TCP

- Introducción
- Diseño
- Formato de Segmento
- Establecimiento de la Conexión
- Sliding Windows y TCP
- Control de Flujo

INTRODUCCIÓN (1)

- Protocolo de transporte que ofrece un servicio confiable, orientado a la conexión y al flujo de bytes.
- Las aplicaciones no tienen que preocuparse de ordenar o retransmitir los datos, ni de dividir los datos en pedazos.
- Las conexiones son bidireccionales, con un flujo en cada dirección.

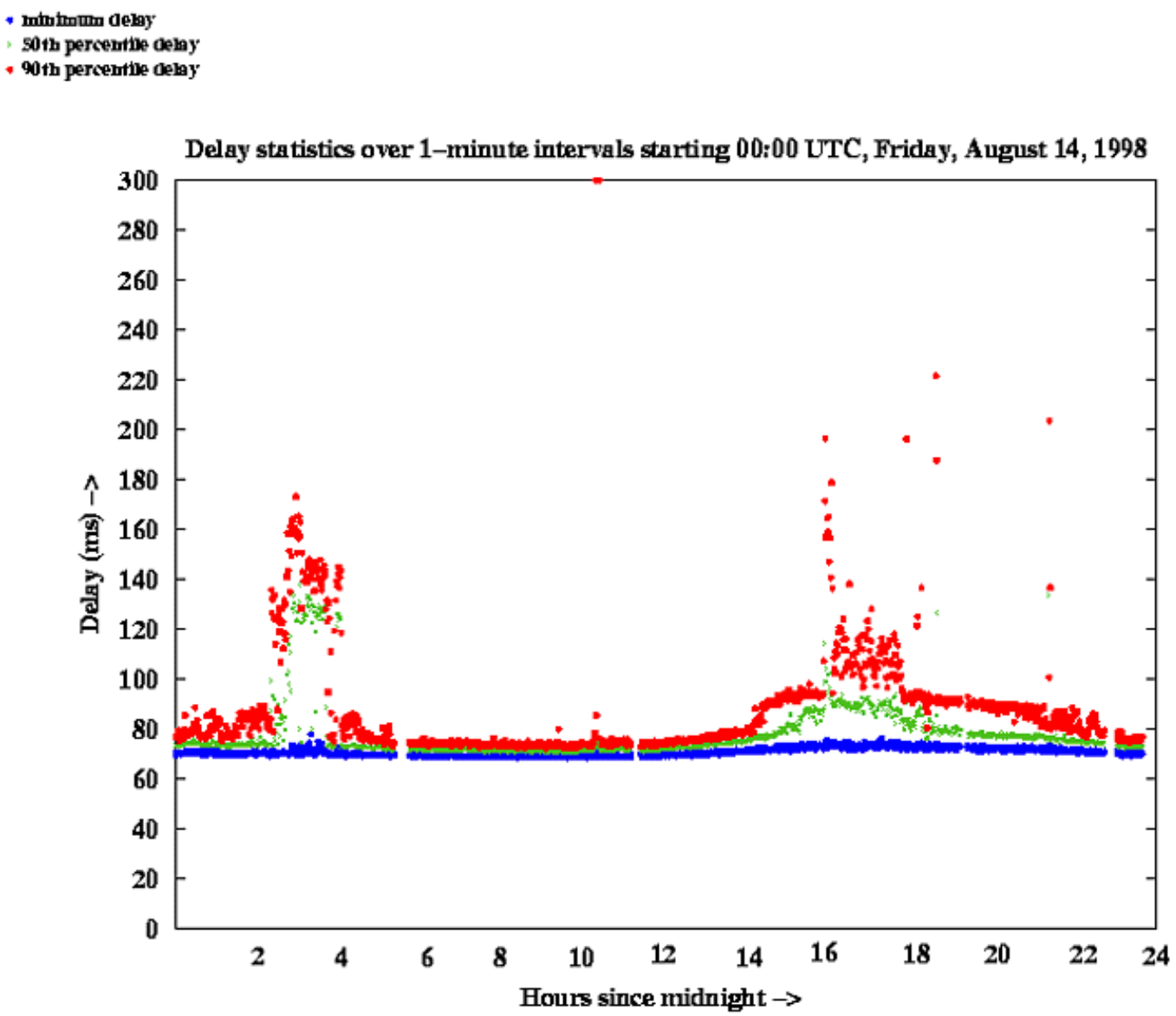
INTRODUCCIÓN (2)

- Incluye control de flujo en cada dirección (de manera de que el receptor pueda limitar la cantidad de datos despachada por el emisor).
- Incluye control de congestión, que le permite ajustar la velocidad con que se envían datos para evitar saturar la red.

DISEÑO (1)

- El protocolo de ventana de corredera es esencial en el funcionamiento de TCP, pero la implementación supone una serie de retos:
  - Establecimiento y término explícito de la conexión entre las partes.
  - RTT variables (muy variables)
  - Congestión debido a tráfico (bandwidth variable)
  - Control de flujo en los extremos (la red siempre acepta paquetes)
  - Escenciales: cálculo de timeouts y tamaño ventana (variables durante una comunicación)

Timeouts (1)


minimum delay
50th percentile delay
90th percentile delay
Delay statistics over 1–minute intervals starting 00:00 UTC, Friday, August 14, 1998
Delay (ms) ->
300
280
260
240
220
200
180
160
140
120
100
80
60
40
20
0
2
4
6
8
10
12
14
16
18
20
22
24
Hours since midnight ->

Timeouts (2)

- El Timeout controla el tiempo de espera que debiera ser mayor que el RTT
- Pero el RTT tiene varianza muy alta en caso de congestión (colas en routers)
- En ese caso, incluso 2*RTT promedio puede ser poco timeout
- Debemos estimar la varianza junto con el RTT promedio
- Y en qué intervalos? (historia reciente vs largo plazo)
- Timeout = RTT+K*MDEV

Timeouts (3)

- ¿Por qué la varianza es alta?
- -> Sin congestión: RTT es constante
- -> Congestión hace que las colas de espera en los routers crezcan: mientras más RAM, más varianza
- -> Congestión extrema genera pérdidas, cuando se acaba la RAM
- Mientras más routers en el camino, mayor probabilidad de varianzas altas

Timeouts (4)
Ver: RFC6298

- DIFF = RTT_sample – RTT;
- RTT = RTT + alfa*DIFF;
- MDEV = (1-beta)*MDEV + beta*abs(DIFF);
- TIMEOUT = RTT + 4*MDEV
- Alfa = 1/8 típicamente y beta = 1/4

=> ¿historia 8 veces más fuerte que instante?

- Problema: si tuve que retransmitir un segmento, ¿cómo sé cual es su RTT?
- Karn: no actualizar RTT en ese caso
- Duplicar el timeout (max 120 s)

Timeouts (5)

- Inicialmente: TIMEOUT = 3s
- Al recibir primer ACK:
  - RTT = RTT_sample
  - MDEV = RTT/2
- Opción de timestamp en TCP permite aceptar ACKs duplicados recalculando RTT y MDEV
- Pero, con Go-Back-N:
  - Si se pierde un paquete de la ventana,
  - Recibo ACK para el anterior múltiples veces
  - ACKs duplicados => pérdida (fast retransmit)

Ventana Congestión (1)

- En 1988 Van Jacobson descubrió la congestión: si todos retransmitían sus ventanas todo empeoraba
- En TCP suponemos pérdida == congestión
- Algoritmo AIMD (Additive Increase, Multiplicative Decrease), frente a congestión:
  - *Multiplicative Decrease*: cada retransmisión decrementa la ventana de transmisión a la mitad y multiplica el timeout por dos. Converge a un segmento muy rápido.
  - *Additive Increase*: cuando comienzo a recibir los ACKs, incremento la ventana de a un segmento por RTT!

