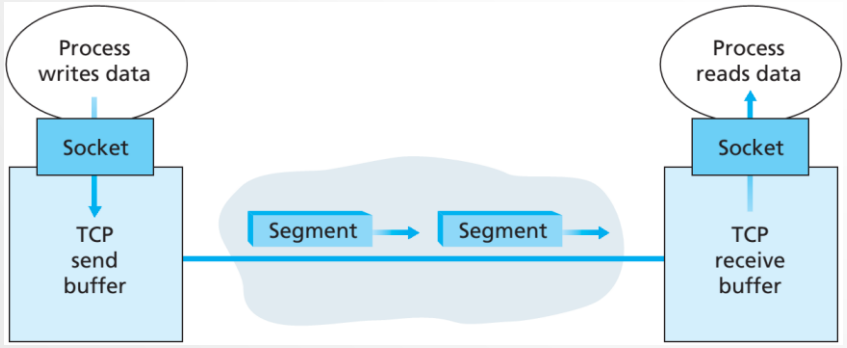
# TCP

# Transmission Control Protocol

Basado en Kurose & Ross – *“Computer Networking. A Top-Down Approach”*

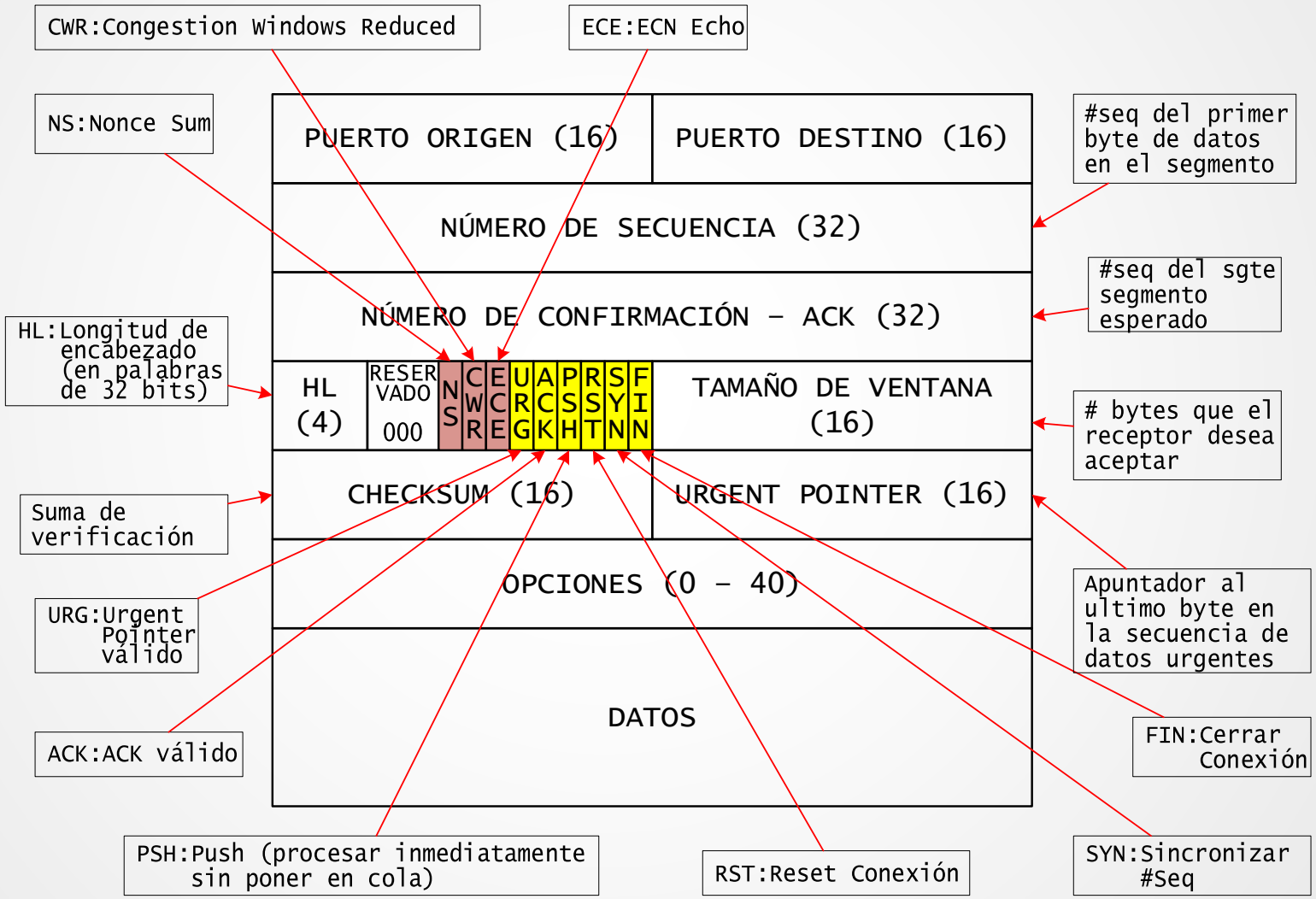
# TCP: RFC: 793, 1122, 1323, 2018, 2581

- Extremo a extremo:
  - Un emisor, un receptor
- Fiable, flujo de bytes en orden:
  - No existe “delimitación de mensajes”
- Encauzado:
  - Control de congestionamiento y flujo.
  - TCP determina el tamaño de la ventana
- Buffers de envío y recepción




- Transmisión full duplex:
  - Flujo de datos bidireccional en la misma conexión
  - MSS: maximum segment size
- Orientado a la conexión:
  - Handshaking (intercambio de mensajes de control) inicia el estado de emisor y receptor antes del intercambio de datos
- Flujo controlado:
  - El emisor no inundará al receptor

# SEGMENTO TCP



# ENCABEZADO TCP

- PUERTO ORIGEN: Puerto del emisor
- PUERTO DESTINO: Puerto del receptor
- NUMERO DE SECUENCIA
  - *Numero de secuencia inicial (si SYN == 1)*
  - *Número de secuencia del primer byte datos del segmento (si SYN == 0)*
- NUMERO DE CONFIRMACION
  - *Si ACK == 1, siguiente NUMERO DE SECUENCIA que el emisor del ACK espera. Este confirma la recepción de todos los bytes previos (ACK acumulativo)*
  - *El primer ACK enviado por las partes confirma el número de secuencia inicial del otro, pero no datos*

# ENCABEZADO TCP

- HEADER LENGHT: Tamaño del encabezado TCP expresado en palabras de 32 bits
  - *El tamaño mínimo es 5 y el máximo 15, permitiéndose hasta 40 bytes para el campo de OPCIONES*
- RESERVADO: Reservado para uso futuro
- BANDERAS
  - *NS (Nonce Sum): Protección ante ocultamiento de paquetes*
  - *CWR (Congestion Window Reduce): Notificación de recepción de segmento con bandera ECE*
  - *ECE (ECN-Echo): Notifica si el par TCP soporta ECN (Explicit Congestion Notification)*
  - *URG: Valida el campo URGENT POINTER*

# ENCABEZADO TCP

- … BANDERAS
  - *ACK: Valida el campo NUMERO DE CONFIRMACION*
  - *PSH: Indica urgencia de procesar el segmento, sin ponerlo en buffer*
  - *RST (Reset): Reiniciar la conexión*
  - *SYN (Synchronize): Se usa solamente al enviar el primer segmento entre pares TCP durante el 3-way-handshake*
  - *FIN (Finished): Se usa en el ultimo segmento del emisor*
- TAMAÑO DE VENTANA: Tamaño de la ventana de recepción. Especifica el número de unidades de tamaño de ventana que el emisor del segmento desea recibir actualmente
- CHECKSUM: Suma de control similar al de UDP
- URGENT POINTER: Apuntador a la posición donde terminan los datos urgentes.

# ENCABEZADO TCP

- OPCIONES
  - *MSS (Maximum Segment Size): usado durante la fase SYN y SYN-ACK del 3-way-handshake para definir el tamaño máximo de segmento que se usará durante una conexión (ocupa 4 bytes)*
  - *Windows Scaling:*
  - *SACK (Selective Acknowledgements)*
  - *Timestamps*
  - *Nop*

# Número de secuencia (#sec) y ACKs TCP

#Sec:

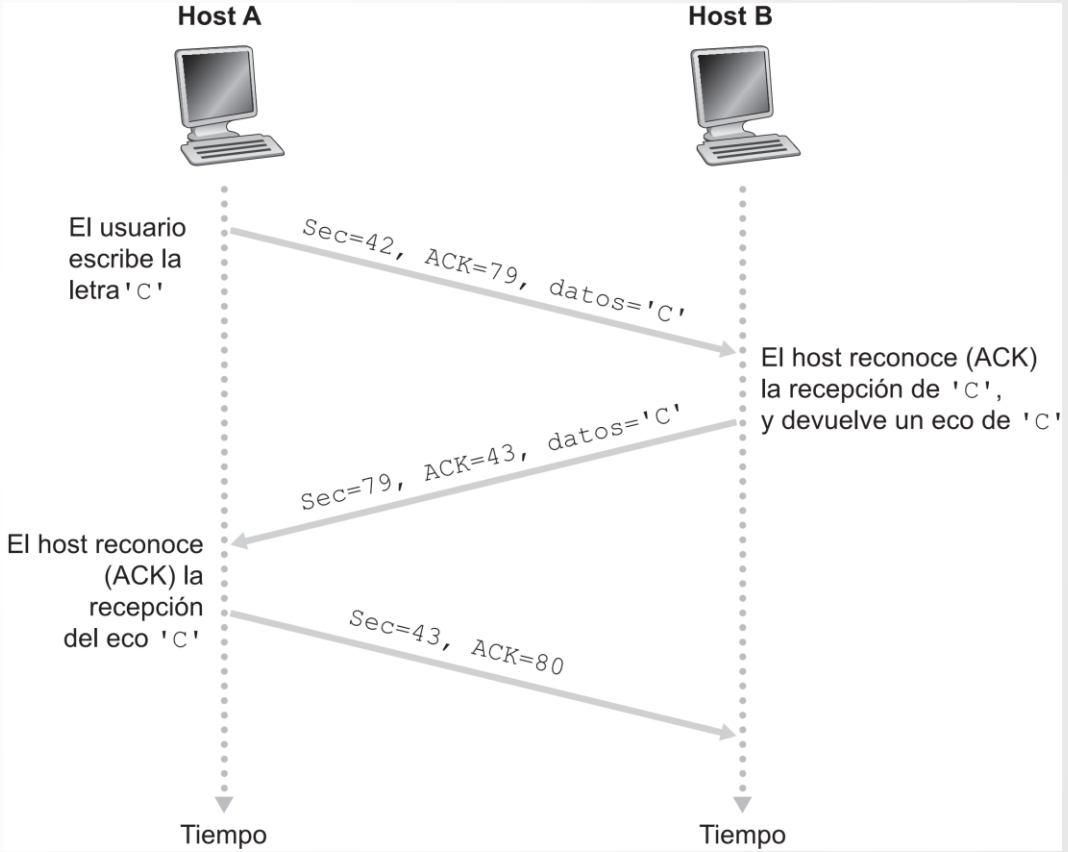
- “número” de byte en el flujo del primer byte en el segmento de datos

ACKs:

- Número de secuencia del siguiente byte esperado del otro lado
- ACK acumulativos

P: ¿Cómo maneja el receptor los segmentos fuera de orden?

R: TCP deja esto al implementador




ESCENARIO SIMPLE DE TELNET

# TCP: Round Trip Time  (RTT) y Retransmission Timeout (RTO)

¿Como establecer el valor del RTO de TCP?

- Debe ser mayor que RTT
  - Pero RTT varía
- Si muy corto: Timeout prematuro
  - Genera retransmisiones innecesarias
- Muy largo:
  - Reacción lenta ante la pérdida de segmentos
- RTT se debe estimar

# TCP: Round Trip Time  (RTT) y Retransmission Timeout (RTO)

¿Como estimar RTT?

