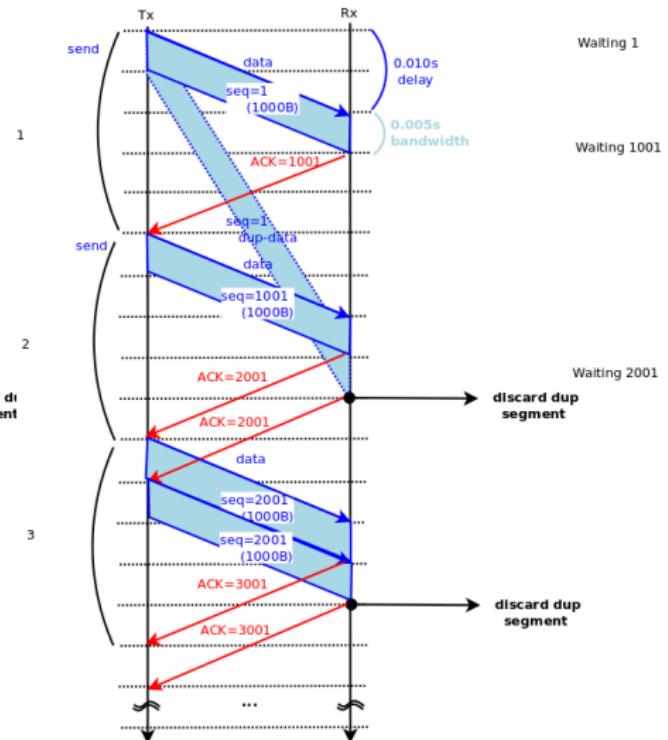
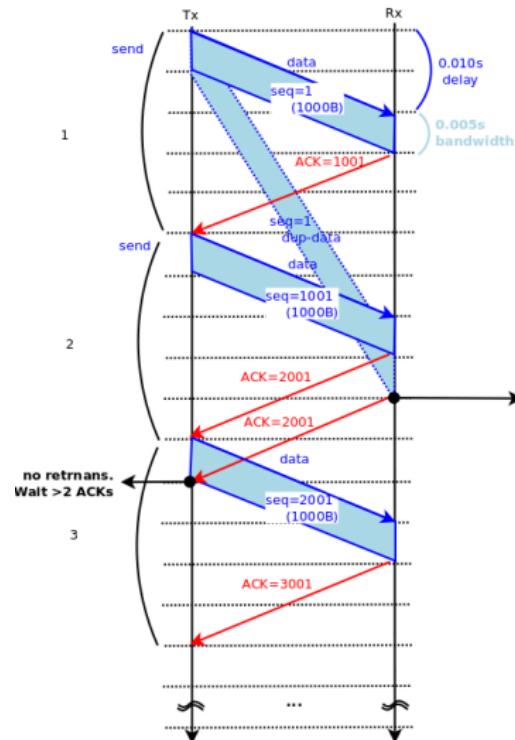
Segmentos perdidos:



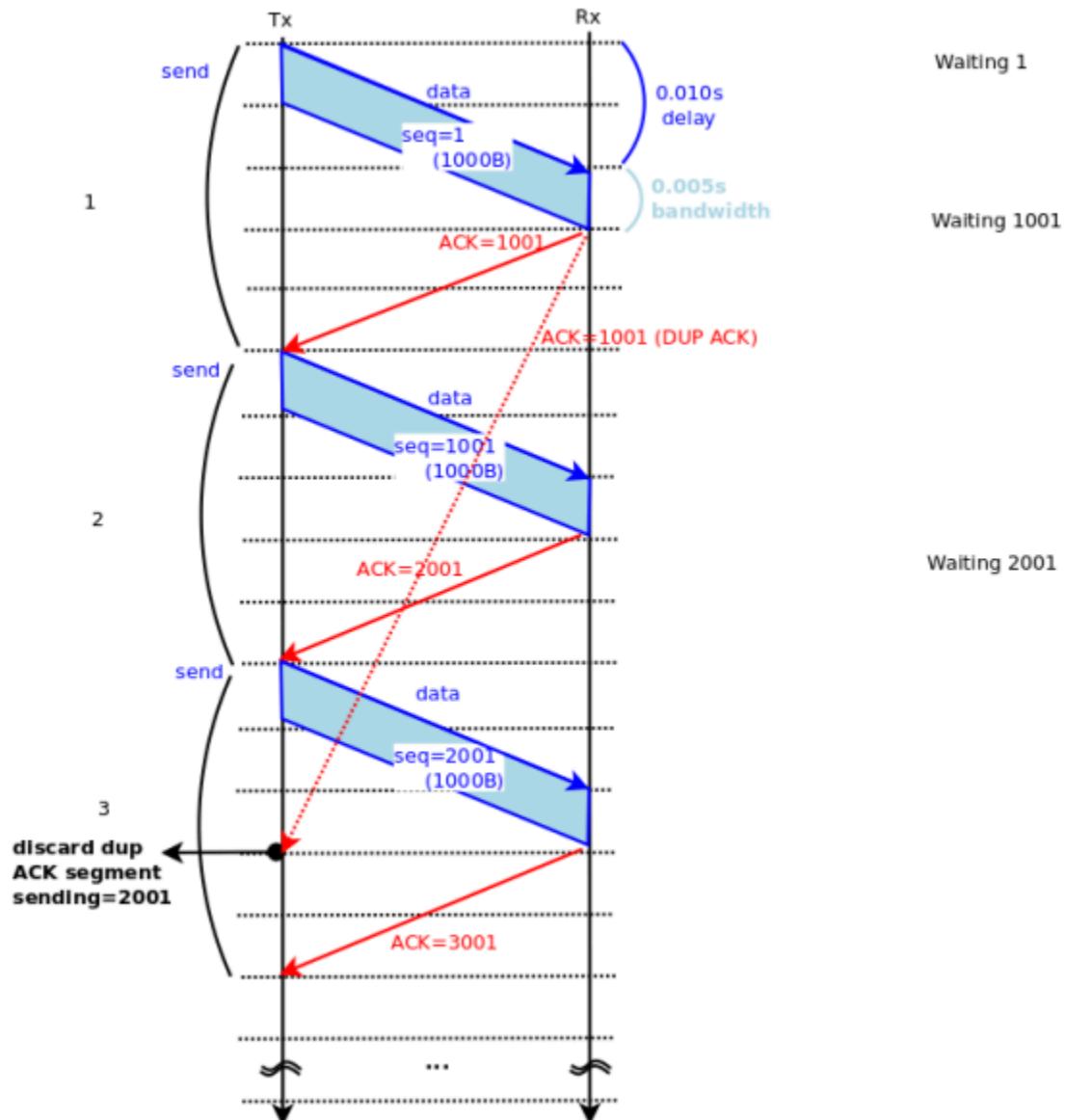
Errores checksum/CRC

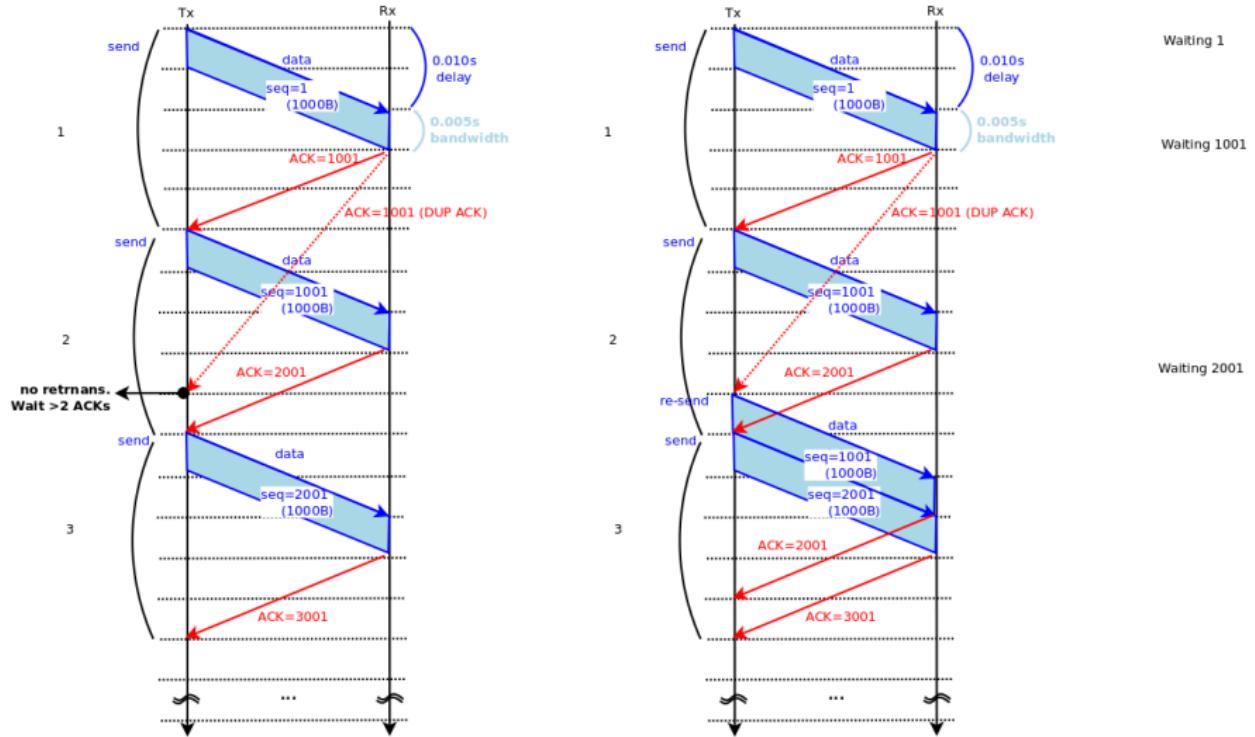


Segmentos duplicados



ACK duplicados





Conclusiones S&W

S&W+SeqNumbers (#) 0..M – 1 en datos/ACK recupera los problemas en la red

Problema: Sequence Number Wrap Around, PAWS (Protection Against Wrapping Sequence) (aumentar el espacio de números de secuencia, puede usarse time-stamping)

El sistema es ineficiente, envía un dato por vez, ventana de transmisión/recepción: $K = 1$, $W = 1$. No se envía el próximo mensaje hasta que no se confirma el que se envió

Sistema simple: no optimiza producto: Delay, Bandwidth: $BDP = D \times B$, $D = RTT$ o $D = RTT + L$

Cada vez que envía un segmento requiere arrancar un timer: RTO

Si no recibe confirmación se vence el timer y se retransmite

Pipelining/Sliding Window

Pipelining: permitir enviar múltiples segmentos o paquetes por RTT (ráfaga) sin aún haber recibido confirmaciones, paquetes “in-flight”

Emisor debe saber la cantidad de segmentos o bytes que se puede enviar sin aún recibir confirmación se llama Ventana, notado como K o W , $W = n$, donde $n > 1$

Requiere buffering de Tx, TxBuf (lo que deja la capa superior y aún no se confirmó) y de Rx, RxBuf (lo que se va recibiendo hasta entregar a la capa superior)

Por cada mensaje enviado se podría iniciar un timer de retransmisión, RTO (no escala)

Se mantiene un RTO por ráfaga, para el segmento más viejo no confirmado

Receptor debe confirmar los segmentos recibidos (se confirma indicando lo que se espera)

Confirmaciones deslizan la ventana sobre TxBuf habilitando nuevos segmentos a transmitir

Podría generarse confirmaciones negativas: NAK (NO-Acknowledge), implícitas o explícitas. (TCP usa implícitas con varios ACK dups)

Algoritmo más eficiente, óptimo, si llena el pipe

Los números de secuencia, $0..M - 1$, si se numeran en módulo, no necesariamente son $M = K$, Sucede que $K < M$. Puede ser por bytes o por PDU (segmento). (TCP lo hace por bytes)

Aumentar M (números de secuencia) permite ser tolerante a fallas de ACK delayed

Alternativas:

- Go-back-N (confirma en orden)
- Selective-Repeat (confirma fuera de orden)

Notas sobre buffers Tx y Rx

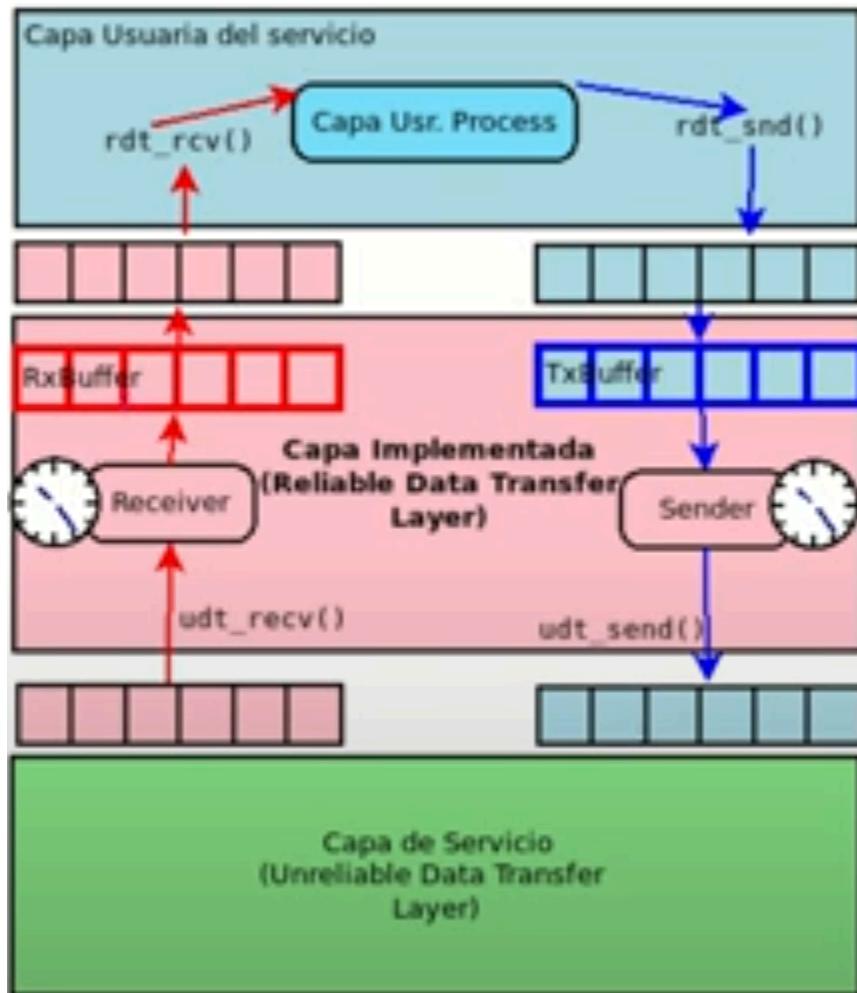
El tamaño de la ventana (W) va a ser como máximo igual al tamaño de el menor de estos buffers