Ventana Congestión (2)

- *Slow Start*: Cuando no hay congestión, la ventana crece en un segmento por ACK recibido
- Esto no es nada de *Slow:* Cada ACK corre la ventana en un segmento y la agranda en otro => genera el envío de dos segmentos, que generan dos ACKS los que generan cuatro segmentos… En realidad es exponencial
- Aquí hablamos en segmentos, en TCP los contadores siempre son en bytes
- Usaremos dos ventanas en el emisor: la normal y la de congestión. Parten iguales, pero la de congestión manda. La normal trata de aproximar el BDP.
- Pueden haber paquetes transmitidos, sin ACK, fuera de la ventana de congestión (y dentro de la normal) que NO se retransmiten

Ventana Congestión (3)

- *Slow Start*/AIMD en pseudo-código:

```
Init: cwnd = 1;
    ssthresh = infinite;
ACK y RTT desde el último cambio:
if(cwnd < ssthresh) /* slow start */
     cnwd = cwnd + 1; /* en segmentos */
else /* congestion avoidance: Additive
                                    Increase */
     cwnd = cwnd +1/cwnd;
TIMEOUT: /* Multiplicative decrease */
     ssthresh = cwnd/2;
     cwnd = 1;
```

Ventana Congestión (4)

The big picture (with timeouts)

Initially:

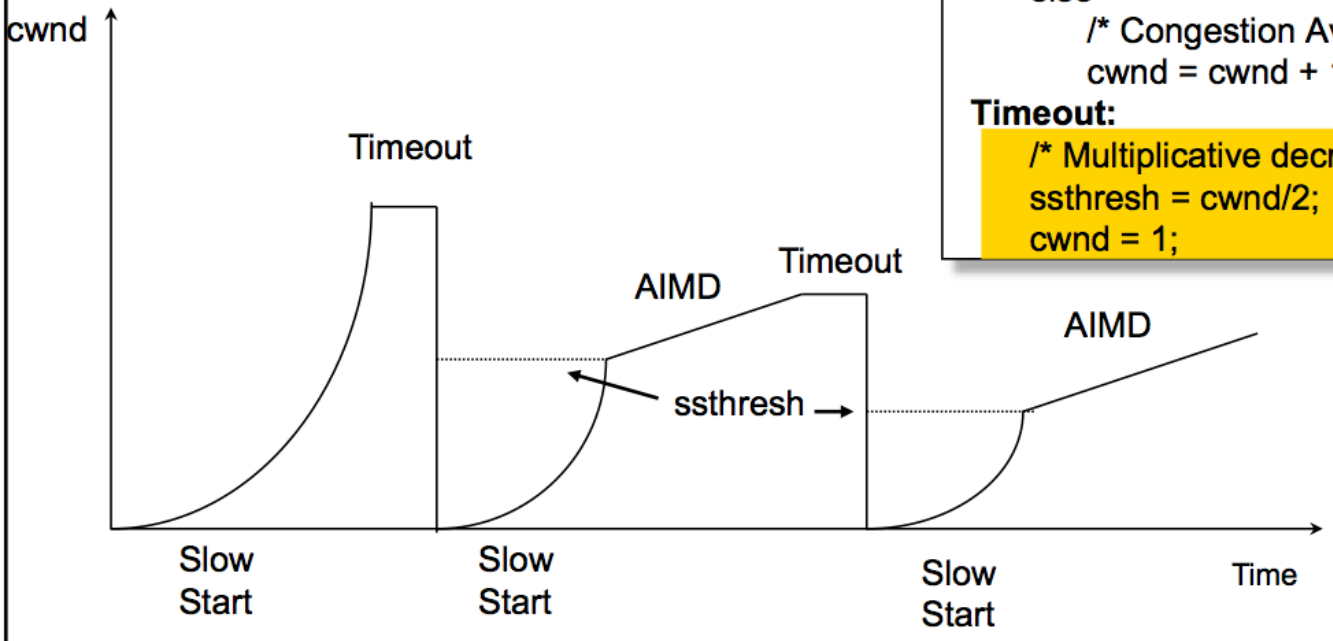
```
cwnd = 1;
ssthresh = infinite;
```

New ack received:

```
if (cwnd < ssthresh)
    /* Slow Start*/
    cwnd = cwnd + 1;
else
    /* Congestion Avoidance */
    cwnd = cwnd + 1/cwnd;
```

Timeout:

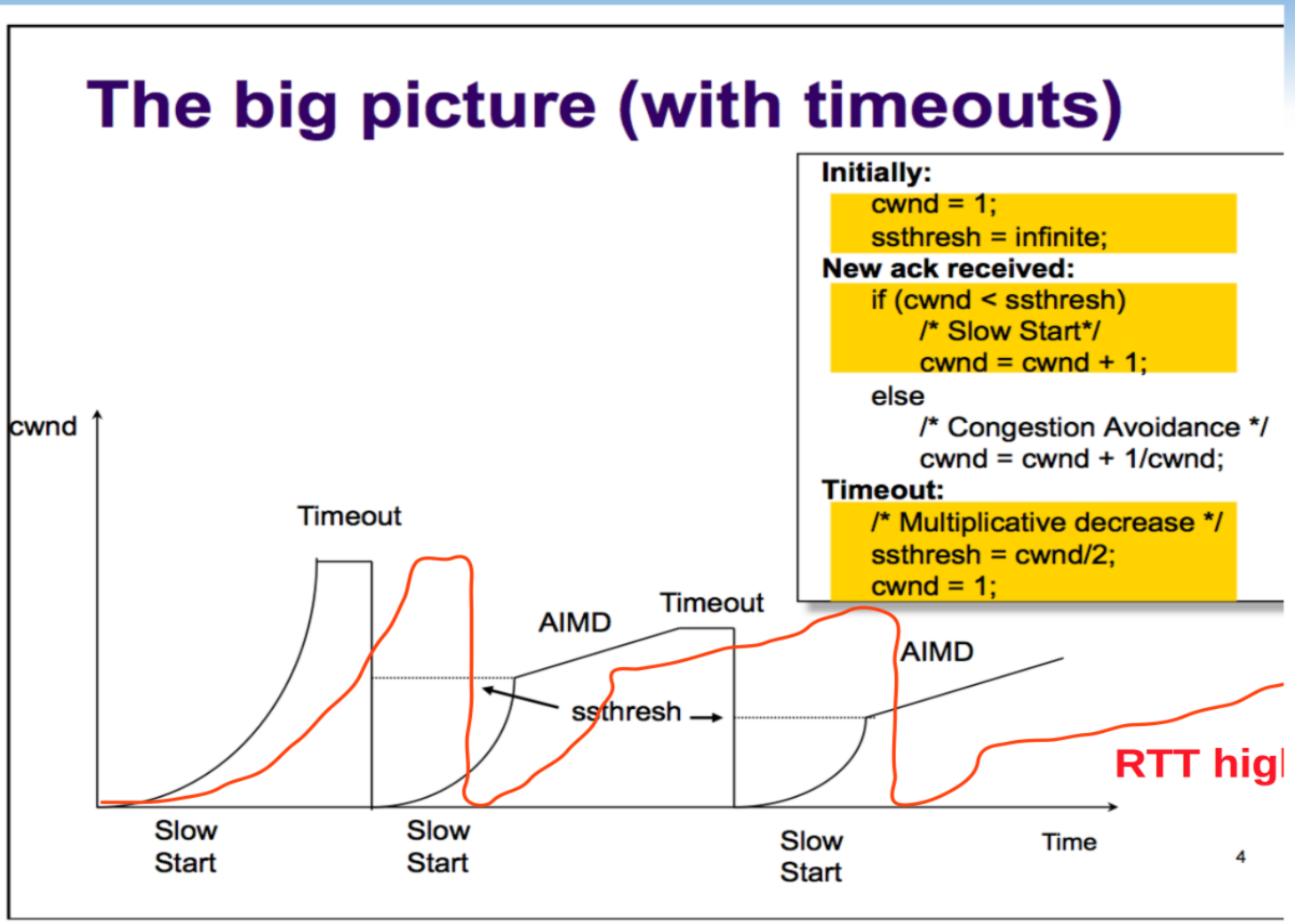
```
/* Multiplicative decrease */
ssthresh = cwnd/2;
cwnd = 1;
```