- ***SampleRTT***: Tiempo medido desde la transmisión del segmento hasta la recepción del ACK
  - Ignorar retransmisiones

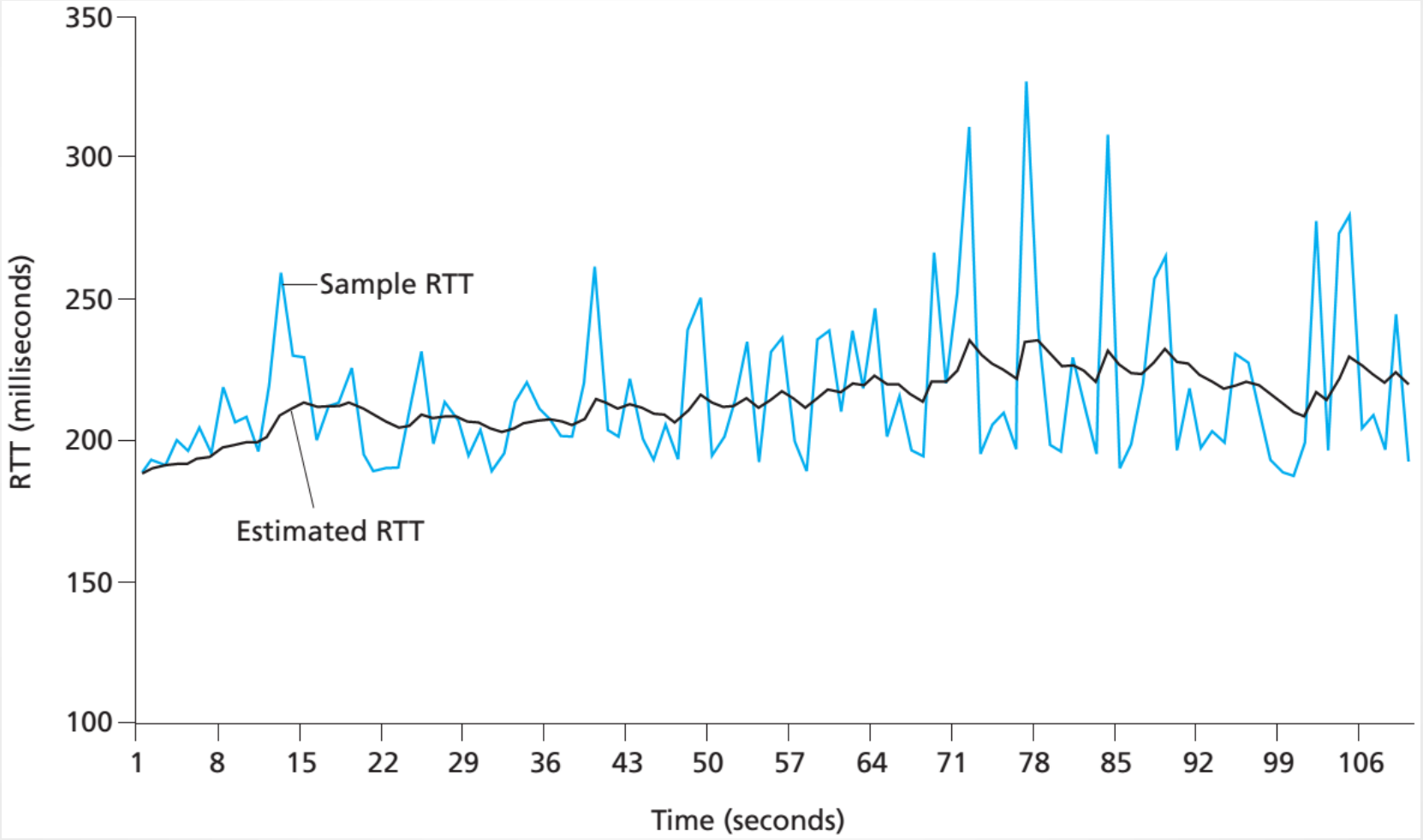
- ***SampleRTT*** Variará , se requiere un RTT estimado “suave”
  - Promediar varias mediciones recientes, no solo el ***SampleRTT*** actual

# TCP: Round Trip Time (RTT) y Retransmission Timeout (RTO)

$$EstimatedRTT = (1 - \alpha) \times EstimatedRTT + \alpha \times SampleRTT$$

- Media móvil ponderada exponencial (Exponential Weighted Moving Average – EWMA)
- La influencia de muestras anteriores decrece exponencialmente rápido
- Valor típico de $\alpha = 0.125$

# Ejemplo de estimación RTT:



# TCP: Round Trip Time (RTT) y Retransmission Timeout (RTO)

Establecimiento del RTO

- $EstimatedRTT$ más un "margen de seguridad"
  - Mayor variación en $EstimatedRTT$ → Margen de seguridad mayor
- Primero estimar en cuánto $SampleRTT$ se desvía del $EstimatedRTT$:

$$DevRTT = (1 - \beta) \times DevRTT + \beta \times |SampleRTT - EstimatedRTT|$$

(valor típico de $\beta = 0.25$)

Luego establecer el intervalo del timeout :

$$RetransmissionTimeout = EstimatedRTT + 4 \times DevRTT$$

# TCP: Round Trip Time (RTT) y Retransmission Timeout (RTO)

Valores iniciales

- $RetransmissionTimeout$ (RTO) se inicia en 1 segundo
- Cuando llega el primer ACK, su valor $RTT$ se almacena en $EstimatedRTT$
- $DevRTT$ se establece en $RTT/2$
- Luego $RetransmissionTimeout$ se calcula con la fórmula antes indicada

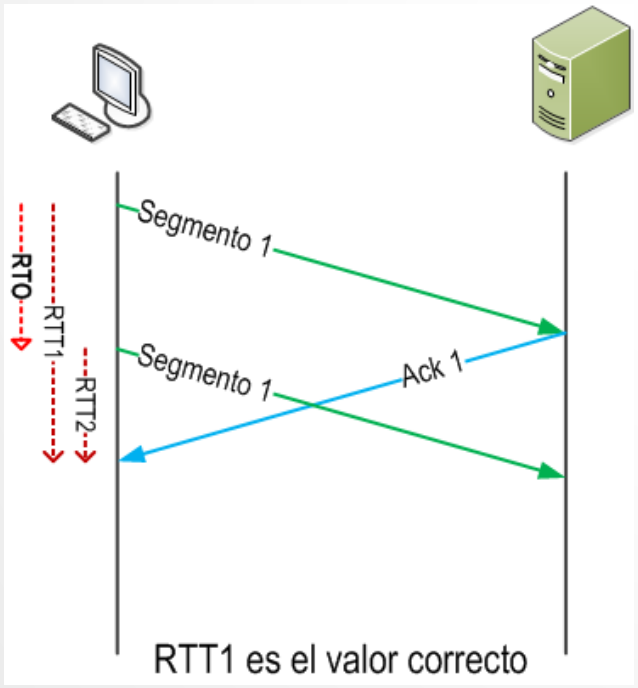
$$RetransmissionTimeout = EstimatedRTT + 4 \times DevRTT$$

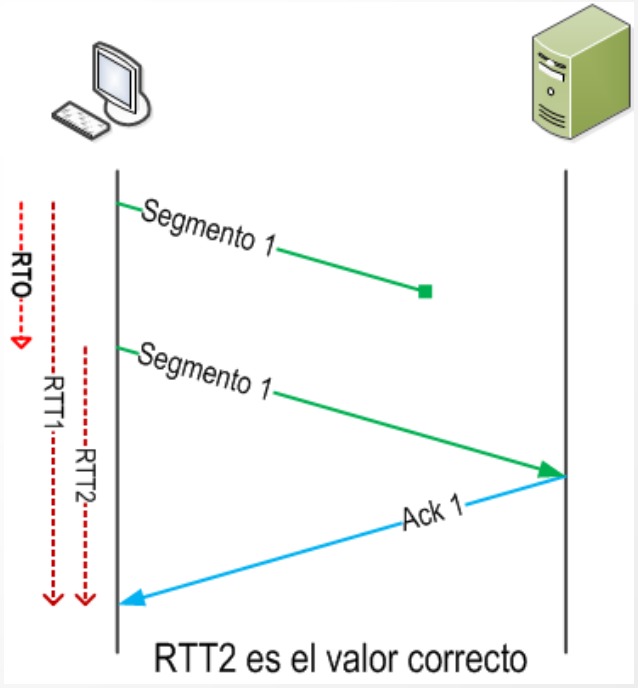
- Si el $RetransmissionTimeout$ fuese menor a 1, este debe redondearse a 1
- El ACK de un segmento retransmitido no se toma en consideración para el cálculo del RTO (Algoritmo de Khan)

# TCP: Round Trip Time (RTT) y Retransmission Timeout (RTO)

Algoritmo de Khan

- El ACK de un segmento retransmitido no se toma en consideración para el cálculo del $TimeoutInterval$ pues puede conducir a resultados erróneos



# Transferencia de datos fiable TCP

- TCP crea un servicio TDF sobre el servicio no fiable de IP
- Segmentos encauzados
- ACKs acumulativos
- TCP utiliza un temporizador de retransmisión único
- Las retransmisiones ocurren cuando:
  - Ocurre un timeout
  - Se recibe ACKs duplicados
- Considere inicialmente un emisor TCP simplificado:
  - Ignora ACKs duplicados
  - Ignora control de flujo y control de congestionamiento

# Eventos del emisor TCP:

Datos recibidos de la aplicación:

- Crear segmento con *seq#*
- *seq#* es el número en el flujo de bytes del primer byte de datos del segmento
- Iniciar timer si éste aún no esta activado (el timer se asocia al segmento no confirmado más antiguo)
- Intervalo de expiración: `TimeOutInterval`

Timeout:

- Retransmitir el segmento que ocasionó el timeout
- Reiniciar timer

ACK recibido:

- Si confirma segmentos previamente no confirmados
  - Actualizar aquello que debía ser confirmado
  - Iniciar timer si existen segmentos pendientes

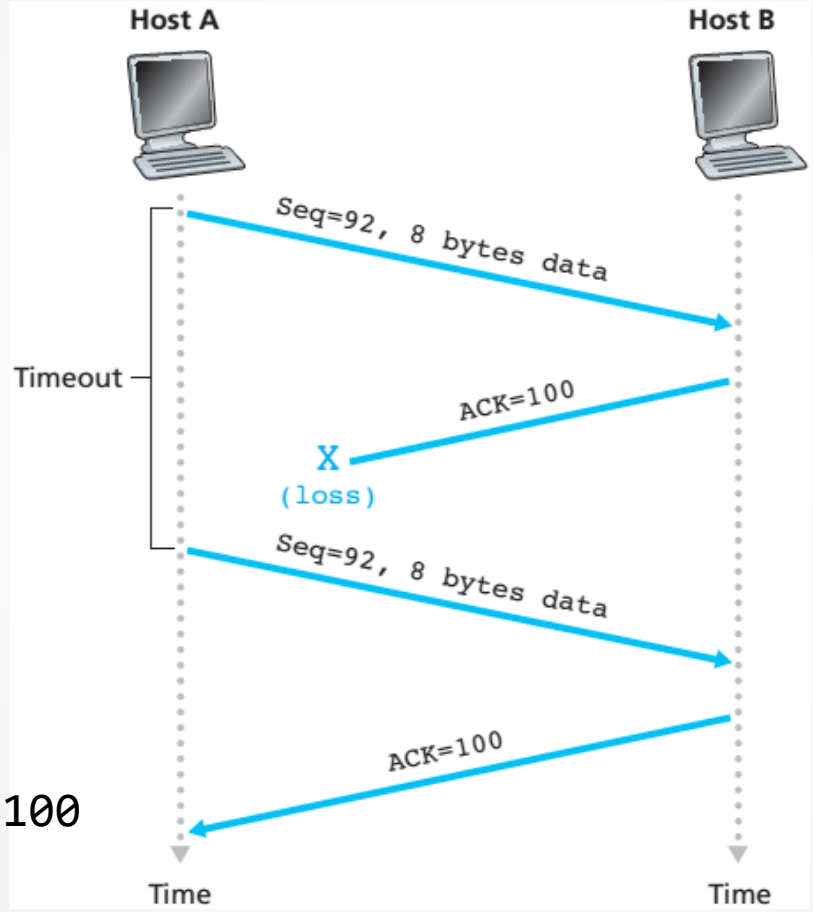
# Emisor TCP (simplificado)

```
NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber
loop (eternamente) {
    switch(event)
        event: Datos recibidos desde desde aplicacion
                Crear segmento TCP con numero de secuencia NextSeqNum
                if    (timer no esta corriendo)
                   Iniciar timer
                pasar segmento a IP
                NextSeqNum = NextSeqNum + length(datos)
                break;
        event: Timeout de timer
                Retransmitir segmento aun-no-confirmado con el menor numero de secuencia
                Iniciar timer
                break;
        event: ACK recibido, con campo ACK igual a y
                if    (y > SendBase) {
                   SendBase=y
                   if (existen segmentos aun-no-reconocidos)
                      Iniciar timer
                }
                break;
}
```

`SendBase-1`: último byte confirmado acumulativamente

Ejemplo:
Si `SendBase-1 = 71; y = 73`, entonces el receptor espera 73+ ;
Si `y > SendBase`, entonces se confirma segmentos previamente no confirmados