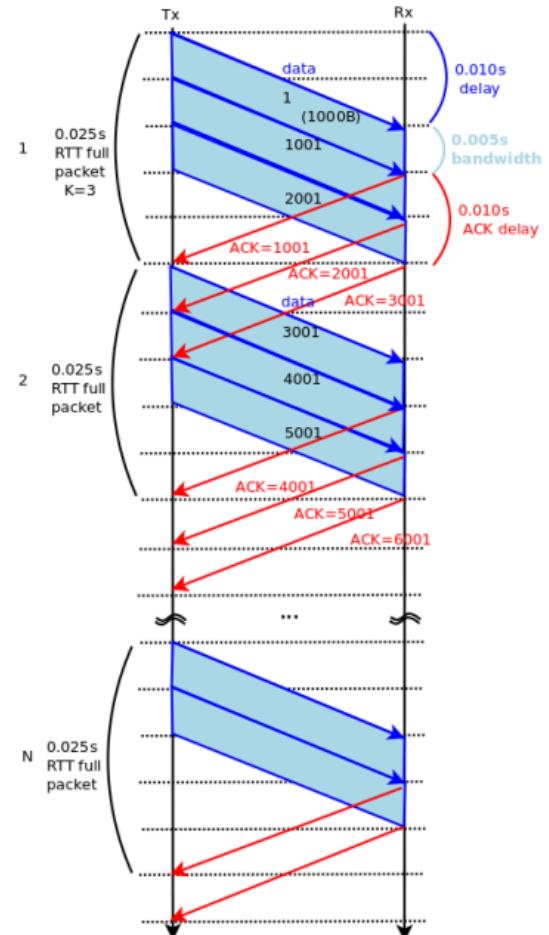
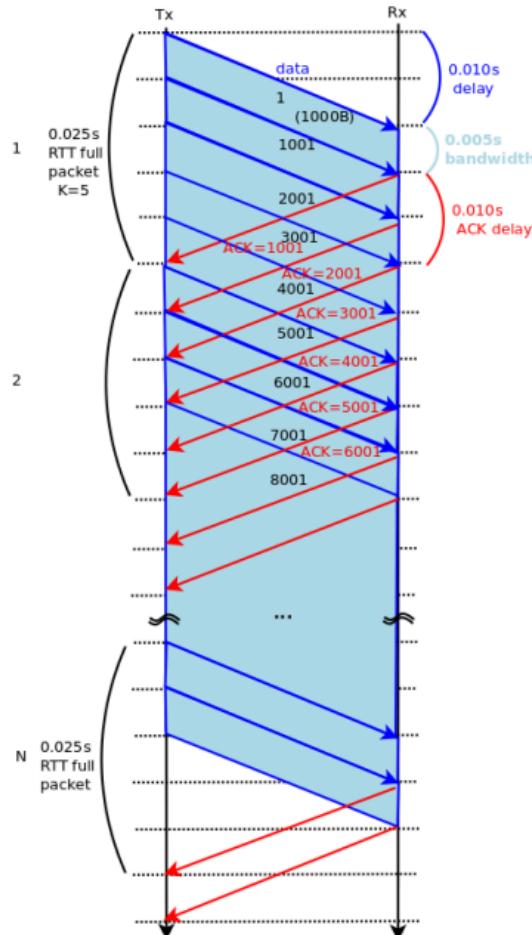
El buffer Tx es un buffer manejado por el sender donde quedan los datos que llegan de la capa de aplicación

Los datos se mantienen en Tx por si hay un error en la transmisión y se deben retransmitir (como máximo se va a retransmitir un segmento W espacios atrás)

El buffer Rx es un buffer manejado por el receiver donde quedan los datos que llegan de la capa de red

Los datos se mantienen en Rx para poder reordenarlos, en caso de que hayan llegado desordenados, antes de mandarlos a la capa de aplicación





Go-back N

Se tiene una ventana “estática” de tamaño $W=n$, $n>1$

Numeración de segmentos, se realiza en módulo M , $W \leq (M - 1)$

Go-back N 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 (puedo confirmar los segmentos 0, 1 y 2 diciendo que espero el 3)

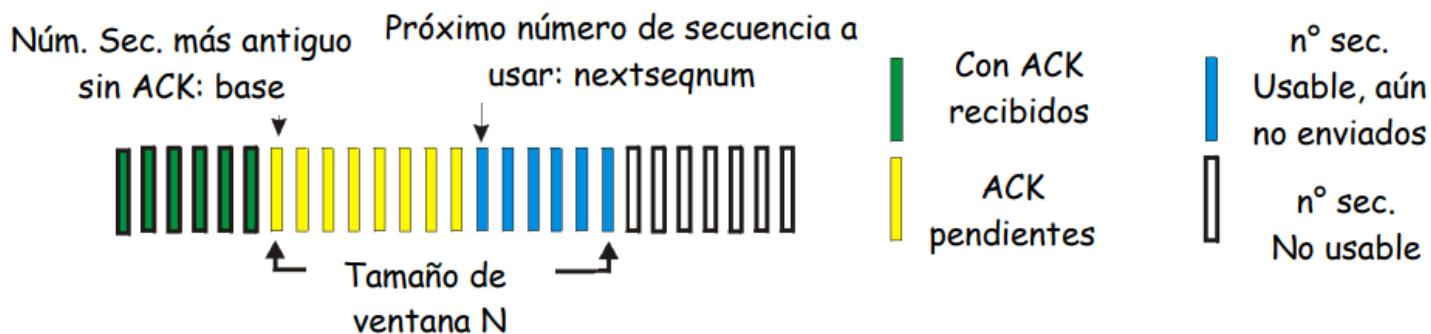
Confirmar cada segmento tiene ventajas ante la pérdida de los ACK

Se puede retransmitir ante un timeout o un NAK. Requiere buffering extra en el emisor, no se pueden descartar del buffer de comunicación con la capa de aplicación

Si se retransmite ante un timeout se hace desde N hacia adelante (si se pierde un segmento del medio se retransmiten todos los posteriores igualmente)

El emisor puede mantener solo un timer para el segmento más viejo enviado y aún no confirmado. Si este expira, retransmite todos los no confirmados. Si llega una confirmación en orden arranca un nuevo timer. Ejemplo: si mando los segmentos 0, 1, 2 y 3 tengo un timer para el 0, cuando confirman el 0, arranco un nuevo timer para el 1 y así siguiendo. Si alguna de las confirmaciones no llega, se retransmiten todos los no confirmados (a partir de donde estaba el timer, ya que no se admiten ACK desordenados)

Emisor transmite tantos segmentos o bytes que admite W de acuerdo a lo que tiene en el buffer dejados por la capa superior



Emisor

Eventos en el lado del Emisor (Tx):

- Si tiene datos en el buffer TxBuf y la ventana lo permite, $W > \text{in flight segments}$, los transmite
- Si no tiene RTO activo arranca un nuevo RTO para el primero que manda
- Si recibe un ACK en orden, desliza/actualiza la ventana, si hay segmentos "in-flight" re-arranca nuevo RTO, sino lo descarta
- Si vence RTO, re-envía todo a partir del segmento más viejo aún no confirmado

A diferencia de S&W, más de un segmento "in-flight", más de un ACK "in-flight"

Receptor

Eventos en el lado del Receptor (Rx):

- Si se recibe un segmento en orden confirma por la positiva indicando el próximo que espera
- Si se recibe un segmento corrupto lo descarta como si se hubiese perdido, espera retransmisión
- Si recibe segmento fuera de orden confirma indicando que espera uno anterior (NAK):
 - Puede descartar el segmento fuera de orden, ya que será retransmitido
 - Puede buffear el fuera de orden, pero no entregar a capa superior, esperando que llegue el/los segmento/s que llena/n el hueco y luego confirmarlo

Se aprovechan segmentos de datos para confirmar: Piggy-backing

El receptor puede usar: timer de ACK, DelACK (Delayed ACK) (es un timer como el RTO pero lo tiene el receptor), para confirmar. $RTO > DelACK + RTT$ y aprovechar datos en el otro sentido

DelACK debe aprovechar piggy-backing, y confirmaciones acumulativas pero sin demorar demasiado tiempo el flujo de datos, sino podría generar retransmisiones innecesarias

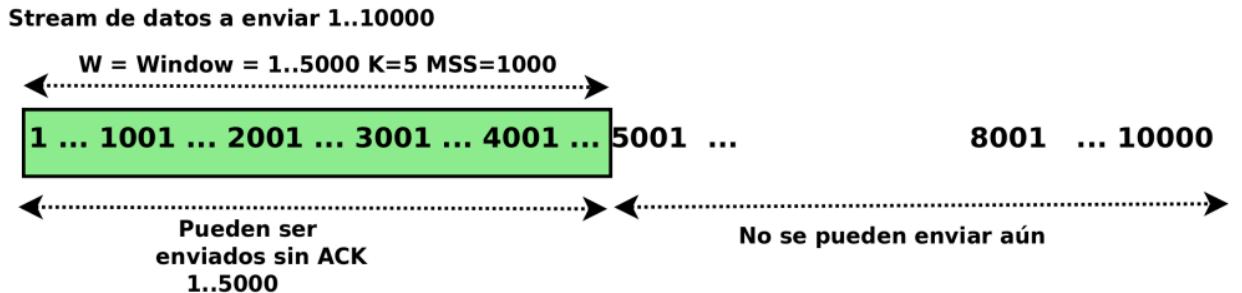
Explicación porque las diapositivas no se entienden un choto 😊:

- El DelACK se inicia cuando llega un dato desde la capa de red y sirve para retrasar el ACK lo más posible, pero sin generar una retransmisión innecesaria
- De esta manera, se espera a la llegada de un dato desde la capa de aplicación para enviar y hacer piggy-backing (al confirmar y enviar datos a la vez)
- También sirve para acumular varios ACK
- El RTO debe ser mayor que el $RTT + DelACK$ así no se causan retransmisiones innecesarias
- Si fuera menor, entre el viaje de ida y vuelta (RTT) y el retraso para confirmar (DelACK), se superaría el RTO, por lo que el sender volvería a transmitir el segmento

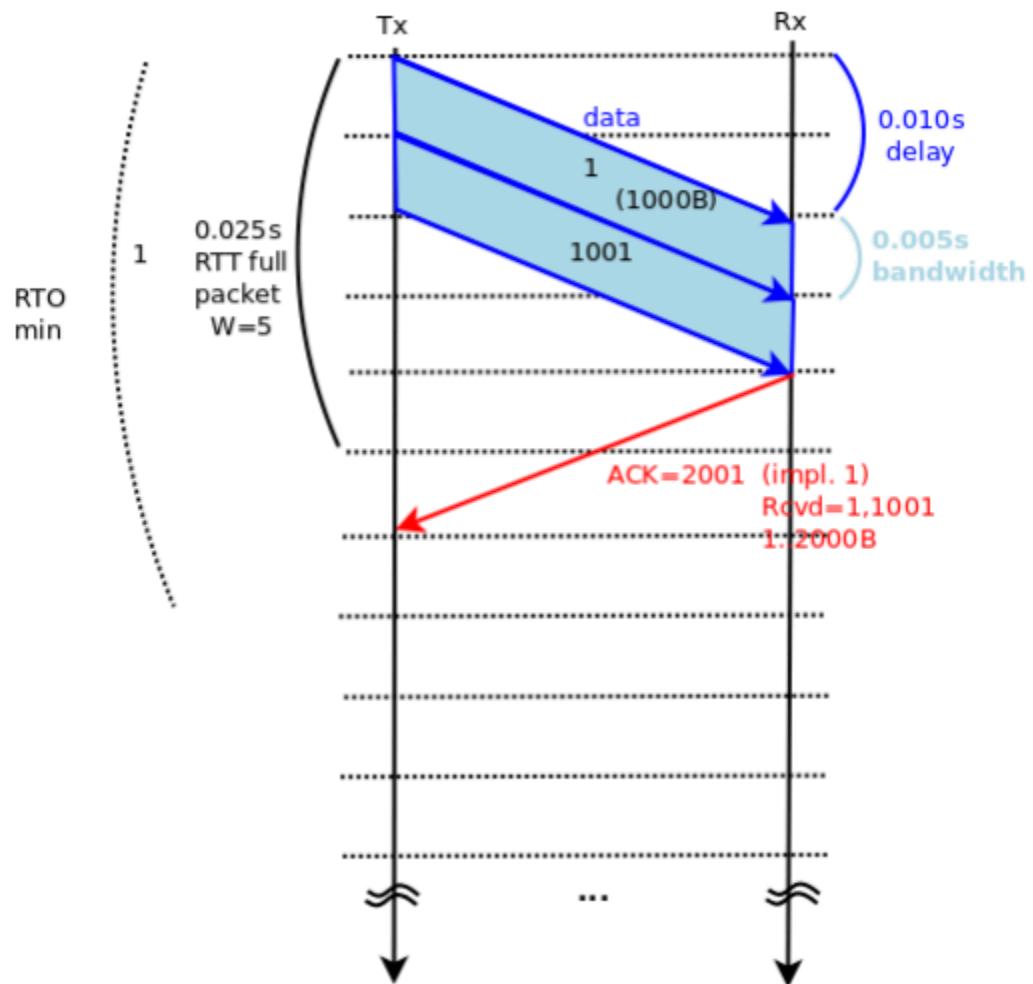
Ejemplo en acción 😎

Se quieren enviar los datos del 1 al 10.000

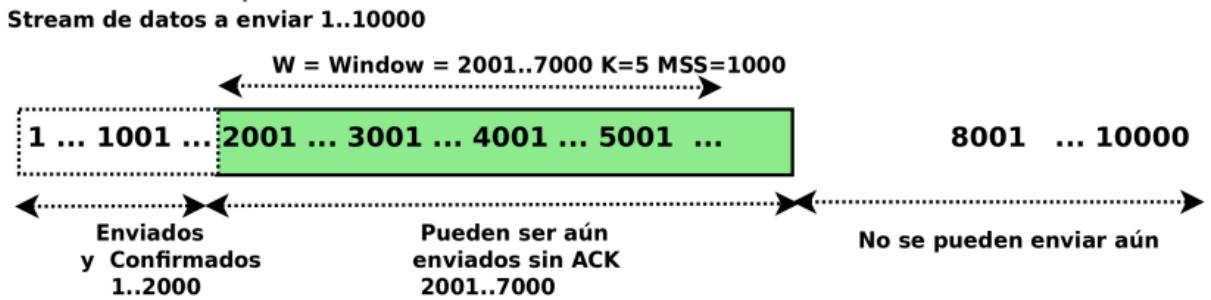
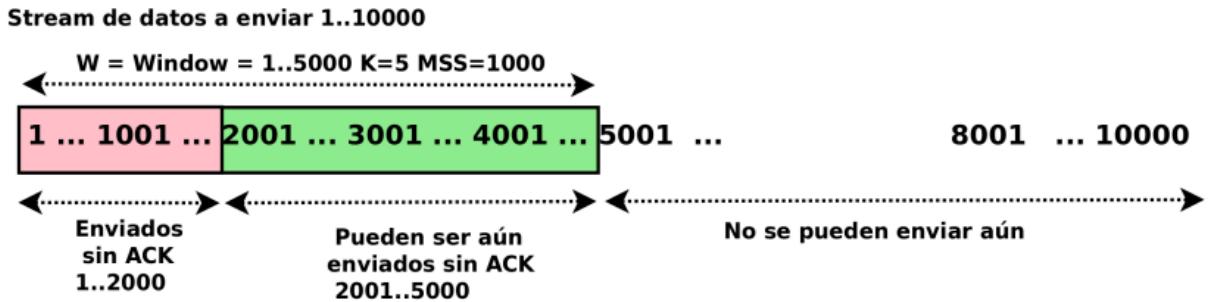
Ventana inicial