Ventana Congestión (5)

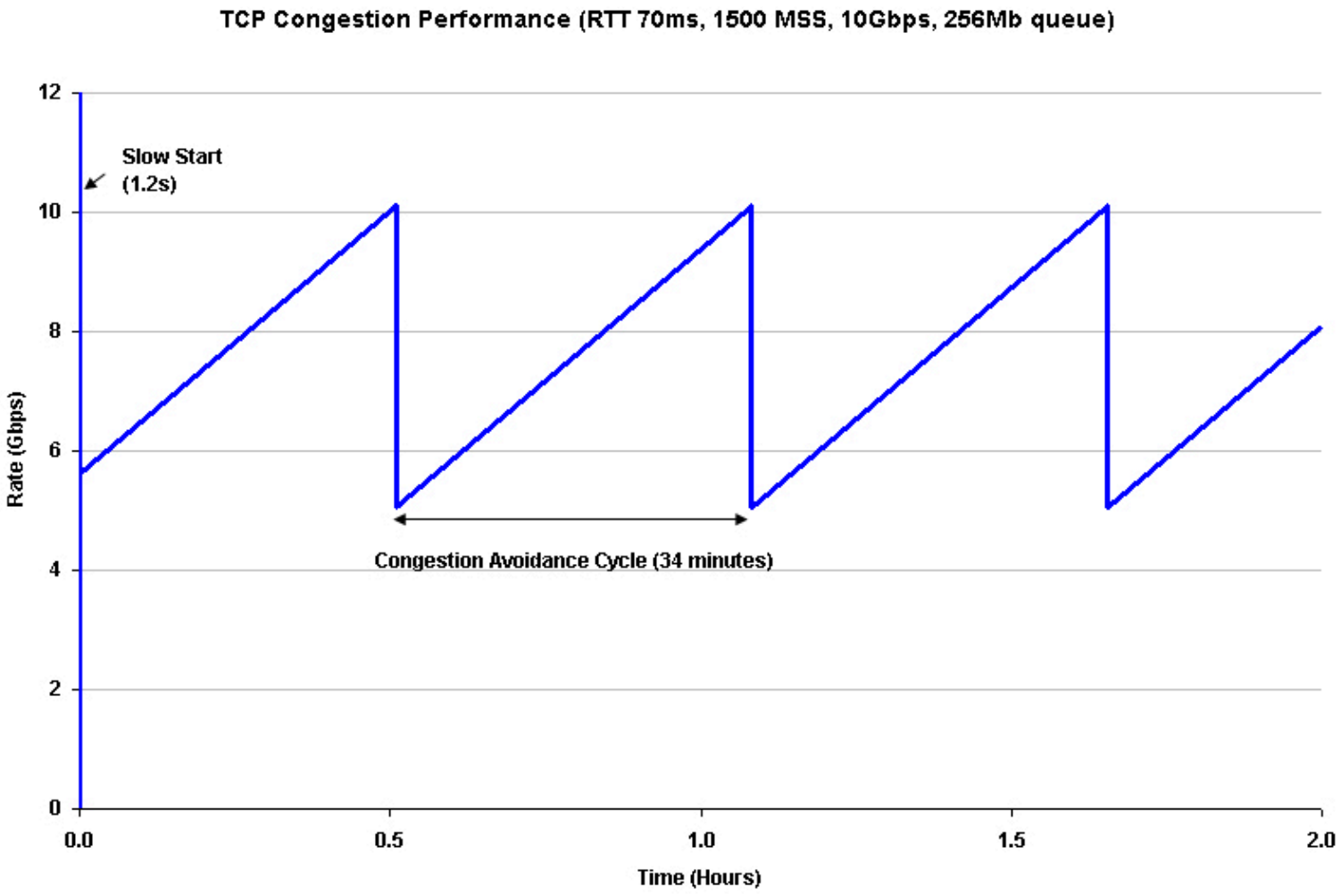
- *Fast Retransmit:* 3 ACKs duplicados => timeout
- *Fast Recovery:* hacer cwnd = ssthresh en vez de 1
- Ventana inicial + grande
- SACK: Selective Repeat en vez de Go-back-N
- Varios “sabores” de TCP (Reno, Vegas, BIC, CUBIC, etc.)
- En linux se puede elegir
- Pero aún tenemos varios problemas pendientes: ¿es posible hacer un TCP para todos los escenarios posibles?
- Mucha investigación: DCTCP, …, TFRC:
- Throughput ~= (1/RTT)*sqrt(3/2p)

Ventana Congestión (6)
El problema del RTT



Ventana Congestión (7)

Ancho de Banda se ve como una “sierra” en el tiempo:



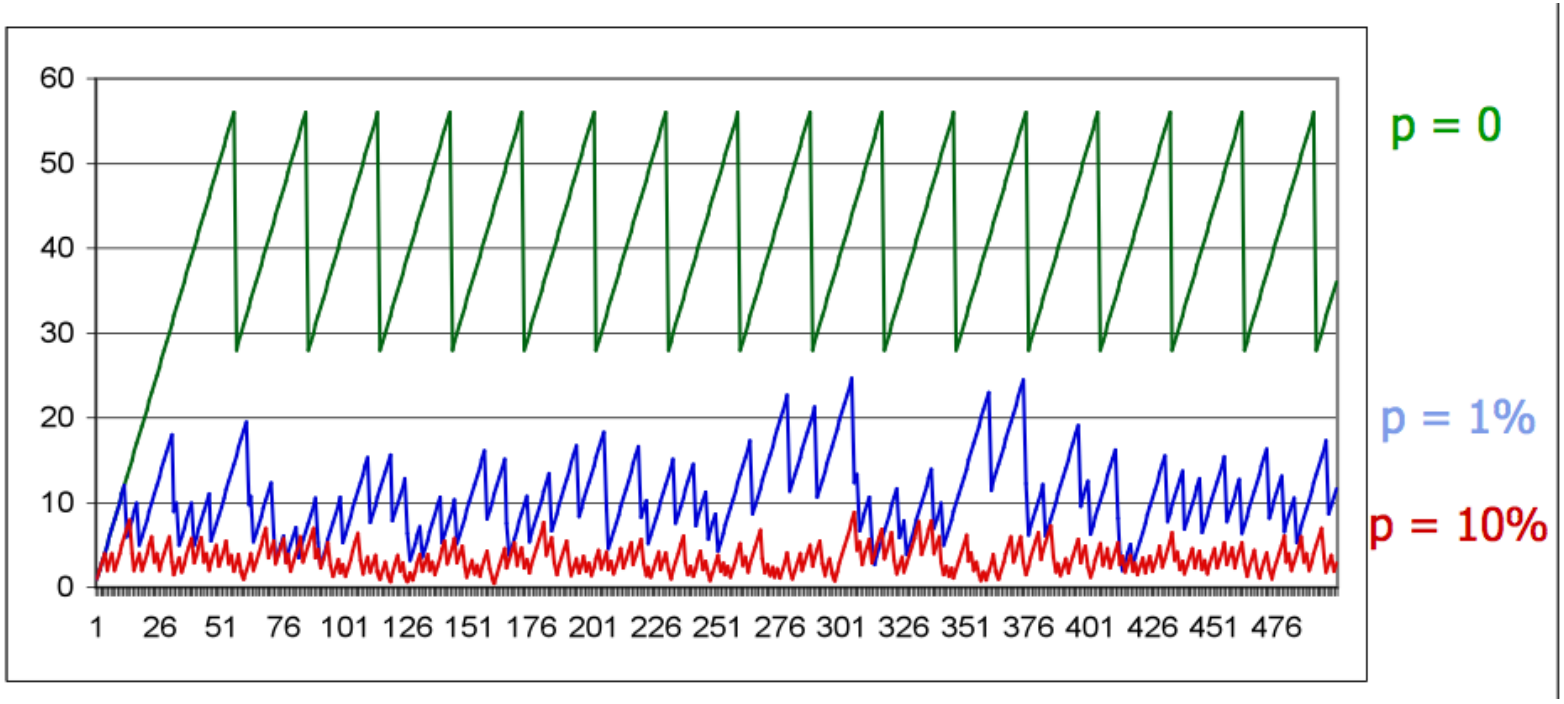
Ventana Congestión (8)

- Enlace de 1.5 Mbps, 70ms, en 0.5s llega a máximo, después de 150 Kbytes
- Enlace de 10Gbps, 70ms, demora 34 mn (sin pérdida!), después de 2 Tbytes!
- Enlace de 100 Mbps, 250ms, 10mn!
- -> TCP entre Chile y USA demora al menos 5 minutos en estabilizarse
- -> Imposible sacar ancho de banda completo
- -> Al sacar RTT de la ecuación dañamos la *fairness*
- -> Se usan varias conexiones simultáneas en esos casos: ¿trampa? (ej: speedtest.net)

Ventana Congestión (9)

- TCP es un buen ciudadano: fairness, *TCP-friendly*
- Pero si la pérdida es física (señal inalámbrica, por ejemplo) el resultado es nefasto => stop-and-wait con timeouts enormes
- Discusión hoy: *cross-layer information,* se trata de que TCP mire el layer físico para ver si los errores son locales, saltándose IP
- En caso límite, conviene matar una conexión y empezar otra vez, cuando la señal es fuerte

Ventana Congestión (10)
Pérdidas Físicas (ruido)



Ventana Congestión (11)

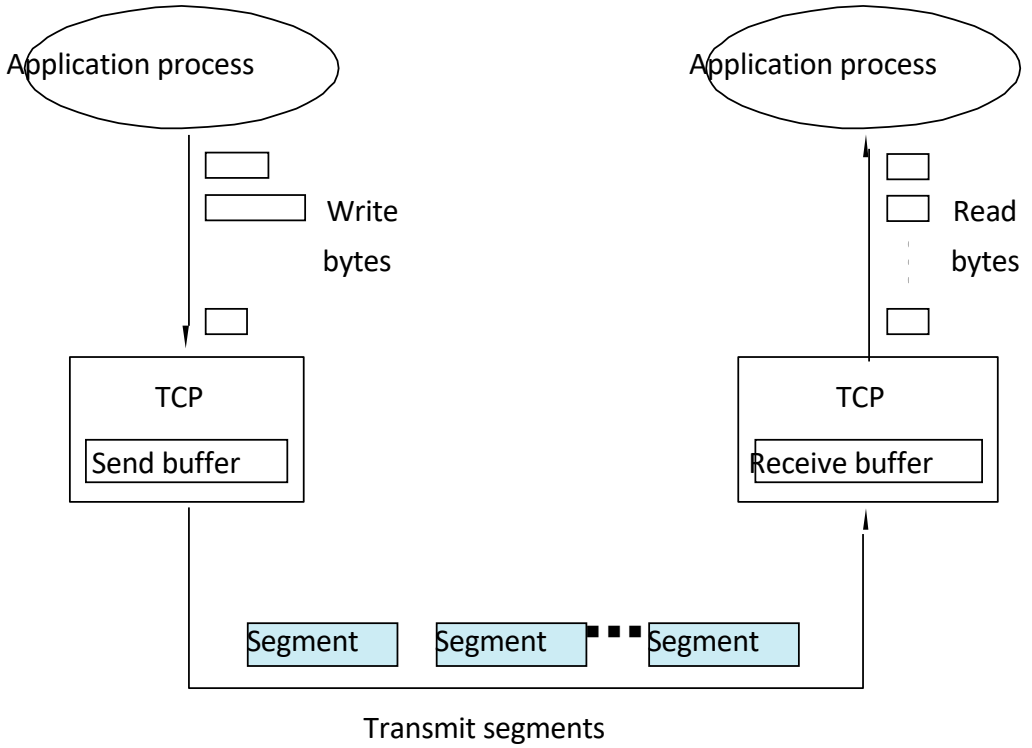
- El manejo de congestión en TCP se considera aún un problema de investigación
- No hemos encontrado mejores soluciones genéricas que Van Jacobson 1988-1990
- Muchas propuestas específicas
- Timeout, Backoff, ventana de congestión han operado increíblemente bien por más de 20 años!