# TCP: Escenarios de retransmisión

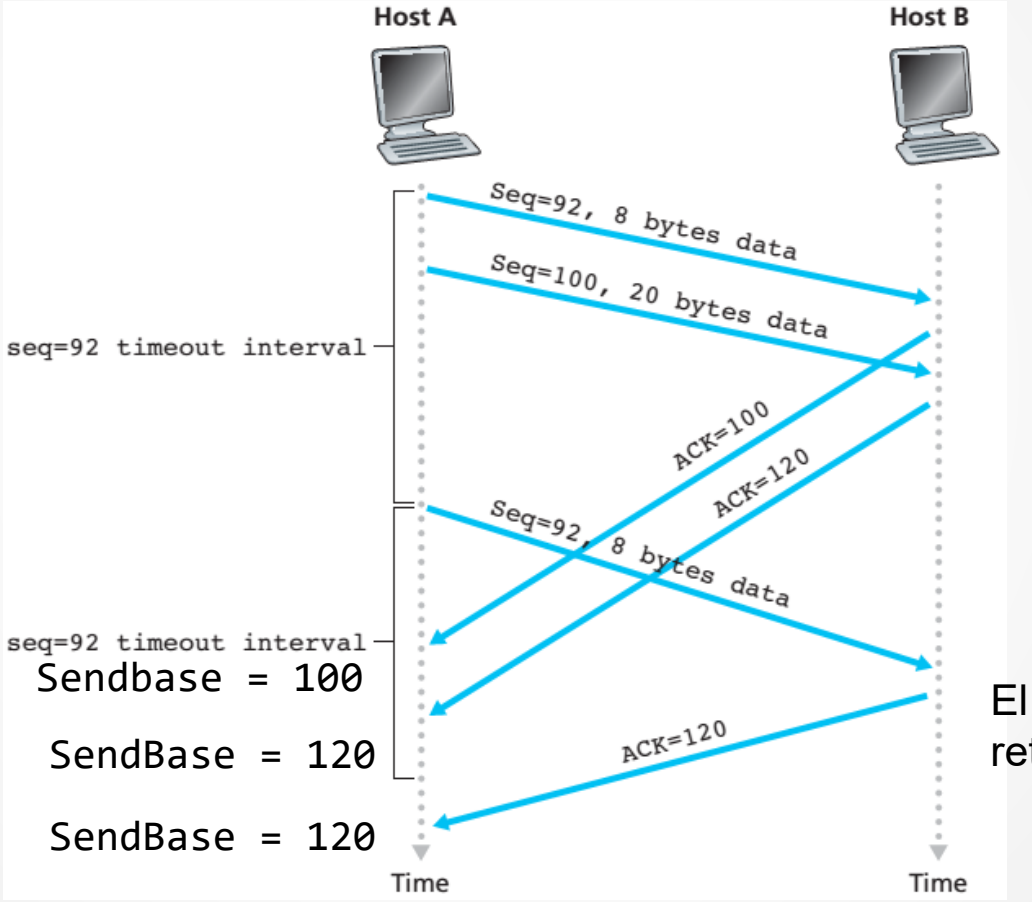



SendBase = 100

**ACK PERDIDO**

La pérdida de un ACK obliga una retransmisión

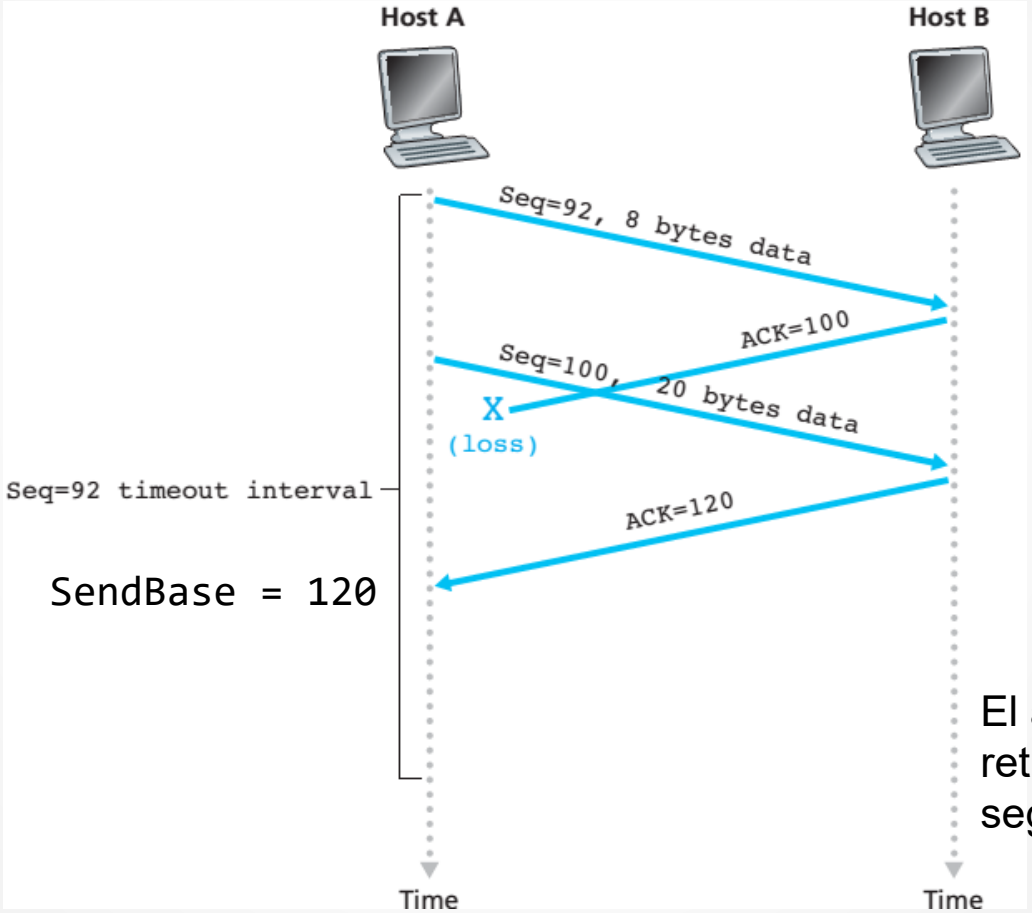
# TCP: Escenarios de retransmisión




El segmento 100 no se retransmite

**TIMEOUT PREMATURO**

# TCP – Escenarios de retransmisión




El ack acumulativo evita la retransmisión del primer segmento

**ACK ACUMULATIVO**

# Recomendaciones para la generación de ACK TCP [RFC 5681]

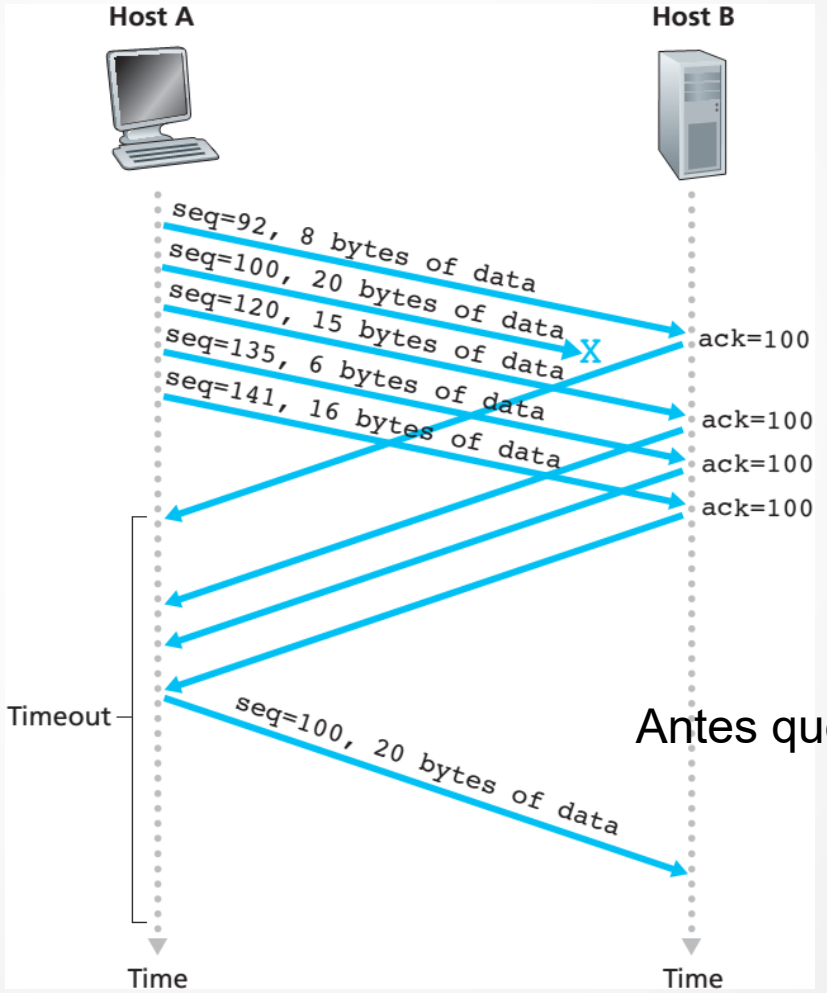
| Evento en el Receptor | Acción en el Receptor TCP |
|---|---|
| Llegada de segmento con seq# esperado. Todos los datos hasta el seq# esperado ya confirmados | ACK retrasado. Esperar hasta 500 ms por un siguiente segmento-en-orden. Si no hay, enviar ACK |
| Llegada de segmento con seq# esperado. Otro segmento tiene un ACK pendiente | Enviar inmediatamente un único ACK Acumulativo, confirmando los dos segmentos llegados en orden |
| Llegada de un segmento fuera de orden con seq# mayor al esperado. Se detecta un hueco | Enviar inmediatamente un *ACK duplicado*, Indicando el seq# del sgte byte esperado (que es el limite inferior del hueco) |
| Llegada de un segmento que llena parcial o completamente un hueco en datos recibidos | Enviar inmediatamente un ACK, siempre que el segmento comience en el limite inferior del hueco |

# Retransmisión rápida

- Periodo de Timeout suele ser relativamente largo:
  - Retardo largo antes de reenviar un paquete perdido
- Detección de segmentos perdidos mediante ACKs duplicados.
  - El emisor suele enviar muchos segmentos seguidos (back-to-back)
  - Si se pierde un segmento, puede haber muchos ACKs duplicados.
- Si el emisor recibe 3 ACKs para el mismo dato, este supone que el segmento después de los datos confirmados se perdió:
- Retransmisión rápida: reenviar segmento antes que expire el timer

# Retransmisión rápida

Reenvio de un segmento después de un ACK duplicado **TRES** veces




Antes que venza el temporizador

# Algoritmo de Retransmisión Rápida:

```
event:    ACK recibido, con campo ACK igual a y
        if (y > SendBase) {
            SendBase = y
            if    (existen segmentos aun-no-confirmados)
                Iniciar timer
        }
        else { /* un ACK duplicado para un segmento ya confirmado */
            Incrementar contador de ACKs duplicados recibidos para y
            if (numero de ACKs duplicados para y == 3)
                /* Retransmisión rápida TCP */
                reenviar segmento con numero de secuencia y
        }
        break;
```

Un ACK duplicado para un segmento ya confirmado

Retransmisión rápida

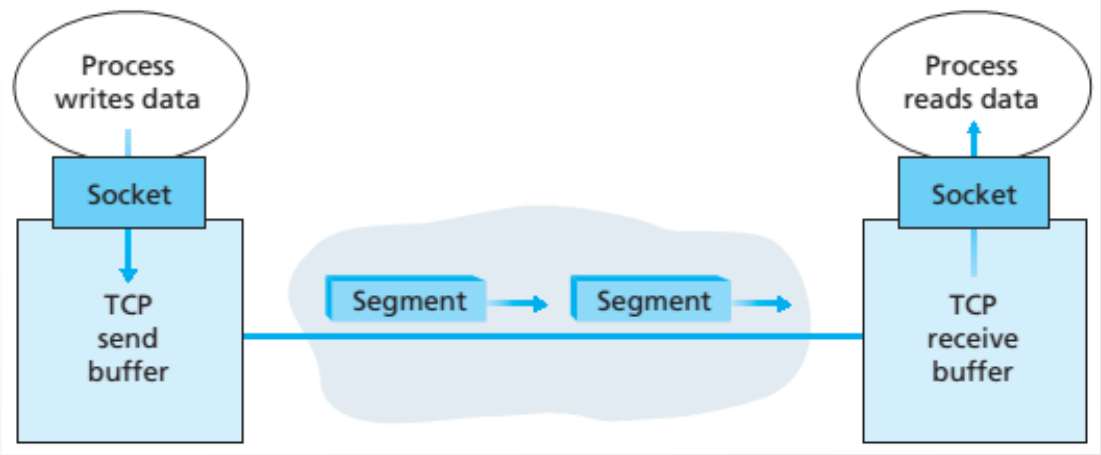
# Control de flujo TCP

# Control de flujo TCP

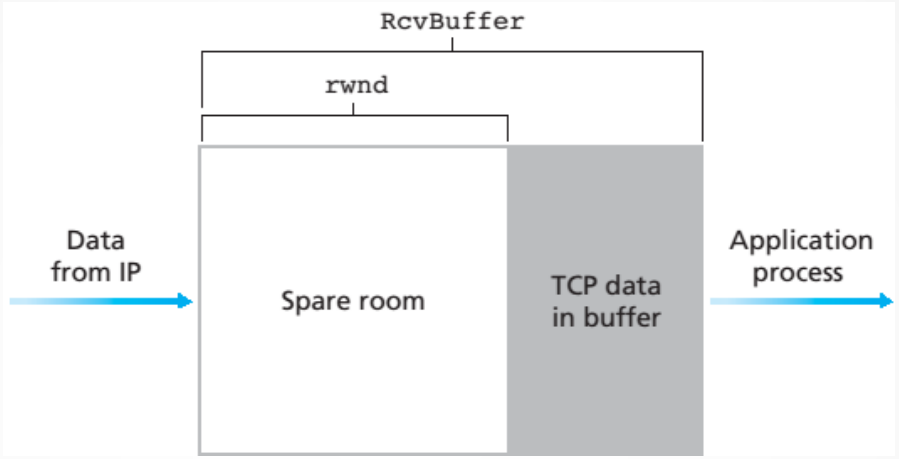
Propósito: El emisor no desbordará el buffer del receptor transmitiendo demasiado a excesiva velocidad

- El control de flujo es un servicio de conciliación de velocidad: Conciliar la tasa de transmisión con la tasa de recuperación de la aplicación receptora

- El receptor TCP tiene un buffer de recepción
- El proceso de la aplicación puede ser lento al leer del buffer