Envíos



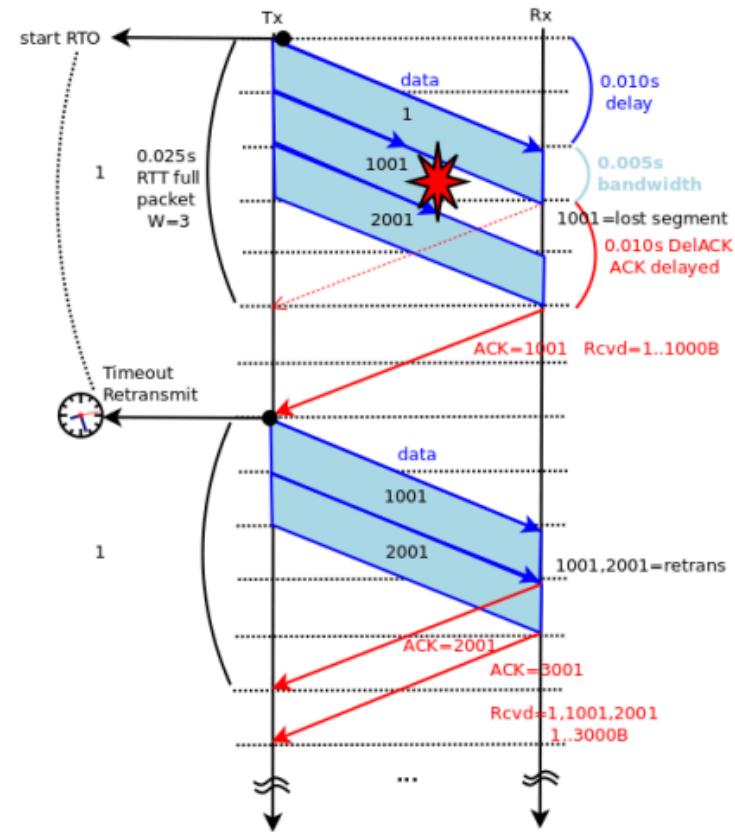
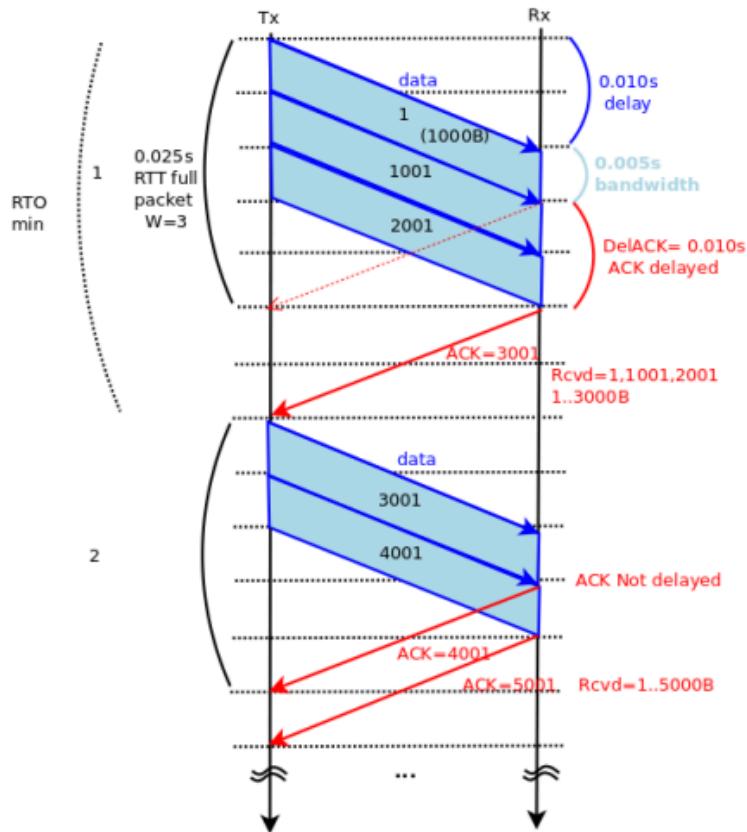
Ventana actualizada



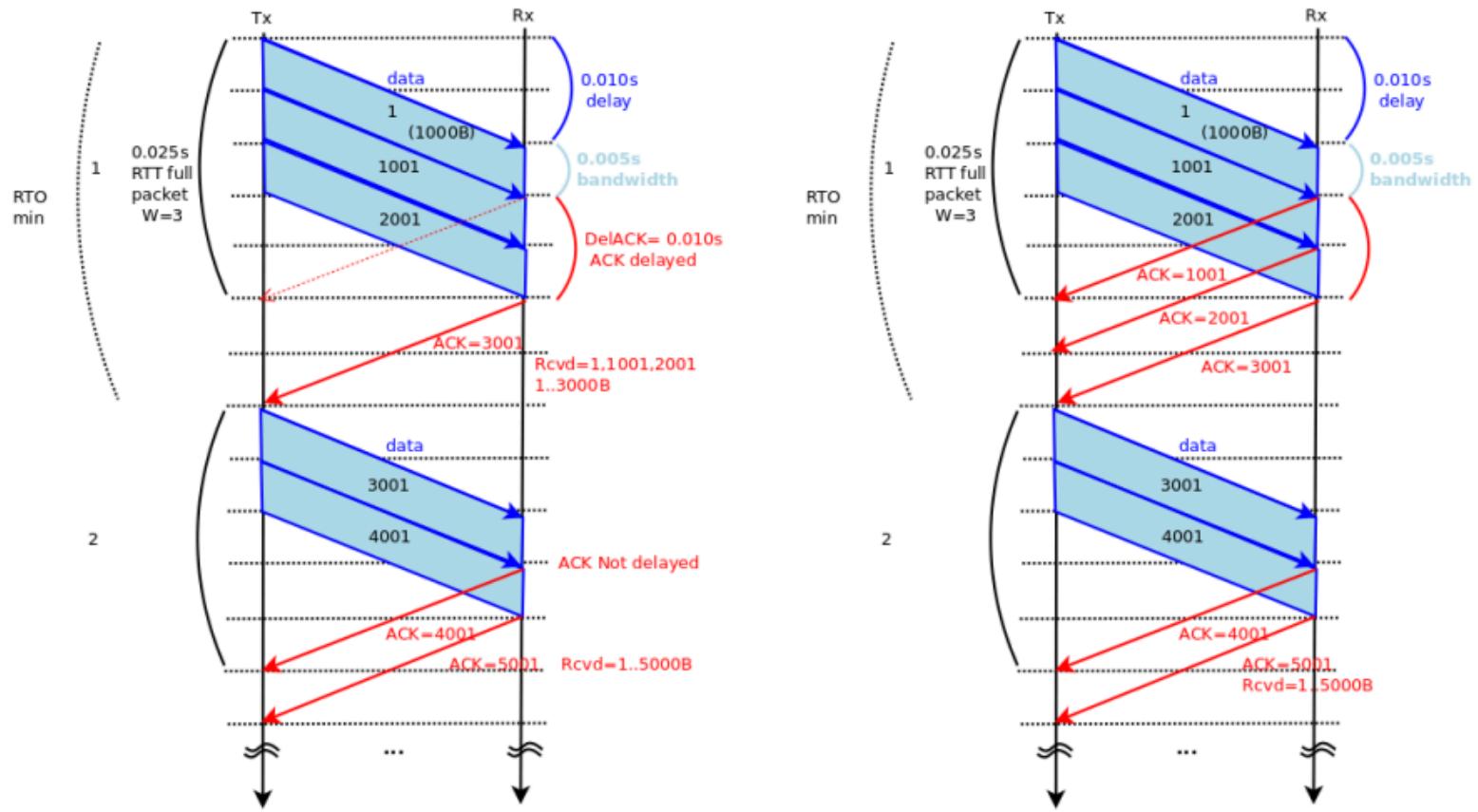
Como ya se enviaron (y confirmaron) los primeros 2K datos, se desliza la ventana (W=5) para que se puedan enviar del 2000 al 7000 hasta tener que para para esperar ACK

Me cambiaron el ejemplo lpqmp 😭 (y se metieron el otro en el * 😡) así que hasta acá llegó este viaje 😭

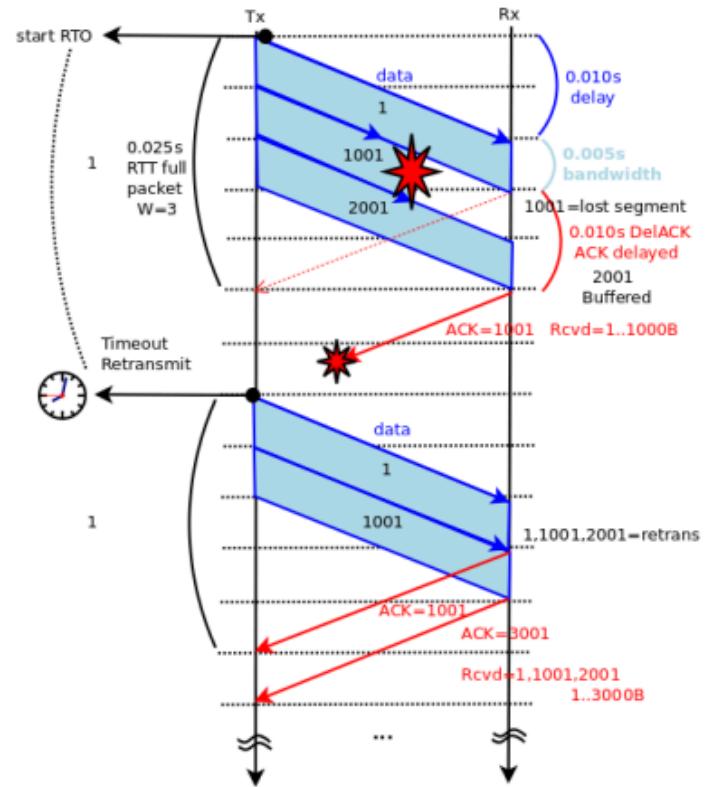
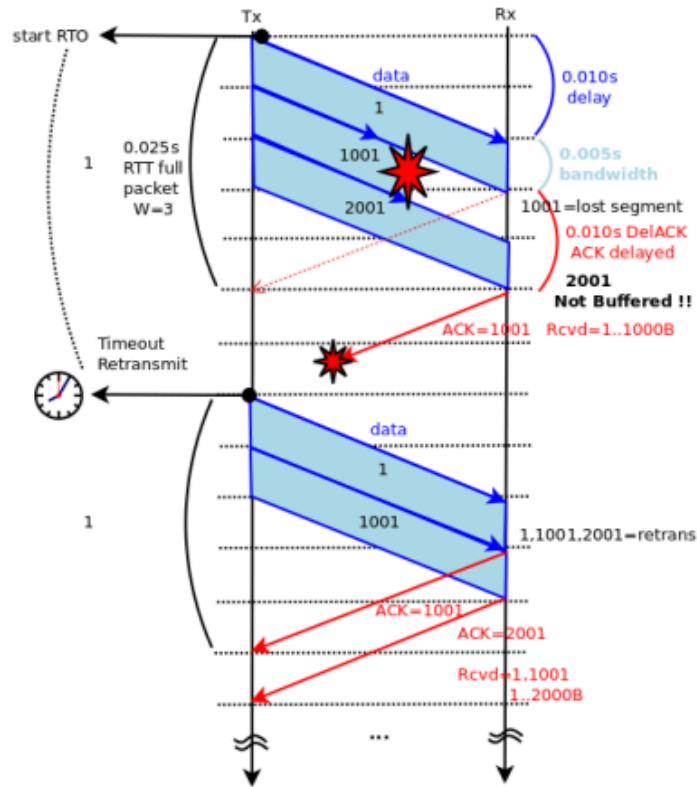
Segmento perdido



ACK acumulados



Buffer fuera de orden



Las confirmaciones por la negativa pueden generar ACK duplicados (NAK)

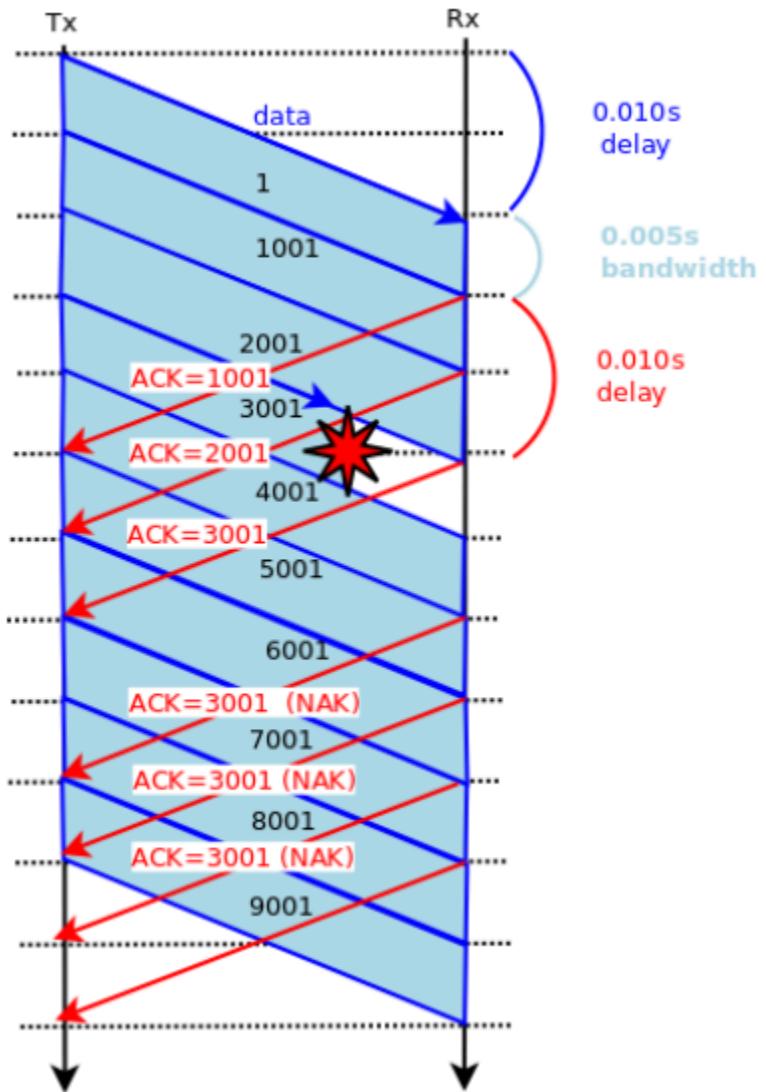
Mensajes fuera de orden:

- Buffering hasta recibir los que llenan los huecos (requiere buffering de Rx antes de pasarlos a la capa usuaria)
- Descartarlos y esperar retransmisiones (no requiere buffering del receptor, solo recordar la secuencia que se espera)

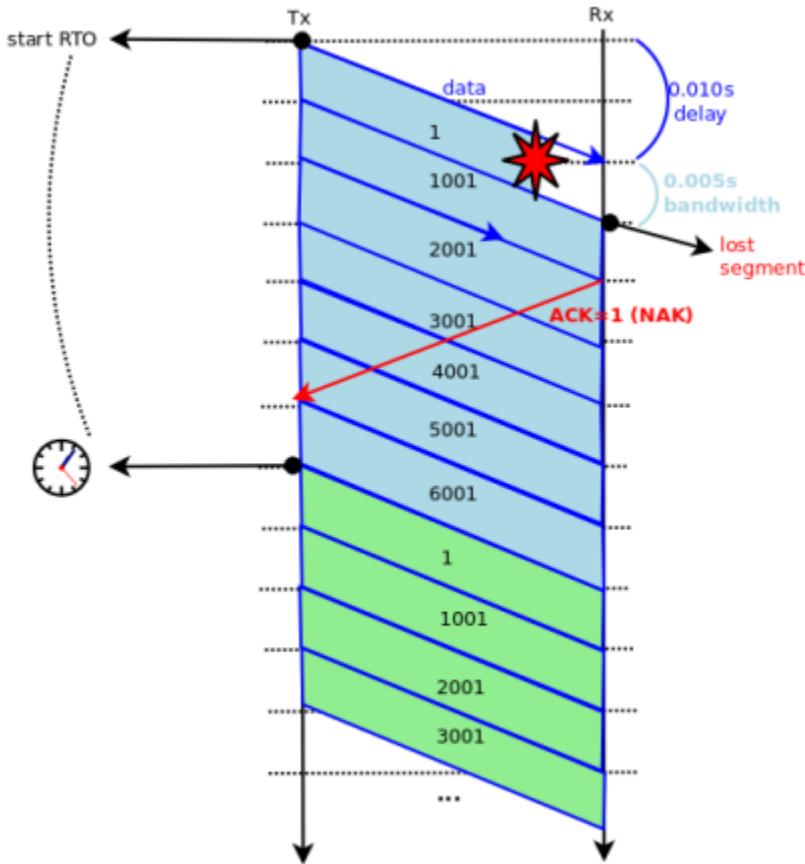
Los NAK por sí solos no generan retransmisiones, requiere varios duplicados para retransmitir