Ventana Congestión (12) Bibliografía

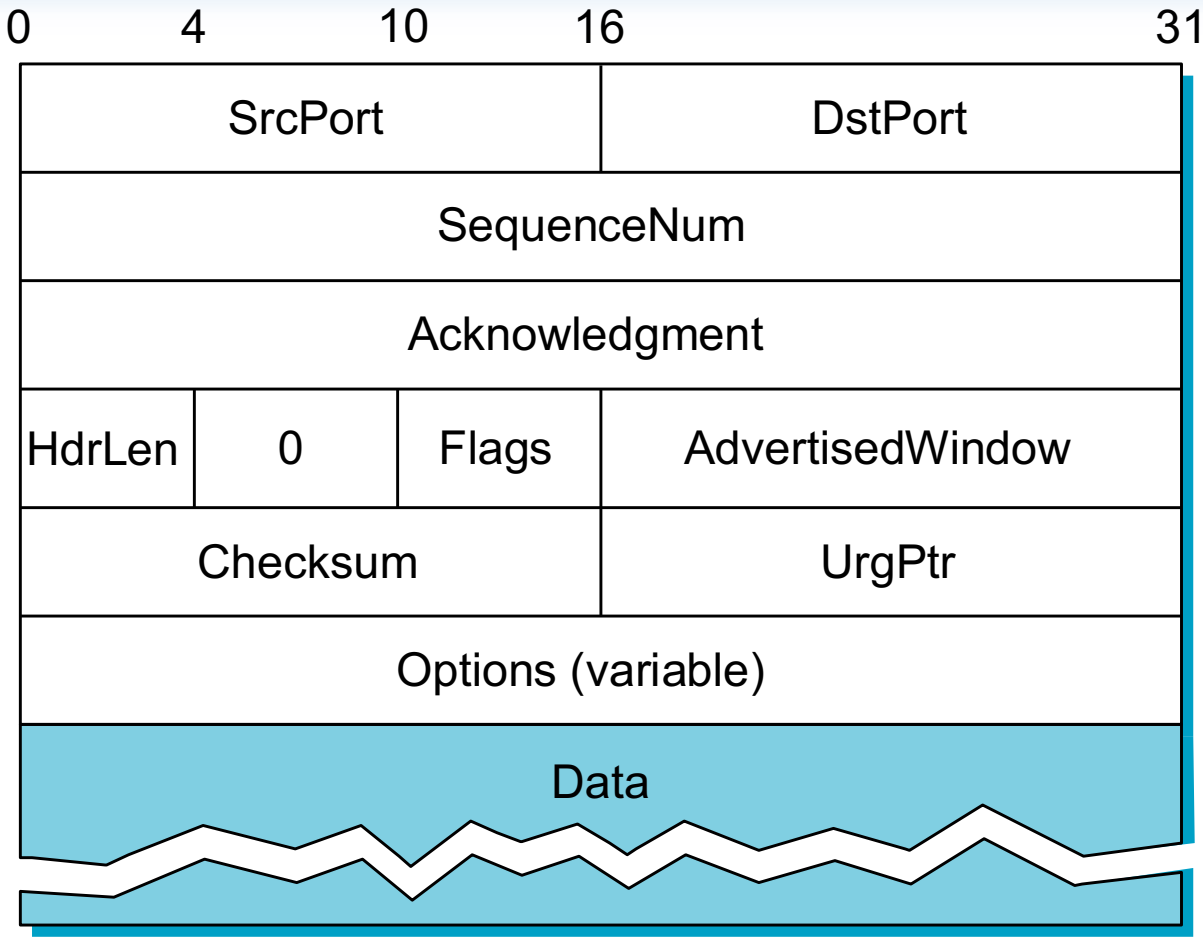
- Van Jacobson, 1988: https://ee.lbl.gov/papers/congavoid.pdf
- RFCs: rfc4614, rfc6077, rfc7323
- Material pirateado para este curso:
- http://www.potaroo.net/ispcol/2005-06/faster.html
- http://inst.eecs.berkeley.edu/~ee122/fa09/
- http://www.pcvr.nl/tcpip/tcp_time.htm
- CUBIC: TCP “moderno”: https://www.cs.princeton.edu/courses/archive/fall16/cos561/papers/Cubic08.pdf

FORMATO DEL SEGMENTO (1)

- TCP es orientado al flujo de bytes, pero para transmitir utiliza bloques de bytes llamados segmentos, donde cada uno transporta un "*pedazo*" del flujo de bytes.

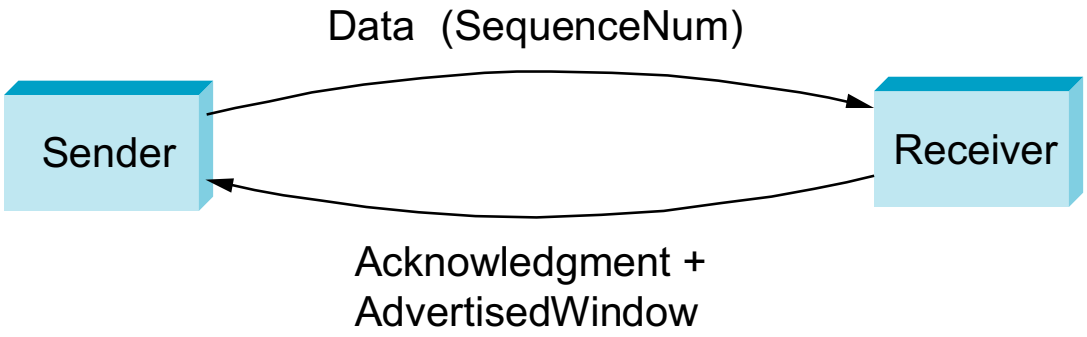


FORMATO DEL SEGMENTO (2)



FORMATO DEL SEGMENTO (3)

- SrcPort y DstPort son los puertos que identifican completamente el flujo: <SrcIP, SrcPort, DstIP, DstPort> identifican una conexión.
- Acknowledgment, SequenceNum y AdvertisedWindow están relacionados con el protocolo de ventana de corredera. SequenceNum indica la secuencia del primer byte en el segmento. Ack indica que el emisor está esperando ese número de secuencia. AdvWin indica la cantidad de espacio disponible en el receptor.