# Control de flujo TCP



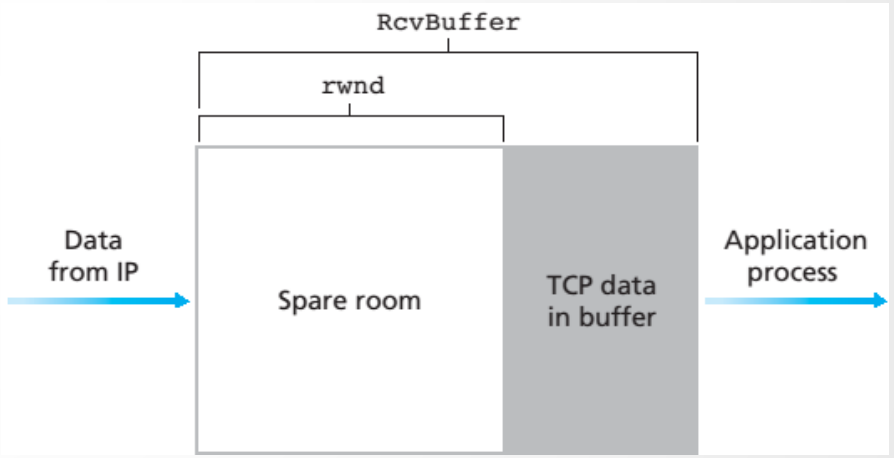

- Para que TCP no rebase el buffer asignado se debe cumplir:

$$LastByteRcvd - LastByteRead \leq RcvBuffer$$

$$Espacio\ Libre\ en\ buffer = rwnd$$

# Control de flujo TCP

- El receptor notifica espacio libre incluyendo el valor de ***rwnd*** en los segmentos enviados al emisor
- El emisor limita datos no confirmados a ***rwnd***
  - Garantiza que el buffer de recepción no se desbordará




$$rwnd = RcvBuffer - [LastByteRcvd - LastByteRead]$$

$$LastByteSent - LastByteAcked \leq rwnd$$

# Gestión de conexión TCP

# Gestión de conexión TCP

- El emisor y el receptor TCP establecen una conexión antes de intercambiar segmentos de datos
- Inicializar variables TCP :
  - seq. #s
  - buffers, información de control de flujo (e.g. **rwnd**)

- *Cliente: iniciador de la conexión*

```
Socket clientSocket = new Socket("hostname","port number");
```

- *Servidor: contactado por cliente*

```
Socket connectionSocket = welcomeSocket.accept();
```

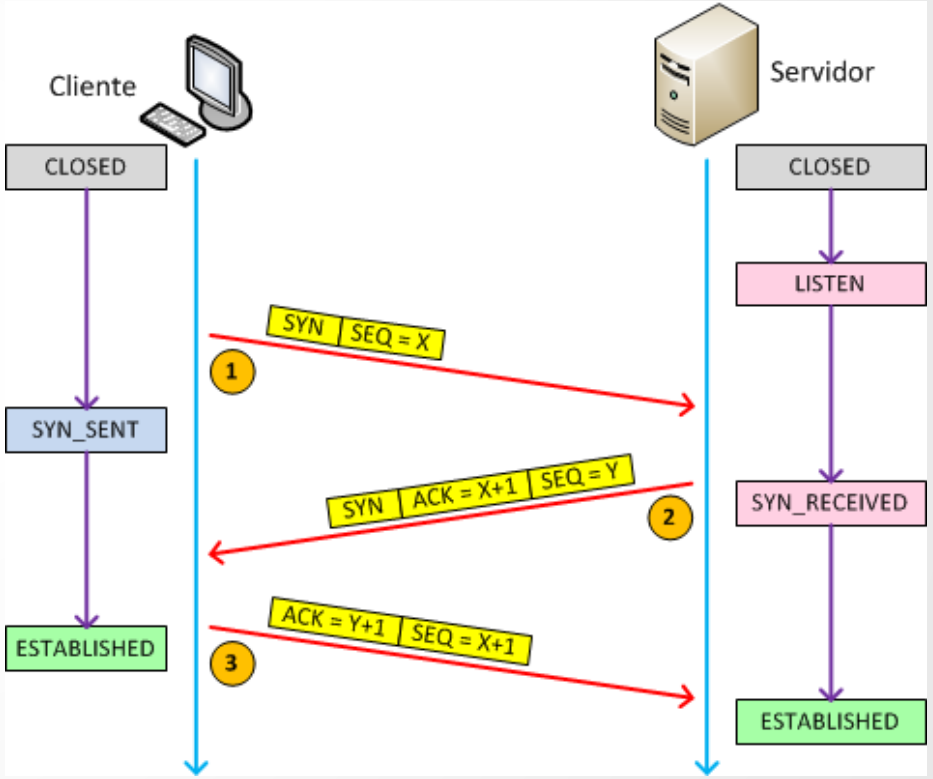
# Gestión de conexión TCP

Three way handshake:

1: El cliente envía un segmento TCP SYN al servidor
- Especifica su seq# inicial
- No envía datos

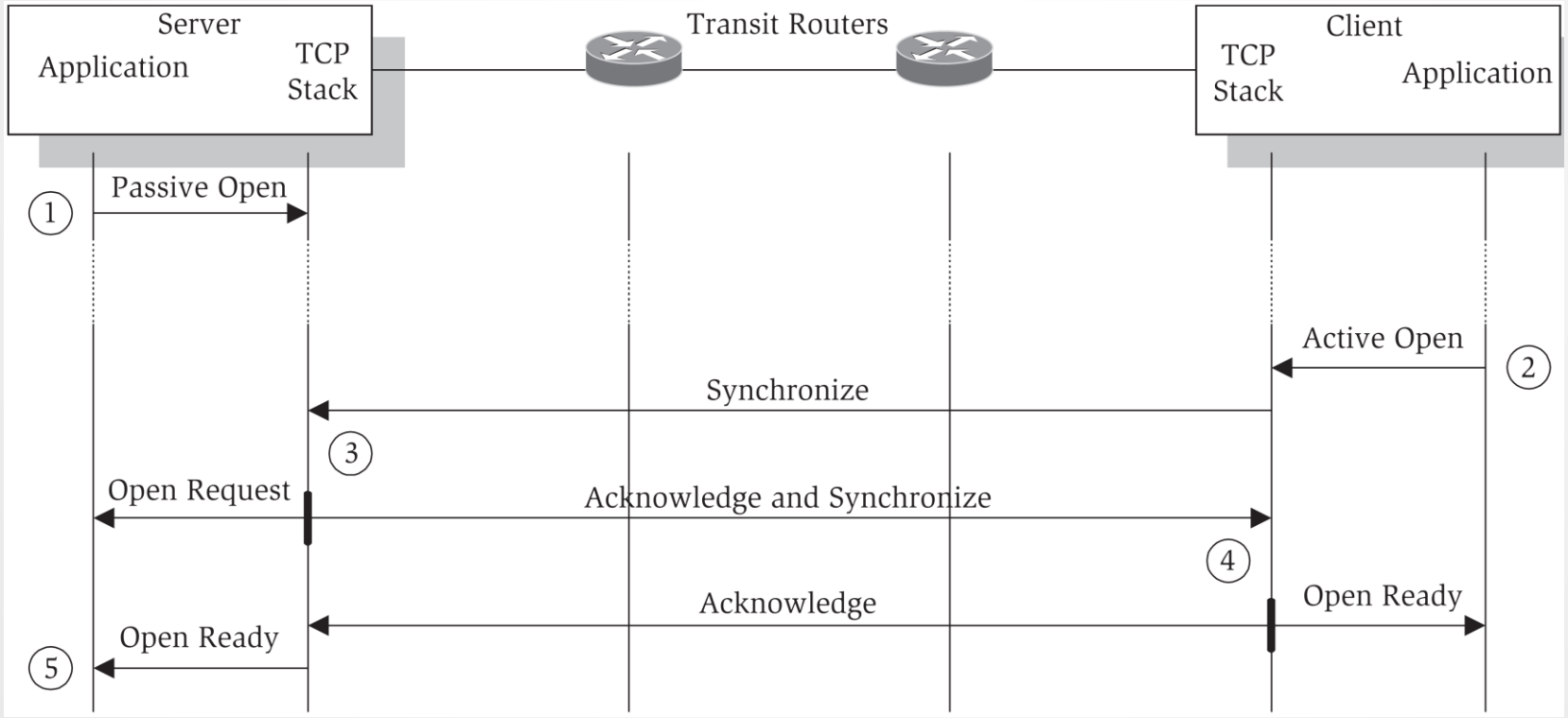
2: El servidor recibe SYN, responde con un segmento SYN-ACK
- Servidor asigna buffers
- Especifica el seq# inicial del servidor

3: El cliente recibe SYN-ACK; responde con un segmento ACK, que puede contener datos



# Gestión de conexión TCP

- Conexión TCP desde la perspectiva de las aplicaciones



# Gestión de conexión TCP

Cierre de conexión:

Cliente cierra socket:

```
clientSocket.close();
```

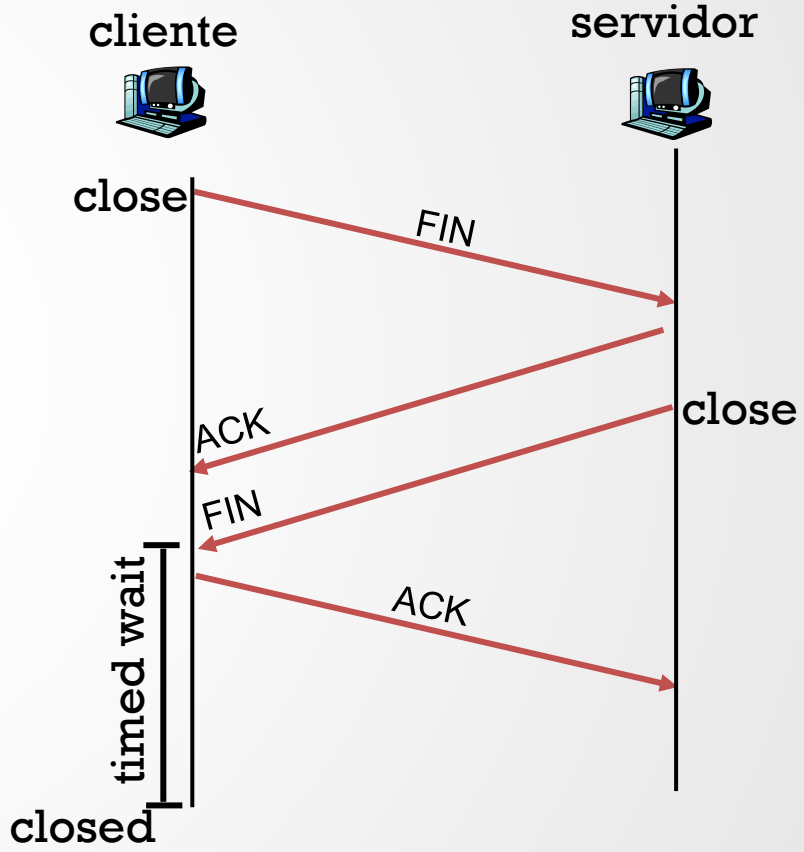
Paso 1: *cliente envía un segmento de control TCP FIN al servidor*

Paso 2: *servidor recibe FIN, responde con ACK. Cierra conexión, envía FIN.*

Paso 3: *cliente recibe FIN, responde con ACK.*

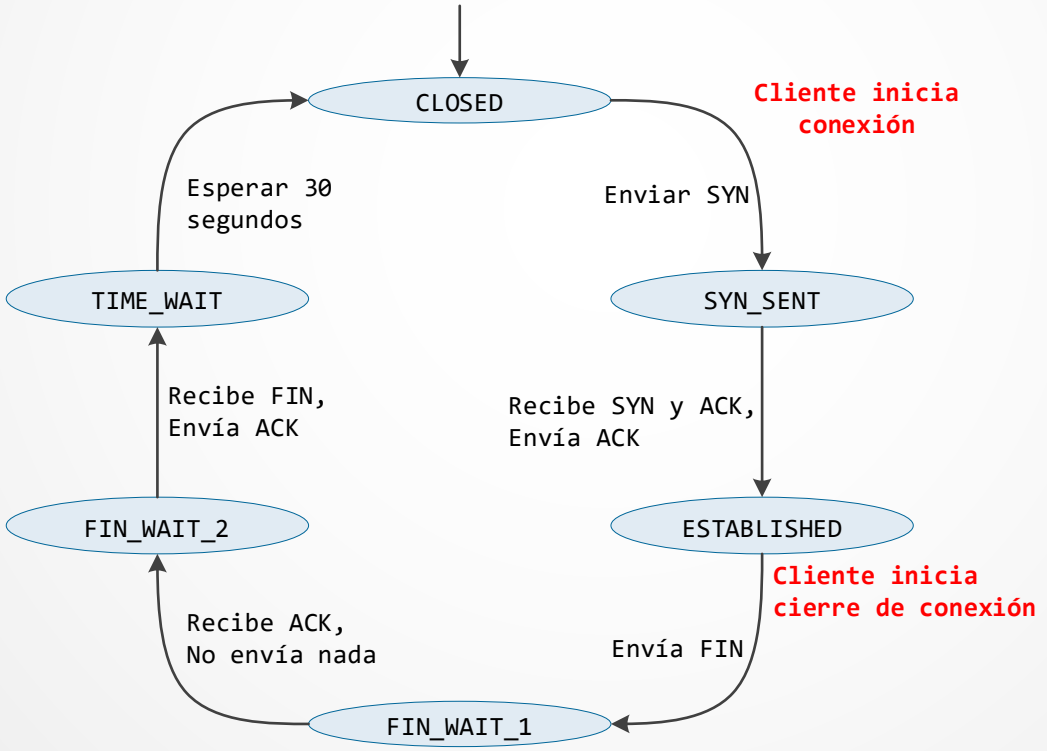
- Ingresa en “timed wait” – responderá con ACK a los FINs recibidos