UN NAK solo no necesariamente indica pérdida, puede ser re-ordering en la red

ACK duplicados, NAK



RTO



Si pierde un segmento y se enviaron varios siguientes es como “vaciar” el pipe y volver atrás: Go-Back-N

Si se mantienen segmentos en el receptor más fácil la recuperación

Selective repeat

Go-Back-N ante pérdidas retransmite segmentos innecesarios si se perdió uno del medio del stream de datos solamente y hubo timeout

Selective Sliding Window/Selective Repeat solo retransmite los que no se confirmaron

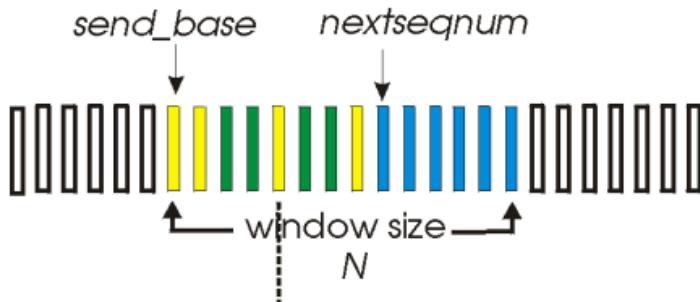
El receptor puede confirmar de a uno o usar bit vectors/intervalos de confirmaciones

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

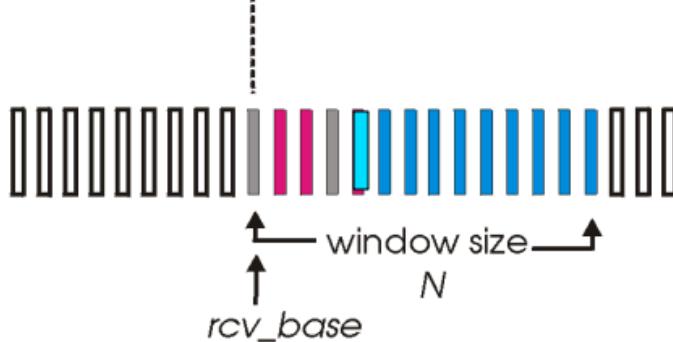
Se realiza en módulo M, $W \leq (M-1)/2$, para evitar confundir los ACK de segmentos

La ventana se desliza sin dejar huecos, desde los confirmados más viejos



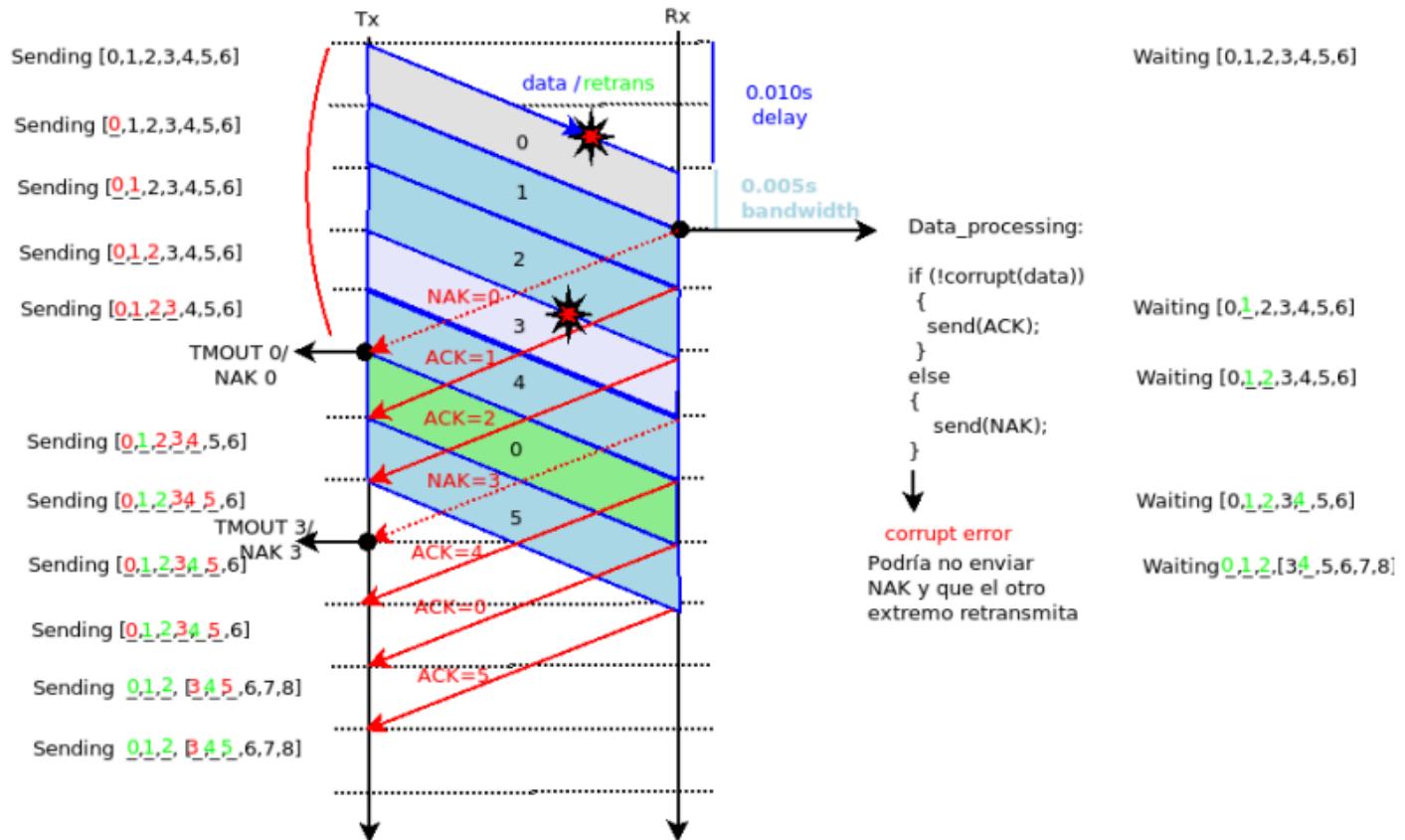
Con ACK recibidos
ACK pendientes
Usable, aún no enviados
No usable

a) Vista de los número de secuencia del transmisor



Fuera de orden (almacenados) con ACK enviado
Esperado, aún no recibido
Aceptable (en ventana)
No usable

b) Vista de los número de secuencia del receptor



Emisor

Eventos en el lado del Emisor (Tx):

- Si tiene datos en el buffer y la ventana lo permite los transmite igual que GBN
 - A diferencia que GBN cada segmento requiere su RTO, se puede simular con un solo timer
 - Si recibe un ACK en orden, desliza/actualiza la ventana, detiene el RTO para el segmento confirmado
 - Si recibe un ACK fuera de orden, no desliza/actualiza la ventana, detiene el RTO para el segmento confirmado y lo marca como ya recibido
 - Si vence RTO, re-envía el segmento asociado y reinicia el RTO

Receptor

Eventos en el lado del Receptor (Rx):

- Si recibe segmento dentro de la ventana esperada confirma el segmento particular (grupo de bytes)
- Si recibe segmento dentro de la ventana esperada y no lo tiene lo almacena, si ya lo tiene lo descarta. Siempre confirma. Requiere bufferear los segmentos RxBuf
- El receptor actualiza su ventana conforme recibe los segmentos en orden

Control de errores en TCP

Utiliza Go-back-N (GBN) con ventana dinámica (flow-control), utiliza piggy-backing y permite negociar Selective Repeat con opciones SACK (Selective ACK)

TCP mantiene el control de errores por bytes (byte oriented), no por segmentos

Los segmentos se numeran de acuerdo a bytes enviados (nro. del primer byte) de datos dentro del segmento

Los números se negocian al establecer la sesión y cada implementación los elige libremente (ISN)

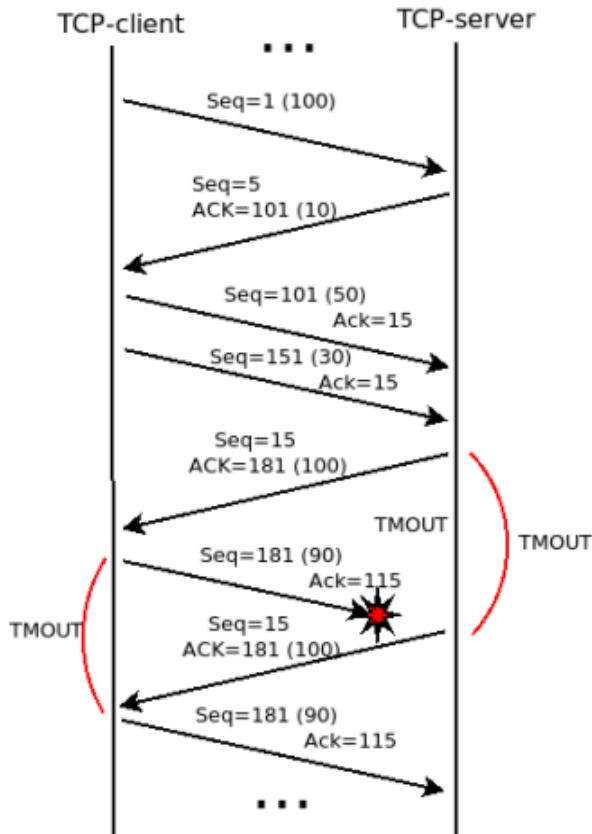
Las confirmaciones son “anticipativas”, indican el nro. de byte que esperan

Para control de errores TCP utiliza los campos: #SEQ, #ACK, flag ACK más timer y algunas opciones

Los campos de un segmento TCP asociados al control de errores son:

- Sequence number
- Acknowledgement number
- Flag ACK
- El campo de opciones se puede usar para realizar una negociación para hacer uso de selective repeat (SACK) y Timestamps

Como TCP utiliza IP, el cual no es estable (RTT puede variar), no se puede establecer un valor fijo del tamaño de la ventana (el tamaño de la ventana también sirve para que el receptor lo reduzca en caso de que su buffer Rx disminuya el tamaño (por ejemplo si la aplicación no solicita leer esos datos))



Time	172.20.1.1 172.20.1.100	Comment
0.000	SYN (41749) → (11111)	Seq = 0
0.001	SYN, ACK (41749) ← (11111)	Seq = 0 Ack = 1
0.001	ACK (41749) → (11111)	Seq = 1 Ack = 1
90.730	PSH, ACK - Len: 5 (41749) ← (11111)	Seq = 1 Ack = 1
90.730	ACK (41749) → (11111)	Seq = 1 Ack = 6
100.150	PSH, ACK - Len: 16 (41749) ← (11111)	Seq = 1 Ack = 6
100.150	ACK (41749) → (11111)	Seq = 6 Ack = 17
104.580	PSH, ACK - Len: 5 (41749) ← (11111)	Seq = 6 Ack = 17
104.580	ACK (41749) → (11111)	Seq = 17 Ack = 11
112.290	PSH, ACK - Len: 6 (41749) ← (11111)	Seq = 17 Ack = 11
112.290	ACK (41749) → (11111)	Seq = 11 Ack = 23
114.890	PSH, ACK - Len: 6 (41749) ← (11111)	Seq = 23 Ack = 11
114.890	ACK (41749) → (11111)	Seq = 11 Ack = 29
120.620	FIN, ACK (41749) → (11111)	Seq = 29 Ack = 11
120.620	FIN, ACK (41749) ← (11111)	Seq = 11 Ack = 30
120.620	ACK (41749) → (11111)	Seq = 30 Ack = 12

Por cada segmento (con datos) que envía TCP es como si iniciara un Timer local, RTO y pone copia del segmento en cola local TxBuf (RFC-793)

Por cada segmento ACK'd en orden descarta el timer asociado y descarta la copia del segmento (RFC-793) del TxBuf. Hace lugar para nuevos segmentos en Tx

Si RTO expira antes que se confirme el segmento TCP lo copia del TxBuf y retransmite (RFC-793)

Segmentos ACK'd no indica leído por aplicación, sí recibido por TCP (RFC-793) (ubicado en el RxBuf del receptor)

Si el receptor detecta error en el segmento simplemente descarta y espera que expire RTO

TCP no arranca un RTO por cada segmento, solo mantiene un por el más viejo enviado y no ACK'd y arranca uno nuevo solo si no hay RTO activo y hay mensajes sin confirmar

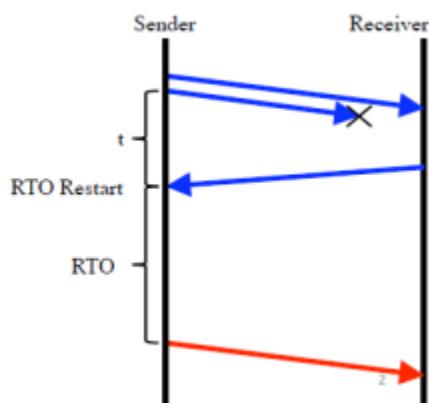
Receptor con segmentos fuera de orden se descartan directamente, mejor podrá re-enviar último ACK solicitando lo que espera (podría dejar en RxBuf pero no entregar a la aplicación, tiene huecos)

Se puede confirmar con ACK acumulativos

Si se confirman datos, se inicia un nuevo RTO (RFC-6298) recomendado. Si todo fue confirmado se detiene el RTO

El nuevo RTO le está dando más tiempo al segmento más viejo aún no confirmado

Mejor: RFC-7765: $RTOnew = RTO - T_{earliest}$ (menos el tiempo que pasó del pendiente más viejo)



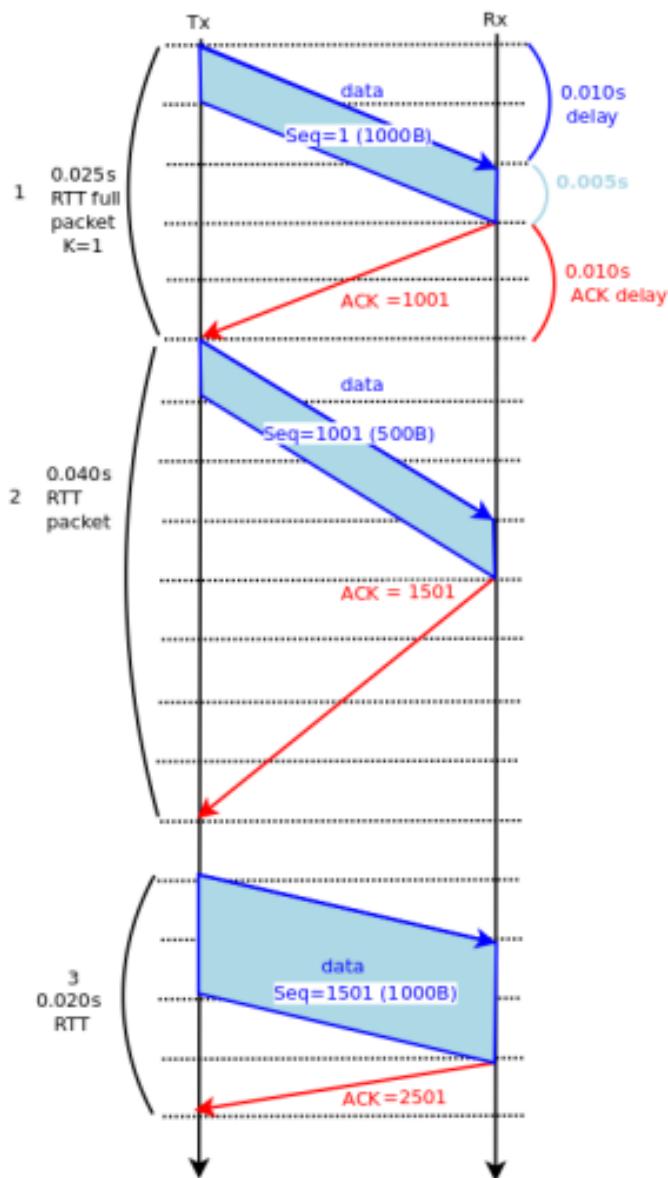
Si vence un RTO se debe retransmitir el segmento más viejo no ACK'd y se debe duplicar: Back-off timer RTOnew = RTO * 2. RTOMAX = 60s (RFC-6298) recomendado