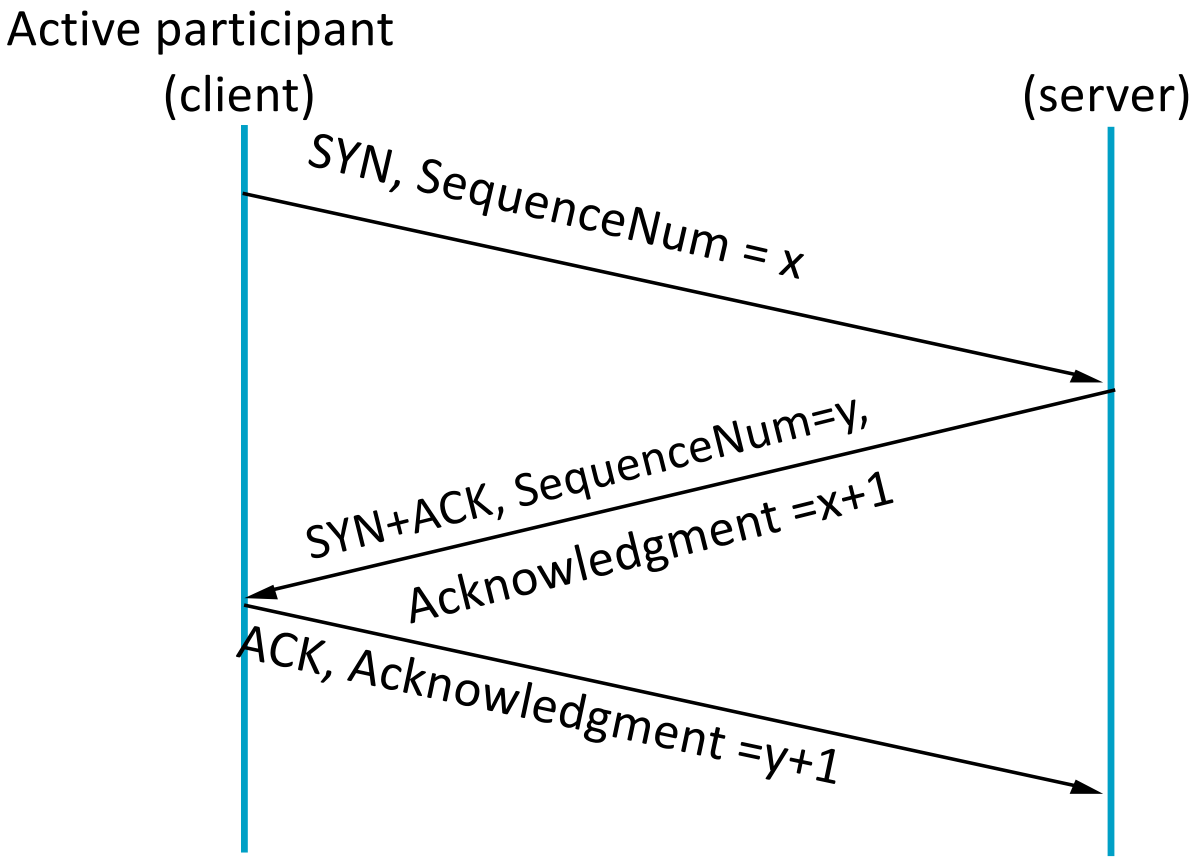
ESTABLECIMIENTO DE LA CONEXIÓN (1)

- Una conexión tiene un cliente, que realiza una “apertura” activa y un servidor que está pasivamente esperando un requerimiento.
- Sólo después del establecimiento de la conexión se transmiten datos.
- En cuanto el cliente ha terminado de enviar datos, inicia una ronda de mensajes para cerrar.
- Aún así, es posible que una de las partes cierre (no desee enviar más datos) pero el otro lado mantendrá su “dirección” del flujo abierta para continuar enviando datos.

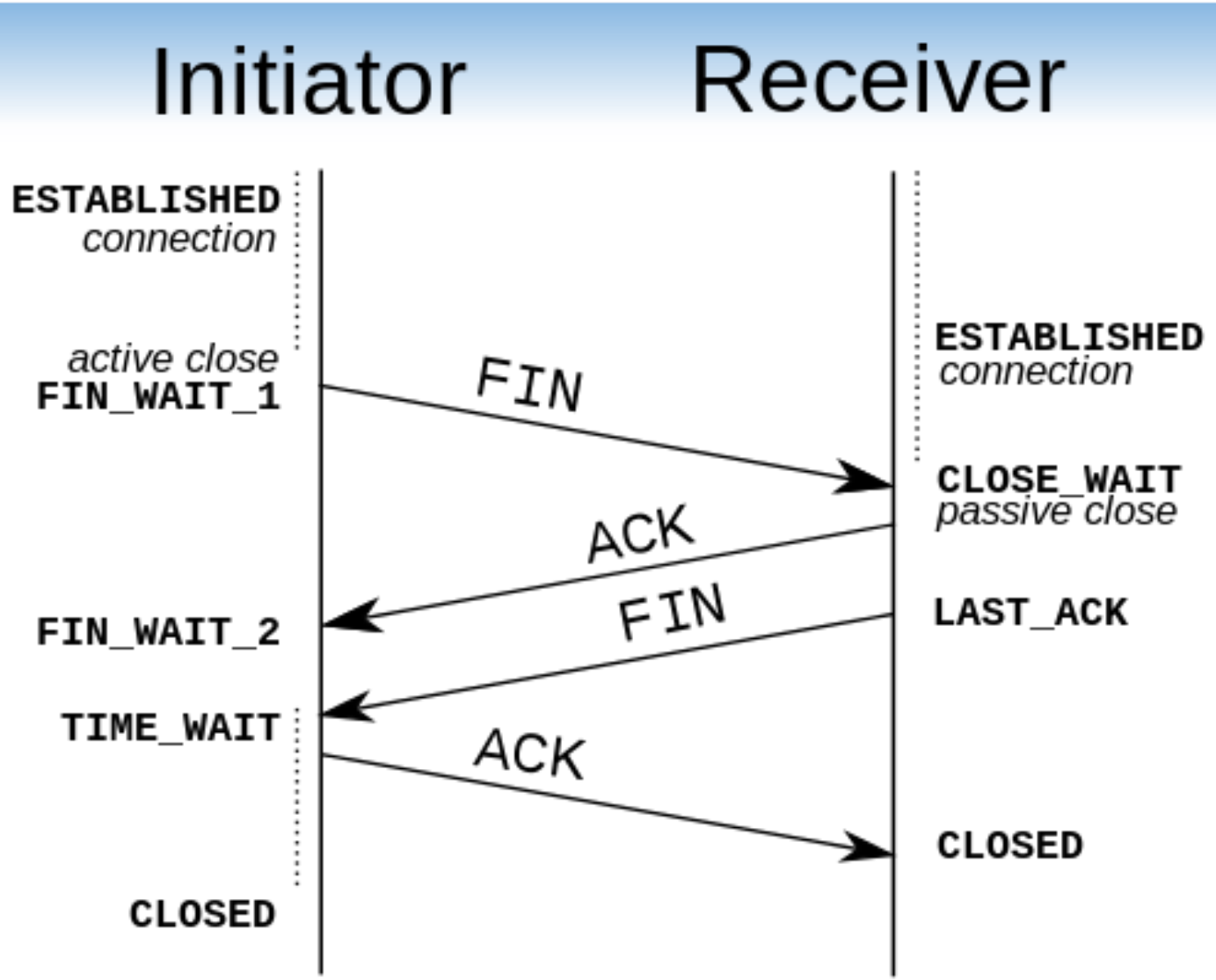
ESTABLECIMIENTO DE LA CONEXIÓN (2)

- Negociación de tres vías
  - El proceso de creación de una conexión en TCP se llama three-way handshake.
  - El objetivo es que las partes se pongan de acuerdo en ciertos parámetros que gobernarán la conexión, algunos de ellos definitivos y otros no.
    - Se fijan los números de secuencia en cada sentido, el tamaño de las ventanas y otras opciones.
    - El objetivo de fijar los números de secuencia (y no partir cada conexión con un número fijo) es evitar que un segmento retrasado de una conexión anterior pueda confundir o interferir con el actual estado.
  - Para los dos primeros segmentos enviados (SYN y SYN+ACK) existe retransmisión.