Paso 4: *servidor, recibe ACK. Conexión cerrada.*



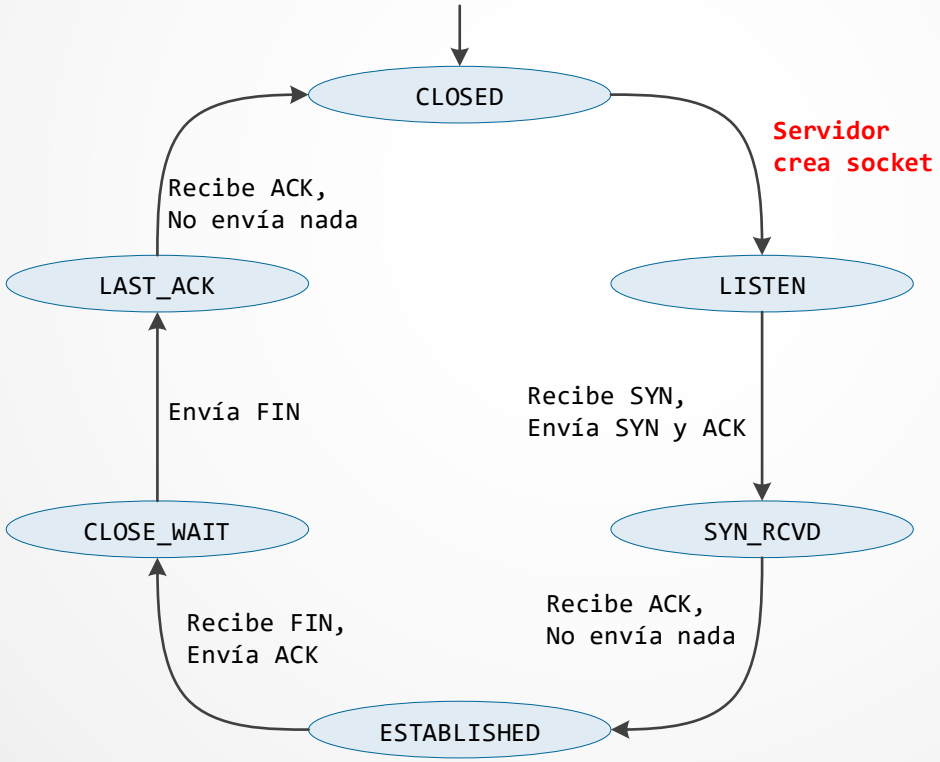
# Gestión de conexión TCP

Ciclo de vida de cliente TCP



# Gestión de conexión TCP

Ciclo de vida de servidor TCP



# PRINCIPIOS DE CONTROL DE CONGESTIONAMIENTO

# PRINCIPIOS DE CONTROL DE CONGESTIONAMIENTO

Congestión:

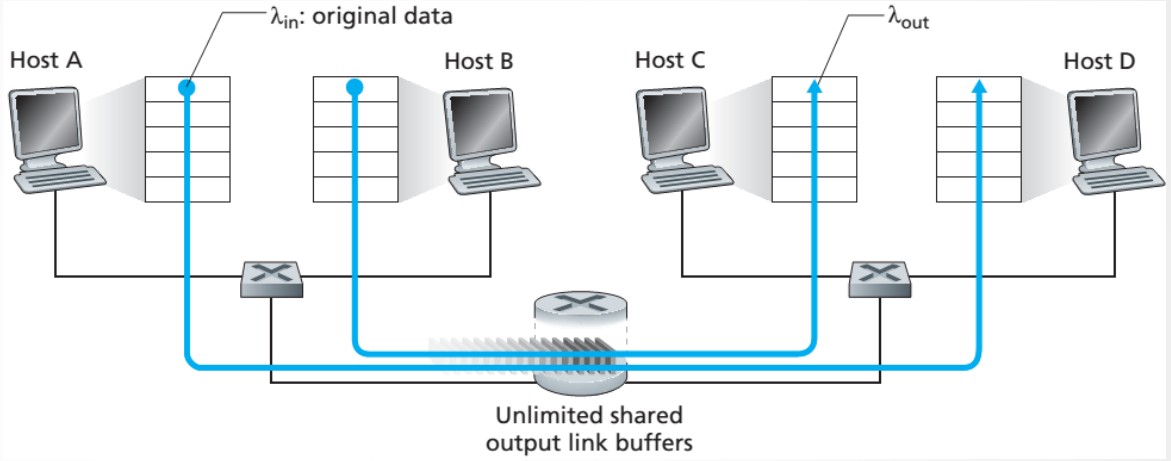
- Se presenta cuando múltiples fuentes transmiten datos mas rápido de lo que la red puede manejar

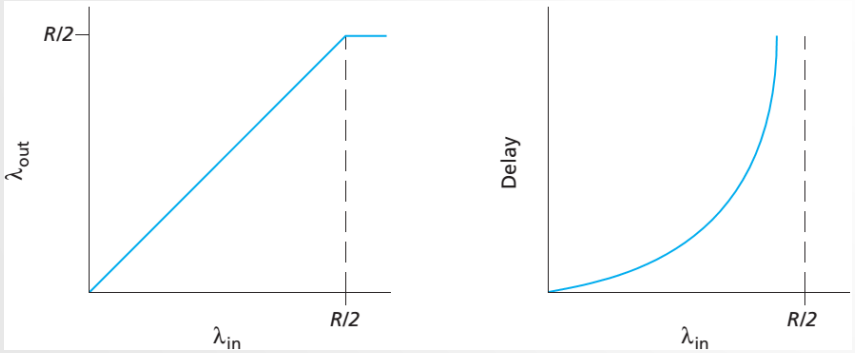
- Manifestación de la congestión:
  - Paquetes perdidos (desbordamiento de buffers en los enrutadores)
  - Retardos largos (encolamiento en buffer de enrutadores)

- Diferente del control de flujo!

# CAUSAS/COSTOS DE CONGESTIÓN: ESCENARIO 1

- Dos emisores, dos receptores
- Un enrutador con buffer infinito
- Sin retransmisión
- *R* – Capacidad del enlace

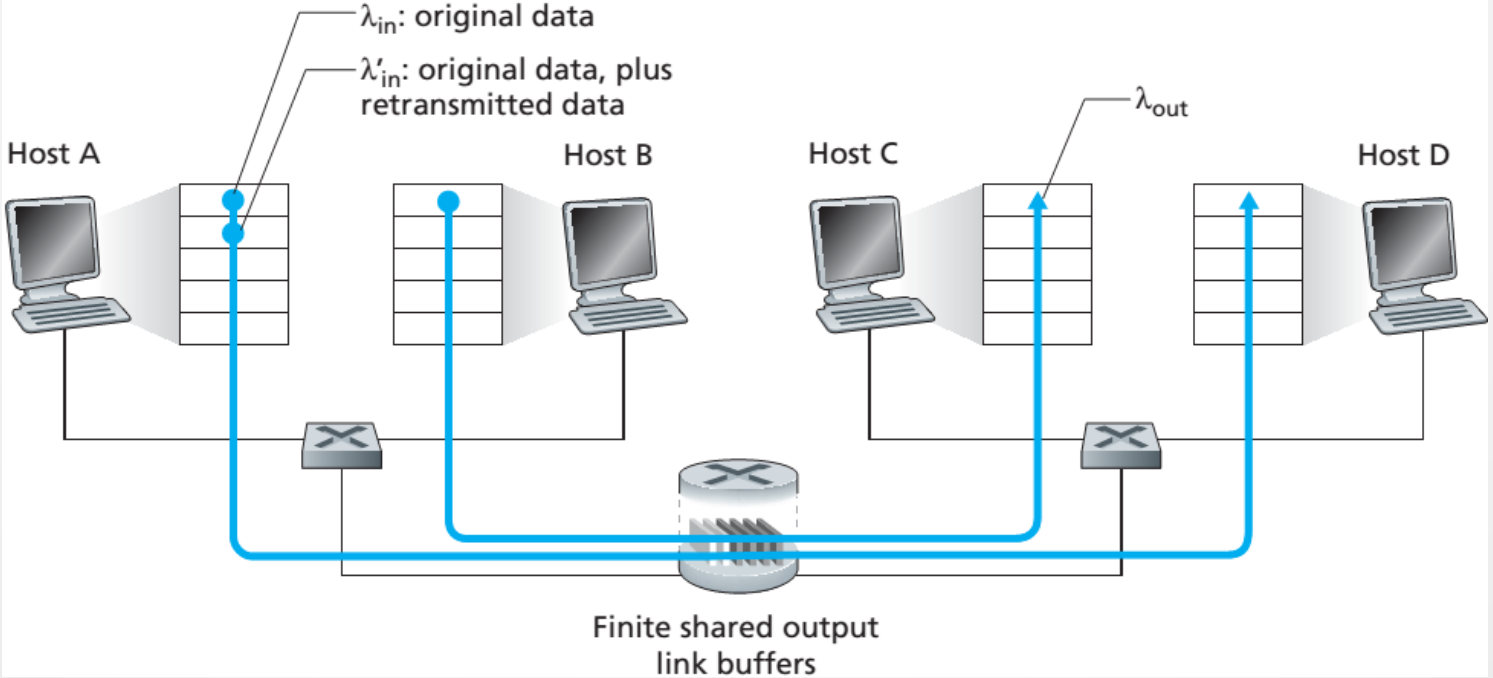



- Retardos largos cuando la red se congestiona
- Rendimiento máximo alcanzable = R/2
- Surgen retardos de encolamiento largos a medida que la tasa de llegada se aproxima a la capacidad del enlace

# CAUSAS/COSTOS DE CONGESTION: ESCENARIO 2

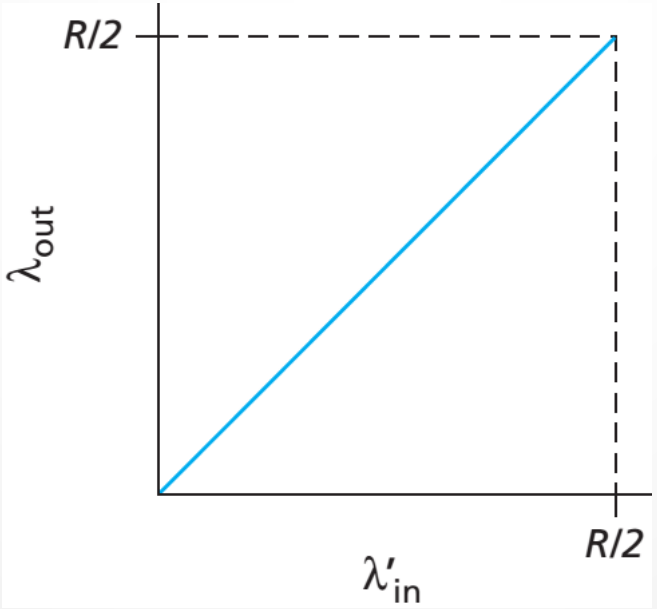
- Un enrutador con buffer *finito*, los paquetes pueden descartarse
- Conexiones fiables (pueden retransmitirse datos):



# CAUSAS/COSTOS DE CONGESTIÓN: ESCENARIO 2

Si asumimos que no ocurren pérdidas,

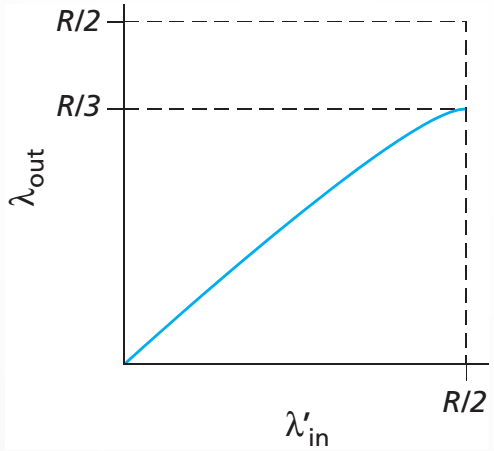
- Entonces, : $\lambda'_{in} = \lambda_{out}$ (goodput)
- Tasa de transmisión promedio no puede exceder de R/2



# CAUSAS/COSTOS DE CONGESTIÓN: ESCENARIO 2

El emisor retransmite paquetes que sabe con certeza que se han perdido (timeout grande)

- La carga ofertada (la tasa de transmisión original mas retransmisiones) = R/2:
  - $\lambda_{out} = R/3$ , entonces, de 0.5R unidades de dato transmitidos, 0.333R bytes/s son datos originales y 0.166R bytes/s son datos retransmitidos

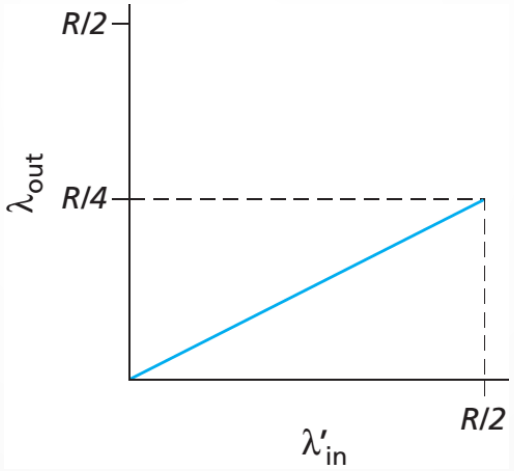



"Costo" de congestión:

- El emisor debe retransmitir para compensar paquetes perdidos por desbordamiento de buffer

# CAUSAS/COSTOS DE CONGESTIÓN: ESCENARIO 2

- El emisor genera timeout prematuros y podría retransmitir paquetes aun no perdidos (en cola de espera)
- Si cada paquete se retransmite 2 veces (en promedio), entonces:
  - Cuando $\lambda'_{in} = R/2, \lambda_{out} = R/4$

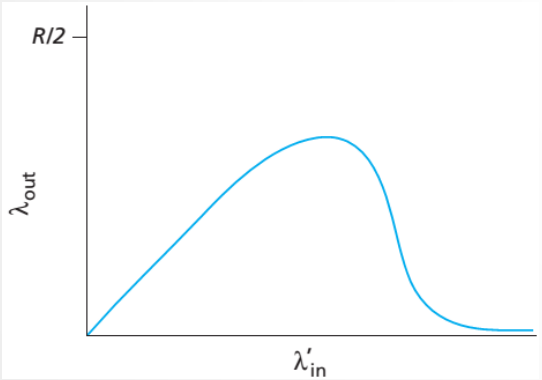


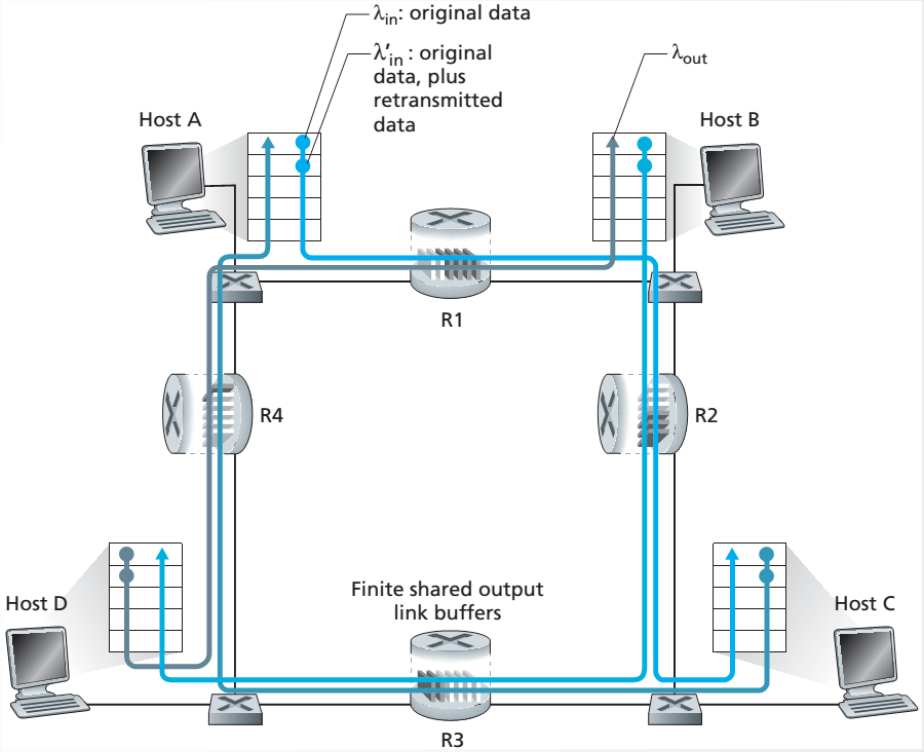

"Costos" de congestión:

- Retransmisiones innecesarias: el enlace transporta múltiples copias de un paquete

# CAUSAS/COSTOS DE CONGESTIÓN: ESCENARIO 3

- Cuatro emisores
- Rutas multisalto
- Timeout / retransmisión
- Enlaces con capacidad R bytes/s




Otro “costo” de la congestión:

- Cuando un paquete es descartado, toda capacidad de transmisión usada para dicho paquete fue desperdiciada

# ENFOQUES PARA EL CONTROL DE CONGESTIONAMIENTO

## Dos enfoques para el control de congestionamiento:

### Control de congestionamiento extremo-extremo:

- No hay retroalimentación explicita de la red
- El congestionamiento se infiere de las pérdidas y retardos observados por el sistema final
- Es el enfoque tomado por TCP

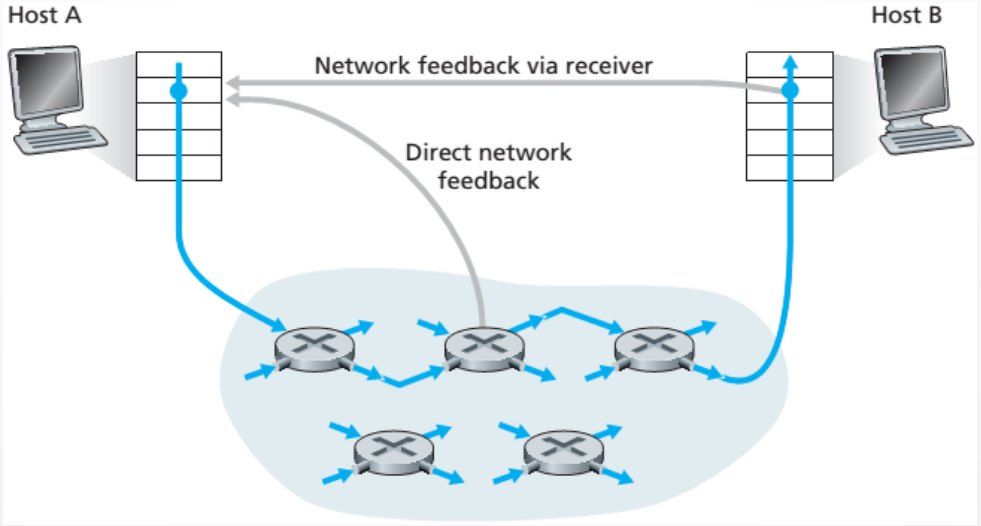
### Control de congestionamiento asistido por la red:

- Los enrutadores proveen retroalimentación  a los sistemas finales
  - Bit indicador de congestión (SNA, DECbit, TCP/IP ECN, ATM ABR)
  - Tasa de transmisión explicita a la cual debe transmitir el emisor

# ENFOQUES PARA EL CONTROL DE CONGESTIONAMIENTO

## Control de congestionamiento asistido por la red :

- La información de congestión se retroalimenta desde la red al emisor en dos formas:
  - Retroalimentación directa de un enrutador al emisor (paquete de estrangulamiento)
  - El enrutador marca un campo en un paquete que va del emisor al receptor para indicar la congestión. Al recibir el paquete marcado, el receptor notifica al emisor de la indicación de congestión



# CONTROL DE CONGESTIONAMIENTO TCP

# CONTROL DE CONGESTIONAMIENTO TCP

- TCP utiliza control de congestionamiento extremo a extremo
- Cada emisor limita la tasa de transmisión en función del congestionamiento de red percibido
  - ¿Cómo limita el emisor la tasa a la que envía datos a su conexión?
  - ¿Cómo percibe el emisor TCP que hay congestión en el camino hasta el receptor?
  - ¿Qué algoritmo debe usar el emisor para cambiar su tasa de envío como función del congestionamiento extremo a extremo percibido?

# CONTROL DE CONGESTIONAMIENTO TCP

- ¿Cómo limita el emisor la tasa de envío de datos a su conexión?
  - Cada extremo de una conexión TCP comprende:
    - Buffer de recepción y envío
    - Variables:
      - LastByteRead
      - rwnd
      - etc.
  - El mecanismo de control de congestionamiento TCP en el emisor vigila una variable adicional: La ventana de congestionamiento
    - Ventana de congestión (*cwnd*): impone una restricción en la tasa a la que un emisor TCP puede enviar datos a la red
    - La cantidad de datos no confirmados en el emisor no puede exceder el mínimo de *cwnd* y *rwnd*

$$LastByteSent - LastByteAcked \leq \min\{cwnd, rwnd\}$$

# CONTROL DE CONGESTIONAMIENTO TCP

- Emisor limita la transmisión:

$$LastByteSent - LastByteAcked \leq cwnd$$

- La tasa de transmisión es aproximadamente:

$$Tasa\ de\ transmisión = \frac{cwnd}{RTT}\ bytes/s$$

- ***cwnd*** es dinámica y función del congestionamiento de red percibido y permite ajustar la tasa de transmisión

# CONTROL DE CONGESTIONAMIENTO TCP

- ¿Cómo percibe el emisor TCP que hay congestión en el camino hasta el receptor?

  - Mediante la detección de un evento de pérdida ("loss event")
  - Un "loss event" se deduce de timeout o la recepción de tres ACK duplicados
    - Timeout: Por descarte de paquete de la cola de un enrutador en el camino
    - Tres ACK duplicados: Paquetes perdidos
  - TCP utiliza los ACK como un indicador del estado de transmisión
    - Si recibe ACKs, todo esta bien e incrementa **cwnd**

# CONTROL DE CONGESTIONAMIENTO TCP

- ¿Qué algoritmo debe usar el emisor para cambiar su tasa de envío como función del congestionamiento extremo a extremo percibido?

  - Un segmento perdido implica congestión, por tanto, la tasa del emisor debe decrementarse
  - Un segmento confirmado indica que la red esta entregando los segmentos del emisor al receptor, entonces, puede incrementarse la tasa cuando llega un ACK por un segmento previamente no confirmado
  - Prueba de ancho de banda. La estrategia de TCP es incrementar la tasa mientras reciba ACKs hasta que ocurra un “loss event”, en cuyo caso reduce su tasa. El emisor entonces, incrementa su tasa para explorar la tasa a la que comienza la congestión, luego retrocede y luego comienza a probar nuevamente
  - Cada emisor TCP actúa de forma independiente de los demás

# ALGORITMO DE CONTROL DE CONGESTIONAMIENTO TCP

- Estandarizado mediante RFC 5681
- Tiene tres componentes:
  - Slow start
  - Congestion avoidance
  - Fast recovery
- Slow start y congestion avoidance son componentes obligatorios de TCP
- Fast recovery es recomendado pero no obligatorio.

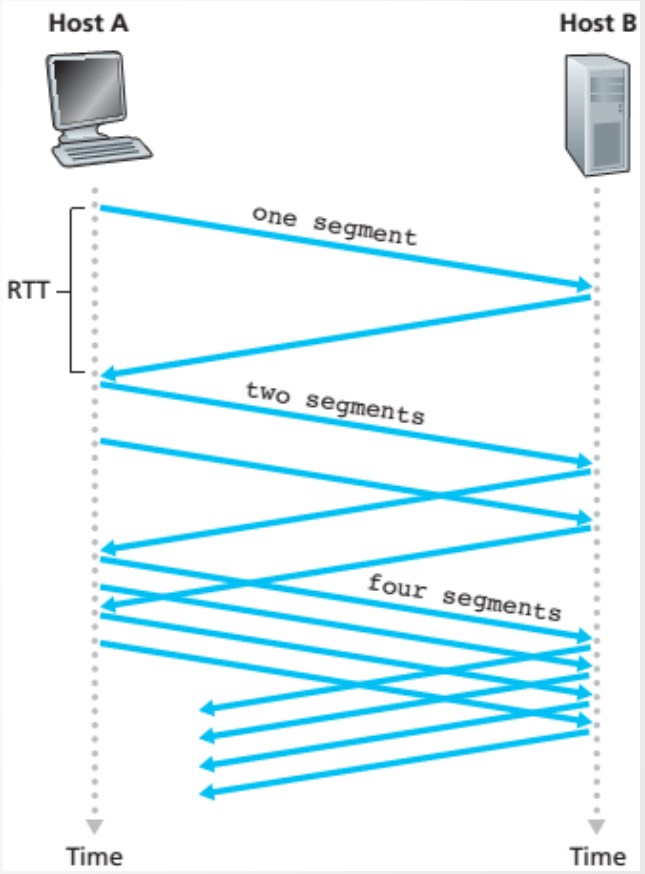
# ALGORITMO DE CONTROL DE CONGESTIONAMIENTO TCP

- Al iniciar una conexión TCP, se inicializa *cwnd* = 1 *MSS*
- Tasa inicial = 1 *MSS/ RTT*
- Ejemplo si
  - *RTT* = 200 ms y *MSS* = 500 bytes
  - Tasa inicial = 20 kbps

- Ejercicio
  - En Ethernet
    - *MSS* = 1460 bytes y *RTT* = 0.025 ms
    - Tasa inicial = ?