RTT dinámico

RTT calculado en base el RTT

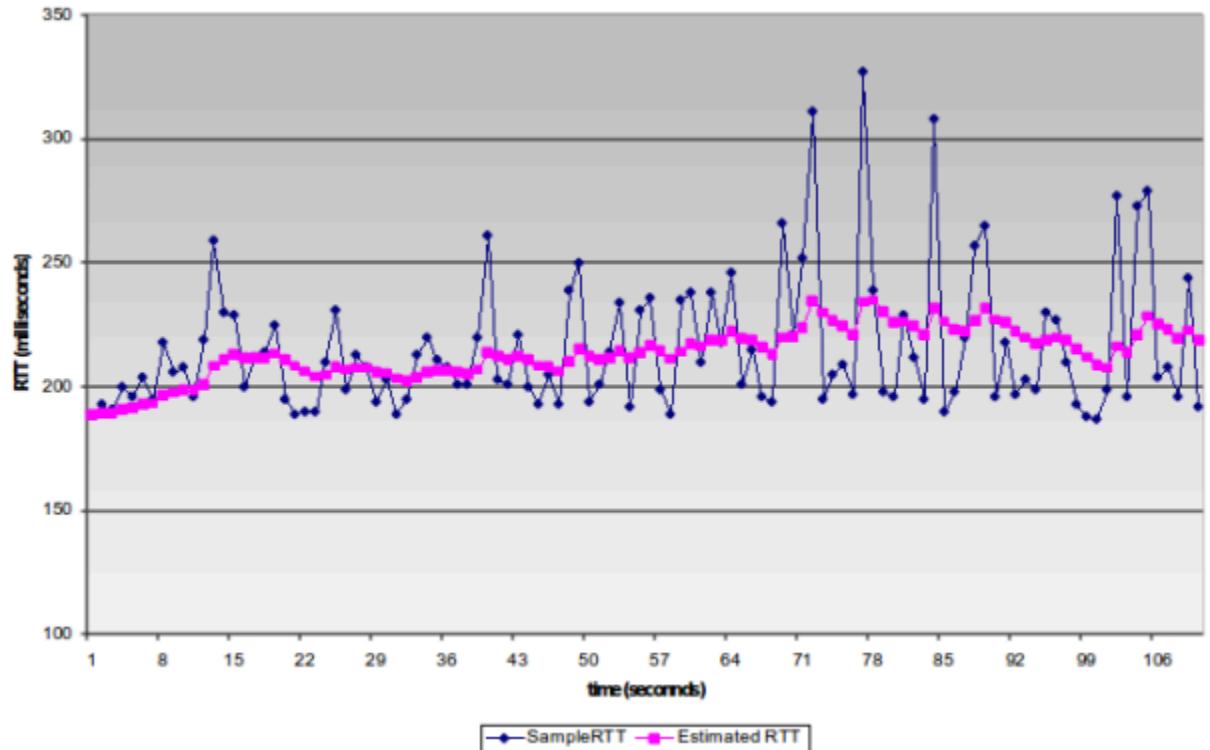
RTT debe ser dinámico, debe contemplar estado de la red

RTT estático sólo serviría para L2 (directamente conectados)



Para calcular RTO se estima RTT. RTT inicial = 1 seg (hasta 2011 era de 3 segundos)

Se hace un promedio pesado dándole más importancia a los valores más recientes de RTT, pero evitando grandes saltos del RTO ante grandes cambios del RTT



Timestamp

Se utiliza para evitar tener un timer para calcular el RTT

Se envía en el primer SYN el timeStamp local, opcional

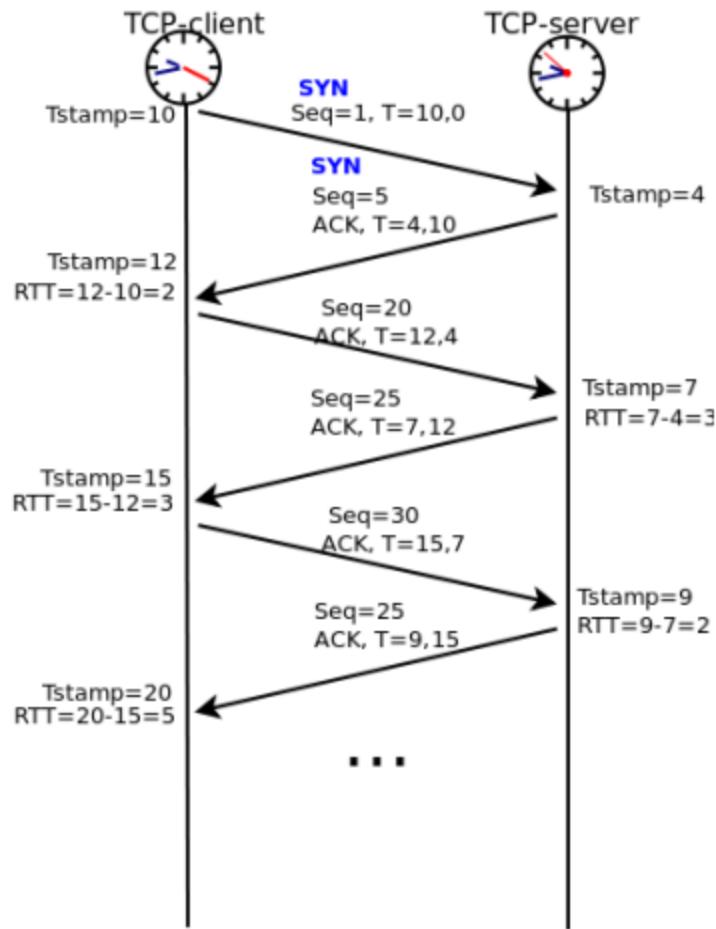
En cada mensaje TCP con esta opción, se copia el timeStamp local y se hace echo del último timeStamp recibido desde otro extremo

Con el valor recibido como echo y el valor del reloj local se calcula el RTT

Si el mensaje no es un ACK válido no se actualiza la estimación del RTT/SRTT

Relaja la necesidad de usar timer por cada segmento para estimar RTT

Protección contra Wraparounds de #secuencia (PAWS)



Clase 3 - Control de flujo en TCP

Servicios de TCP

- 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
- Control de Flujo (Flow-Control):
 - Mecanismo protocolar, algoritmo, que permite al receptor controlar la tasa a la que le envía datos el transmisor
 - Controla cuánto puede enviar una aplicación sabiendo que la receptora tiene capacidad de recibirla y procesarla
 - Objetivo: prevenir que el emisor sobrecargue al receptor con datos evitando un mal uso de la red

Control de errores y de flujo

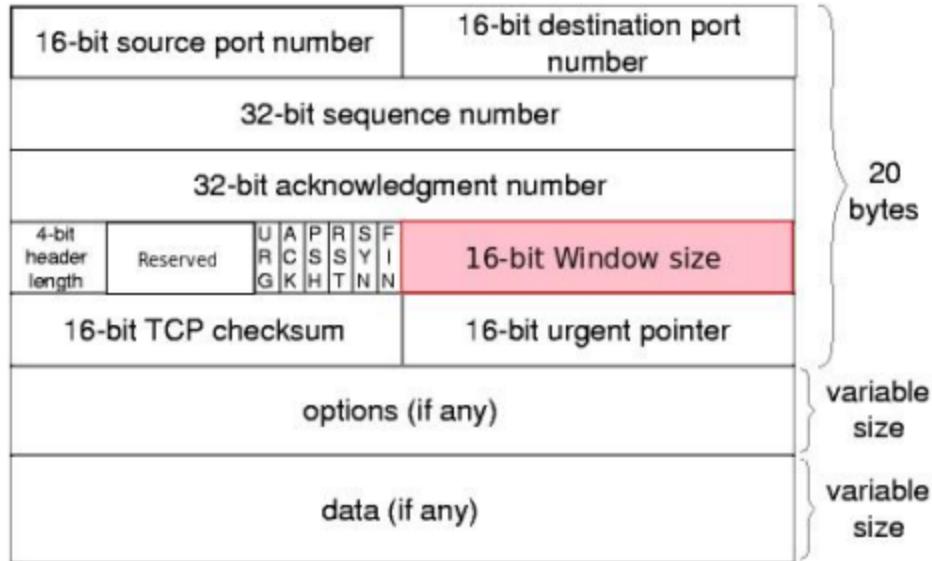
Para realizar control de errores y control flujo se utilizan técnicas de ARQ (Automatic Repeat reQuest), Transferencia de Datos Fiables

ARQ solo no hace control de flujo, requiere de otros mecanismos como RNR (Receive-Not-Ready), o Dynamic Window (Ventana Dinámica). TCP usa Ventana Dinámica

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

Control de flujo

El receptor (cada extremo puede recibir, es full duplex) 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 (mbufs)

Cada conexión mantiene su propio buffer en espacio del kernel (TCP)

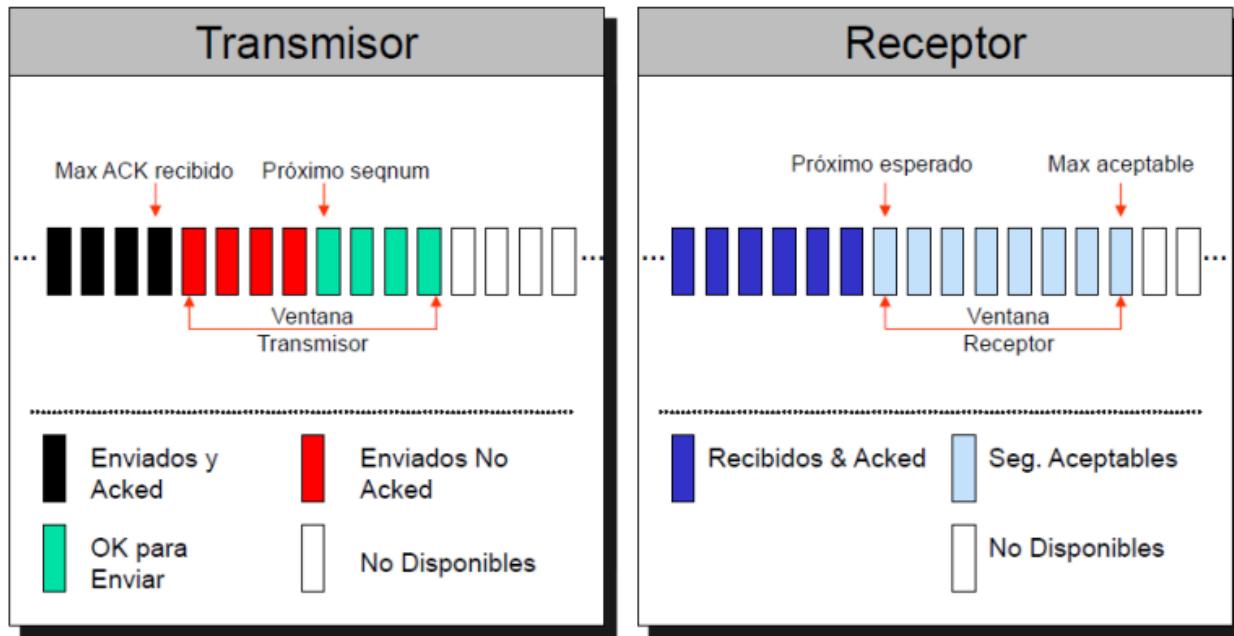
Window (Ventana) indica la cantidad de datos en bytes que el emisor le puede enviar sin esperar confirmación (mejora notablemente contra Stop & Wait)

La ventana de recepción de cada extremo es independiente

Cada vez que llega un segmento nuevo en orden es puesto por TCP en el Rx Buffer, TCP lo debe confirmar

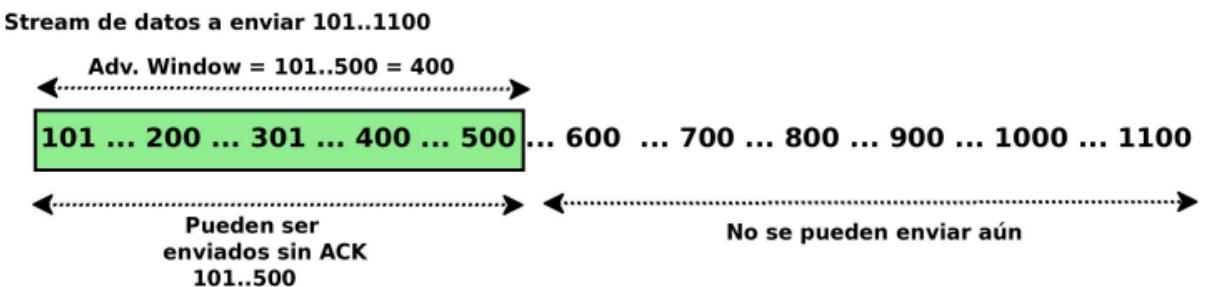
Cada vez que la aplicación lee se hace espacio en el Rx Buffer. Se va modificando el tamaño de la ventana. Se comunica con los segmentos de ACK (y de datos)

Cada vez que llega un ACK en orden se mueve la ventana en el Transmisor, se descartan segmentos confirmados del Tx Buffer



Ejemplo ventana deslizante

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



Luego, la aplicación que envía, escribe, `write()`, y se envían 400 bytes usando toda la ventana (los 400 bytes se pudieron enviar en múltiples segmentos)

Se recibe un segmento con $ACK = 301$ y $WIN = 400$

Se desliza ventana

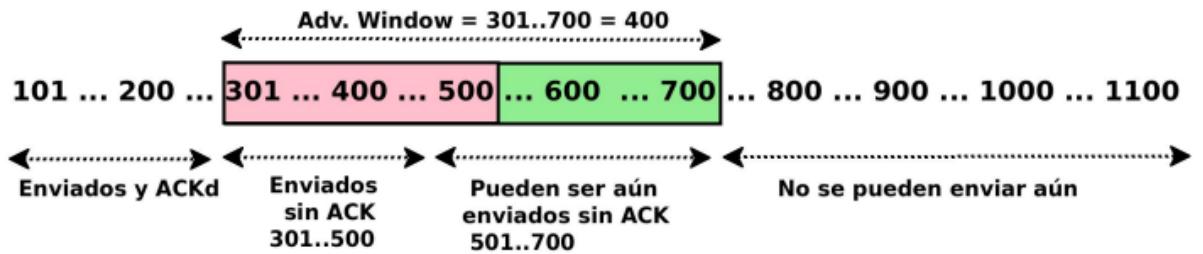
101..300 en ningún buffer, enviados y leídos