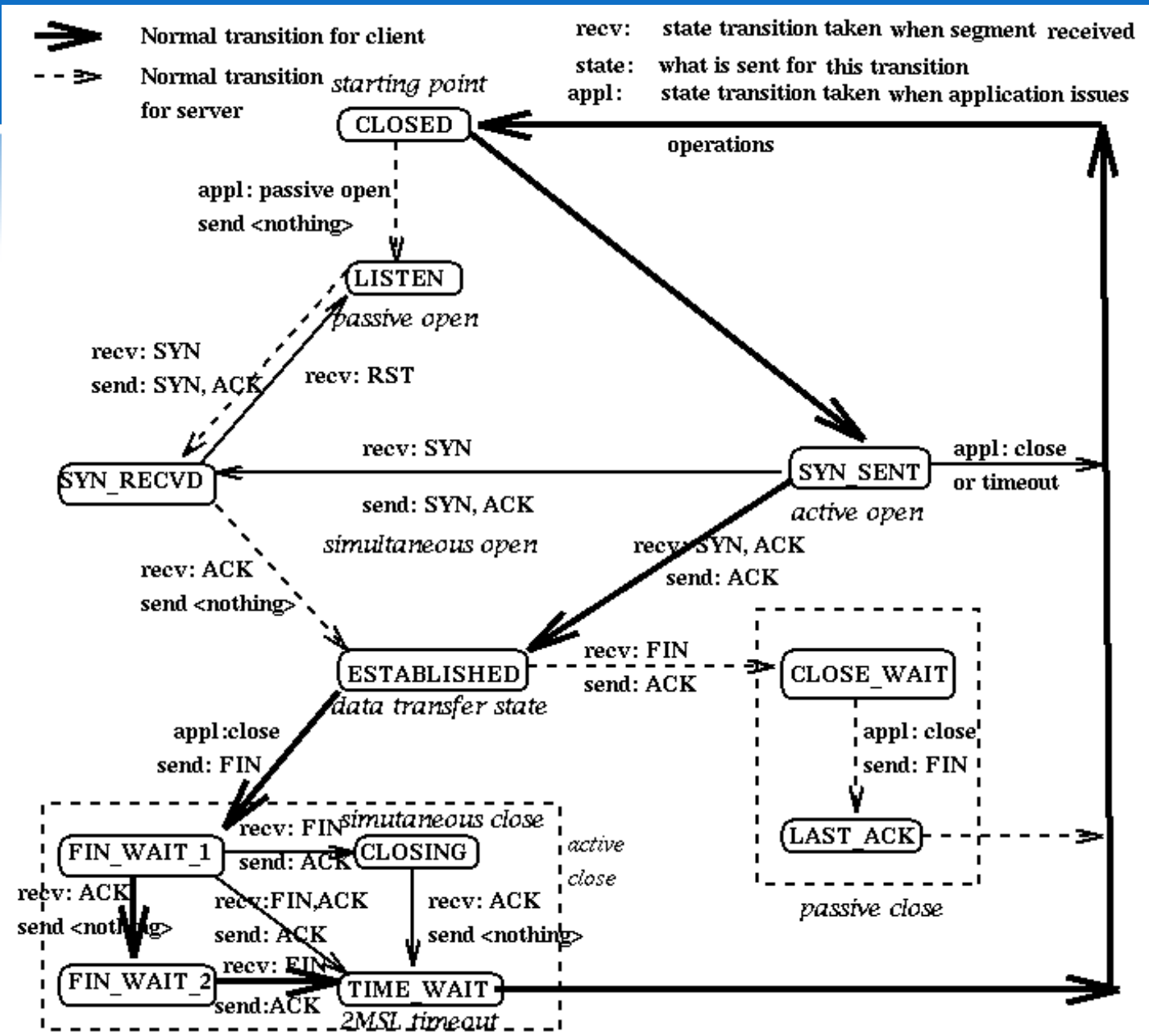
ESTABLECIMIENTO DE LA CONEXIÓN (3)



FIN DE LA CONEXIÓN



Estados TCP

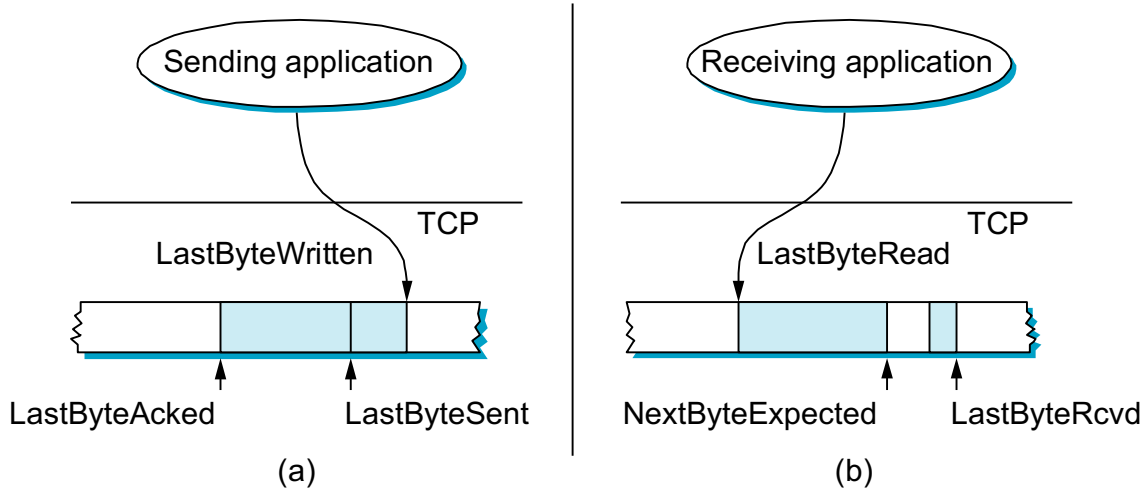

Normal transition for client
Normal transition for server
recv: state transition taken when segment received
state: what is sent for this transition
appl: state transition taken when application issues operations
starting point
CLOSED
appl: passive open
send <nothing>
LISTEN
passive open
recv: SYN
send: SYN, ACK
recv: RST
recv: SYN
send: SYN, ACK
simultaneous open
SYN_RECVD
SYN_SENT
appl: close
or timeout
active open
recv: SYN, ACK
send: ACK
recv: ACK
send <nothing>
ESTABLISHED
data transfer state
recv: FIN
send: ACK
CLOSE_WAIT
appl: close
send: FIN
LAST_ACK
passive close
appl:close
send: FIN
simutaneous close
recv: FIN
send: ACK
FIN_WAIT_1
CLOSING
active close
recv: ACK
send <nothing>
recv:FIN,ACK
send: ACK
recv: ACK
send <nothing>
recv: FIN
send:ACK
FIN_WAIT_2
TIME_WAIT
2MSL timeout

Estados TCP

- La cuadrupla: <SrcIP, SrcPort, DstIP, DstPort> identifica una conexión
- Eso permite tener más de un cliente conectado al mismo servidor, incluso desde la misma IP de origen
- Los S.O. tratan de no re-utilizar los ports demasiado rápido para no confundir paquetes de otras conexiones
- Una conexión TCP requiere 3+4 paquetes para la conexión a lo menos, y dos RTT: ¡para enviar un solo paquete de datos no es muy buena idea!

VENTANAS, CONGESTIÓN Y TCP (1)

- Los tamaños de las ventanas no son fijos, sino variables y se establecen inicialmente en la negociación previa a la conexión (utilizando el campo *AdvertisedWindow* del header TCP.