# ALGORITMO DE CONTROL DE CONGESTIONAMIENTO TCP

- **SLOW START**:
  - En el estado slow – start, el valor de *cwnd* comienza con 1 MSS
  - *cwnd* se incrementa en 1 *MSS* cada vez que un segmento transmitido es confirmado la primera vez.
  - Al incrementarse en 1 *MSS* por cada ACK recibido, *cwnd* crece exponencialmente
  - *cwnd* se duplica cada *RTT*
  - tasa inicial es baja pero escala exponencialmente rápido



# ALGORITMO DE CONTROL DE CONGESTIONAMIENTO TCP

- **SLOW START** - Finalización
  - Si ocurre un *timeout*, el emisor establece *cwnd* en 1 y comienza el proceso slow – start nuevamente
  - Inicia una segunda variable de estado, *ssthresh* (“slow start threshold”) en *cwnd*/2; la mitad del valor de *cwnd* cuando se detecto la congestión
  - Slow start puede terminar en función de *ssthresh*. Puesto que *ssthresh* era *cwnd*/2 la ultima vez que se detectó la congestión, sería displicente seguir duplicando *cwnd* cuando alcance o supere *ssthresh*.
  - Cuando *cwnd* == *ssthresh*, termina slow start y TCP transita al modo de evasión de congestionamiento (congestion avoidance)
  - Si se detectan 3 ACK duplicados, TCP realiza una retransmisión rápida y pasa al estado de recuperación rápida (fast recovery)

# ALGORITMO DE CONTROL DE CONGESTIONAMIENTO TCP

- **CONGESTION AVOIDANCE**
  - Al pasar al estado de evasión de congestión, *cwnd* es igual a la mitad de su valor cuando se detecto la ultima congestión
  - TCP incrementa *cwnd* en 1 *MSS* cada *RTT*
  - El emisor TCP  incrementa *cwnd* en (*MSSxMSS/cwnd) bytes* cuando llega un ACK nuevo
    - Si *MSS* = 1460 B y *cwnd* = 14600; entonces 10 segmentos se están enviando por *RTT*
    - Cada ACK que llegue incrementará *cwnd* en 1/10 *MSS*
    - El valor de *cwnd* se habrá incrementado en un *MSS* después de los ACK cuando todos los 10 segmentos hayan sido recibidos

# ALGORITMO DE CONTROL DE CONGESTIONAMIENTO TCP

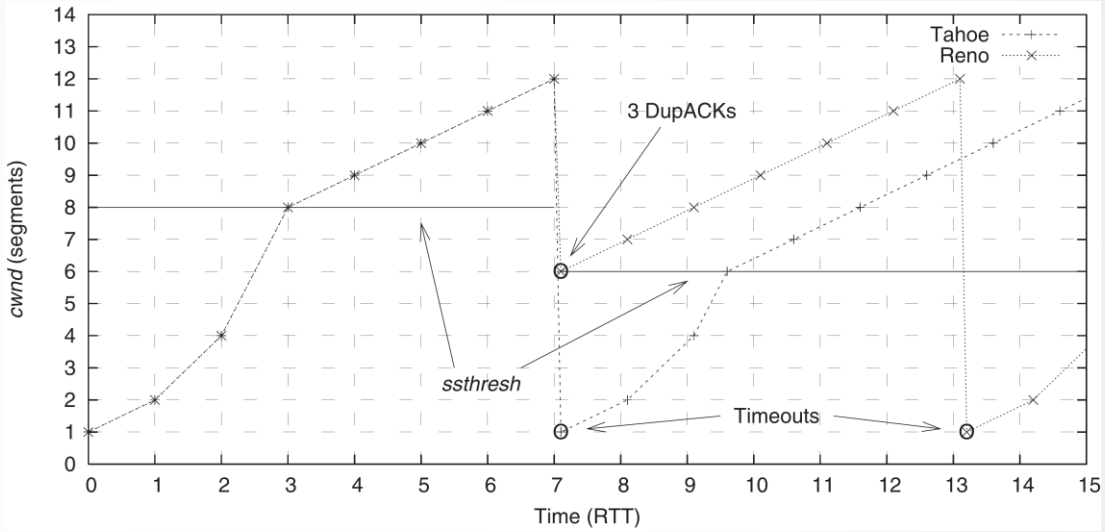
- **CONGESTION AVOIDANCE**: Finalización
  - ¿Cuándo termina el crecimiento lineal de CA?
  - Si ocurre un `timeout`, el valor de `cwnd` se fija en 1 `MSS` y `ssthresh` se actualiza a la mitad el tamaño de que tenia `cwnd` al ocurrir el "`loss event`"
  - Si se reciben 3 ACK duplicados, TCP divide `cwnd` entre 2 (agregando 3 `MSS` por los tres ACK duplicados) y establece `ssthresh` en la mitad de `cwnd` cuando se recibieron los 3 ACK duplicados
  - Luego de estos eventos, se pasa al estado de recuperación rápida (fast – recovery)

# ALGORITMO DE CONTROL DE CONGESTIONAMIENTO TCP

- **FAST RECOVERY**
  - En fast recovery, `cwnd` se incrementa en 1 *MSS* por cada ACK duplicado que se recibe por el segmento perdido que causó que TCP pase al estado fast – recovery
  - Cuando llega un ACK por el segmento perdido, TCP pasa al estado CA después de reducir `cwnd`
  - Si ocurre un timeout, fast recovery transita al estado slow – start luego de realizar las mismas acciones que en slow start y CA: el valor de `cwnd` se fija en 1 *MSS* y el valor de `ssthresh` se fija en la mitad del valor que tenia `cwnd` al momento de ocurrir “loss event“
  - TCP Tahoe, reduce `cwnd` a 1 MSS incondicionalmente y pasa a la fase SS luego de un timeout o 3 ACK duplicados
  - TCP Reno incorpora fast recovery

# ALGORITMO DE CONTROL DE CONGESTIONAMIENTO TCP

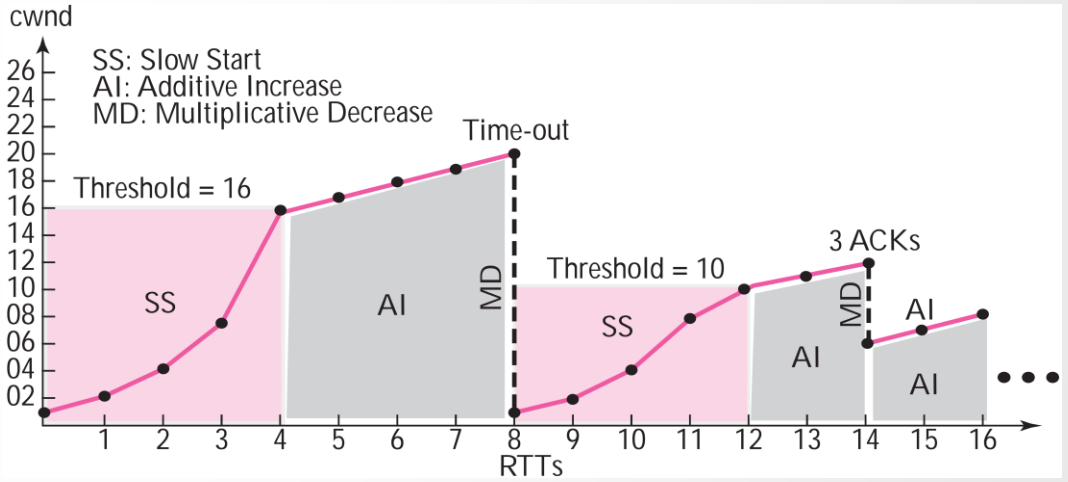
- Evolución de *cwnd* en TCP Tahoe y Reno
  - Inicialmente *ssthresh* = 8 *MSS*
  - Para las primeras 8 rondas, Tahoe y Reno toman acciones idénticas.




- La ventana de congestión escala exponencialmente durante SS y alcanza *ssthresh* en la cuarta ronda de transmisión. Entonces *cwnd* crece linealmente hasta que ocurre un evento de 3 ACK duplicados justo después de la ronda de transmisión 8, cuando *cwnd* = 12 *MSS*. Entonces, *ssthresh* = 0.5 *cwnd* = 6 *MSS*
- En TCP Reno, *cwnd* = 6 *MSS* y luego crece linealmente
- En TCP Tahoe, *cwnd* = 1 *MSS* y crece exponencialmente hasta alcanzar *ssthresh* desde donde crece linealmente

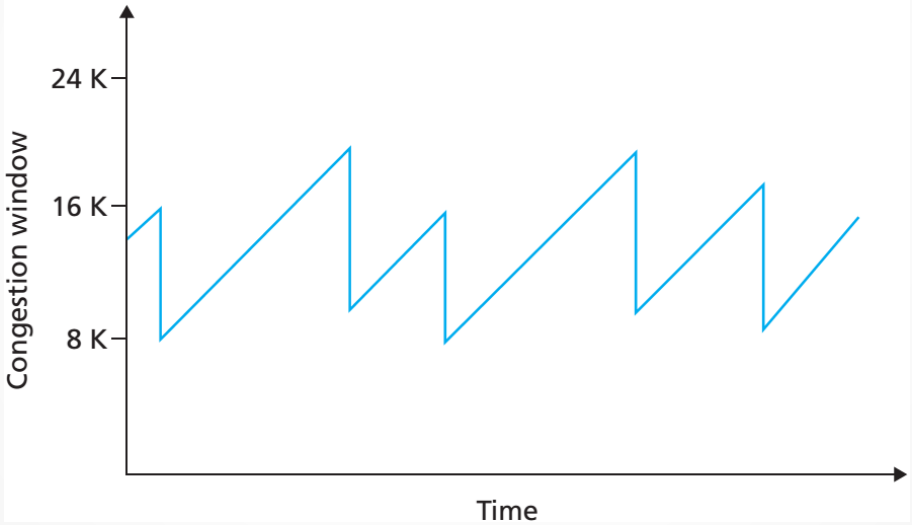
# **AIMD** – INCREMENTO ADITIVO, DECREMENTO MULTIPLICATIVO

- El control de congestionamiento TCP consiste de un incremento lineal aditivo en 1 *MSS* cada *RTT* de *cwnd* y luego en el decremento en la mitad (multiplicativo) de *cwnd* ante un evento de 3 ACK duplicados
- Por esta razón, el control de congestionamiento TCP es referido como *Additive – Increase, Multiplicative – Decrease* (AIMD)
- TCP incrementa linealmente *cwnd* hasta que ocurre un evento de 3 ACK duplicados.
- Luego reduce *cwnd* en un factor de 2 , para luego incrementarla linealmente, probando a ver si hay ancho de banda disponible adicional



# **AIMD** – INCREMENTO ADITIVO, DECREMENTO MULTIPLICATIVO

- AIMD da origen al comportamiento “diente de sierra”




- TCP AIMD se desarrolló en base a experimentación y no deja de ser un campo abierto a la investigación

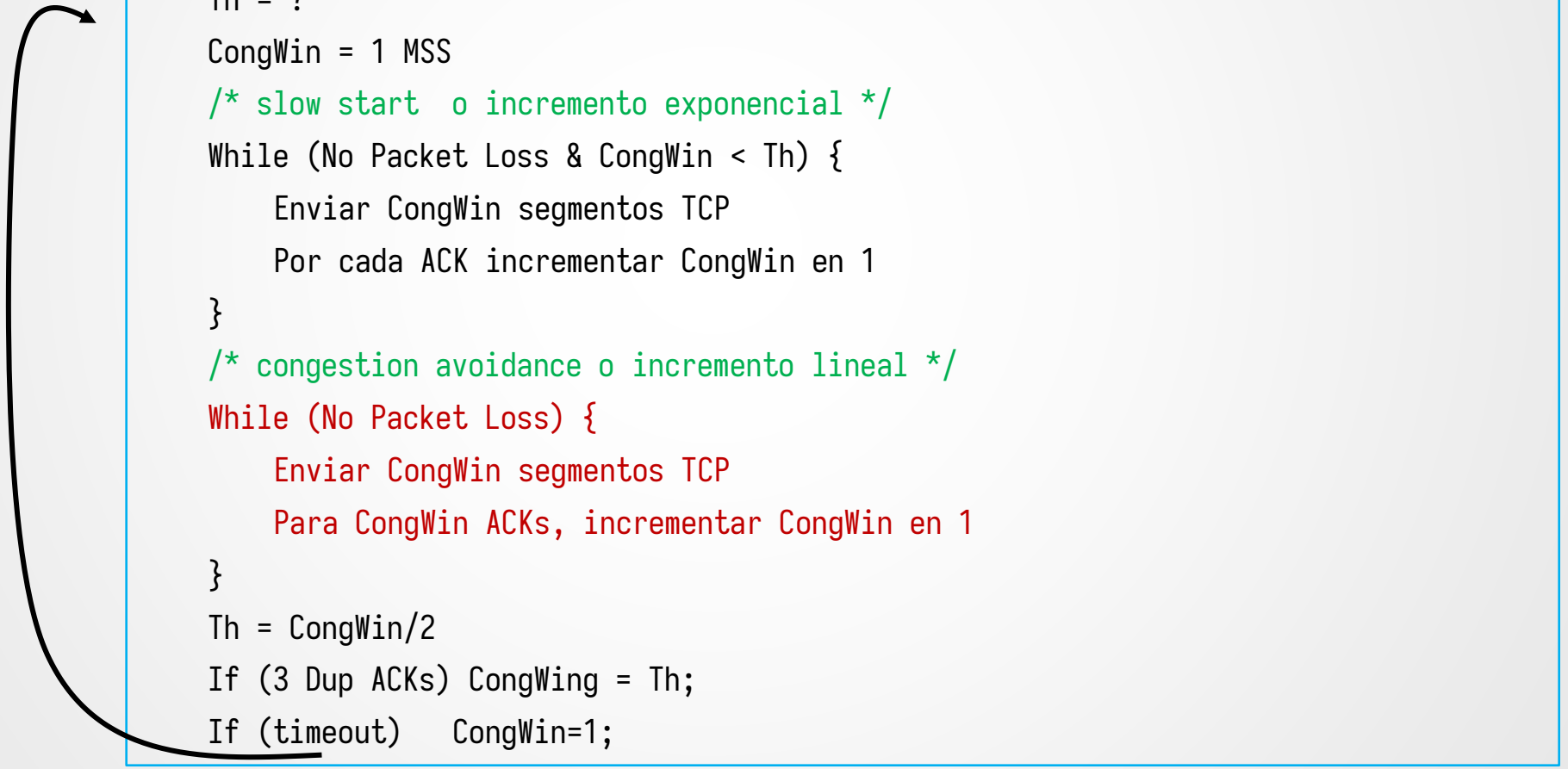
# RESUMEN: CONTROL DE CONGESTIONAMIENTO TCP

- Cuando *CongWin* esta por debajo del *Threshold*, el emisor esta en fase slow-start, la ventana crece exponencialmente
- Cuando *CongWin* esta por encima del *Threshold*, el emisor esta en fase congestion-avoidance, la ventana crece linealmente
- Cuando ocurre triple ACK duplicado, el *Threshold* se iguala a *CongWin*/2 y *CongWin* se iguala al *Threshold*
- Cuando ocurre un *timeout*, el *Threshold* se iguala a *CongWin*/2 y *CongWin* se iguala a 1 *MSS*