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

Stream de datos a enviar 101..1100



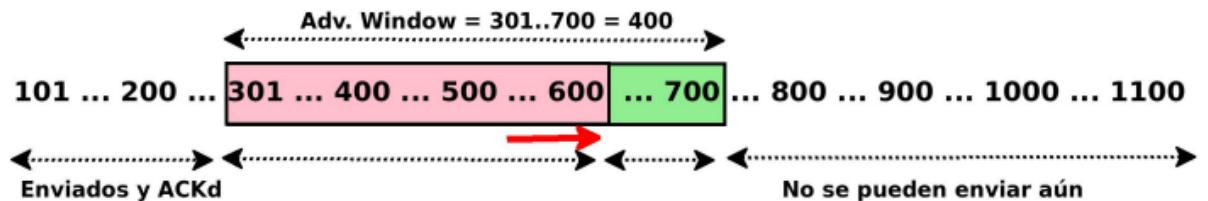
Se envía un segmento con los bytes 501..600

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

Stream de datos a enviar 101..1100



Se recibe un segmento ACK = 401, WIN = 300

Ventana se cierra, la aplicación receptora no lee, read()

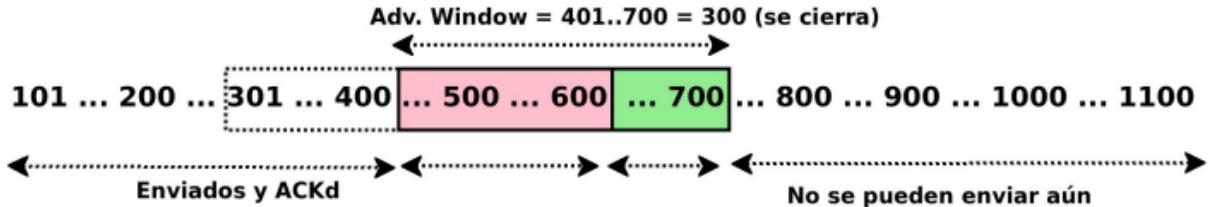
101..300 ya estaban procesados

301..400 en Rx Buffer, no se han leído aún, pero ACKd, fuera del Tx Buffer

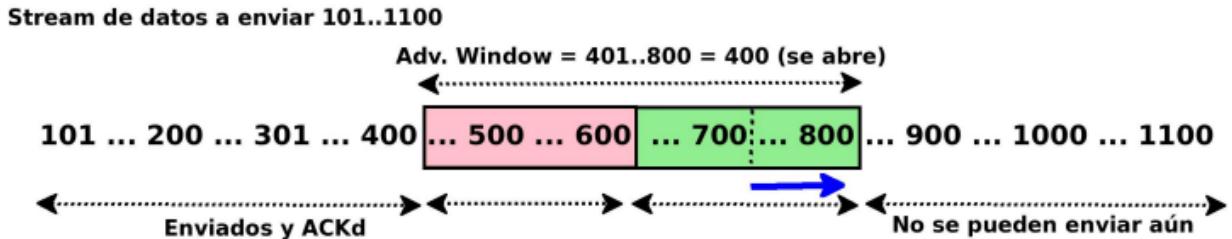
401..600 en Tx Buffer, aún en “en vuelo”

601..700 en Tx Buffer, aún no han sido enviados

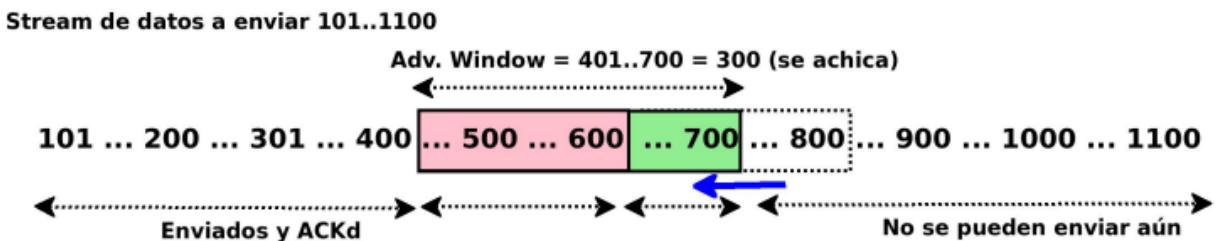
Stream de datos a enviar 101..1100



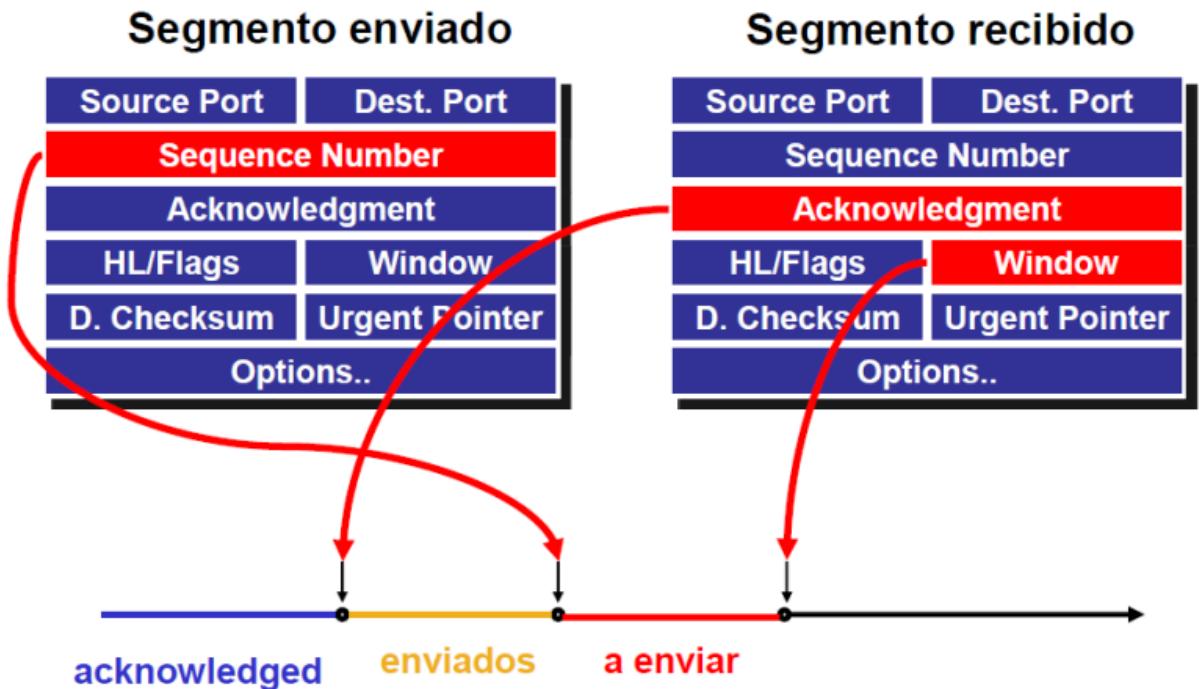
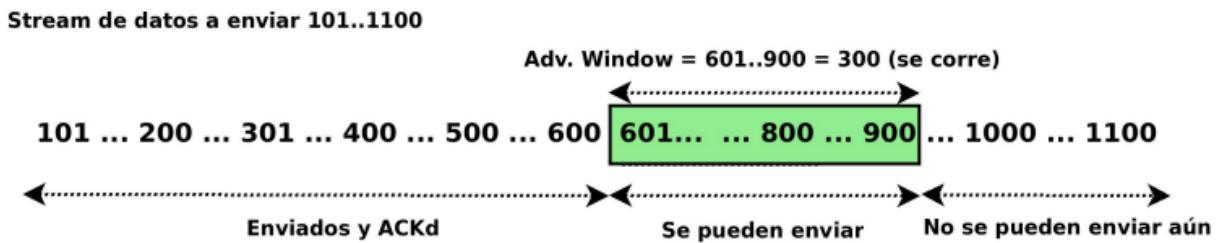
Ventana se abre, la aplicación receptora leyó, llamó a read(). Se manda nuevo ACK
 Se recibe un segmento ACK = 401, WIN = 400
 401..600 en Tx Buffer, “en vuelo”
 601..800 en Tx Buffer, aún no han sido enviados
 101..400 no están más en Rx Buffer, se procesaron



Se recibe un segmento ACK = 401, WIN = 300
 Ventana se achica
 401..600 en Tx Buffer, “en vuelo”
 601..700 en Tx Buffer, aún no han sido enviados
 Raro que suceda



Se leyeron los datos del Rx Buffer y se confirma
 Se recibe un segmento ACK = 601, WIN = 300
 Ventana se corre
 Rx Buffer vacío
 601..900 en Tx Buffer, aún no han sido enviados, se pueden enviar



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: $Win - Sent.No.ACK'd$ (es decir, el tamaño de la ventana menos la cantidad de segmentos enviados no ACK'd)

$Effective_Win = Win - (LastByteSent - LastByteAcked)$ si no se tiene en cuenta la congestión

Al recibir ACKs de TCP (Aplic. no lee aún) se cierra ventana

Al recibir ACKs y Win fijo desliza ventana (Aplic. lee a tasa(rate) fija)

Al achicarse Win se reduce ventana (Aplic. no lee)

Al agrandarse Win tiene posibilidad de enviar más (Aplic. lee más rápido)

Tamaño de ventana seleccionado por el kernel o por aplicación usando: `setsockopt()`

TCP bulk e interactivo

Delayed ACKs: no enviar ACK sin esperar de enviar datos antes: piggy-back (200ms, MAX=500ms)

Algoritmo Nagle:

- No enviar datos en chunks pequeños, esperar juntar información
- Perjudica aplicaciones interactivas

Tinygrams:

- Segmentos chicos
- Beneficia a las aplicaciones interactivas
- Win casi vacía

Silly Window: Win casi llena se “ofrecen” pequeños incrementos. Enfoque: mostrar incrementos de $\text{Min}(\text{MSS}, \text{RecvBuf})/2$, no menores. Es decir, hasta que no haya una modificación del window size del mínimo entre MSS y el buffer de recepción dividido 2, no se va a enviar un window update

Escalado de Ventana

El campo Ventana del segmento TCP es de 16bits dando MAX=64KB

Para obtener mejor rendimiento ante BDP (BW*DELAY) grandes se requiere aumentar, para LFNs (Long Fat Networks)

RFC-7323: Opción TCP de escalado de ventana en SYN

Multiplicar el valor del campo por 2, 4, 8, ..., 128, ..., 16KB (shift 1,2,3...7...14)

Valor máximo con escalado $W = 65535 \times 2^{14} \approx 2^{16} \times 2^{14} = 2^{30} = 1\text{GB}$

```
Decompression failed.
[Stream index: 0]
[Conversation completeness: Complete, WITH_DATA (31)]
[TCP Segment Len: 0]
Sequence Number: 0 (relative sequence number)
Sequence Number (raw): 2219352712
[Next Sequence Number: 1 (relative sequence number)]
Acknowledgment Number: 1 (relative ack number)
Acknowledgment number (raw): 3436242235
1010 .... = Header Length: 40 bytes (10)
> Flags: 0x012 (SYN, ACK)
Window: 14480
[Calculated window size: 14480]
Checksum: 0x1842 [unverified]
[Checksum Status: Unverified]
Urgent Pointer: 0
- Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation (NOP), W
  > TCP Option - Maximum segment size: 1460 bytes
  > TCP Option - SACK permitted
  > TCP Option - Timestamps: TSval 214042, TSecr 214023
  > TCP Option - No-Operation (NOP)
  - TCP Option - Window scale: 4 (multiply by 16)
    Kind: Window Scale (3)
    Length: 3
    Shift count: 4
```

```
> Flags: 0x010 (ACK)
Window: 15023
[Calculated window size: 240368]
[Window size scaling factor: 16]
Checksum: 0x183a [unverified]
```

tcp.analysis.zero_window

No.	Time	Source	Destination	Protocol	Length	Info
279	8.953425	10.0.3.10	10.0.1...	TCP	66	5001 → 51373 [ACK] Seq=1 Ack=254873 Win=
280	12.168621	10.0.1.10	10.0.3...	TCP	1090	[TCP Window Full] 51373 → 5001 [PSH, ACK]
281	12.168664	10.0.3.10	10.0.1...	TCP	66	[TCP ZeroWindow] 5001 → 51373 [ACK] Seq=
282	15.336591	10.0.1.10	10.0.3...	TCP	66	[TCP Keep-Alive] 51373 → 5001 [ACK] Seq=
283	15.336634	10.0.3.10	10.0.1...	TCP	66	[TCP ZeroWindow] 5001 → 51373 [ACK] Seq=
284	21.521222	10.0.1.10	10.0.3...	TCP	66	[TCP Keep-Alive] 51373 → 5001 [ACK] Seq=

Frame 281: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
 Ethernet II, Src: 00:00:00_aa:00:07 (00:00:00:aa:00:07), Dst: 00:00:00_aa:00:06 (00:00:00:aa:00:06)
 Internet Protocol Version 4, Src: 10.0.3.10, Dst: 10.0.1.10
 Transmission Control Protocol, Src Port: 5001, Dst Port: 51373, Seq: 1, Ack: 255897, Len: 0

Source Port: 5001
 Destination Port: 51373
 [Stream index: 0]
 [Conversation completeness: Complete, WITH_DATA (31)]
 [TCP Segment Len: 0]
 Sequence Number: 1 (relative sequence number)
 Sequence Number (raw): 2219352713
 [Next Sequence Number: 1 (relative sequence number)]
 Acknowledgment Number: 255897 (relative ack number)
 Acknowledgment number (raw): 3436498131
 1000 = Header Length: 32 bytes (8)
 Flags: 0x010 (ACK)
 Window: 0
 [Calculated window size: 0]
 [Window size scaling factor: 16]
 Checksum: 0x183a [unverified]
 [Checksum Status: Unverified]
 Urgent Pointer: 0
 Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps

"tcp.analysis.zero" is neither a field nor a protocol name. Packets: 607 - Displayed: 607 (100.0%) Profile: Classic

Clase 4 - Control de congestión

El control de flujo sólo tiene en cuenta el estado del receptor, no el de la red (el receptor podría poder recibir segmentos, pero la red podría no poder enviarlos)

Se realiza por cada conexión: End-to-End, App-to-App

Permite que aplicaciones no saturen la capacidad de la red

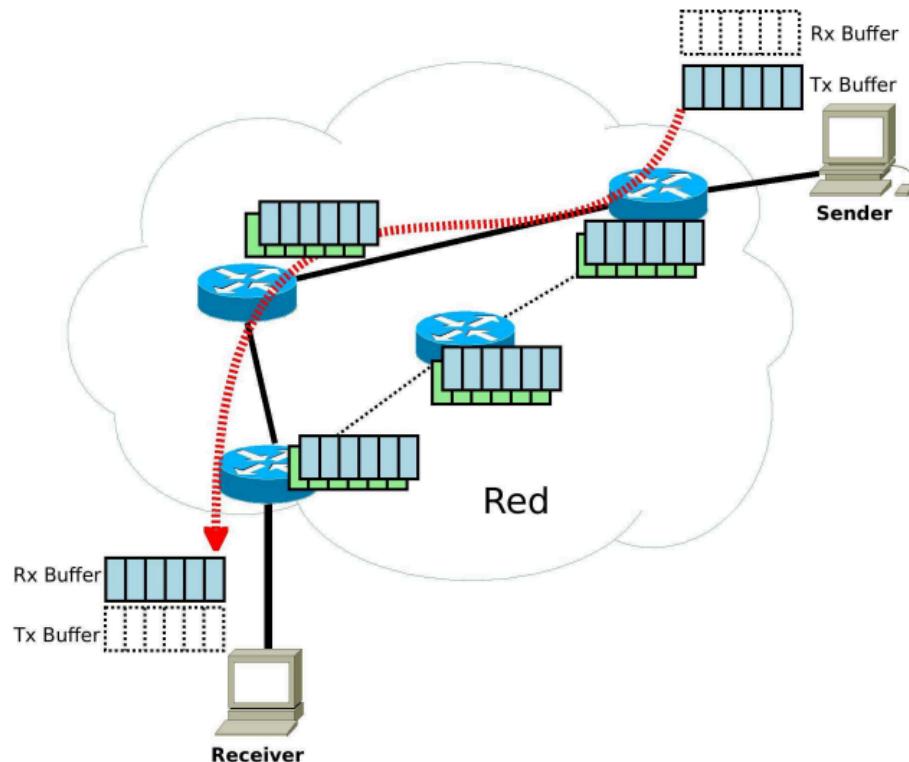
Tiene en cuenta el estado de la red a diferencia del control de flujo que solo ve el receptor