SLIDING WINDOWS Y TCP (2)

- Cada punta mantiene un buffer de envío y recepción. Por simplicidad estudiaremos un solo sentido.
- En el emisor tenemos las siguientes variables: LastByteAcked, LastByteSent y LastByteWritten. Se cumple que LastByteAcked <= LastByteSent y que LastByteSent <= LastByteWritten.
- En el receptor tenemos: LastByteRead, NextByteExpected y LastByteRcvd. Se cumple que LastByteRead <= NextByteExpected y que NextByteExpected <= LastByteRcvd+1.

CONTROL DE FLUJO (1)

- Se definen *MaxSendBuffer* y *MaxRcvBuffer*, para especificar el tamaño de la ventana de envío y recepción respectivamente.
- Recordando “sliding window”, el tamaño de la ventana de envío especifica la cantidad de datos que se pueden enviar sin esperar por un ACK.
- El receptor debe cumplir que

$$\text{LastByteRcvd} - \text{LastByteRead} \leq \text{MaxRcvBuffer}$$

para evitar saturar su buffer de recepción.

- Para ello anuncia una ventana de

$$\text{AdvertisedWindow} = \text{MaxRcvBuffer} - ((\text{NextByteExpected} - 1) - \text{LastByteRead})$$

CONTROL DE FLUJO (2)

- El valor anterior representa el espacio libre en el buffer de recepción. En la medida que se reciben datos, los receptor los confirma.
- El valor de LastByteRcvd se incrementa, lo que puede producir que la ventana se achique.
  - Esto dependerá de que tan rápido pueda la aplicación “consumir” los datos.
    - Si LastByteRead se incrementa con la misma tasa de LastByteRcvd, entonces la ventana conserva tu tamaño.
    - Si no, la ventana anunciada se achicará hasta llegar a cero.

CONTROL DE FLUJO (3)

- Por otro lado, el emisor debe respetar que:

$$LastByteSent - LastByteAcked \leq AdvertisedWindow$$

- por lo que se define una ventana efectiva que limita cuantos datos se pueden enviar.

$$EffectiveWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)$$

- El valor efectivo tiene que ser mayor a cero para que salgan datos desde el origen. El emisor debe cumplir con:

$$LastByteWritten - LastByteAcked \leq MaxSendBuffer$$

CONTROL DE FLUJO (4)

- Para no saturar el buffer de emisión. Si un proceso intenta escribir y bytes, pero se cumple que:

$$(LastByteWritten - LastByteAcked) + y > MaxSendBuffer$$

- TCP bloqueará al emisor y no permitirá generar más datos.
- Si el receptor libera espacio y lo anuncia mediante un aviso incluído en un ACK, entonces el origen podrá retomar el envío de datos.
- ¿Qué pasa cuando la ventana de recepción llega a cero? ¿Es el receptor quién anuncia posteriormente la habilitación de espacio?
  - La respuesta es NO. El receptor sólo anuncia la ventana en respuesta a segmentos originados en el emisor.
  - Es por ello que cuando se llega a un anuncio de Ventana=0, el emisor envía a intervalos regulares segmentos con payload de 1 byte y el mismo número de secuencia, para generar la respuesta.
  - Esto sigue el principio de emisor inteligente/receptor tonto y que justifica la no existencia de los NAK.

SILLY WINDOWS SINDROME (1)

- Si un emisor se encuentra detenido a la espera de espacio y le llega un anuncio de que se desocupó MSS/2 bytes, ¿qué se debería hacer?
  - Esperar a que hayan suficientes bytes para llegar un segmento
  - Enviar cuanto antes los datos en espera
- Originalmente se actuaba agresivamente, enviando datos en cuanto hubiera espacio en el receptor. Esto derivó en el problema de la ventana tonta. Si llega una serie de anuncio de ventana de 1 byte, se enviarán segmentos con 1 byte de datos (claramente ineficiente)

SILLY WINDOWS SINDROME (2)

- Para resolver el problema, Nagle propuso el siguiente algoritmo.

```
When the application produces data to send
If both the available data and the windows >= MSS
     send a full segment
else
     if there is unACKed data in flight
          buffer the new data until an ACK arrives
     else
          send all the new data now
```

NÚMEROS DE SECUENCIA (1)

- En TCP, así como lo vimos en el protocolo de ventana original, existen números de secuencia que pueden reutilizarse.
- Descubrimos que el espacio de números debía al menos duplicar el tamaño de ventana posible. En TCP se cumple, pues los números de secuencia son de 32 bits y las ventanas de a lo más 16 bits (64K). *Ver Window Scaling y Timestamps* para ver cómo tener ventanas de 1 Gbyte.
- Aún así, es importante considerar que los números de secuencia pueden dar la vuelta en una conexión, por lo que para distinguir diferentes encarnaciones del mismo segmento, suponemos que no pueden sobrevivir en Internet más allá de MSL segundos (Maximum Segment Lifetime, actualmente en 120 segundos).

NÚMEROS DE SECUENCIA (2)

- Esto pasará en la medida de la rapidez con que utilicemos números de secuencia.

Bandwidth	Time until Wraparound
T1 (1.5 Mbps)	6.4 hours
Ethernet (10 Mbps)	57 minutes
T3 (45 Mbps)	13 minutes
FDDI (100 Mbps)	6 minutes
STS-3 (155 Mbps)	4 minutes
STS-12 (622 Mbps)	55 seconds
STS-24 (1.2 Gbps)	28 seconds

VENTANA DE EMISIÓN

- Con el objetivo de mantener el enlace en máximo uso, los 16 bits de la ventana podrían resultar insuficientes.
- Se ha creado un mecanismo opcional llamado Window Scaling que permite definir una ventana más allá del límite de los 16 bits, para nodos que lo soporten.

Bandwidth	BW Delay Product (Delay = 100ms)
T1 (1.5 Mbps)	18 KB
Ethernet (10 Mbps)	122 KB
T3 (45 Mbps)	549 KB
FDDI (100 Mbps)	1.2 MB
STS-3 (155 Mbps)	1.8 MB
STS-12 (622 Mbps)	7.4 MB
STS-24 (1.2 Gbps)	14.8 MB

ENVÍO DE SEGMENTOS

- TCP mantiene una variable llamada MSS (Maximum Segment Size), que se calcula en base a la MTU local, menos el tamaño del header IP menos el tamaño del header TCP, para evitar la fragmentación.
  - Si se soporta MTU Path Discovery, se utiliza el valor descubierto en vez de la MTU local.
- ¿En qué momento envío un segmento?
  - Tan pronto como ha juntado bytes para llegar un segmento.
  - Cuando el proceso explícitamente lo ha solicitado, a través de la opción PUSH.
  - Cuando se acaba cierto tiempo de espera, donde se envía lo que hubiere estado esperando para despacho.

MTU Path Discover

- Envío Segmentos con el bit de No Fragmentar
- Si recibo ICMP “Need to Fragment”
  - Adapto el MSS al nuevo tamaño (viene en el ICMP, o intento con uno más pequeño)
  - Si timeout: intento con uno más pequeño

- Siempre mantengo el bit de No Fragmentar
- Me adapto en caso de cambio de rutas
- Pero MTU Path nunca crece
- En IPv6 eliminamos fragmentación en ruta por esto mismo