# CONTROL DE CONGESTIONAMIENTO EN EMISOR TCP

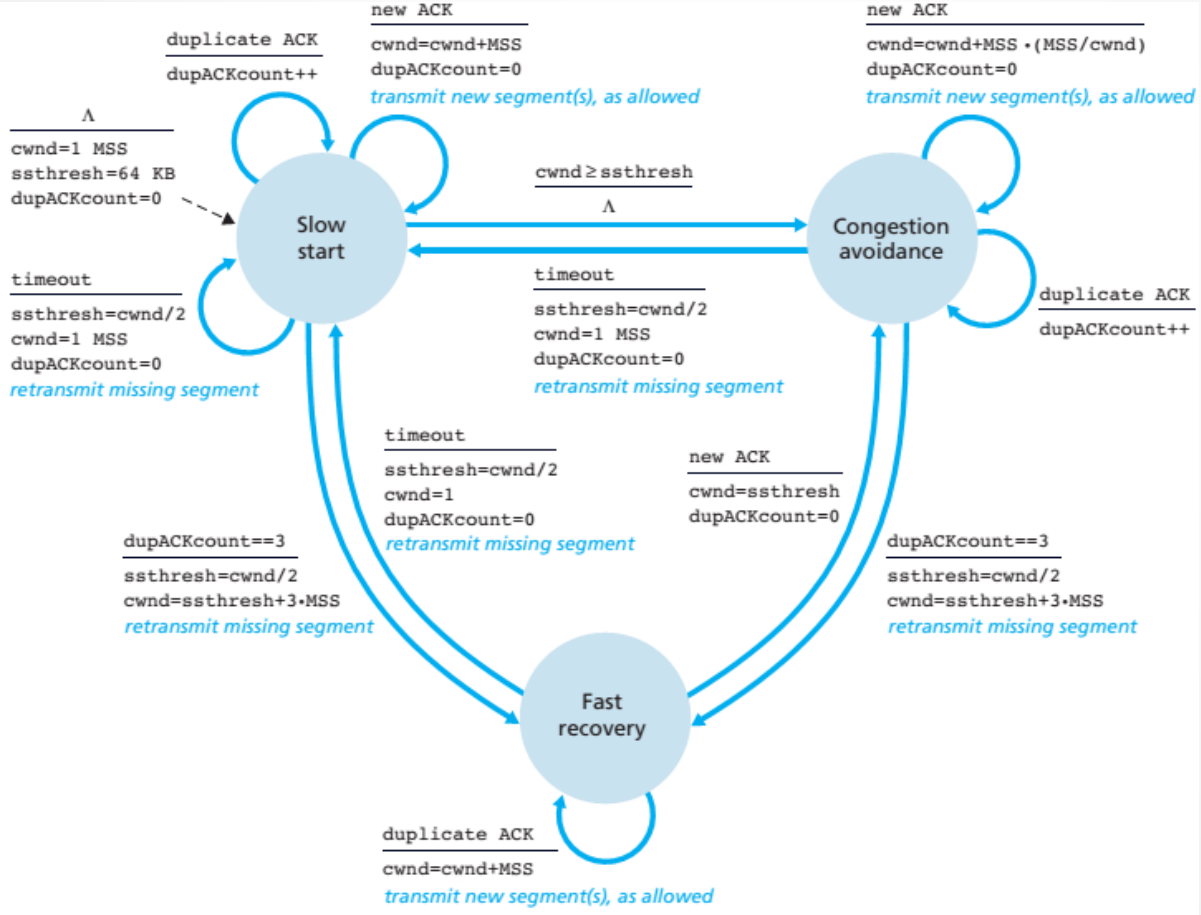
| Estado | Evento | Acción de Emisor TCP | Comentario |
|---|---|---|---|
| Slow Start (SS) | ACK recibido por datos previamente no confirmados | CongWin = CongWin + MSS;<br>Si (CongWin > Threshold)<br>pasar a estado “Congestion Avoidance” | CongWin se duplica cada RTT |
| Congestion Avoidance (CA) | ACK recibido por datos previamente no confirmados | CongWin = CongWin+MSS * (MSS/CongWin) | Incremento aditivo: CongWin se incrementa en 1 MSS cada RTT |
| SS o CA | Pérdida detectada por tres ACK duplicados | Threshold = CongWin/2;<br>CongWin = Threshold;<br>pasar a estado “Congestion Avoidance” | Recuperación rápida. Decremento multiplicativo: CongWin no se reduce por debajo de 1MSS |
| SS o CA | Timeout | Threshold = CongWin/2;<br>CongWin = 1 MSS;<br>pasar a estado“Slow Start” | Slow start |
| SS o CA | ACK duplicado | Incrementar contador de ACK duplicados para el segmento que se esta confirmando | CongWin y Threshold no cambian |

# ALGORITMO DE CONTROL DE CONGESTIONAMIENTO



```
Th = ?
CongWin = 1 MSS
/* slow start  o incremento exponencial */
While (No Packet Loss & CongWin < Th) {
    Enviar CongWin segmentos TCP
    Por cada ACK incrementar CongWin en 1
}
/* congestion avoidance o incremento lineal */
While (No Packet Loss) {
    Enviar CongWin segmentos TCP
    Para CongWin ACKs, incrementar CongWin en 1
}
Th = CongWin/2
If (3 Dup ACKs) CongWing = Th;
If (timeout)   CongWin=1;
```

# MEF DE CONTROL DE CONGESTIONAMIENTO TCP

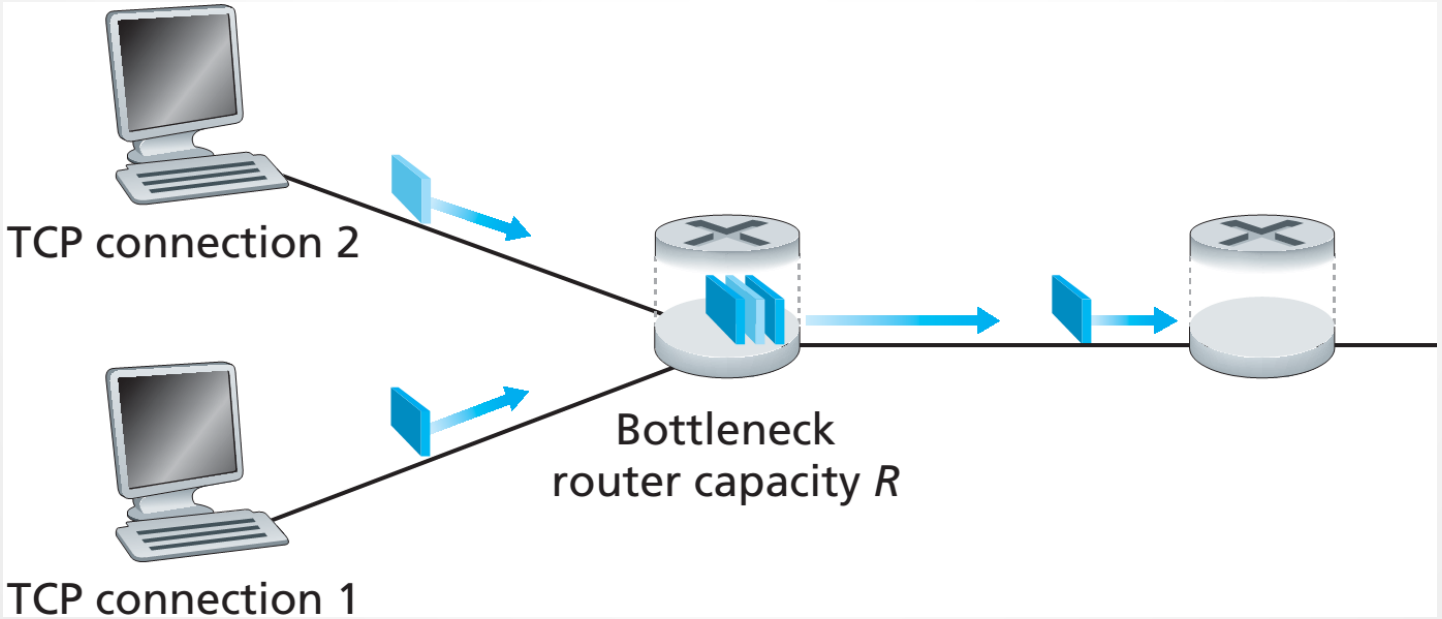


# RENDIMIENTO TCP

- ¿Cuál es el rendimiento promedio de TCP en función del tamaño de ventana y el RTT?
  - Ignore slow start
  - Sea W el tamaño de la ventana cuando ocurre una perdida.
  - Cuando la ventana es W, el rendimiento es W/RTT
  - Justo después de la perdida, la ventana cae a W/2 y el rendimiento a W/2RTT.
  - Rendimiento promedio = 0.75W/RTT

# EQUIDAD DE TCP

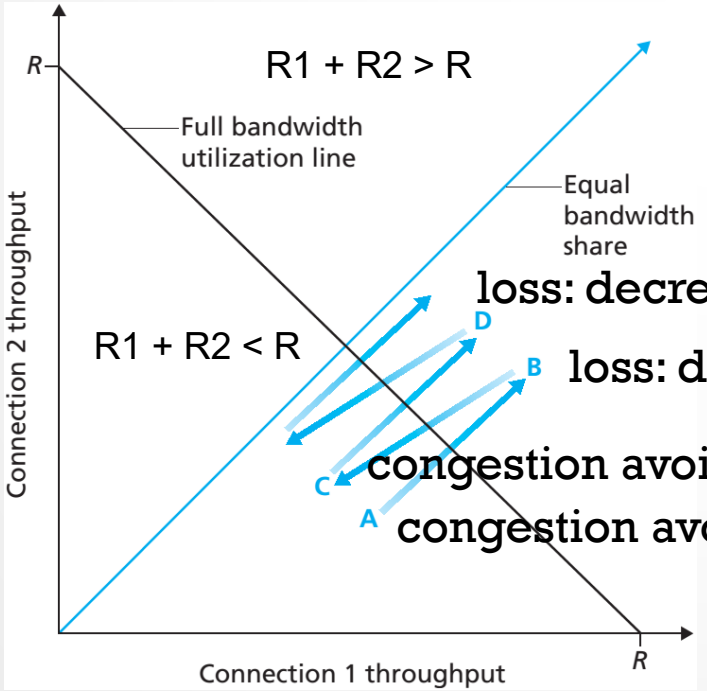
Objetivo de equidad: Si K sesiones TCP comparten el mismo enlace con ancho de banda R, cada uno debe tener una tasa promedio de R/K



# EQUIDAD DE TCP

Dos sesiones concurrentes:

- Incremento aditivo genera una pendiente de 1 a medida se incrementa
- Decremento multiplicativo disminuye el rendimiento proporcionalmente




loss: decrementa ventana a la mitad

loss: decrementa ventana a la mitad

congestion avoidance: incremento aditivo

congestion avoidance: incremento aditivo

# EQUIDAD TCP

## Equidad y UDP

- Las aplicaciones multimedia generalmente no usan TCP
  - No desean ahogar la tasa por el control de congestionamiento
- En su lugar usan UDP:
  - Bombean audio/video a tasa constante, toleran pérdida de paquetes

## Equidad y conexiones TCP paralelas

- Nada prohíbe a las aplicaciones de abrir conexiones paralelas entre 2 hosts.
- Los navegadores web lo hacen
- Por ejemplo: un enlace de tasa R que soporta 9 conexiones;
  - Una nueva aplicación pide 1 TCP, recibe una tasa de R/10
  - Una nueva aplicación pide 11 TCPs, recibe R/2 !

# EJERCICIOS

1. Calcule TimeoutInterval (RTO) si
    1. RTT estimado = 10
    2. RTT muestreado = 8
    3. Alfa = 0.15
    4. Desviación RTT = 0.05
    5. Beta = 0.5
    6. RTO = ?
2. si cwnd = 1 MSS = 600B y RTT = 300ms, la tasa inicial de transmisión será: ?
3. Determine el valor de rwnd que notificará el receptor al emisor si RcvBuffer = 2048, LastByteRead = 456 y LastByteRcvd = 789
    1. rwnd = ?
4. Estudie el mecanismo de control de congestionamiento ABR de ATM