Objetivo: controlar el tráfico que se envía evitando que se colapse la red y se descarte teniendo que retransmitirse

Se puede implementar End-to-End (caso TCP, RFC-5681 (ant. RFC-2581))

O tomando como partida a la red. Modelo basado en la Red: IP+TCP:RFC-3168 (similar a mecanismos L2/L3: Frame-Relay, ATM)

Congestión: problemas de delay en los routers, problemas de overflow y descarte



Causas de congestión en la red

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)

Utilización de la red:

- Demasiado tráfico en la red (modelo de red compartida)
- Se detecta por los nodos intermedios (router/switches) por ejemplo: cuando las colas sobrepasan un umbral. Se utiliza simple umbral o doble umbral (min,max)

Modelo end-to-end

Modelo en el cual no participa la red (más implementado)

Se utilizan nuevos parámetros (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 es la ventana de recepción, usada para el control de flujo)
- FlightSize = (LastByteSent – LastByteACKed) segmentos en vuelo, enviados y aún no confirmados (Sent.No.ACKed)
- Puede enviar: EffectiveWin = MaxWin – FlightSize
- MaxWin = Min(rwnd, cwnd) se calcula en base a quién está más cargado, el receptor o la red

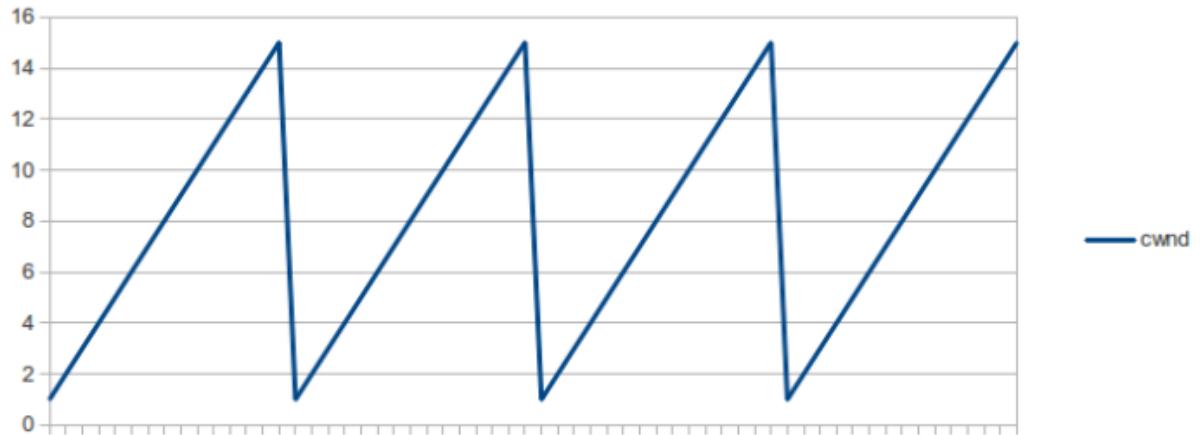
Diferentes Fases:

- Si $cwnd < ssthresh$: fase de crecimiento inicial: SS (Slow Start)
- Si $cwnd \geq ssthresh$: fase de mantenimiento: CA (Congestion Avoidance)

Versión vieja de TCP

cwnd se incrementa al recibir un ACK

Al perder un ACK o tener un segmento erróneo se comienza de cero
Para saber si había congestión se usaba la expiración del RTO
Esto no sirve para un entorno WAN como lo es Internet



Tahoe

Old-Tahoe sin Fast Retransmit, solo SS y CA
Tahoe agrega Fast Retransmit, aunque no está bien implementado

Slow Start

La ventana crece exponencialmente, de forma rápida, no es slow
Se le llama Slow Start porque comienza a probar con pocos paquetes, menos agresivo que el enfoque de enviar tanto como la ventana de recepción permita
Inicia $cwnd = IW = 1 * MSS$ (a veces se usa 2 o 3)
Transmite y espera ACK
ACK recibido, $cwnd ++$
El crecimiento exponencial se da ya que, cuando $cwnd$ es 1, entonces envío 1 segmento y recibo 1 ACK, lo que me lleva a incrementar $cwnd$ en 1, pasando a valer 2. Cuando envío 2 segmentos, recibo 2 ACKs, por lo que incremento $cwnd$ en 2, pasando a valer 4, y así sucesivamente
Si se usa delayed ACK, podría no crecer a esa misma tasa

Congestion Avoidance

Un evento de congestión se da cuando vence el RTO (o se reciben 3DUP ACKs si hay fast retransmit), deriva en ambos casos en Slow Start:

- $ssthresh = cwnd/2 * MSS$
- $cwnd = LW = 1 * MSS$ (Loss Window) (mejor implementado en TCP Reno)

Puede suceder que MSS sea diferente entre emisor y receptor, para este caso se considera SMSS y RMSS. Los cálculos se hacen en base a SMSS

Ante el primer evento de congestión se calcula el $ssthresh$, primer rafaga SS puro

Una vez que $cwnd \geq ssthresh$ crece de forma lineal, es decir, se incrementa $cwnd$ por cada RTT (en lugar de por cada ACK)

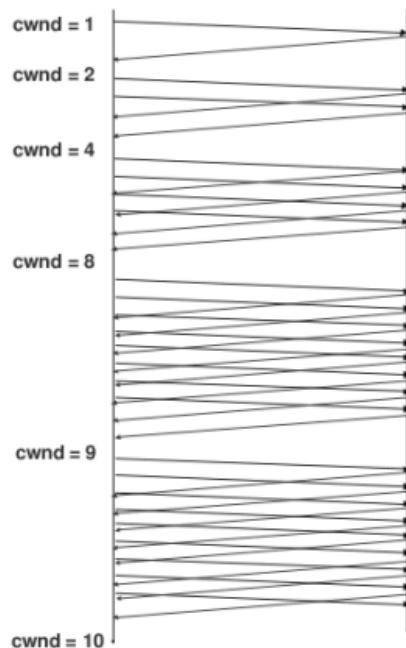
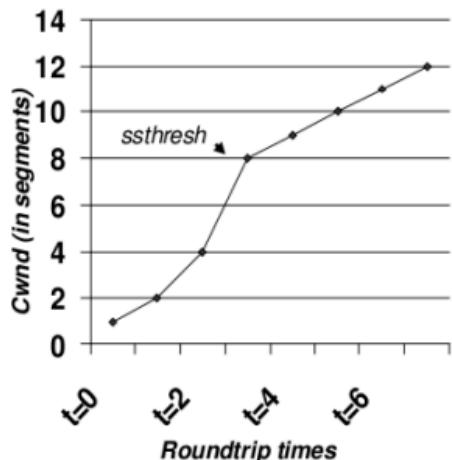
Fast Retransmit

Hace uso de los 3DUP ACKs, para no esperar hasta que venza el RTO

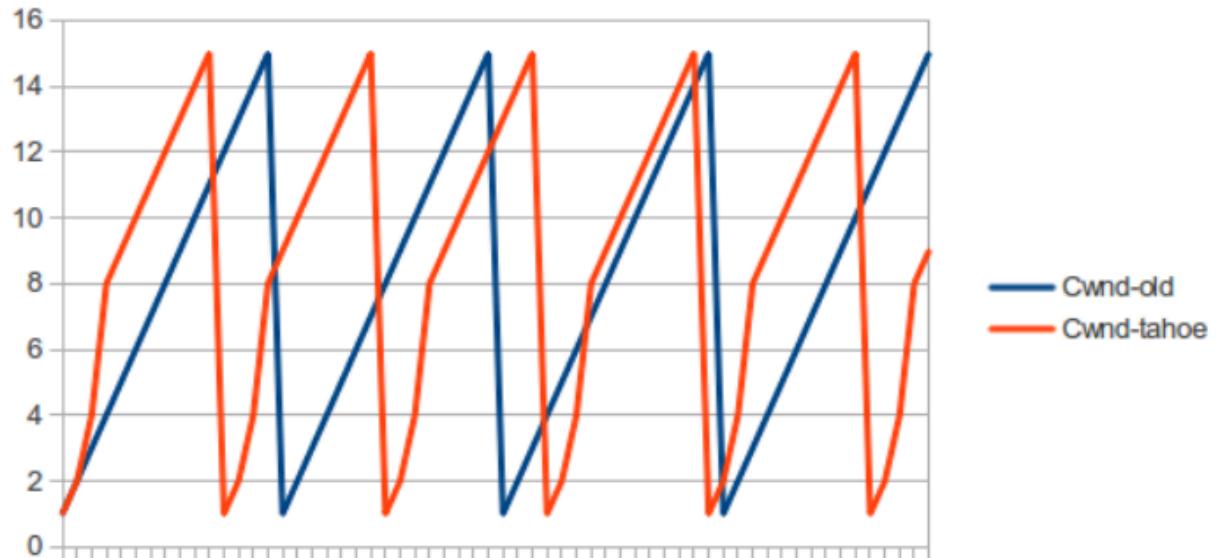
Objetivo, recuperarse más rápido que un timeout

En Tahoe, FRT seguido por Slow Start. Vuelve al inicio $cwnd = 1 * MSS$

Assume that $ssthresh = 8$



Old Tahoe vs Tahoe



TCP Reno

TCP Reno (BSD 4.3 - 1990) Van Jacobson. RFC-2001/RFC-2581/RFC-5681

Se implementan de forma correcta Fast Retransmit y Fast Recovery

Usa:

- Slow Start (SS)
- Congestion Avoidance (CA)
- Fast Retransmit (FRT)
- Fast Recovery (FR)

FlightSize=cwnd si siempre tuvo datos para enviar, aunque en general cwnd>FlightSize, donde cwnd crece sin enviar realmente datos, por lo tanto no refleja en ese caso el límite de la red

Si el RTO expira

1. Retransmite el segmento perdido

2. $ssthresh = cwnd/2 * MSS$ (en RFC $ssthresh = \text{FlightSize}/2 * SMSS$)
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

Fast retransmit

Intenta recuperarse más rápido que un timeout (expira RTO)

El receptor inmediatamente al recibir un segmento fuera de orden no debe esperar RTO del emisor sino generar un ACK (DUP ACK). No se debe retardar

Debido a que el emisor no sabe si se perdió o llegó fuera de secuencia: espera por más ACK duplicados

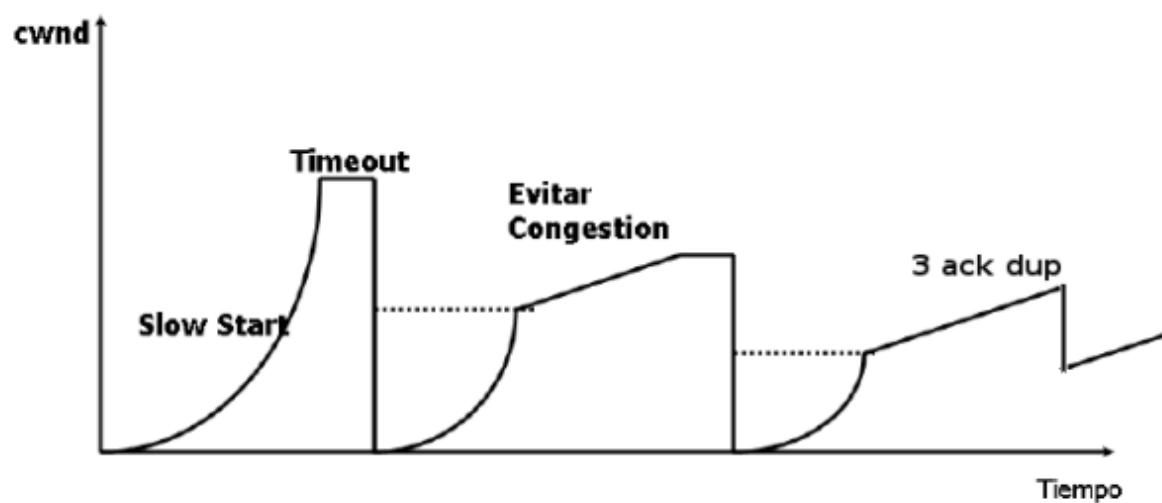
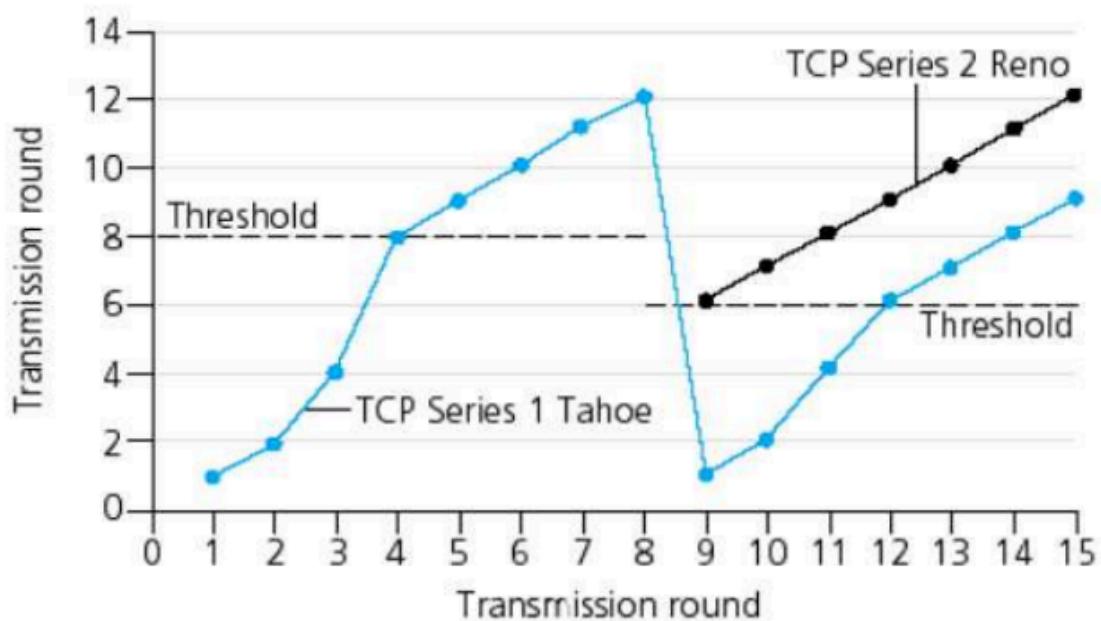
El emisor, si recibe 3 ACK duplicados (4 ACK para el mismo segmento, el original y los 3 duplicados), considera que se perdió (no es re-ordenamiento) y re-envía el segmento solicitado sin esperar RTO

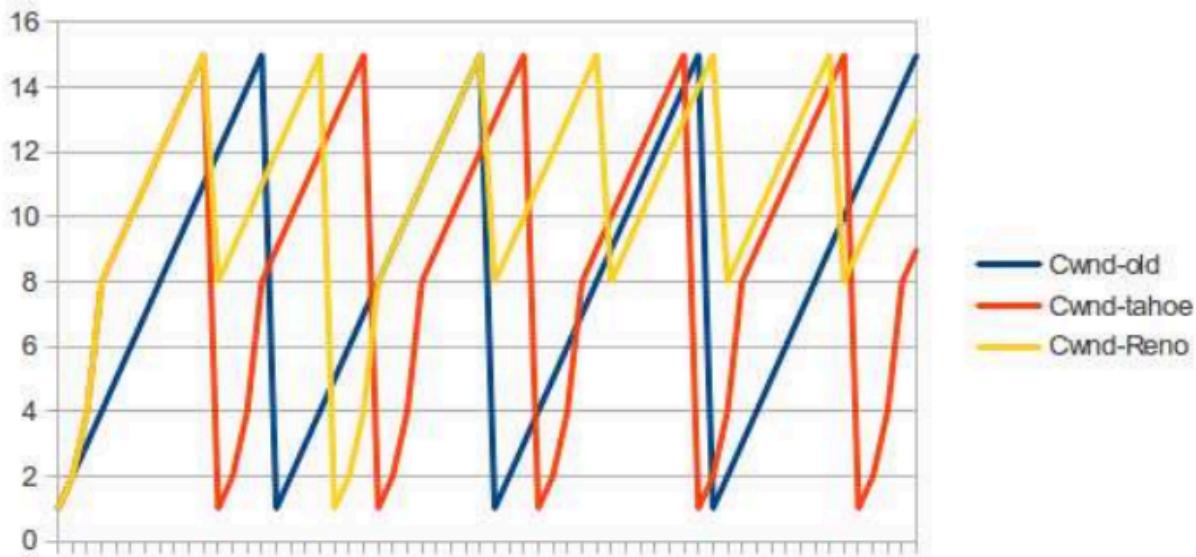
Fast Recovery

Si se detectan 3DUP ACKs (Fast Retransmit):

1. $ssthresh = cwnd/2 * MSS$ (en RFC: $ssthresh = \text{FlightSize}/2 * SMSS$)
2. Fast Retransmit (se retransmite el segmento perdido)
3. Fast Recovery: $cwnd = (ssthresh + 3) * MSS$
4. Por cada ACK de un segmento distinto que recibe incrementa cwnd en 1
5. Una vez recuperado (cuando se recibe el ACK del segmento perdido), finaliza Fast Recovery. 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)

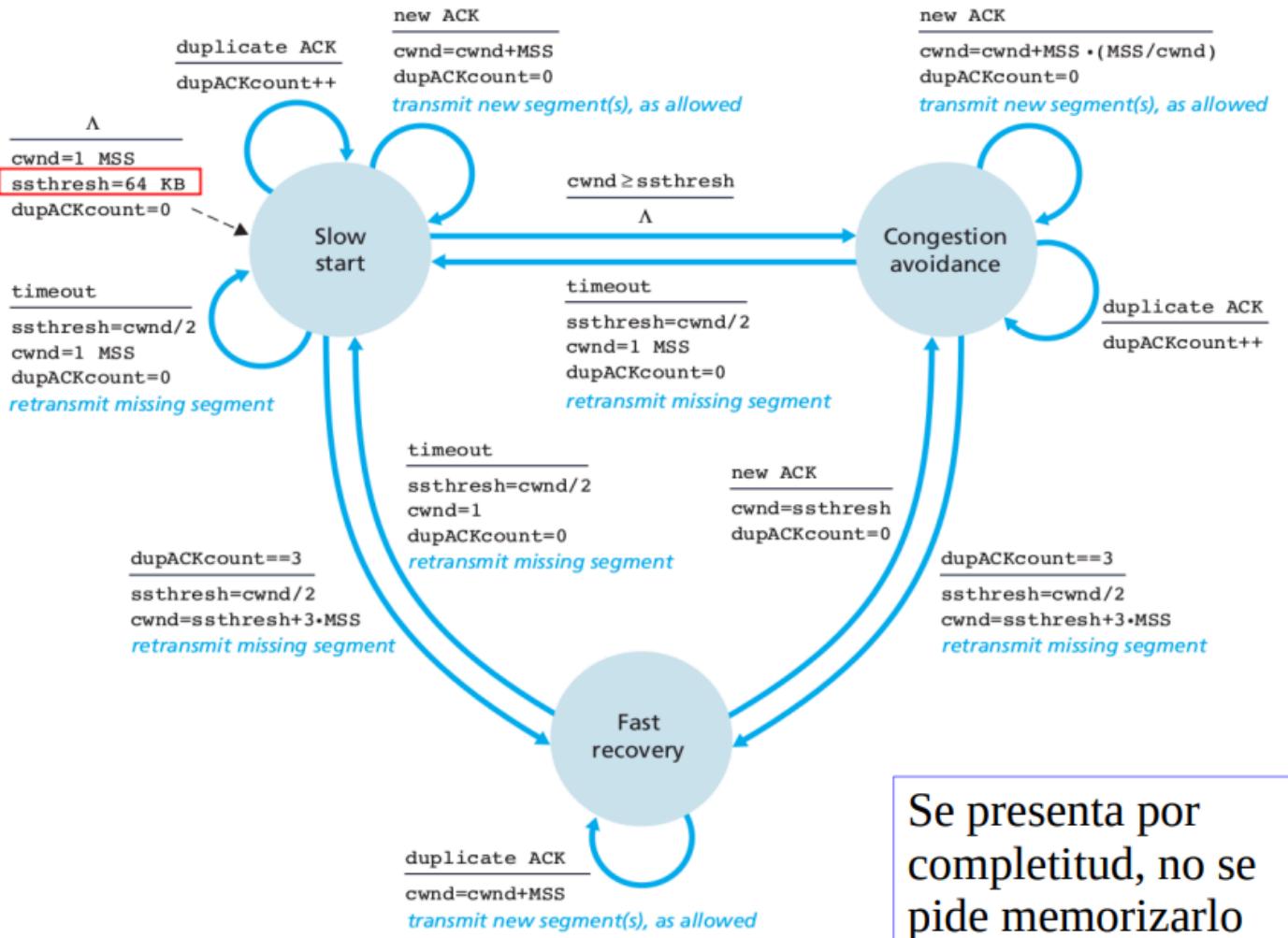




```

void tcp_snd(event_t ev)
{
    switch (ev) {
        case (init):
            cwnd      = 1;
            ssthresh = INF;
            ...
            break;
        case (newack):
            if (cwnd<=ssthresh) {
                /* Slow Start */
                /* 1 MSS for each ACK */
                cwnd++; // cwnd = cwnd + 1
            } else {
                /* Congestion Avoidance */
                /* 1 MSS for each RTT */
                cwnd = cwnd + 1/cwnd;
            }
            ...
            break;
    ...
    case (timeout):
        ssthresh = cwnd / 2;
        cwnd = 1;
        ...
        break;
    case (3ackdup):
        fast_retrans_fast_recover();
        ssthresh=cwnd / 2;
        cwnd = ssthresh + 3;
        ...
    }
}

```



Se presenta por
completitud, no se
pide memorizarlo

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to "Congestion Avoidance"	Resulta en una duplicación de CongWin cada RTT.
Congestion Avoidance (CA)	ACK receipt for previously unacked data	CongWin = CongWin + MSS * (MSS/CongWin)	Aumento aditivo, resulta en aumento de CongWin en apox. 1 MSS cada RTT
SS or CA	Loss event detected by triple duplicate ACK	Threshold = CongWin/2, CongWin = Threshold, Set state to "Congestion Avoidance"	Recuperación rápida, implementando reducción multiplicativa. CongWin no caerá a 1 MSS.
SS or CA	Timeout	Threshold = CongWin/2, CongWin = 1 MSS, Set state to "Slow Start"	Ingresa a Partida Lenta (slow start)
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin y Threshold no cambian

AIMD (Additive Increase/Multiplicative Decrease)

De acuerdo al modelo de TCP parece ser estable y justo

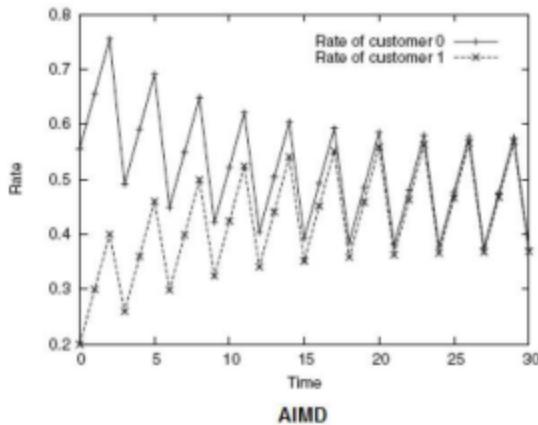
Utiliza un enfoque AIMD, crece de forma aditiva con ACK positivos y decrece de forma multiplicativa ante 3 ACK DUP (a la mitad)

Se tratan de autorregular entre flujos

$$a_i > 0; a_d = 0$$

$$b_i = 1; 0 < b_d < 1$$

$$w(t+1) = \begin{cases} a_i + b_i w(t) & \text{si } (+) \\ a_d + b_d w(t) & \text{si } (-) \end{cases}$$



Si se recibe un ACK correcto, $w(t+1)$ (la próxima ventana) será igual $w(t)$ (ventana actual) * 1 (b_i) + a_i (incremento de la ventana)

Si se reciben 3DUP ACKs, $w(t+1)$ (la próxima ventana) será igual $w(t)$ (ventana actual) * un valor entre 0 y 1 (b_d) + 0 (a_d)

New Reno

TCP Reno ante la pérdida de múltiples segmentos probablemente termine en timeout (RTO expira) y vuelve a Slow Start

TCP New Reno, última versión RFC 6582(2012) hace obsoletas RFC 3782 (2004), RFC 2582(1999) S. Floyd, T. Henderson

New Reno trata de recuperarse de la pérdida de varios segmentos dentro de una misma ventana de una manera más eficiente

Una vez que está en Fast Recovery, permite enviar varios segmentos perdidos y recuperarse de ACK parciales (las confirmaciones parciales implican que me confirmaron el

segmento que causó FRT, pero siguen sin confirmar todos los segmentos que envíe, indicando que hubo pérdidas entre medio)

New Reno obtiene buen rendimiento con pérdida en varios segmentos y sin SACK

Control de congestión por la red

No tan implementado

TCP se monta sobre IP, best effort protocol, no considera en su origen QoS

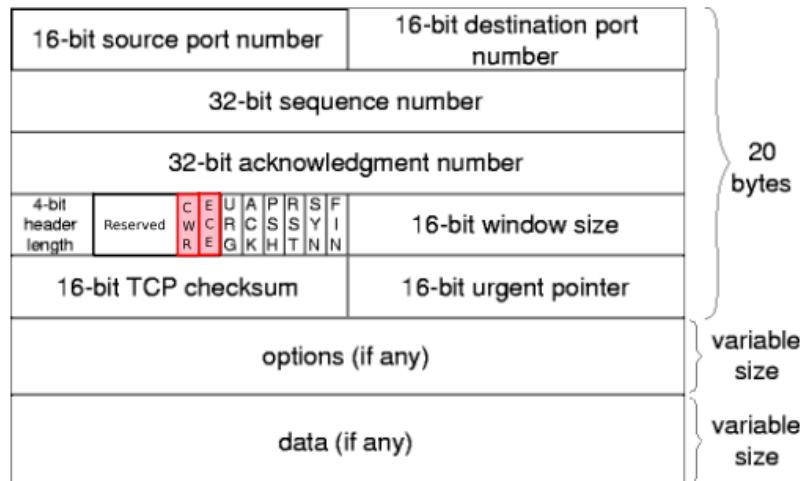
Se utilizan campos reservados para marcar tráfico

TCP+IP con ECN es un mecanismo más adecuado

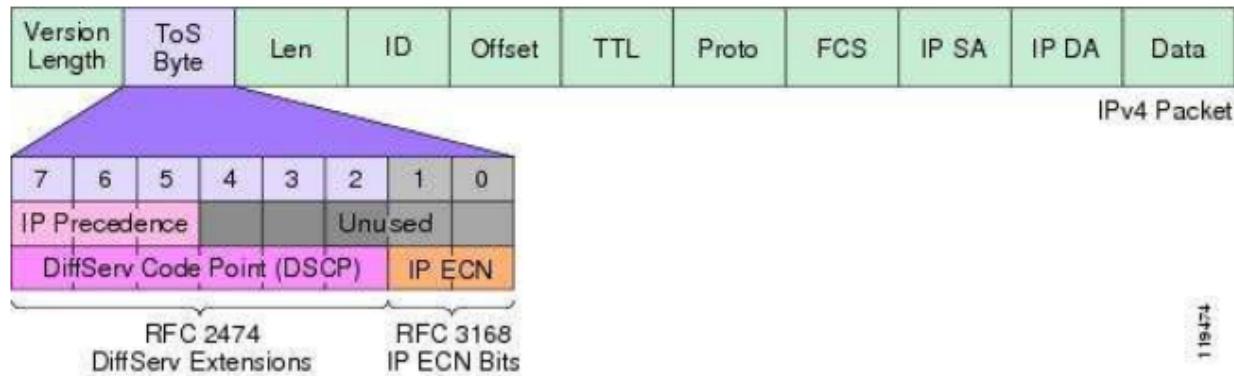
ECN (Explicit Congestion Notification)

Se usan los flags CWR (Congestion Window Reduced) y ECE (Explicit Congestion Notification Echo)

Segmento TCP Marcado con ECE y CWR



Datagrama IP Marcado con ECN: ECT, CE



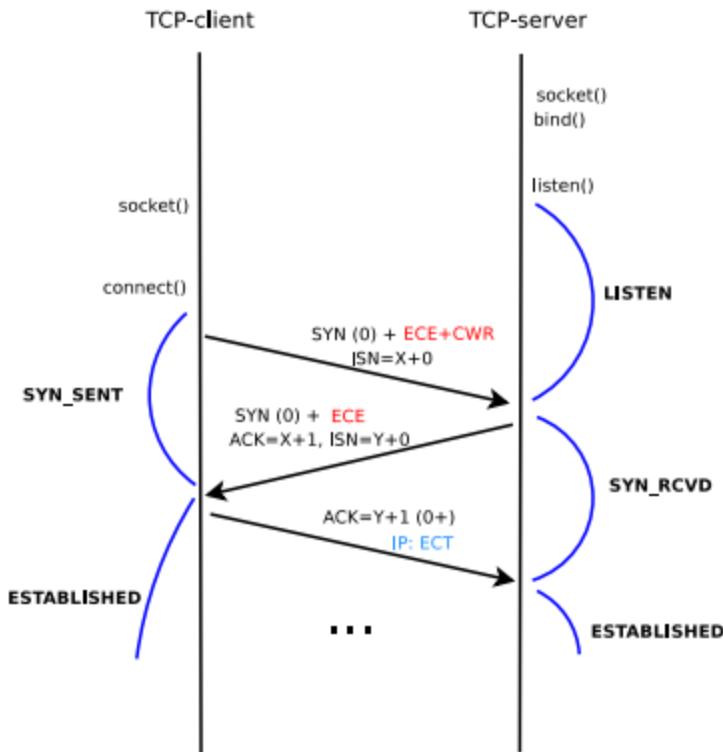
Funcionamiento

Al establecerse la conexión TCP, se negocia capacidad de ECN

En el SYN: ECE = 1 y CWR = 1 (ECN - Echo)

En el SYN+ACK: ECE = 1 y CWR = 0

TCP ECN Setup



Luego en datagramas IP (en los bits ECN) se configuran $ECT(0) = 01$ o $ECT(1) = 10$ (ECN-Capable Transport), ambos indican que ECN está habilitado

Si un router (nodo intermedio) detecta congestión, tamaño de cola por encima de umbral aplica RED (Random Early Detection)

Al aplicar RED, en lugar de descartar, marca $CE = 11$ (Congestion Experienced) (en los bits ECN)

El receptor al recibir $CE = 11$, setea ECE en el próximo segmento de ACK

El emisor detecta $ACK = 1$ y $ECE = 1$ y aplica TCP CC clásico como si el mensaje hubiese sido descartado

El emisor también configura $CWR = 1$ (Congestion Window Reduced) para notificar que reaccionó